Projet Ajacs

Deliverable WP3

Enforcement mechanisms for JavaScript and modular analyses for APIs

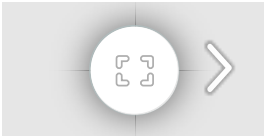
December 2017

This deliverable includes the following articles describing work done on WP3 during the last 18 months.

- **Using JavaScript Monitoring to Prevent Device Fingerprinting**, Frdric Besson, Nataliia Bielova, Thomas Jensen.
- **Spot the Difference: Secure Multi-Execution and Multiple Facets**, Nataliia Bielova, Tamara Rezk.
- **DOM: Specification and Client Reasoning**, Azalea Raad, Jos Fragoso Santos, Philippa Gardner.
- **Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate**, Karthikeyan Bhargavan, Bruno Blanchet, Nadim Kobeissi.
- **Formal Modeling and Verification for Domain Validation and ACME**, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Nadim Kobeissi.
- **Automated Verification for Secure Messaging Protocols and their Implementations: A Symbolic and Computational Approach**, Nadim Kobeissi, Karthikeyan Bhargavan, Bruno Blanchet.
- **On the Content Security Policy Violations Due to the Same-Origin Policy**, Dolire Francis Som, Nataliia Bielova, Tamara Rezk.
- **Type Abstraction for Relaxed Noninterference**, Raimil Cruz, Tamara Rezk, Bernard Serpette, ric Tanter.
- **Control What You Include ! Server-Side Protection against Third Party Web Tracking**, Dolire Francis Som, Nataliia Bielova, and Tamara Rezk.

Next issue: **January 2018**
Special theme:
Quantum Computing
Call for the next issue (/call)

(/images /stories/EN106/EN106.epub)
This issue in ePub format (/images/stories/EN106 /EN106.epub)



Powered by P

Using JavaScript Monitoring to Prevent Device Fingerprinting (/en106/special/using-javascript-monitoring-to-prevent-device-fingerprinting)

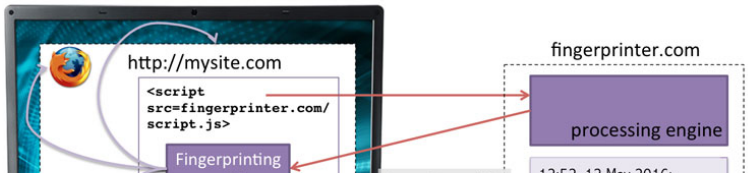
Special Theme (/En106/special) Hits: 1694

by Nataliia Bielova, Frédéric Besson and Thomas Jensen (Inria)

Today's Web users are continuously tracked as they browse the Web. One of the techniques for tracking is device fingerprinting that distinguishes users based on their Web browser and operating system properties. We propose solutions to detect and prevent device fingerprinting via runtime monitoring of JavaScript programs.

The use of sophisticated web tracking technologies has grown enormously in the last decade. Advertisement companies and tracking agencies are collecting increasing amounts of data about Web users in order to better advertise their products. Social media plugins also collect data to learn about online habits and preferences of their users. In the last five years, researchers have started to examine the mechanisms used for Web tracking. Recent research has shown that advertising agencies and networks use a wide range of techniques in order to track users across the Web.

Web tracking via cookies is well known. Cookies are stored in a user's browser so that the tracking script can immediately recognise the user. However, another group of tracking techniques, called 'device fingerprinting', does not require storing anything in a user's browser. Fingerprinting scripts make a snapshot of the configuration of the Web browser and operating system properties and then are able to distinguish a particular user from all other website visitors. Unlike cookies, this technique also works perfectly across sites, meaning that the tracker will know all the web sites that the user has visited if this tracker's script is present on these sites. The Panopticlick project [L1] by Electronic Frontier Foundation was the first to demonstrate the power of fingerprinting in 2010. Since then, researchers have found new ways to distinguish Web users, for example through HTML5 Canvas fingerprinting, which was discovered only in 2012.



Get the latest issue to your desktop
RSS 2.0 (/?format=feed&type=rss)

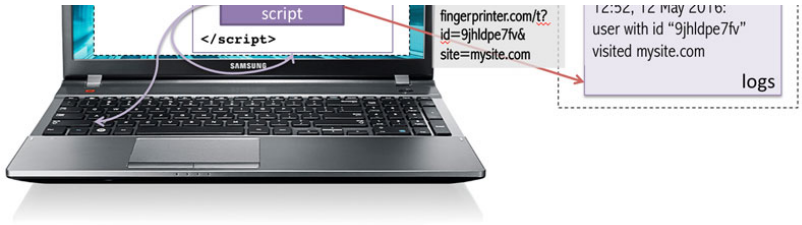


Figure 1: Device fingerprinting: a fingerprinting script collects data about Web browser and operating system properties, such as Web browser version, list of installed plugins, screen resolution, time zone etc., encodes it into a string and sends it back to fingerprinter.com.

Within the French ANR projects Seccloud (Security of cloud programming) and AJACS (Analyses of JavaScript Applications: Certification and Security), in INRIA, we have proposed a new solution to protect Web users from being fingerprinted. We are developing a tool that formally guarantees that the scripts run in a browser are not fingerprinting the user. This can be done either by detecting and blocking tracking scripts, or by modifying their tracking behaviour. To do so, we developed a monitor that analyses a potentially tracking script, and computes how much fingerprinting information this script collects. The more information it collects, the more easily it can distinguish the user from all other visitors.

As a first step, we have developed a methodology to analyse how much identifiable information a tracker may learn about a user through an execution of a script. While a script runs, it collects some data about the Web browser and operating system configuration and sends this data back to the server. How much identifiable information did the tracker learn by observing this data? We have shown that this problem can be stated as an information-flow problem that answers the question: what is the probability that a server can identify a user after analysing the output of the script? If the probability is low, the user is unlikely to be tracked. If the probability is high, this is a tracking script trying to identify the user.

We have also developed a quantitative information flow monitor [1] computing how much the tracker learns when running a script in the user's browser. The monitor uses a combination of dynamic and static analysis and over-approximates on-the-fly the amount of information that is learnt by running the script. If the amount of information is below a threshold, it is safe to send the output to the server. Otherwise, counter-measures need to be taken, such as shutting down the connection or providing forged but credible output. The theoretical foundations of such browser randomisation were developed in [2].

Next, we recently proposed a new version of a quantitative information flow monitor that is more precise in computing the knowledge of the tracker [3]. This new version expresses the monitoring of attacker knowledge in a general framework of semantics-based program analysis, and shows how a knowledge monitor can be combined with existing monitoring techniques for information flow control, such as the 'no-sensitive-upgrade' principle.

Link:
[L1] https://panopticlick.eff.org/ (https://panopticlick.eff.org/)

References:
[1] F. Besson, N. Bielova, T. Jensen: "Hybrid Information Flow Monitoring Against Web Tracking", IEEE CSF 2013.
[2] F. Besson, N. Bielova, T. Jensen: "Browser Randomisation against

Fingerprinting: A Quantitative Information Flow Approach", NordSec 2014.
[3] F. Besson, N. Bielova, T. Jensen: "Hybrid Monitoring of Attacker Knowledge", IEEE CSF 2016.

Please contact:
Nataliia Bielova, Inria, France
+33 4 92 38 77 87
nataliia.bielova@inria.fr (mailto:nataliia.bielova@inria.fr)

ERCIM articles (sections "special theme" and "research and innovation) are referenced by DBLP (http://dblp.uni-trier.de/db/journals/ercim/index.html)

ERCIM News is a joint publication of CNR (https://www.ercim.eu/cnr) CWI (https://www.ercim.eu/cwi) Fraunhofer (https://www.ercim.eu/fhg) FNR (https://www.ercim.eu/fnr) FORTH (https://www.ercim.eu/forth) INESC (https://www.ercim.eu/inesc) INRIA (https://www.ercim.eu/inria) ISI (https://www.ercim.eu/isi) NTNU (https://www.ercim.eu/ntnu) SBA (https://www.ercim.eu/sba) SICS (https://www.ercim.eu/sics) SZTAKI (https://www.ercim.eu/sztaki) UCY (https://www.ercim.eu/ucy) UWAW (https://www.ercim.eu/uwaw) VTT (https://www.ercim.eu/vtt) © ERCIM 2016 | legal information (http://www.ercim.eu/about/legal)

Spot the Difference: Secure Multi-Execution and Multiple Facets

Nataliia Bielova and Tamara Rezk

Université Côte d'Azur, Inria, France
name.surname@inria.fr

Abstract. We propose a rigorous comparison of two widely known dynamic information flow mechanisms: Secure Multi-Execution (SME) and Multiple Facets (MF). Informally, it is believed that MF simulates SME while providing better performance. Formally, it is well known that SME has stronger soundness guarantees than MF.
Surprisingly, we discover that even if we approach them to enforce the same soundness guarantees, they are still different. While modeling them in the same language, we are able to precisely identify the features of the semantics that lead to their differences. In the process of comparing them, we also discovered four new mechanisms that share features of MF and SME. We prove that one of them simulates SME, which was falsely believed to be true for MF.

1 Introduction

Information flow security [22] is an important guarantee for computer systems. A common security guarantee, called *noninterference*, requires that the secret inputs to the program do not influence (*flow into*) public outputs. In recent years, with the growing impact of highly dynamic languages such as JavaScript, a significant number of dynamic mechanisms [2–4, 12, 15, 19, 24] were proposed for information flow control and enforcement of noninterference.

A dynamic information flow mechanism is *sound* if it ensures equal observable outputs when executions start in equal observable inputs. In other words, a sound dynamic mechanism must detect all insecure executions and enforce noninterference by modifying the insecure executions. An important property of dynamic mechanisms is transparency [6, 13]. A dynamic mechanism is *transparent* if it does not modify the executions of the program that are already secure. In other words, a transparent mechanism does not have any "false positives" when it comes to detecting secure executions.

Secure Multi Execution (SME) [12] and Multiple Facets (MF) [4] are two dynamic sound mechanisms to enforce noninterference. For brevity, we call these mechanisms SME monitor and MF monitor.

The main idea behind SME is to execute a program multiple times, one for each security level. Each execution receives only input visible to its level, and a default value for inputs that should not be visible. In this way, executions cannot depend on non observable inputs. Moreover, SME uses a low priority scheduler

so that non-termination does not depend on high inputs. This allows SME to prevent information leaks due to program non-termination.

The main idea behind MF is to execute a program using faceted values, one facet for each security level. When a facet possesses nothing to be observed, there is a special value to signal this. Moreover, based on the Fenton strategy [14], MF also skips assignment to public variables in a context that depends on a secret to prevent implicit information flows.

By appropriately manipulating the faceted values, a single execution of MF is claimed to *simulate* the multiple executions of SME with the primary benefit of being more performant [1, 4, 25]:

> "Faceted evaluation is a technique for *simulating secure multi-execution* with a single process" – from [25, p. 4]
> "Austin and Flanagan [6] show how secure multi-execution can be *optimized* by executing a single program on faceted values" – from [15, p.15]

One of the two formally studied differences between SME and MF before this work is their soundness guarantee. SME enforces the soundness guarantee of *Termination-Sensitive Noninterference* (TSNI), by preventing information flows when a program has a different termination behaviour based on a secret input. However, MF is only proven to enforce *Termination-Insensitive Noninterference (TINI)*, a weaker information flow policy [22] that does not prevent leaks due to program non-termination. For transparency, SME has been proved to be TSNI precise [12,27], a flavour of monitor transparency which means that SME outputs without changes any execution of a noninterferent program. In contrast, MF is recently demonstrated not to be TINI precise [9].

In this work, we investigate if the generalized belief on the equivalence of SME and MF can be formally supported by appropriate hypotheses. Hence, we raised the following questions:

- Are these monitors essentially different or do they become semantically equivalent when adapted to the same soundness guarantees ?
- Can SME and MF actually be adapted to other soundness guarantees?

Our contributions are the following:

- A formal demonstration of the differences between SME and MF in a simple programming language. We underline their different guarantees in Section 4.
- A comparison of different SME-based and MF-based monitors with respect to soundness and transparency. We have discovered four new monitors:
  - SME-TINI monitor, based on SME, which enforces a weaker termination-insensitive noninterference policy than SME.
  - MFd monitor, based on MF, which is semantically equivalent to SME-TINI (Section 5).
  - MFd-TSNI monitor, based on MFd, semantically equivalent to the original SME (Section 6).
  - MF-TSNI monitor, based on MF, which enforces a stronger termination-sensitive noninterference policy than original MF.

The comparison of the guarantees of all the monitors described in this paper is summarized in Fig. 8 (Section 8). The companion technical report [8] includes all the proofs as well as more details and a formalization of the MFd-TSNI monitor in a language with input and output channels as the one of [12].

2 Soundness and Transparency

The syntax of the language to demonstrate our technical results is:

(programs) $P ::= \mathsf{skip} \mid x{:=}\ e \mid P_1; P_2 \mid \mathsf{if}\ x\ \mathsf{then}\ P_1\ \mathsf{else}\ P_2 \mid \mathsf{while}\ x\ \mathsf{do}\ P$
(expressions) $e ::= \ v \ \mid x \mid e_1 \oplus e_2$

The language's expressions include constants or values (v), variables (x) and operators ($\oplus$) to combine them. We present the standard big-step deterministic semantics denoted by $(P, \mu) \Downarrow \mu'$, where P is the program, and μ is a memory mapping variables to values (Fig. 1).

$$\textsc{skip}\ \frac{}{(\mathsf{skip}, \mu) \Downarrow \mu} \qquad \textsc{assign}\ \frac{}{(x := e, \mu) \Downarrow \mu[x \mapsto [\![e]\!]_\mu]} \qquad \textsc{seq}\ \frac{(P_1, \mu) \Downarrow \mu' \qquad (P_2, \mu') \Downarrow \mu''}{(P_1; P_2, \mu) \Downarrow \mu''}$$

$$\textsc{if}\ \frac{[\![x]\!]_\mu = \alpha \qquad (P_\alpha, \mu) \Downarrow \mu'}{(\mathsf{if}\ x\ \mathsf{then}\ P_{true}\ \mathsf{else}\ P_{false}, \mu) \Downarrow \mu'} \qquad \textsc{while}\ \frac{(\mathsf{if}\ x\ \mathsf{then}\ P; \mathsf{while}\ x\ \mathsf{do}\ P\ \mathsf{else}\ \mathsf{skip}, \mu) \Downarrow \mu'}{(\mathsf{while}\ x\ \mathsf{do}\ P, \mu) \Downarrow \mu'}$$

where $[\![x]\!]_\mu = \mu(x)$, $[\![v]\!]_\mu = v$ and $[\![e_1 \oplus e_2]\!]_\mu = [\![e_1]\!]_\mu \oplus [\![e_2]\!]_\mu$

Fig. 1: Language semantics

Noninterference We assume a two-element security lattice with $L \sqsubseteq H$ and a security environment Γ that maps program variables to security levels. By $\mu_{\mathtt{L}}$ we denote the projection of the memory μ on low variables, according to an implicitly parameterized security environment Γ. We first define noninterferent executions, following [9].

Definition 1 (Termination-Sensitive Noninterference for $\mu_{\mathtt{L}}$). *Given a semantics relation $\Downarrow$, program P is* termination-sensitive noninterferent for an initial low memory $\mu_{\mathtt{L}}$, *written $TSNI_{\Downarrow}(P, \mu_{\mathtt{L}})$, if and only if for all memories μ^1 and μ^2, such that $\mu^1_{\mathtt{L}} = \mu^2_{\mathtt{L}} = \mu_{\mathtt{L}}$ we have $\exists \mu'.(P, \mu^1) \Downarrow \mu' \Rightarrow \exists \mu''.(P, \mu^2) \Downarrow \mu'' \wedge \mu'_{\mathtt{L}} = \mu''_{\mathtt{L}}$.*

Program P is *termination-sensitive noninterferent*, written $TSNI_{\Downarrow}(P)$, if all its executions are TSNI, that is, for all $\mu_{\mathtt{L}}$, $TSNI_{\Downarrow}(P, \mu_{\mathtt{L}})$ holds.

Example 1. Consider Program 1, where variable `h` can take only two possible values: 0 and 1.

```
1 l = 1; if h = 1 then (while true do skip)
```
Program 1

If an attacker observes that `l=1`, she learns that `h` was 0, and if she doesn't see any program output (divergence), then she learns that `h` was 1. TSNI captures this kind of information leakage, hence TSNI doesn't hold.

A weaker security condition, called *termination-insensitive noninterference* (*TINI*), allows information leakage through program divergence.

Definition 2 (Termination-Insensitive Noninterference for $\mu_{\mathtt{L}}$). *Given a semantics relation $\Downarrow$, program P is* termination-insensitive noninterferent for an initial low memory $\mu_{\mathtt{L}}$*, written $TINI_{\Downarrow}(P, \mu_{\mathtt{L}})$, if and only if for all μ^1 and μ^2, such that $\mu^1_{\mathtt{L}} = \mu^2_{\mathtt{L}} = \mu_{\mathtt{L}}$, we have $\exists \mu'.(P, \mu^1) \Downarrow \mu' \wedge \exists \mu''.(P, \mu^2) \Downarrow \mu'' \Rightarrow \mu'_{\mathtt{L}} = \mu''_{\mathtt{L}}$.*

Program P is *termination-insensitive noninterferent*, written $TINI_{\Downarrow}(P)$, if all its executions are TINI, that is, for all $\mu_{\mathtt{L}}$, $TINI_{\Downarrow}(P, \mu_{\mathtt{L}})$ holds[1].

Termination-insensitive noninterference is a strictly weaker property than termination-sensitive noninterference [22]. For example, Program 1 is insecure with respect to TSNI, however it is secure with respect to TINI since whenever a program execution terminates, it always finishes in a memory with `l=1`.

Monitor Soundness and Transparency To define a *sound monitor* for termination-sensitive (resp., -insensitive) noninterference, we only substitute the semantics relation $\Downarrow$ with the monitor semantics relation $\Downarrow_M$ in the definitions of TINI and TSNI. Instead of using a subscript $\Downarrow_M$ (e.g., in $TINI_{\Downarrow_M}$) for a semantics of a monitor M, we will use a subscript M (e.g., $TINI_M$).

Definition 3 (Soundness). *Monitor M is* termination-sensitive (resp., -insensitive) sound *if for all programs P, $TSNI_M(P)$ (resp., $TINI_M(P)$).*

A number of works on dynamic information flow monitors try to analyse transparency of monitors. Intuitively, transparency describes how often a monitor accepts (doesn't block or modify) secure program executions without changing the original semantics. Different approaches have been taken to compare transparency of monitors (see [9] for a survey): in this work, we adhere to the standard meaning [6, 13] of "transparency" as the capability of a monitor to accept secure executions and use the term "precision" as the capability to accept all executions of secure programs. To formally define transparency, we first define a predicate $\mathcal{A}(P, \mu, M)$ (where $\mathcal{A}$ stands for "accepted") that holds if:

- whenever a program P terminates for an initial memory μ, then the monitor M will also terminate on μ, producing the same final memory as the original program: $\exists \mu'.\ (P, \mu) \Downarrow \mu' \Rightarrow (P, \mu) \Downarrow_M \mu'$, and
- whenever a program P does not terminate for an initial memory μ (denoted by $\perp$), then the monitor does not terminate for μ: $(P, \mu) \Downarrow \perp \Rightarrow (P, \mu) \Downarrow_M \perp$.

[1] In the following, we don't write the semantics relation $\Downarrow$ when we mean the original program semantics and the semantics is clear from the context.

The notion of *transparency* for TSNI (TINI) requires a monitor to accept all the TSNI (TINI) executions of a program. Our choice of transparency definition is based on the original literature on runtime monitors [6,13], which requires that if a program execution is secure (noninterferent), then the monitor must accept this execution without modifications. Our definition is similar to the one of [21], which considers both terminating and nonterminating executions, however it differs because we don't require the set of executions accepted by a monitor and the set of noninterferent executions to be equal.

Definition 4 (Transparency). *Monitor M is* TSNI (resp., TINI) transparent *if for any program P, and any memory μ, $TINI(P, \mu_{\mathtt{L}}) \Rightarrow \mathcal{A}(P, \mu, M)$ (resp., $TSNI(P, \mu_{\mathtt{L}}) \Rightarrow \mathcal{A}(P, \mu, M)$).*

3 SME and MF Original Semantics

In order to compare SME and MF, we first model them in the same language defined in Section 2. The semantics relation of a command P is denoted by $\Gamma \vdash (P, \mu) \Downarrow_M \mu'$ where Γ is a security environment, and M is the name of the monitor and $\Downarrow_M$ relates a program configuration and a memory. Both SME and MF monitors have deterministic semantics.

Secure Multi-Execution (SME) Devriese and Piessens proposed secure multi-execution (SME) [12]. The idea of SME is to execute the program multiple times: one for each security level. SME has two mechanisms to enforce noninterference:

- Each execution receives only inputs visible to its security level and a fixed default value for each input that should not be visible to the execution. This default value predefines a so-called "default" execution, so that under SME all the interferent executions would behave like a "default" execution.
- A low priority scheduler ensures that lower executions do not depend on the termination of higher executions. Therefore, the low priority scheduler ensures that the program termination based on a secret input does not influence a public output, and hence enforces TSNI.

$$\text{SME}\ \frac{(P, \mu|_\Gamma) \Downarrow \mu_2 \qquad \mu_1 = \begin{cases} \mu' & \mathsf{if}\ \exists \mu'.(P, \mu) \Downarrow \mu' \\ \bot & \mathsf{otherwise} \end{cases}}{\Gamma \vdash (P, \mu) \Downarrow_{\mathbf{SME}} \mu_1 \odot_\Gamma \mu_2}$$

where

$$\mu|_\Gamma(x) = \begin{cases} \mathsf{def}_H & \Gamma(x) = H \\ \mu(x) & \Gamma(x) = L \end{cases} \qquad \mu_1 \odot_\Gamma \mu_2(x) = \begin{cases} \mu_1(x) & \Gamma(x) = H \\ \mu_2(x) & \Gamma(x) = L \end{cases}$$

Fig. 2: Secure Multi-Execution semantics (SME)

The SME adaptation for the while language, taken from [9], is given in Fig. 2, with executions for levels L and H. The $\mu|_\Gamma$ function substitutes the values of all the high variables in μ with the default value def_H, such that all the insecure program executions will behave as an execution predefined by def_H.

Example 2 (SME imitates "default" executions). Consider the following program and assume that the SME's default value is $\texttt{def}_H\texttt{=0}$.

```
1 l = 1; if h = 0 then l = 0
```
Program 2

A "default" execution would take $\texttt{def}_H$ value instead of a real high value and compute the final memory with `l=0`. This program is not TINI, and therefore all its executions will terminate under SME with the memory where `l=0`.

In our SME semantics, the special runtime value $\perp$ represents the idea that no value can be observed (notice that original programs use only standard values). We overload the symbol to also denote a memory that maps every variable to $\perp$. Using memory $\perp$ we simulate the low priority scheduler of SME in our setting: *if the high execution does not terminate*, the low observer will still see the low part of the memory in the SME semantics. In this case all the high variables, whose values should correspond to values obtained in the normal execution of the program, are given value $\perp$. We model the final memory by a merging function $\odot_\Gamma$ that combines high and low parts of two final memories from high and low executions. Notice that even though the semantics becomes non computable, this model allows us to prove the same results as for the original SME and further use it for comparison with MF.

Example 3 (SME prevents leakage through non-termination). Consider Program 3:

```
1 if l = 0 then (while h=0 do skip)
2 else (while h=1 do skip)
```
Program 3

This program is TINI but not TSNI. Assume $\mu = [\texttt{h=1, l=0}]$ and that the default high value used by SME is $\texttt{def}_H\texttt{=1}$. The program terminates on memory μ, producing `l=0`, however there exists a memory $\mu' = [\texttt{h=0, l=0}]$, low-equal to μ, on which the original program doesn't terminate, thus leaking secret information through non-termination. SME prevents such leakage, because SME terminates on both memories μ and μ' producing `l=0`.

Multiple Facets (MF) Austin and Flanagan [4] proposed multiple facets (MF). In MF, each variable is mapped to several values or facets, one for each security level: each value corresponds to the view of the variable for an observer at different security level. The main mechanisms used by MF are the following:

- MF uses a special value $\perp$ to signal that a variable contains no information to be observed at a given security level.
- MF uses the Fenton strategy [14] that skips *sensitive upgrades*. A sensitive upgrade is an assignment to a low variable in a high security context that may cause an implicit information flow. If there is a sensitive upgrade, MF semantics does not update the observable facet. Otherwise, if there is no sensitive upgrade, MF semantics updates it according to the original semantics.

$$\text{MF}\ \boxed{\frac{(P, \mu\uparrow_\Gamma) \downarrow_{MF} \hat{\mu}}{\Gamma \vdash (P,\mu) \Downarrow_{\mathbf{MF}} \hat{\mu}\downarrow_\Gamma}} \qquad \text{SKIP}\ \frac{}{(\mathsf{skip}, \hat{\mu}) \downarrow_{MF} \hat{\mu}}$$

$$\text{ASSIGN}\ \frac{}{(x := e, \hat{\mu}) \downarrow_{MF} \hat{\mu}[x \mapsto [e]_{\hat{\mu}}]} \qquad \text{SEQ}\ \frac{(P_1, \hat{\mu}) \downarrow_{MF} \hat{\mu}' \qquad (P_2, \hat{\mu}') \downarrow_{MF} \hat{\mu}''}{(P_1; P_2, \hat{\mu}) \downarrow_{MF} \hat{\mu}''}$$

$$\text{IF-BOT}\ \frac{[x]_{\hat{\mu}} = \langle \alpha : \bot \rangle \qquad (P_\alpha, \hat{\mu}) \downarrow_{MF} \hat{\mu}'}{(\mathsf{if}\ x\ \mathsf{then}\ P_{true}\ \mathsf{else}\ P_{false}, \hat{\mu}) \downarrow_{MF} \hat{\mu}' \otimes \hat{\mu}}$$

$$\text{IF-VAL}\ \frac{[x]_{\hat{\mu}} = \langle \alpha_1 : \alpha_2 \rangle \qquad \alpha_2 \neq \bot \qquad (P_{\alpha_1}, \hat{\mu}) \downarrow_{MF} \hat{\mu}_1 \qquad (P_{\alpha_2}, \hat{\mu}) \downarrow_{MF} \hat{\mu}_2}{(\mathsf{if}\ x\ \mathsf{then}\ P_{true}\ \mathsf{else}\ P_{false}, \hat{\mu}) \downarrow_{MF} \hat{\mu}_1 \otimes \hat{\mu}_2}$$

$$\text{WHILE}\ \frac{(\mathsf{if}\ x\ \mathsf{then}\ P; \mathsf{while}\ x\ \mathsf{do}\ P\ \mathsf{else}\ \mathsf{skip}, \hat{\mu}) \downarrow_{MF} \hat{\mu}'}{(\mathsf{while}\ x\ \mathsf{do}\ P, \hat{\mu}) \downarrow_{MF} \hat{\mu}'}$$

where

$$\mu\uparrow_\Gamma(x) = \begin{cases} \langle \mu(x) : \bot \rangle & \text{if } \Gamma(x) = H \\ \langle \mu(x) : \mu(x) \rangle & \text{if } \Gamma(x) = L \end{cases} \qquad \hat{\mu}\downarrow_\Gamma(x) = \begin{cases} \hat{\mu}(x)_1 & \text{if } \Gamma(x) = H \\ \hat{\mu}(x)_2 & \text{if } \Gamma(x) = L \end{cases}$$

and $\hat{\mu}_1 \otimes \hat{\mu}_2(x) = \langle \hat{\mu}_1(x)_1 : \hat{\mu}_2(x)_2 \rangle$

Fig. 3: Multiple Facets semantics (MF)

Our adaptation of MF semantics is given in Fig. 3, where we use the following notation: a faceted value, denoted $\hat{v} = \langle v_1 : v_2 \rangle$, is a pair of values v_1 and v_2. The first value presents the view of an observer at level H and the second value the view of an observer at level L. In the syntax, we interpret a constant v as the faceted value $\langle v : v \rangle$. The evaluation of faceted expressions is strict in $\bot$ – if an expression contains $\bot$ then it evaluates to $\bot$ – and it is defined as follows:

$$[\hat{v}]_{\hat{\mu}} = \hat{v} \qquad [x]_{\hat{\mu}} = \hat{\mu}(x) \qquad [e_1 \oplus e_2]_{\hat{\mu}} = [e_1]_{\hat{\mu}} \oplus [e_2]_{\hat{\mu}}$$

where $\langle v_1 : v_1' \rangle \oplus \langle v_2 : v_2' \rangle = \langle v_1 \oplus v_2 : v_1' \oplus v_2' \rangle$.

Faceted memories, ranged over by $\hat{\mu}$, are mappings from variables to faceted values. A function $\mu\uparrow_\Gamma$ creates a faceted memory from a memory μ using the labelling function Γ, and function $\hat{\mu}\downarrow_\Gamma$ erases facets from the faceted memory $\hat{\mu}$ and returns a normal memory μ. We use the notation $\hat{\mu}(x)_i$ $(i \in \{1, 2\})$ for the first or second projection of a faceted value stored in x. Similar to the formalisation of SME, the special runtime value $\bot$ represents the idea that no value can be observed (program syntax only uses standard values). Moreover, MF skips any operation that depends on a value $\bot$ (see rule IF-BOT in Fig. 3).

Example 4 (MF uses $\bot$ to signal "no information"). Consider the following program, that copies the secret from `h` to low variable `l` : `l = h` . Given an initial environment μ= [`h=1, l=0`], the function $\mu\uparrow_\Gamma$ creates a faceted memory $\hat{\mu}$, where `h` = $\langle 1 : \bot \rangle$. After assignment, the variable `l` will contain the faceted value of `h`, that will be projected to the $\bot$ value using the function $\hat{\mu}\downarrow_\Gamma$ to erase facets in the end of the execution.

Example 5 (MF "skips" sensitive upgrades). Consider Program 4.

```
1 l = 0; if h = 1 then l = 1 else l = 2
```
Program 4

In MF, the L facet of variable `l` will be the initial value of variable `l` since MF will not update a low variable in a high context. Therefore, all the executions of Program 4 are modified by MF, producing the final memory with `l=0`.

4 Differences between SME and MF

Even though MF is claimed to simulate SME, the example below demonstrates that even in a simple language, SME and MF semantics are different.

Example 6 (SME and MF semantics are different). Consider Program 5 and an initial memory [`h=0, l=0`].

```
1 l = 0;
2 if h = 0 then l = 1;
3 if l = 1 then l = 2 else l = 3
```
Program 5

This program terminates in MF with the final memory where `l=3` because the value of `l` is not updated due to a sensitive upgrade. In contrast, under SME with $\mathsf{def}_H = 0$, the program terminates with the final memory where `l=2`.

Projection Theorem of MF For MF semantics, a Projection Theorem [4, Thm. 1] states that a computation over a 2-faceted memory simulates 2 non-faceted computations, one per each security level. The theorem uses a projection of a faceted memory into a normal memory using the following functions (simplified for our setting), where Lev represents either a high viewer H or a low viewer L, so that $H(\langle v_1 : v_2\rangle) = v_1, L(\langle v_1 : v_2\rangle) = v_2$, and $Lev(\hat{\mu}) = \lambda x.Lev(\hat{\mu}(x))$.

The Projection Theorem states that whenever the monitor terminates[2] for some memory $\hat{\mu}$, $\Gamma \vdash (P, \hat{\mu}) \downarrow_{MF} \hat{\mu}'$, then for any viewer Lev,

$$(P, Lev(\hat{\mu})) \Downarrow Lev(\hat{\mu}').$$

The Projection Theorem may resemble to an equivalence between SME and MF semantics, however, as we have shown above, MF is not equivalent to SME.

Example 7 (Projection Theorem of MF doesn't imply equivalence to SME). Consider Program 6 and an initial memory μ=[`h=1, l=1`]. This program is TINI and TSNI and SME would terminate in the memory μ' = [`h=1, l=0`].

```
1 if h = 0 then l = 0 else l = 0
```
Program 6

[2] Notice that the original program semantics in [4] already contains rules that deal with special $\perp$ values, that skip any operation that involves a $\perp$ value.

The Projection theorem is based on the assumption that MF terminates on a given initial faceted memory. We use the function $\mu \uparrow_\Gamma$ that creates a faceted memory from a normal memory μ given a security labelling Γ. The obtained memory is $\hat{\mu} = [\texttt{h} = \langle 1 : \bot \rangle, \texttt{l} = \langle 1 : 1 \rangle]$. Upon a faceted execution of the program, the final faceted memory is $\hat{\mu}' = [\texttt{h} = \langle 1 : \bot \rangle, \texttt{l} = \langle 0 : 1 \rangle]$.

For a viewer at level H, the initial projected memory is $H(\hat{\mu}) = [\texttt{h=1, l=1}]$, and the final projected memory is $H(\hat{\mu}')$=[h=1, l=0], which corresponds to the original final memory μ'. However, only a viewer a level H is able to see this memory, while a viewer at level L will see a different memory.

For a viewer at level L, the projected initial memory is $L(\hat{\mu}) = [\texttt{h=}\bot\texttt{, l=1}]$, and the final projected memory is also $L(\hat{\mu}') = [\texttt{h=}\bot\texttt{, l=1}]$, since the mechanism of MF skips the sensitive upgrades and the value of l is not changed. It means that a viewer at level L will see l=1 in MF, however will see l=0 in SME.

Soundness Monitors that enforce TSNI and TINI are comparable with respect to soundness thanks to the fact that TSNI is a stronger guarantee than TINI [22]. SME was previously proven TSNI sound [12], and therefore SME is also TINI sound. Example 3 demonstrated how SME enforces TSNI and hence TINI soundness. In contrast, MF was previously proven TINI sound [4], however it is unable to enforce TSNI.

Example 8 (MF is not TSNI sound). Consider Program 1. When h=1, the MF semantics will diverge because the faceted value of h is $\langle 1 : \bot \rangle$ and the premises of the IF-BOT rule are not satisfied (the program diverges on line 3). However when h=0, the MF semantics will terminate with final memory where l=1.

Transparency Devriese and Piessens [12, Thm. 2] have proven that SME is TSNI precise, meaning that for TSNI secure programs, all their executions are not modified by SME.

Theorem 1 ([12, Thm. 2]). *SME is TSNI precise, meaning that for any program P, the following holds: $TSNI(P) \Rightarrow \forall \mu.\ \mathcal{A}(P, \mu, SME)$.*

In this paper, we prove a more fine-grained guarantee for SME, which is TSNI transparency. Notice that TSNI transparency is stronger than TSNI precision because it requires that the monitor not only does not modify any executions of secure programs, but also secure executions of insecure programs.

Theorem 2. *SME is TSNI transparent.*

Example 9. Consider Program 7. This program is not TSNI, however there are TSNI-secure executions of this program when initially l=1. For an initial memory where l=1, and for any default high value $\texttt{def}_H$, SME will terminate in a final memory, where l=1, like the original program.

---

1 `if l=0 then (while h=0 do skip)` **Program 7**

$$\text{SME-TINI}\ \frac{(P,\mu|_\Gamma) \Downarrow \mu_2 \qquad (P,\mu) \Downarrow \mu_1}{\Gamma \vdash (P,\mu) \Downarrow_{\textsc{smetini}} \mu_1 \odot_\Gamma \mu_2}$$

Fig. 4: SME semantics for TINI (SME-TINI)

Example 10 (MF is not TSNI and not TINI transparent). Consider Program 6, which is TSNI secure, and an initial memory [`h=1, l=1`]. The MF semantics will modify this execution. Since the test depends on a high variable `h`, the IF-BOT rule will be used to evaluate the conditional, and only the high facet of the value in `l` will be updated, getting the value 0, while the low facet will not be updated, hence the new faceted value of `l` is $\langle 0 : 1 \rangle$. Following the definition of the $\downarrow_\Gamma$ function, the final memory will contain `l=1` because $\Gamma(\texttt{l}) = L$, while the original program would terminate in the memory where `l=0`. Hence, this is a counter example for TSNI and TINI transparency of MF.

5 SME vs MF by downgrading SME to TINI

The first reason for SME and MF to be incomparable is that SME enforces termination-sensitive noninterference (TSNI), whereas MF enforces a weaker version of noninterference called termination-insensitive noninterference (TINI). To formally compare SME and MF, we modify SME semantics in order for SME to enforce the same version of noninterference as MF, which is TINI.

SME that enforces TINI (SME-TINI) We propose a version of SME, that we call SME-TINI and present its semantics in Fig. 4. SME-TINI runs the program multiple times like SME, but it does not have a low priority scheduler and hence is not sensitive to termination leaks.

Example 11 (SME-TINI does not enforce TSNI). Consider Program 1 and a default value for SME is $\texttt{def}_H = 0$. In an initial memory where `h=1`, the program will diverge in the SME-TINI semantics whereas it will terminate with the memory `l=1` in the SME semantics. In an initial memory where `h=0`, the program will terminate with `l=1` in both SME-TINI and SME semantics. This example shows that, in contrast to SME, SME-TINI does not enforce TSNI.

Theorem 3. *SME-TINI is TINI sound.*

Example 12 (SME-TINI is TINI sound). Consider Program 4 and an initial memory [`h=1,l=0`]. SME-TINI with $\texttt{def}_H = 0$ enforces TINI by always terminating in a final memory where `l=2`.

However, differently from original SME, SME-TINI does not provide transparency guarantee.

$$\text{MFD}\ \boxed{\frac{(P, \mu \uparrow_\Gamma^{\mathsf{def}}) \downarrow_{MF} \hat{\mu}}{\Gamma \vdash (P, \mu) \Downarrow_{\mathbf{MFd}} \hat{\mu} \downarrow_\Gamma}}$$

Fig. 5: Multiple Facets with default (MFd).

Example 13 (SME-TINI is not TINI transparent). Consider Program 3, an initial memory μ=[h=0, l=1] and def_H=1. The original program terminates on memory μ= [h=0, l=1]. Though program is TINI, SME-TINI does not terminate on μ because its low execution does not terminate since def_H=1.

Surprisingly, we find out that even if we downgrade SME to only enforce TINI, and SME-TINI and MF now have the same soundness guarantees, still SME-TINI and MF semantics are different.

Example 14 (SME-TINI and MF semantics are different). Consider again Program 4 and an initial memory [h=1, l=0]. SME-TINI with $\mathsf{def}_H = 0$ enforces TINI by always producing an output 2, however MF does not execute an alternative else-branch, and keeps an initial value of l, terminating with the final memory where l=0.

The main reason for a different semantics now is the way in which SME-TINI and MF treat insecure executions: while SME forces all insecure executions to behave like the "default" executions, MF uses the Fenton strategy to skip sensitive upgrades.

Multiple Facets with Default (MFd) To propose a version of MF that has the same semantics as SME-TINI, we replace the $\perp$ value of MF with def_H as the default high value (this is exactly as the default of SME). In fact, there is a maybe different default high value for each high variable, so in fact def_H is a vector of variables but for simplicity of presentation (and without loss of generality), we call it a default value and use only one high variable in our examples.

The new version of MF, that we call MFd, uses the semantics rules of MF, and instead of a $\mu \uparrow_\Gamma$ function that creates a faceted memory in the MF rule, it uses a $\mu \uparrow_\Gamma^{\mathsf{def}}$ function, that is defined as follows:

$$\mu \uparrow_\Gamma^{\mathsf{def}} (x) = \begin{cases} \langle \mu(x) : \mu(x) \rangle & \text{if } \Gamma(x) = L \\ \langle \mu(x) : \mathsf{def}_H \rangle & \text{if } \Gamma(x) = H \end{cases}$$

Therefore, the MFd semantics is presented with only one rule shown in Fig. 5. Since the MFd semantics never introduces a runtime value $\perp$, the MFd rules do not include the rule IF-BOT of the original MF semantics (Fig. 3). Notice that, the fact that the rule IF-BOT is not included implies that one of the bases of original MF, which is to skip sensitive upgrades as originally proposed by Fenton [14], is made obsolete.

Theorem 4. *MFd is TINI sound.*

To prove that MFd is equivalent to SME-TINI, we first propose the following definition of an equivalence relation on two monitor semantics.

Definition 5. *A monitor A is semantically equivalent to a monitor B, written $A \approx B$, if and only if for all programs P, all memories μ and μ', and all labelling functions Γ, the following holds:*

$$\Gamma \vdash (P, \mu) \Downarrow_A \mu' \iff \Gamma \vdash (P, \mu) \Downarrow_B \mu'.$$

Theorem 5. *MFd $\approx$ SME-TINI.*

Example 15 (MFd and SME-TINI semantics are equivalent). Consider Program 4 and an initial memory [h=1, l=0]. SME-TINI with $\mathsf{def}_H = 0$ always terminates in a final memory where l=2. MFd also terminates in a final memory with l=2, because differently from original MF, it does not skip the sensitive upgrades but rather uses the results of the "default" execution, like SME.

6 SME vs MF by upgrading MFd to TSNI

By analysing SME and MF semantics, we concluded that they are different for two reasons. First, SME enforces TSNI, while MF enforces TINI. In the previous section we have downgraded SME to enforce a weaker property TINI, however the resulting SME-TINI monitor was not semantically equivalent to MF. Therefore, we have found the second reason for their difference: while SME is using a default value for high variables in the low execution, MF uses special runtime values $\bot$, allowing the execution of some branches to be skipped.

In the previous section we proposed a new version of MF, called MFd, that solves the second difference of SME and MF, but does not solve the first one: MFd does not have the same strong soundness guarantee, TSNI, that original SME has. Therefore, we propose modifications to the MFd semantics in order for MFd to enforce TSNI.

We propose a new monitor, that we call MFd-TSNI, and present its semantics in Fig. 6. The main difference between MFd and MFd-TSNI is the embedding of a low priority scheduler to schedule with priority the low facet in the execution. This can be observed in the rules IF-VAL and IF-BOT-VAL. The rule IF-VAL simulates the idea behind the low priority scheduler from original SME. The symbol $\bot$ is overloaded to denote a memory that maps every variable to $\bot$ when the high execution does not terminate. We illustrate the efficiency of MFd-TSNI in enforcing TSNI in the following example.

Example 16. Consider Program 8 which is not TSNI and initial memory μ= [h=1, l=1], the default value used to create a faceted memory is def_H= 0.

Program 8

```
1 if h=1 then (while true skip);
2 if h=0 then l=0
```

$$\text{MFd-TSNI}\ \boxed{\frac{(P, \mu \uparrow_\Gamma^{\mathsf{def}}) \downarrow_{MFT} \hat{\mu}}{\Gamma \vdash (P, \mu) \Downarrow_{\mathbf{MFdT}} \hat{\mu} \downarrow_\Gamma}} \qquad \text{SKIP}\ \frac{}{(\mathsf{skip}, \hat{\mu}) \downarrow_{MFT} \hat{\mu}}$$

$$\text{ASSIGN}\ \frac{}{(x := e, \hat{\mu}) \downarrow_{MFT} (\hat{\mu}[x \mapsto [e]_{\hat{\mu}}])} \quad \text{SEQ}\ \frac{(P_1, \hat{\mu}) \downarrow_{MFT} \hat{\mu}' \qquad (P_2, \hat{\mu}') \downarrow_{MFT} \hat{\mu}''}{(P_1; P_2, \hat{\mu}) \downarrow_{MFT} \hat{\mu}''}$$

$$\text{IF-BOT-VAL}\ \frac{[x]_{\hat{\mu}} = \langle \bot : \alpha \rangle \qquad \alpha \neq \bot \qquad (P_\alpha, \hat{\mu}) \downarrow_{MFT} \hat{\mu}'}{(\mathsf{if}\ x\ \mathsf{then}\ P_{true}\ \mathsf{else}\ P_{false}, \hat{\mu}) \downarrow_{MFT} \bot \otimes \hat{\mu}'}$$

$$\text{IF-VAL}\ \frac{\begin{array}{c}[x]_{\hat{\mu}} = \langle \alpha_1 : \alpha_2 \rangle \qquad \alpha_1 \neq \bot \\ \alpha_2 \neq \bot \qquad (P_{\alpha_2}, \hat{\mu}) \downarrow_{MFT} \hat{\mu}_2 \qquad \hat{\mu}_1 = \begin{cases} \hat{\mu}' & \mathsf{if}\ \exists \hat{\mu}'.(P_{\alpha_1}, \hat{\mu}) \downarrow_{MFT} \hat{\mu}' \\ \bot & \mathsf{otherwise} \end{cases}\end{array}}{(\mathsf{if}\ x\ \mathsf{then}\ P_{true}\ \mathsf{else}\ P_{false}, \hat{\mu}) \downarrow_{MFT} \hat{\mu}_1 \otimes \hat{\mu}_2}$$

$$\text{WHILE}\ \frac{(\mathsf{if}\ x\ \mathsf{then}\ P; \mathsf{while}\ x\ \mathsf{do}\ P\ \mathsf{else}\ \mathsf{skip}, \hat{\mu}) \downarrow_{MFT} \hat{\mu}'}{(\mathsf{while}\ x\ \mathsf{do}\ P, \hat{\mu}) \downarrow_{MFT} \hat{\mu}'}$$

Fig. 6: Multiple Facets semantics with default for TSNI (MFd-TSNI)

An initial value of `h` in the new faceted memory $\hat{\mu}$ is `h`=$\langle 1 : 0 \rangle$, while `l` = $\langle 1 : 1 \rangle$. Upon the first test, the IF-VAL rule is applied. This rule first requires that the execution corresponding to the low facet terminates, which is the case and the final faceted memory after the first test is $\hat{\mu}_2 = \hat{\mu}$. However, the program does not terminate if we use the high facet of `h`, therefore all the program variables get assigned to $\bot$ in a memory $\hat{\mu}_1$. After the combination of memories, we get the final memory after the first test, which is $\hat{\mu}_1 \otimes \hat{\mu}_2$, where `h` = $\langle \bot : 0 \rangle$ and `l` = $\langle \bot : 1 \rangle$.

Upon the second test, the IF-BOT-VAL rule is applied since the high facet of variable `h` is now $\bot$. Therefore, MFd-TSNI executes only one branch where `h`=0 and computes the final memory where `l` = $\langle 0 : 0 \rangle$.

We prove that the new monitor MFd-TSNI is semantically equivalent to original SME.

Theorem 6. *MFd-TSNI* $\approx$ *SME.*

As a direct consequence of the semantical equivalence to SME, MFd-TSNI is TSNI sound and TSNI transparent. Notice that MFd was not transparent.

Theorem 7. *MFd-TSNI is TSNI sound and TSNI transparent.*

Example 17 (MFd-TSNI is TINI sound and TSNI sound). Consider Program 4 and $\mathsf{def}_H = 0$. For any initial memory, MFd-TSNI always terminates in final memory where `l=2`, thus enforcing TINI and TSNI.

Example 18 (MFd-TSNI is TSNI transparent). Consider again Program 7. For the initial memory where `l=1`, and for any default high value def_H, MFd-TSNI will terminate in a final memory, where `l=1`, like the original program.

$$\text{MF-TSNI}\ \boxed{\frac{(P, \mu \uparrow_\Gamma) \downarrow_{MFT} \hat{\mu}}{\Gamma \vdash (P, \mu) \Downarrow_{\mathbf{MFT}} \hat{\mu} \downarrow_\Gamma}}$$

$$\text{IF-BOT}\ \frac{[x]_{\hat{\mu}} = \langle \alpha : \bot \rangle \qquad \alpha \neq \bot \qquad \hat{\mu}_1 = \begin{cases} \hat{\mu}' & \text{if } \exists \hat{\mu}'.(P_\alpha, \hat{\mu}) \downarrow_{MFT} \hat{\mu}' \\ \bot & \text{otherwise} \end{cases}}{(\text{if } x \text{ then } P_{true} \text{ else } P_{false}, \hat{\mu}) \downarrow_{MFT} \hat{\mu}_1 \otimes \hat{\mu}}$$

$$\text{IF-BOT-BOT}\ \frac{[x]_{\hat{\mu}} = \langle \bot : \bot \rangle}{(\text{if } x \text{ then } P_{true} \text{ else } P_{false}, \hat{\mu}) \downarrow_{MFT} \hat{\mu}}$$

Fig. 7: Additional rules for the Multiple Facet semantics for TSNI (MF-TSNI)

MF that enforces TSNI Given the technique we used to upgrade MFd to MFd-TSNI to enforce termination-sensitive noninterference, in this section we show how to upgrade the original MF in order to enforce TSNI using the same low priority scheduler.

The new monitor, that we call MF-TSNI, uses the $\uparrow_\Gamma$ function from MF to create a faceted memory, and uses all the rules of MFd-TSNI, with additional two rules to incorporate the possibility of having a special $\bot$ value in the low facet of the faceted value. We present these additional rules in Fig. 7.

We now prove that the new MF-TSNI monitor indeed enforces termination-sensitive noninterference.

Theorem 8. *MF-TSNI is TSNI sound.*

Example 19 (MF-TSNI is TSNI sound and TINI sound). Consider Program 1. When `h=1`, the IF-BOT rule of MF-TSNI (Fig. 7) will construct a memory $\hat{\mu}_1$, where all the variables are assigned to $\bot$ value since the high facet execution does not terminate. Therefore, MF-TSNI will terminate with the final memory where $\texttt{l} = \langle \bot : 1 \rangle$. When `h=0`, the MF-TSNI will terminate in the final memory where $\texttt{l} = \langle 1 : 1 \rangle$, thus enforcing TINI and TSNI.

MF-TSNI is not TSNI transparent for the same reason that MF is not TSNI transparent: the Fenton strategy of skipping sensitive upgrades prevents a mechanism from being transparent.

Example 20 (MF-TSNI is not TSNI transparent). Consider again Program 6, which is TSNI and TINI. For an initial memory where `l=1`, MF-TSNI will terminate in a final memory, where `l=1`, thus being not transparent.

7 Related Work

We present only SME and MF closely related work. We refer to [22, 23] for a wider overview on information flow properties, to [6, 13, 18] for a wider overview on transparency properties of monitors, and to [9, 17] for a wider overview on information flow monitors.

Originally, Secure Multi-Eexecution is presented in a while language featuring input/output commands and channels [12]. An output command produces a value that is queued in the output channel. An input command reads a value that is read from the input channel. We model SME as in [9], in a while language without channels. Instead of channels, we use memories mapping variables to values. To simulate an input (resp. output) command, our language reads (resp. writes) a variable from memory. In the original SME semantics [12], a configuration contains a pool of threads, one thread for each level. Then, a scheduler selects to execute first all steps of the lower level threads. Hence, all outputs of a low execution appear first in the output channel in the original SME semantics. The low priority scheduler is simulated in our model by the only rule of Fig. 2. In this rule, the low thread executes to the end to obtain the low part of the final memory and, *if the high thread does not terminate*, the high part of the final memory is $\perp$. Hence, the semantics becomes non computable. With the current model we can at least prove the same results as in the original SME monitor, and further use it for comparison with MF. Notice that, at the cost of simplicity we could have used the original SME language and semantics in order to have computability (we have modelled the MFd-TSNI monitor in the original SME language as a proof of concept in the companion technical report [8]).

SME is proved to be TSNI sound in Theorem 1 of [12]. Kashyap *et al.* [16] investigate different strategies for SME to also enforce several flavours of time-sensitive noninterference. Intuitively, time-sensitive noninterference is stronger than termination-sensitive noninterference because it requires that two executions starting in low-equal memories must terminate within the same number of program execution steps. Other works [10, 20, 26] have proposed other information flow properties, declassification properties, for modified SME monitors. We do not study in this work SME-based monitors for declassification. SME is proved to be TSNI precise in Theorem 2 of [12]. Notice that TSNI precision is a weaker property than transparency since a program which is not secure may still have some secure executions.

TSNI transparency does not hold for original SME because the low priority scheduler may reorder outputs compared to the original program semantics, letting outputs of low executions appear first in the output channel. Zanarini et al. [27] propose a modification to SME in order to prove a version of TSNI transparency (In fact, they prove a property called CP precision in Theorem 23 of [27], which is a weaker notion that TSNI transparency because it recognizes as secure a program that silently diverges on one branch, and terminates without producing any outputs on the other branch). In contrast, we can prove TSNI transparency in our SME model (and also CP precision) without need of the SME modifications proposed in Zanarini et al. because reordering is not visible in our model due to the lack of output channels, and intermediate outputs.

Zanarini et al. [27] also prove a version of TINI transparency for their TSNI sound SME-based monitor (Theorem 22 of [27]). Using our notations, their notion of TINI transparency is different from ours since if an execution is secure, if the original program terminates in a final memory μ and *if the monitor ter-*

minates in final memory μ', then μ and μ' should be low equal (in fact, they prove a property called ID-transparency in Theorem 22 of [27], which recognizes as transparent a monitor that always diverges).

SME is also shown TSNI sound and TSNI precise for a language featuring dynamic code evaluation [5] and adapted to reactive systems [7]. SME is implemented in a real browser called FlowFox [11], and SME guarantees via program transformations are implemented in JavaScript and Python [5].

Originally, Multiple Facets is presented in a lambda calculus with mutable reference cells and reactive input/output [4]. In contrast, we model MF in an imperative while language without mutable references. Moreover, since our language features memories that map variables to values, we use security environments as a means to create faceted values in our MF model. As we do, the original MF semantics [4] uses the special value $\perp$ in order to model the Fenton strategy [14], which roughly means to skip sensitive upgrades [2, 28] to prevent implicit flows.

MF is also modelled in [9] using an imperative while language as ours. The semantics in [9] uses security environments and program counters in order to implement the Fenton strategy. Our formalisation is simpler since we use facets and $\perp$ to do this, as in [4]. MF is proved to be TINI sound in Theorem 2 of [4] and also is extended to declassification and proved sound in Theorem 6 of [4].

Transparency guarantees of MF are studied in [9]. It was first shown that MF is not TINI transparent (more precisely, TINI transparency is called true transparency in [9]). Using a notion of false transparency, it is then shown that MF can accept more insecure executions than any other information flow monitor with the exception of SME (Table 1 of [9]). Moreover, Theorems 3 and 4 of [4] prove that MF generalizes no-sensitive upgrade monitor (NSU) [2, 28] and permissive-upgrade monitor (PU) [3]. These theorems imply that MF is relatively more transparent than NSU and PU [9].

MF has been implemented in JavaScript as a Firefox browser extension [4] and also as a Haskell Library using monads [25].

8 Conclusion

We have formally compared SME, MF, and other mechanisms derived from them. We present a summary of the comparison in Fig. 8. We have first downgraded SME to enforce only TINI, and proposed a new version of MF, called MFd, which is indeed semantically equivalent to a TINI version of SME. We then upgraded the MFd monitor to enforce TSNI and proposed a new monitor that we call MFd-TSNI. We have proven that

	Soundness		Transparency	
	TINI	TSNI	TINI	TSNI
SME	✓	✓	✗	✓
MF	✓	✗	✗	✗
SME-TINI	✓	✗	✗	✗
MFd	✓	✗	✗	✗
MFd-TSNI	✓	✓	✗	✓
MF-TSNI	✓	✓	✗	✗

Fig. 8: Summary of our results

MFd-TSNI is semantically equivalent to SME, and therefore enjoys the same TSNI soundness and TSNI transparency guarantees as SME. Finally, we propose to upgrade MF semantics so that it can also enforce termination-sensitive noninterference (TSNI). The new monitor, that we call MF-TSNI, is not semantically equivalent to MFd-TSNI, and is not TSNI transparent. Both SME [10, 20, 26], and MF [4] have been extended to handle declassification, a security property more versatile than noninterference. It is left as future work to understand if our results generalize to declassification properties in order to compare SME and MF.

Acknowledgment

We would like to thank Frank Piessens on valuable feedback on earlier versions of this paper and anonymous reviewers who helped us to improve the paper. This work has been partially supported by the ANR project AJACS ANR-14-CE28-0008.

References

1. T. Austin, K. Knowles, and C. Flanagan. Typed faceted values for secure information flow in haskell. Technical Report UCSC-SOE-14-07, University of California, Santa Cruz, 2014.
2. T. H. Austin and C. Flanagan. Efficient purely-dynamic information flow analysis. In *PLAS'09*, pages 113–124, 2009.
3. T. H. Austin and C. Flanagan. Permissive dynamic information flow analysis. In *PLAS'10*, pages 3:1–3:12. ACM, 2010.
4. T. H. Austin and C. Flanagan. Multiple facets for dynamic information flow. In *Proc. of the 39th Symposium of Principles of Programming Languages*. ACM, 2012.
5. G. Barthe, J. M. Crespo, D. Devriese, F. Piessens, and E. Rivas. Secure multi-execution through static program transformation. In *Formal Techniques for Distributed Systems - Joint 14th IFIP WG 6.1 International Conference, FMOODS 2012 and 32nd IFIP WG 6.1 International Conference, FORTE*, 2012.
6. L. Bauer, J. Ligatti, and D. Walker. Edit Automata: Enforcement Mechanisms for Run-time Security Policies. *International Journal of Information Security*, 4(1-2):2–16, 2005.
7. N. Bielova, D. Devriese, F. Massacci, and F. Piessens. Reactive non-interference for a browser model. In *Proc. of the 5th International Conference on Network and System Security (NSS 2011)*, pages 97–104. IEEE, 2011.
8. N. Bielova and T. Rezk. Spot the Difference: Secure Multi-Execution and Multiple Facets Technical Report. `https://goo.gl/b7yoQ9`.
9. N. Bielova and T. Rezk. A taxonomy of information flow monitors. In *International Conference on Principles of Security and Trust (POST 2016)*, volume 9635, pages 46–67. Springer, 2016.
10. I. Bolosteanu and D. Garg. Asymmetric secure multi-execution with declassification. In *International Conference on Principles of Security and Trust (POST 2016)*, volume 9635, pages 24–45. Springer, 2016.

11. W. De Groef, D. Devriese, N. Nikiforakis, and F. Piessens. Flowfox: a Web Browser with Flexible and Precise Information Flow Control. In *Proc. of the 19th ACM Conference on Communications and Computer Security*, pages 748–759, 2012.
12. D. Devriese and F. Piessens. Non-interference through secure multi-execution. In *Proc. of the 2010 Symposium on Security and Privacy*, pages 109–124. IEEE, 2010.
13. U. Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Cornell University, 2003.
14. J. S. Fenton. Memoryless subsystems. *Comput. J.*, 17(2):143–147, 1974.
15. D. Hedin, L. Bello, and A. Sabelfeld. Value-sensitive hybrid information flow control for a javascript-like language. In *IEEE 28th Computer Security Foundations Symposium, CSF*, 2015.
16. V. Kashyap, B. Wiedermann, and B. Hardekopf. Timing- and termination-sensitive secure information flow: Exploring a new approach. In *Security and Privacy (SP), 2011 IEEE Symposium on*, pages 413–428, 2011.
17. G. Le Guernic. *Confidentiality Enforcement Using Dynamic Information Flow Analyses*. PhD thesis, Kansas State University and University of Rennes 1, 2007.
18. J. Ligatti, L. Bauer, and D. Walker. Enforcing Non-safety Security Policies with Program Monitors. In *Proc. of the 10th European Symposium on Research in Computer Security*, volume 3679 of *LNCS*, pages 355–373. Springer-Verlag Heidelberg, 2005.
19. A. G. A. Matos, J. F. Santos, and T. Rezk. An Information Flow Monitor for a Core of DOM - Introducing References and Live Primitives. In *Trustworthy Global Computing - 9th International Symposium, TGC*, 2014.
20. W. Rafnsson and A. Sabelfeld. Secure multi-execution: Fine-grained, declassification-aware, and transparent. In *2013 IEEE 26th Computer Security Foundations Symposium*, 2013.
21. W. Rafnsson and A. Sabelfeld. Secure multi-execution: Fine-grained, declassification-aware, and transparent. *Journal of Computer Security*, 24(1):39–90, 2016.
22. A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communication*, 21(1):5–19, 2003.
23. A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 17(5):517–548, 2009.
24. J. F. Santos and T. Rezk. An Information Flow Monitor-Inlining Compiler for Securing a Core of Javascript. In *ICT Systems Security and Privacy Protection - 29th IFIP TC 11 International Conference, SEC 2014*, 2014.
25. T. Schmitz, D. Rhodes, T. H. Austin, K. Knowles, and C. Flanagan. Faceted dynamic information flow via control and data monads. In *International Conference on Principles of Security and Trust (POST 2016)*, volume 9635, pages 3–23. Springer, 2016.
26. M. Vanhoef, W. D. Groef, D. Devriese, F. Piessens, and T. Rezk. Stateful declassification policies for event-driven programs. In *IEEE 27th Computer Security Foundations Symposium, CSF 2014*, pages 293–307, 2014.
27. D. Zanarini, M. Jaskelioff, and A. Russo. Precise enforcement of confidentiality for reactive systems. In *IEEE 26th Computer Security Foundations Symposium*, pages 18–32, 2013.
28. S. A. Zdancewic. *Programming languages for information security*. PhD thesis, Cornell University, 2002.

DOM: Specification and Client Reasoning

Azalea Raad, José Fragoso Santos and Philippa Gardner

Imperial College London

Abstract. We present an axiomatic specification of a key fragment of DOM using structural separation logic. This specification allows us to develop modular reasoning about client programs that call the DOM.

1 Introduction

The behaviour of JavaScript programs executed in the browser is complex. Such programs manipulate a heap maintained by the browser and call a wide range of APIs via specific objects in this heap. The most notable of these is the Document Object Model (DOM) API and the DOM document object, which are used to represent and manipulate the web page. JavaScript programs must run uniformly across all browsers. As such, the English standards of JavaScript and DOM are rather rigorous and are followed closely by browser vendors. While there has been work on formal specifications of JavaScript [14], including mechanised specifications [4], and some work on the formal specification of DOM [9,22] and on the verification of JavaScript programs [7], we are not aware of any work on the verification of JavaScript programs that call the DOM.

The W3C DOM standard [1] describes an XML update library used by all browsers. This English standard is written in an axiomatic style that lends itself well to formalisation. The first formal axiomatic DOM specification has been given in [9,22], using context logic (CL) [6,5], which extends ideas from separation logic (SL) [19] to complex data structures. However, this work has several shortcomings. First, it is not simple to integrate SL reasoning about e.g. C [19], Java [16] and JavaScript [7] with the DOM specifications in CL. The work in [9,22] explores the verification of simple client programs manipulating a variable store and calling the DOM. It does not verify clients manipulating a standard program heap. Second, this specification does not always allow *compositional* client-side reasoning. Finally, this specification makes simplifying choices (e.g. with live collections), and does not always remain faithful to the standard.

We present a faithful axiomatic specification of a key fragment of the DOM and verify substantial client programs, using structural separation logic (SSL) introduced in [25,8]. SSL provides fine-grained reasoning about complex data structures. The SSL assertion language contains the commutative separating conjunction ($*$), as in SL, that serves to split the DOM tree into smaller subtrees. By contrast, the CL assertion language contains the non-commutative separating application ($\bullet$), that splits the DOM tree into a tree context with a hole applied to a partial DOM tree. These two operators are not compatible with each other.

In particular, the integration of the CL DOM specification with an SL-based program logic involves extending the program logic to include a frame rule for the separating application. By contrast, the integration of our SSL DOM specification with an SL-based program logic requires no extensions. We can reason about DOM client programs written in e.g. C, Java and JavaScript, by simply using a combination of the appropriate SL-based program logic for reasoning about the particular programming language and our DOM axioms. We illustrate this by verifying several realistic ad-blocker client programs written in JavaScript, using the program logic of [7]. Our reasoning abstracts the complexities of JavaScript, simply using standard SL assertions, an abstract variable store predicate, and JavaScript heap assertions. It is thus straightforward to transfer our ideas to other languages, as we show in §3.

As the authors noted in [9,22], CL does not always allow for *local* reasoning. As we demonstrate in§2, it also does not provide *compositional* reasoning. In contrast, SSL provides both local and compositional client reasoning. We demonstrate this by presenting a simple client program which can be specified using a *single* SSL triple whose precondition captures its intuitive footprint, compared to *six* CL triples, whose preconditions are substantially larger than the footprint.

The DOM English standard [1] is written in an axiomatic style, allowing for a straightforward comparison of our formal axiomatic specification with the standard. A typical way to justify an axiomatic specification of a library is to compare it against an operational semantics, as in [9,22,25] for DOM. However, this approach seems unsuitable as it involves inventing an operational semantics for the library, even though the standard is written in an axiomatic style. Instead, we justify our specification with respect to a reference implementation that can be independently tested. In [17] we present a JavaScript implementation of our DOM fragment, and prove its correctness with respect to our specification.

Related work There has been much work on simple models of semi-structured data, following the spirit of DOM, such as [6,2,3] (axiomatic, program logic) and [20] (operational, information flow). We do not detail this work here. Instead, we concentrate on axiomatic and operational models, with a primary focus on DOM. Smith et al. developed an axiomatic specification of the DOM [9,22] in CL [6,5], as discussed above. Others have also studied operational models of DOM. Lerner et al. were the first to formalise the DOM event model [13]. This model is executable and can be used as an oracle for testing browser compliance with the standard. Unlike our work, this model was not designed for proving functional properties of client programs, but rather meta-properties of the DOM itself. The main focus of this work is the event dispatch model in DOM. Rajani et al. [18] have developed an operational model for DOM events and live collections, in order to study information flow. We aim to study DOM events in the future.

There has been much work on type analysis for client programs calling the DOM. Thiemann [24] developed a type system for establishing safety properties of DOM client programs written in Java. He extended the Java type system of [10] with recursion and polymorphism, and used this extension to specify the DOM data structures and operations. Later, Jensen et al. added DOM types

to JavaScript [12,21,11], developing a flow sensitive type analysis tool TAJS. They used DOM types to reason about control and data flow in JavaScript applications that interact with the DOM. Recently, Park et al. developed a framework for statically analysing JavaScript web applications that interact with the DOM [15]. As with TAJS, this framework uses configurable DOM abstraction models. However, the proposed models are significantly more fine-grained than those of TAJS in that they can precisely describe the structure of DOM trees whereas TAJS simply treats them as black boxes. In [23], Swamy et al. translate JavaScript to a typed language and type the DOM operations. The DOM types are intentionally restrictive to simplify client analysis (e.g. modelling live collections as iterators in [23]). In contrast, there has been little work on the verification of programs calling the DOM. Smith et al. [9,22] look at simple client programs which manipulate the variable store and the DOM. However, their reasoning is not compositional, as previously discussed and formally justified in §2.

Outline In §2, we summarise our contributions. In §3, we present our DOM specification and describe how our specification may be integrated with an arbitrary SL-based program logic. In §4, we verify a JavaScript ad-blocker client program which calls the DOM, and we finish with concluding remarks.

2 Overview

2.1 A Formal DOM Specification

The W3C DOM standard [1] is presented in an object-oriented (OO) and language-independent fashion. It consists of a set of interfaces describing the fields and methods exposed by each DOM datatype. A DOM object is a tree comprising a collection of *node* objects. DOM defines twelve specialised node types. As our goal is to present our specification methodology, we focus on an expressive fragment of DOM Core Level 1 (CL1) that allows us to create, update, and traverse DOM documents. We thus model the four most commonly used node types: *document*, *element*, *text* and *attribute* nodes. Additionally, we model *live collections of nodes* such as the *NodeList* interface in DOM CL1-4 (discussed in §3.5). Our fragment underpins DOM Core Levels 1-4. As shown in [22], it is straightforward to extend this fragment to the full DOM CL1 without adding to the complexity of the underlying program logic. It will be necessary to extend the program logic as we consider additional features in the higher levels of the standard (e.g. DOM events). However, these features will not affect the fragment specified here. We proceed with an account of our DOM fragment, hereafter simply called DOM.

DOM nodes Each node in DOM is associated with a *type*, a *name*, an optional *value*, and information about its surroundings (e.g. its parent, siblings, etc.). Given the OO spirit of the standard, each node object is uniquely identified by its reference. To capture this more abstractly (and admit non-OO implementations), we associate each node with a unique *node identifier*. As mentioned earlier, the standard defines twelve different node types of which we model the following

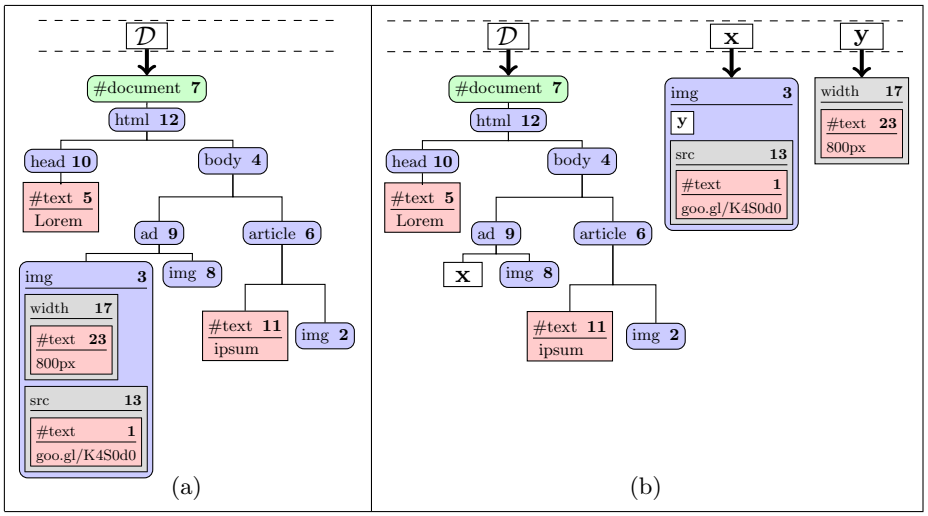


Fig. 1: A complete DOM heap (a); same DOM heap after abstract allocation (b)

four. *Document nodes* represent entire DOM objects. Each DOM object contains exactly one document node, named #document, with no value and at most a single child, referred to as the *document element*. In Fig. 1a, the document node is the node with identifier 7 (with document element 12). *Text nodes* (named #text) represent the textual content of the document. They have no children and may have arbitrary strings as their values. In Fig. 1a, node 5 is a text node with string data "Lorem". *Element nodes* structure the contents of a DOM object. They have arbitrary names (not containing the '#' character), no values and an arbitrary number of text and element nodes as their children. In Fig. 1a, node 12 is an element node with name "html" and two children with identifiers 10 and 4. *Attribute nodes* store information about the element nodes to which they are attached. The attributes of an element must have unique names. Attribute nodes may have arbitrary names (not containing the '#' character) and an arbitrary number of text nodes as their children. The value of an attribute node is given by concatenating the values of its children. In Fig. 1a, the element node with identifier 3 has two attributes: one with name "width", identifier 17, and value "800px" (i.e. the value of text node 23); and another with name "src", identifier 13, and value "goo.gl/K4S0d0" (i.e. the value of text node 1).

DOM operations The complete set of DOM operations and their axioms are given in [17]. In §3, we present the axioms for the operations used in the examples of this paper. Here, we describe the `n.getAttribute(s)` and `n.setAttribute(s,v)` operations and their axioms to give an intuitive account of SSL. The `n.getAttribute(s)` operation inspects the attributes of element node `n`. It returns the value of the attribute named `s` if it exists, or the empty string otherwise. For instance, given the DOM tree of Fig. 1a, when variable `n`

holds value 3 (the element node named "img", placed as the left child of node "ad"), and `s` holds "src", then `r = n.getAttribute(s)` yields `r`="goo.gl/K4S0d0".

Intuitively, the footprint of `n.getAttribute(s)` is limited to the element node `n` and its "src" attribute. To describe this footprint minimally, we need to split the element node at `n` away from the larger surrounding DOM tree. To do this, we introduce *abstract DOM heaps* that store abstract tree fragments. For instance, Fig. 1a contains an abstract DOM heap with one cell at address $\mathcal{D}$ and a complete abstract DOM tree as its value. It is abstract in that it hides the details of how a DOM tree might be concretely represented in a machine. Abstract heaps allow for their data to be split by imposing additional instrumentation using *abstract addresses*. Such splitting is illustrated by the transition from Fig. 1a to Fig. 1b. The heap in Fig. 1a contains a complete tree at address $\mathcal{D}$. This tree can be split using *abstract allocation* to obtain the heap in Fig. 1b with the subtree at node 3 at a fresh, fictional *abstract cell* $\mathbf{x}$, and an incomplete tree at $\mathcal{D}$ with a *context hole* $\mathbf{x}$ indicating the position to which the subtree will return. Since we are only interested in the attribute named "src", we can use abstract allocation again to split away the other unwanted attribute ("width") and place it at a fresh abstract cell $\mathbf{y}$ as illustrated in Fig. 1b. The subtree at node 3 and its "src" attribute correspond to the intuitive footprint of `n.getAttribute(s)`. Once the `getAttribute` operation is complete, we can join the tree back together through *abstract deallocation*, as in the transition from Fig. 1b to 1a.

Using SSL [25], we develop *local* specifications of DOM operations that only touch the intuitive footprints of the operations. The assertion language comprises *DOM assertions* that describe abstract DOM heaps. For example, the DOM assertion $\alpha \mapsto \mathrm{img}_3[\beta \odot \mathrm{src}_{13}[\#\mathrm{text}_1[\text{goo.gl/K4S0d0}]], \varnothing]$ describes the abstract heap cell at $\mathbf{x}$ in Fig. 1b, where α and β denote logical variables corresponding to abstract addresses $\mathbf{x}$ and $\mathbf{y}$, respectively. It states that the heap cell at abstract logical address α holds an "img" element with identifier 3, no children ($\varnothing$) and a set of attributes described by $\beta \odot \mathrm{src}_{13}[\#\mathrm{text}_1[\text{goo.gl/K4S0d0}]]$, which contains a "src" attribute (with identifier 13 and value "goo.gl/K4S0d0") and other attributes to be found at abstract logical address β. The attributes of a node are grouped by the commutative $\odot$ operator. When we are only interested in the value of an attribute, we can write an assertion that is agnostic to the shape of the text content under the attribute. For instance, we can write $\alpha \mapsto \mathrm{img}_3[\beta \odot \mathrm{src}_{13}[\mathrm{T}], \varnothing] * \mathsf{val}(\mathrm{T}, \text{goo.gl/K4S0d0})$ to state that attribute 13 contains some text content described by logical variable T, and that the value of T (i.e. the value of the attribute) is "goo.gl/K4S0d0". Assertion $\mathsf{val}(\mathrm{T}, \text{goo.gl/K4S0d0})$ is *pure* in that it contains no resources and merely describes the string value of T.

Using SSL triples, we can now locally specify `r = n.getAttribute(s)` as[1]:

$$\left\{\begin{array}{l}\mathsf{store}(\mathtt{n} : \mathrm{N}, \mathtt{s} : \mathrm{S}, \mathtt{r} : -) \\ * \, \alpha \mapsto \mathrm{S}'_{\mathrm{N}}[\beta \odot \mathrm{S}_{\mathrm{M}}[\mathrm{T}], \gamma] \\ * \, \mathsf{val}(\mathrm{T}, \mathrm{S}'')\end{array}\right\} \ \mathtt{r = n.getAttribute(s)} \ \left\{\begin{array}{l}\mathsf{store}(\mathtt{n} : \mathrm{N}, \mathtt{s} : \mathrm{S}, \mathtt{r} : \mathrm{S}'') \\ * \, \alpha \mapsto \mathrm{S}'_{\mathrm{N}}[\beta \odot \mathrm{S}_{\mathrm{M}}[\mathrm{T}], \gamma] \\ * \, \mathsf{val}(\mathrm{T}, \mathrm{S}'')\end{array}\right\} \quad (1)$$

[1] It is possible to combine multiple cases into one by rewriting the pre- and postconditions as a disjunction of the cases and using logical variables to track each case. For clarity, we opt to write each case separately.

$$\left\{\begin{array}{l}\mathsf{store}(\mathtt{n}:\mathrm{N},\mathtt{s}:\mathrm{S},\mathtt{r}:-)\\ *\ \alpha\mapsto \mathrm{S}'_{\mathrm{N}}[\mathrm{A},\gamma]*\mathsf{out}(\mathrm{A},\mathrm{S})\end{array}\right\}\ \mathtt{r=n.getAttribute(s)}\ \left\{\begin{array}{l}\mathsf{store}(\mathtt{n}:\mathrm{N},\mathtt{s}:\mathrm{S},\mathtt{r}:\text{“”})\\ *\ \alpha\mapsto \mathrm{S}'_{\mathrm{N}}[\mathrm{A},\gamma]*\mathsf{out}(\mathrm{A},\mathrm{S})\end{array}\right\}\quad(2)$$

SSL triples have a fault-avoiding, partial-correctness interpretation as in other separation logics: if an abstract DOM heap satisfies the precondition then either the operation does not terminate, or the operation terminates and the resulting state will satisfy the postcondition. Axiom (1) captures the case when `n` contains an attribute named `s`; axiom (2) when `n` has no such attribute. The precondition of (1) contains three assertions. Assertion $\mathsf{store}(\mathtt{n}:\mathrm{N},\mathtt{s}:\mathrm{S},\mathtt{r}:-)$ describes a variable store where program variables `n`, `s` and `r` have logical values N, S and an arbitrary value $(-)$, respectively.[2] Assertion $\alpha\mapsto \mathrm{S}'_{\mathrm{N}}[\beta\odot \mathrm{S}_{\mathrm{M}}[\mathrm{T}],\gamma]$ describes an abstract DOM heap cell at the logical abstract address α containing the subtree described by assertion $\mathrm{S}'_{\mathrm{N}}[\beta\odot \mathrm{S}_{\mathrm{M}}[\mathrm{T}],\gamma]$. This assertion describes a subtree with a single element node with identifier N and name S′. Its children have been framed off, leaving behind the context hole γ (using abstract allocation as in the transition from Fig. 1a to 1b, then framing off the cell at γ). It has an attribute named S with identifier M and text content T, plus (potentially) other attributes that have been framed off, leaving behind the context hole β. This framing off of the children and attributes other than `s` captures the intuition that the footprint of `n.getAttribute(s)` is limited to element `n` and attribute `s`. Lastly, assertion $\mathsf{val}(\mathrm{T},\mathrm{S}'')$ states that the value of text content T is S″. The postcondition of (1) declares that the subtree remains the same and that the value of `r` in the variable store is updated to S″, i.e. the value of the attribute named S.

The precondition of (2) contains the assertion $\alpha\mapsto \mathrm{S}'_{\mathrm{N}}[\mathrm{A},\gamma]$ where, this time, the attributes of the element node identified by N are described by the logical variable A. With the precondition of (1), all other attributes can be framed off leaving context hole β. With the precondition of (2) however, the attributes are part of the intuitive footprint since we must check the absence of an attribute named S. This is captured by the $\mathsf{out}(\mathrm{A},\mathrm{S})$ assertion. The postcondition of (2) declares that the subtree remains the same and the value of `r` in the variable store is updated to the empty string “”, as mandated by the English specification.

The `n.setAttribute(s,v)` operation inspects the attributes of element node `n`. It then sets the value of the attribute named `s` to `v` if such an attribute exists (3). Otherwise, it creates a new attribute named `s` with value `v` and attaches it to node `n` (4). We can specify this English description as[1]:

$$\left\{\begin{array}{l}\mathsf{store}(\mathtt{n}:\mathrm{N},\mathtt{s}:\mathrm{S},\mathtt{v}:\mathrm{S}'')\\ *\ \alpha\mapsto \mathrm{S}'_{\mathrm{N}}[\beta\odot \mathrm{S}_{\mathrm{M}}[\mathrm{T}],\gamma]\\ *\ \delta\mapsto \varnothing_g\end{array}\right\}\ \mathtt{n.setAttribute(s,v)}\ \left\{\begin{array}{l}\exists \mathrm{R}.\ \mathsf{store}(\mathtt{n}:\mathrm{N},\mathtt{s}:\mathrm{S},\mathtt{v}:\mathrm{S}'')\\ *\ \alpha\mapsto \mathrm{S}'_{\mathrm{N}}[\beta\odot \mathrm{S}_{\mathrm{M}}[\#\mathrm{text}_{\mathrm{R}}[\mathrm{S}'']],\gamma]\\ *\ \delta\mapsto \mathrm{T}\end{array}\right\}\quad(3)$$

$$\left\{\begin{array}{l}\mathsf{store}(\mathtt{n}:\mathrm{N},\mathtt{s}:\mathrm{S},\mathtt{v}:\mathrm{S}'')\\ *\ \alpha\mapsto \mathrm{S}'_{\mathrm{N}}[\mathrm{A},\gamma]*\mathsf{out}(\mathrm{A},\mathrm{S})\end{array}\right\}\ \mathtt{n.setAttribute(s,v)}\ \left\{\begin{array}{l}\exists \mathrm{M},\mathrm{R}.\ \mathsf{store}(\mathtt{n}:\mathrm{N},\mathtt{s}:\mathrm{S},\mathtt{v}:\mathrm{S}'')\\ *\ \alpha\mapsto \mathrm{S}'_{\mathrm{N}}[\mathrm{A}\odot \mathrm{S}_{\mathrm{M}}[\#\mathrm{text}_{\mathrm{R}}[\mathrm{S}'']],\gamma]\end{array}\right\}\quad(4)$$

Recall that attribute nodes may have an arbitrary number of text nodes as their children where the concatenated values of the text nodes denotes the value of the

[2] Since DOM may be called by different client programs written in different languages, store denotes a *black-box* predicate that can be instantiated to describe a variable store in the client language. Here, we instantiate it as the JavaScript variable store.

attribute. As such, when `n` contains an attribute named `s`, its value is set to `v` by removing the existing children (text nodes) of `s`, creating a new text node with value `v` and attaching it to `s` (axiom 3). What is then to happen to the removed children of `s`? In DOM, nodes are not disposed of: whenever a node is removed, it is no longer a part of the DOM tree but still exists in memory. To model this, we associate the document object with a *grove* designating a space for the removed nodes. The $\delta \mapsto \varnothing_g$ assertion in the precondition of (3) simply reserves an empty spot ($\varnothing_g$) in the grove. In the postcondition the removed children of `s` (i.e. T) are moved to the grove. Similarly, when `n` does not contain an attribute named `s`, a new attribute named `s` is created and attached to `n`. The value of `s` is set to `v` by creating a new text node with value `v` and attaching it to `s` (axiom 4).

Comparison with existing work [9,22] In contrast to the *commutative* separating conjunction $*$ in SSL, context logic (CL) and multi-holed context logic (MCL) use a *non-commutative separating application* $\bullet$ to split the DOM tree structure. For instance, the $C \bullet_\alpha P$ formula describes a tree that can be split into a context tree C with hole α and a subtree P to be applied to the context hole. The application is not commutative; it does not make sense to apply a context to a tree. In [9,22], the authors noted that the `appendChild` axiom was not *local*, as it required more than the intuitive footprint of the operation. What they did not observe was that CL client reasoning is not *compositional*. Consider a program $\mathbb{C}$ that copies the value of the "src" attribute in element `p` to that of `q`:

$$\mathbb{C} \triangleq \texttt{s=p.getAttribute("src"); q.setAttribute("src",s)}$$

Let us assume that `p` contains a "src" attribute while `q` does not. Using SSL, we can specify $\mathbb{C}$ as follows, where $S \triangleq \mathsf{store}(\texttt{p}{:}\mathrm{P},\texttt{q}{:}\mathrm{Q},\texttt{s}{:}{-}) * \mathsf{val}(\mathrm{T},\mathrm{S}_1) * \mathsf{out}(\mathrm{A},\mathrm{S})$, $P \triangleq \mathrm{S_P}[\gamma_1 \odot \mathrm{src_N}[\mathrm{T}],\mathrm{F}_1]$, $Q \triangleq \mathrm{S'_Q}[\mathrm{A},\mathrm{F}_2]$ and $Q' \triangleq \mathrm{S'_Q}[\mathrm{A} \odot \mathrm{S_M}[\#\mathrm{text_R}[\mathrm{S}_1]],\mathrm{F}_2]$:

$$\{S * \alpha \mapsto P * \beta \mapsto Q\}\ \mathbb{C}\ \{\exists \mathrm{M},\mathrm{R}.\ S * \alpha \mapsto P * \beta \mapsto Q'\} \tag{5}$$

Observe that the P and Q elements may be in one of three orientations with respect to one another: i) P and Q are not related and describe disjoint subtrees; ii) Q is an ancestor of P; and iii) P is an ancestor of Q. All three orientations are captured by (5). In contrast, using MCL (adapted to our notation) $\mathbb{C}$ is specified as follows where i-iii correspond to the three orientations above.

i) $\{S * ((C \bullet_\alpha P) \bullet_\beta Q)\}\ \mathbb{C}\ \{\exists \mathrm{M},\mathrm{R}.\ S * ((C \bullet_\alpha P) \bullet_\beta Q')\}$
ii) $\{S * (Q \bullet_\alpha P)\}\ \mathbb{C}\ \{\exists \mathrm{M},\mathrm{R}.\ S * (Q' \bullet_\alpha P)\}$ iii) $\{S * (P \bullet_\alpha Q)\}\ \mathbb{C}\ \{\exists \mathrm{M},\mathrm{R}.\ S * (P \bullet_\alpha Q')\}$

When P and Q are not related, the precondition of (i) states that the DOM tree can be split into a subtree with top node Q, and a tree context with hole variable β satisfying the $C \bullet_\alpha P$ formula. This context itself can be split into a subcontext with top node P and a context C with hole α. The postcondition of (i) states that Q is extended with a "src" attribute, and the context $C \bullet_\alpha P$ remains unchanged. This specification is not *local* in that it is larger than the intuitive footprint of $\mathbb{C}$. The only parts of the tree required by $\mathbb{C}$ are the two elements P and Q. However, the precondition in (i) also requires the surrounding *linking* context C: to assert

that P and Q are not related (P is not an ancestor of Q and vice versa), we must appeal to a linking context C that is an ancestor of both P and Q. This results in a significant overapproximation of the footprint. As either C or P, but not both, may contain context hole β, (i) includes the behaviour of (iii), which can thus be omitted. We have included it as it is more local.

More significantly however, due to the non-commutativity of $\bullet$ we need to specify (ii) and (iii) separately. Therefore, the number of CL axioms of a client program may grow rapidly as its footprint grows. Consider the program $\mathbb{C}'$ below:

$$\mathbb{C}' \triangleq \texttt{s=p.getAttribute("src"); s'=r.getAttribute("src");}\\ \texttt{q.setAttribute("src",s+s')}$$

with its larger footprint given by the distinct `p`, `q`, `r`. When `p` and `q` contain a "src" attribute and `r` does not, we can specify $\mathbb{C}'$ in SSL with *one* axiom similar to (5). By contrast, when specifying $\mathbb{C}'$ in MCL, not only is locality compromised in cases analogous to (i) above, but we need *eight* separate specifications. Forgoing locality, as described above, we still require *six* specifications. This example demonstrates that CL reasoning is not compositional for client programs.

2.2 Verifying JavaScript Programs that Call the DOM

We demonstrate how to use our DOM specification to reason about client programs that call the DOM. Our DOM specification is agnostic to the choice of client programming language. In contrast to previous work [9,22], our DOM specification integrates simply with any SL-based program logic such as those for Java [16] and JavaScript [7]. Here, we choose to reason about JavaScript client programs.

We study a JavaScript *image sanitiser* that sanitises the "src" attribute of an element node by replacing its value with a trusted URL if the value is blacklisted. To determine whether or not a value is blacklisted, a remote database is queried. The results of successful lookups are stored in a local cache to minimise the number of queries. In §4, we use this sanitiser to implement an *ad blocker* that filters untrusted contents of a web page. The code of this sanitiser, `sanitiseImg`, is given in Fig. 2. It inspects the `img` element node for its "src" attribute (line 2). When such an attribute exists (line 3), it consults the local cache (`cache`) to check whether its value (`url`) is blacklisted (line 4). If so, it changes its value to the trusted `cat` value. If the cache lookup is unsuccessful (line 6), the database is queried by the `isBlackListed` call (line 7). If the value is deemed blacklisted (line 8), the value of "src" is set to the trusted `cat` value (line 9), and the local cache is updated to store the lookup result (line 10). Observe that `sanitiseImg` does not use JavaScript-specific constructs (e.g. eval) and simply appeals to the standard language constructs of a while language. As such, it is straightforward to transform this proof to verify `sanitiseImg` written in e.g. C and Java.

The behaviour of `sanitiseImg` is specified in Fig. 2. The specifications in (6)-(9) capture different cases of the code as follows: in (6) `img` has no "src" attribute (i.e. the conditional of line 3 fails); in (7) the value of "src" is blacklisted in the local cache (line 5); in (8) the value is blacklisted and the cache has no

$\mathsf{st} \triangleq \mathsf{store}(\mathtt{img}{:}\mathrm{N},\mathtt{cat}{:}\mathrm{S}_2,\mathtt{cache}{:}\mathrm{C},\mathtt{url}{:}-,\mathtt{isB}{:}-)$ $\quad P_{\mathsf{out}} \triangleq \alpha \mapsto \mathrm{S}_\mathrm{N}[\mathrm{A},\gamma] * \mathsf{out}(\mathrm{A},\text{“src”})$

$P \triangleq \alpha \mapsto \mathrm{S}_\mathrm{N}[\beta \odot \mathrm{src}_\mathrm{M}[\mathrm{T}],\gamma] * \mathsf{val}(\mathrm{T},\mathrm{S}_1) * \delta \mapsto \varnothing_g$ $\quad Q \triangleq \exists \mathrm{R}.\, \alpha \mapsto \mathrm{S}_\mathrm{N}[\beta \odot \mathrm{src}_\mathrm{M}[\#\mathrm{text}_\mathrm{R}[\mathrm{S}_2]],\gamma] * \delta \mapsto \mathrm{T}$

$\{\mathsf{st} * P_{\mathsf{out}}\}$ `sanitiseImg(img,cat)` $\{\mathsf{st} * P_{\mathsf{out}}\}$ (6)

$\{\mathsf{st} * P * (\mathrm{C},\mathrm{S}_1) \mapsto 1 * \mathsf{isB}(\mathrm{S}_1)\}$ `sanitiseImg(img,cat)` $\{\mathsf{st} * Q * (\mathrm{C},\mathrm{S}_1) \mapsto 1 * \mathsf{isB}(\mathrm{S}_1)\}$ (7)

$\{\mathsf{st} * P * (\mathrm{C},\mathrm{S}_1) \mapsto 0 * \mathsf{isB}(\mathrm{S}_1)\}$ `sanitiseImg(img,cat)` $\{\mathsf{st} * Q * (\mathrm{C},\mathrm{S}_1) \mapsto 1 * \mathsf{isB}(\mathrm{S}_1)\}$ (8)

$\{\mathsf{st} * P * (\mathrm{C},\mathrm{S}_1) \mapsto 0 * \neg\mathsf{isB}(\mathrm{S}_1)\}$ `sanitiseImg(img,cat)` $\{\mathsf{st} * P * (\mathrm{C},\mathrm{S}_1) \mapsto 0 * \neg\mathsf{isB}(\mathrm{S}_1)\}$ (9)

$\{\mathsf{store}(\mathtt{url}{:}\mathrm{S}_1,\mathtt{isB}{:}-) * \mathsf{isB}(\mathrm{S}_1)\}$ `isB=isBlackListed(url)` $\{\mathsf{store}(\mathtt{url}{:}\mathrm{S}_1,\mathtt{isB}{:}1) * \mathsf{isB}(\mathrm{S}_1)\}$

$\{\mathsf{store}(\mathtt{url}{:}\mathrm{S}_1,\mathtt{isB}{:}-) * \neg\mathsf{isB}(\mathrm{S}_1)\}$ `isB=isBlackListed(url)` $\{\mathsf{store}(\mathtt{url}{:}\mathrm{S}_1,\mathtt{isB}{:}0) * \neg\mathsf{isB}(\mathrm{S}_1)\}$

$\{\mathsf{store}(\mathtt{img}{:}\mathrm{N},\mathtt{cat}{:}\mathrm{S}_2,\mathtt{cache}{:}\mathrm{C},\mathtt{url}{:}-,\mathtt{isB}{:}-) * P * (\mathrm{C},\mathrm{S}_1) \mapsto 0 * \mathsf{isB}(\mathrm{S}_1)\}$

```
1.  sanitiseImg(img,cat) ≜ {
2.    url=img.getAttribute("src");
```
$\{\mathsf{store}(\mathtt{img}{:}\mathrm{N},\mathtt{cat}{:}\mathrm{S}_2,\mathtt{cache}{:}\mathrm{C},\mathtt{url}{:}\mathrm{S}_1,\mathtt{isB}{:}-) * P * (\mathrm{C},\mathrm{S}_1) \mapsto 0 * \mathsf{isB}(\mathrm{S}_1)\}$
```
3.    if (url){ // img has an attribute named “src”
4.      isB=cache.url;
```
$\{\mathsf{store}(\mathtt{img}{:}\mathrm{N},\mathtt{cat}{:}\mathrm{S}_2,\mathtt{cache}{:}\mathrm{C},\mathtt{url}{:}\mathrm{S}_1,\mathtt{isB}{:}0) * P * (\mathrm{C},\mathrm{S}_1) \mapsto 0 * \mathsf{isB}(\mathrm{S}_1)\}$
```
5.      if (isB){img.setAttribute("src",cat)}  // url is in cache (thus blacklisted)
6.      else {  // url is not in cache
```
$\{\mathsf{store}(\mathtt{img}{:}\mathrm{N},\mathtt{cat}{:}\mathrm{S}_2,\mathtt{cache}{:}\mathrm{C},\mathtt{url}{:}\mathrm{S}_1,\mathtt{isB}{:}0) * P * (\mathrm{C},\mathrm{S}_1) \mapsto 0 * \mathsf{isB}(\mathrm{S}_1)\}$
```
7.        isB=isBlackListed(url);
```
$\{\mathsf{store}(\mathtt{img}{:}\mathrm{N},\mathtt{cat}{:}\mathrm{S}_2,\mathtt{cache}{:}\mathrm{C},\mathtt{url}{:}\mathrm{S}_1,\mathtt{isB}{:}1) * P * (\mathrm{C},\mathrm{S}_1) \mapsto 0 * \mathsf{isB}(\mathrm{S}_1)\}$
```
8.        if (isB){  // url is blacklisted
```
$\{\mathsf{store}(\mathtt{img}{:}\mathrm{N},\mathtt{cat}{:}\mathrm{S}_2,\mathtt{cache}{:}\mathrm{C},\mathtt{url}{:}\mathrm{S}_1,\mathtt{isB}{:}1) * P * (\mathrm{C},\mathrm{S}_1) \mapsto 0 * \mathsf{isB}(\mathrm{S}_1)\}$
```
9.          img.setAttribute("src",cat);
```
$\{\mathsf{store}(\mathtt{img}{:}\mathrm{N},\mathtt{cat}{:}\mathrm{S}_2,\mathtt{cache}{:}\mathrm{C},\mathtt{url}{:}\mathrm{S}_1,\mathtt{isB}{:}1) * Q * (\mathrm{C},\mathrm{S}_1) \mapsto 0 * \mathsf{isB}(\mathrm{S}_1)\}$
```
10.         cache.url = 1
```
$\{\mathsf{store}(\mathtt{img}{:}\mathrm{N},\mathtt{cat}{:}\mathrm{S}_2,\mathtt{cache}{:}\mathrm{C},\mathtt{url}{:}\mathrm{S}_1,\mathtt{isB}{:}1) * Q * (\mathrm{C},\mathrm{S}_1) \mapsto 1 * \mathsf{isB}(\mathrm{S}_1)\}$
```
11. } } } }
```
$\{\mathsf{store}(\mathtt{img}{:}\mathrm{N},\mathtt{cat}{:}\mathrm{S}_2,\mathtt{cache}{:}\mathrm{C},\mathtt{url}{:}-,\mathtt{isB}{:}-) * Q * (\mathrm{C},\mathrm{S}_1) \mapsto 1 * \mathsf{isB}(\mathrm{S}_1)\}$

Fig. 2: The specifications of `sanitiseImg` (above); a proof sketch of (8) (below)

record of it (lines 9-10); and in (9) the value is not blacklisted and the cache has no record of it (i.e. the conditional of 8 fails). We focus on (8) here; the remaining ones are analogous. The precondition of (8) consists of four assertions: the st captures the values of program variables; the P describes an element with an attribute named “src” and value S_1; the $(\mathrm{C},\mathrm{S}_1) \mapsto 0$ asserts that the S_1 field of cache C holds value 0 (i.e. value S_1 may or may not be blacklisted but the cache has no record of it); and $\mathsf{isB}(\mathrm{S}_1)$ states that S_1 is blacklisted. This last assertion is used in the `isBlackListed` call of line 7 with its behaviour as specified in Fig. 2. A proof sketch of specification (8) is given in Fig. 2. At each proof point, we have highlighted the effect of the preceding command, where applicable.

3 A Formal DOM Specification

We give our formal axiomatic specification of DOM, comprising the DOM model in §3.1 eliding some details about DOM live collections until §3.5, the DOM

assertions in §3.2, the framework for reasoning about DOM client programs in §3.3, the DOM axioms in §3.4, and DOM live collections in §3.5.

3.1 DOM Model

We model *DOM heaps* (e.g. Fig. 1) as mappings from addresses to DOM data. To this end, we assume a countably infinite set of *identifiers*, $n \in \text{ID}$, a designated *document identifier* associated with the document object, $d \in \text{ID}$, a countably infinite set of *abstract addresses*, $\mathbf{x} \in \text{AADD}$, and a designated *document address* $\mathcal{D}$, where the sets ID, AADD and $\{\mathcal{D}\}$ are pairwise disjoint.

DOM data DOM nodes are the building blocks of DOM data. Formally, we write: i) $\#\text{text}_n[s]_{fs}$ for the text node with identifier n and text data s; ii) $s_n[\mathbf{a}, \mathbf{f}]^{ts}_{fs}$ for the element node with identifier n, tag name s, attribute set $\mathbf{a}$, and children $\mathbf{f}$; iii) $s_n[\mathbf{tf}]_{ts}$ for the attribute node with identifier n, name s, and children $\mathbf{tf}$; and iv) $\#\text{doc}_d[\mathbf{e}]^{ts}_{fs}$ & $\mathbf{g}$ for the document object with the designated identifier d, document element $\mathbf{e}$ (or $\varnothing_e$ for no document element) and *grove* $\mathbf{g}$, ignoring the *fs* and *ts* for now. DOM nodes can be grouped into attribute sets, forests, groves, and text forests, respectively ranged over by $\mathbf{a}$, $\mathbf{f}$, $\mathbf{g}$ and $\mathbf{tf}$. An attribute set represents the attribute nodes associated with an element node and is modelled as an *unordered*, possibly empty collection of attribute nodes. A forest represents the children of an element node, modelled as an *ordered*, possibly empty collection of element and text nodes. A grove is where the orphaned nodes are stored. In DOM, nodes are never disposed of and whenever a node is removed from the document, it is moved to the grove. The grove is also where newly created nodes are placed. The document object is thus associated with a grove, modelled as an *unordered*, possibly empty collection of text, element and attribute nodes. A text forest represents the children of an attribute node, modelled as an *ordered*, possibly empty collection of text nodes. We associate each node with a set of *forest listeners*, *fs*; we further associate element and document nodes with a set of *tag listeners*, *ts*. We delay the motivation for these listeners until §3.5 when we model live collections. DOM data may be either incomplete with context holes (e.g. $\mathbf{x}$), or complete with no context holes. Notationally, data written in **bold** may contain context holes; regular font indicates the absence of context holes.

Definition 1. *The sets of* strings $s \in \mathbb{S}$, texts $t \in \mathbb{T}$, elements $\mathbf{e} \in \mathbb{E}$, documents $\mathbf{doc} \in \mathbb{D}$, attribute sets $\mathbf{a} \in \mathbb{A}$, forests $\mathbf{f} \in \mathbb{F}$, groves $\mathbf{g} \in \mathbb{G}$, *and* text forests $\mathbf{tf} \in \mathbb{TF}$, *are defined below where* $\mathbf{x} \in \text{AADD}$, $n \in \text{ID}$, $fs \in \mathcal{P}(\text{ID})$ *and* $ts \in \mathcal{P}(\mathbb{S} \times \text{ID})$*:*

$$s ::= \varnothing_s \mid c \mid s_1.s_2 \quad t ::= \#\text{text}_n[s]_{fs} \quad \mathbf{e} ::= s_n[\mathbf{a}, \mathbf{f}]^{ts}_{fs} \quad \mathbf{a} ::= \varnothing_a \mid \mathbf{x} \mid s_n[\mathbf{tf}]_{fs} \mid \mathbf{a}_1 \odot \mathbf{a}_2$$

$$\mathbf{doc} ::= \#\text{doc}_d[\varnothing_e]^{ts}_{fs} \,\&\, \mathbf{g} \mid \#\text{doc}_d[\mathbf{e}]^{ts}_{fs} \,\&\, \mathbf{g} \mid \#\text{doc}_d[\mathbf{x}]^{ts}_{fs} \,\&\, \mathbf{g}$$

$$\mathbf{f} ::= \varnothing_f \mid \mathbf{x} \mid t \mid \mathbf{e} \mid \mathbf{f}_1 \otimes \mathbf{f}_2 \quad \mathbf{g} ::= \varnothing_g \mid \mathbf{x} \mid t \mid \mathbf{e} \mid s_n[\mathbf{tf}]_{fs} \mid \mathbf{g}_1 \oplus \mathbf{g}_2 \quad \mathbf{tf} ::= \varnothing_{tf} \mid \mathbf{x} \mid t \mid \mathbf{tf}_1 \oslash \mathbf{tf}_2$$

where the operations $.$, $\oslash$, $\otimes$, $\odot$ *and* $\oplus$ *are associative with identities* $\varnothing_s$, $\varnothing_{tf}$, $\varnothing_f$, $\varnothing_a$ *and* $\varnothing_g$, *respectively; the* $\odot$ *and* $\oplus$ *operations are commutative; and all data are equal up to the properties of* $.$, $\oslash$, $\otimes$, $\odot$ *and* $\oplus$. *Data does not contain*

repeated identifiers and abstract addresses; element nodes contain attributes with distinct names. The set of DOM data *is* $\mathbf{d} \in \textsc{Data} \triangleq \mathbb{E} \cup \mathbb{F} \cup \mathbb{TF} \cup \mathbb{A} \cup \mathbb{G} \cup \mathbb{D}$.

When the type of data is clear from the context, we drop the subscripts for empty data and write e.g. $\varnothing$ for $\varnothing_f$. We drop the forest and tag listeners when not relevant to the discussion and write e.g. $s_n[\mathbf{a}, \mathbf{f}]$ for $s_n[\mathbf{a}, \mathbf{f}]^{ts}_{fs}$. Given the set of DOM data Data, there is an associated address function, $\mathsf{Adds}(.)$, which returns the set of context holes present in the data. Context application $\mathbf{d}_1 \circ_{\mathbf{x}} \mathbf{d}_2$ denotes the standard substitution of $\mathbf{d}_2$ for $\mathbf{x}$ in $\mathbf{d}_1$ ($\mathbf{d}_1[\mathbf{d}_2/\mathbf{x}]$) provided that $\mathbf{x} \in \mathsf{Adds}(\mathbf{d}_1)$ and the result is well-typed, and is otherwise undefined.

DOM heaps A DOM heap is a mapping from addresses, $x \in \textsc{Addr} \triangleq \textsc{AAdd} \uplus \{\mathcal{D}\}$, to DOM data. DOM heaps are subject to structural invariants to ensure that they are well-formed. In particular, a context hole $\mathbf{x}$ must not be reachable from the abstract address $\mathbf{x}$ in the domain of the heap. For instance, $\{\mathbf{x} \mapsto s_n[\varnothing, \mathbf{y}], \mathbf{y} \mapsto s'_m[\varnothing, \mathbf{x}]\}$ is not a DOM heap due to the cycle. We capture this by the reachability relation $\rightsquigarrow$ defined as: $x \rightsquigarrow \mathbf{y} \iff \mathbf{y} \in \mathsf{Adds}(\mathbf{h}(x))$, for heap $\mathbf{h}$ and address $x \in \textsc{Addr}$. We write $\rightsquigarrow^+$ to denote the transitive closure of $\rightsquigarrow$.

Definition 2. *The set of* DOM heaps *is:* $\mathbf{h} \in \textsc{DOMHeap} \subseteq (\{\mathcal{D}\} \rightharpoonup \mathbb{D}) \cup (\textsc{AAdd} \overset{fin}{\rightharpoonup} \textsc{Data})$ *provided that for all* $\mathbf{h} \in \textsc{DOMHeap}$ *and* $x \in \textsc{Addr}$ *the following hold:*

1. *identifiers and context holes are unique across* $\mathbf{h}$*;*
2. $\neg \exists \mathbf{x}.\ \mathbf{x} \rightsquigarrow^+ \mathbf{x}$*;*
3. *context holes in* $\mathbf{h}$ *are associated with data of correct type:*
$$\forall x, \mathbf{y}.\ \mathbf{y} \in \mathsf{Adds}(\mathbf{h}(x)) \wedge \mathbf{y} \in \mathrm{dom}(\mathbf{h}) \Rightarrow \exists \mathbf{d}.\ \mathbf{h}(x) \circ_{\mathbf{y}} \mathbf{h}(\mathbf{y}) = \mathbf{d}$$

DOM Heap composition, $\bullet : \textsc{DOMHeap} \times \textsc{DOMHeap} \rightharpoonup \textsc{DOMHeap}$*, is the standard disjoint function union provided that the resulting heap meets the constraints above. The* empty DOM heap*,* $\mathbf{0}$*, is a function with an empty domain.*

Definition 3. *The* abstract (de)allocation relation*,* $\approx : \textsc{DOMHeap} \times \textsc{DOMHeap}$*, is defined as follows where* $*$ *denotes the reflexive transitive closure of the set.*

$$\approx \triangleq \left\{ (\mathbf{h}_1, \mathbf{h}_2), (\mathbf{h}_2, \mathbf{h}_1) \,\middle|\, \exists x, \mathbf{d}_1, \mathbf{d}_2, \mathbf{x}.\ \mathbf{h}_1(x) = (\mathbf{d}_1 \circ_{\mathbf{x}} \mathbf{d}_2) \wedge \mathbf{h}_2 = \mathbf{h}_1[x \mapsto \mathbf{d}_1] \bullet [\mathbf{x} \mapsto \mathbf{d}_2] \right\}^*$$

During abstract allocation (from $\mathbf{h}_1$ to $\mathbf{h}_2$), part of the data $\mathbf{d}_2$ at address x is split and promoted to a fresh abstract address $\mathbf{x}$ in the heap leaving the context hole $\mathbf{x}$ behind in its place. Dually, during abstract deallocation (from $\mathbf{h}_2$ to $\mathbf{h}_1$) the context hole $\mathbf{x}$ in DOM data $\mathbf{d}_1$ is replaced by its associated data $\mathbf{d}_2$ at abstract address $\mathbf{x}$, removing $\mathbf{x}$ from the domain of the heap in doing so.

3.2 DOM Assertions

DOM assertions comprise heap assertions describing DOM heaps such as those in Fig. 1. DOM heap assertions are defined via DOM data assertions describing the underlying DOM structure such as nodes, forests and so forth. As we show later, pure assertions such as $\mathsf{out}(\mathrm{A}, \mathrm{S})$ in §2 are derived assertions defined in Fig. 4.

Definition 4. *The* DOM assertions, $\psi \in \text{DOMAsst}$, *and* DOM data assertions, $\phi \in \text{DOMDAsst}$, *are defined as follows where* $\alpha, \text{A}, \text{N}, \cdots$ *denote logical variables.*

$$
\begin{array}{rll}
\psi ::= \mathcal{D} \mapsto \phi \mid \alpha \mapsto \phi & & \text{DOM heap assertions} \\
\phi ::= \text{false} \mid \phi_1 \Rightarrow \phi_2 \mid \exists \text{X}.\ \phi \mid \text{V} \mid \alpha \mid \phi_1 \circ_\alpha \phi_2 \mid \Diamond \alpha & & \text{classical} \mid \text{context hole} \\
\quad \mid \#\text{text}_{\text{N}}[\phi]_{\text{F}} \mid \text{s}_{\text{N}}[\phi_1,\phi_2]^{\text{E}}_{\text{F}} \mid \text{s}_{\text{N}}[\phi]_{\text{F}} \mid \#\text{doc}_{\text{N}}[\phi_1]^{\text{E}}_{\text{F}} \,\&\, \phi_2 \mid \varnothing_e & & \text{nodes} \mid \text{empty doc. element} \\
\quad \mid \varnothing_s \mid \phi_1.\phi_2 \mid \varnothing_a \mid \phi_1 \odot \phi_2 \mid \varnothing_f \mid \phi_1 \otimes \phi_2 & & \text{strings} \mid \text{attr. sets} \mid \text{forests} \\
\quad \mid \varnothing_g \mid \phi_1 \oplus \phi_2 \mid \varnothing_{tf} \mid \phi_1 \oslash \phi_2 & & \text{groves} \mid \text{text forests}
\end{array}
$$

The $\mathcal{D} \mapsto \phi$ assertion describes a single-cell DOM heap at document address $\mathcal{D}$; similarly, the $\alpha \mapsto \phi$ describes a single-cell DOM heap at the abstract address denoted by α. For data assertions, classical assertions are standard. The V is a logical variable describing DOM data. The α is a logical variable denoting a context hole; the $\phi_1 \circ_\alpha \phi_2$ describes data that is the result of replacing the context hole α in ϕ_1 with ϕ_2; $\Diamond\alpha$ describes data that contains the context hole α. The node assertions respectively describe element, text, attribute and document nodes with their data captured by the corresponding sub-assertions. The $\varnothing_e$, $\varnothing_s$, $\varnothing_a$, $\varnothing_f$, $\varnothing_g$ and $\varnothing_{tf}$ describe an empty document element, string, attribute set, forest, grove and text forest, respectively. Similarly, $\phi_1.\phi_2$, $\phi_1 \odot \phi_2$, $\phi_1 \otimes \phi_2$, $\phi_1 \oplus \phi_2$ and $\phi_1 \oslash \phi_2$ respectively describe a string, attribute set, forest, grove and text forest that can be split into two, each satisfying the corresponding sub-assertion.

3.3 PLDOMLogic

We show how to reason about client programs that call the DOM. Our DOM specification is agnostic to the client programming language and we can reason about programs in any language with an SL-based program logic. To this end, given an arbitrary programming language, PL, with an SL-based program logic, PLLogic, we show how to extend PLLogic to PLDOMLogic, in order to enable DOM reasoning. Later in §4, we present a particular instance of PLDOMLogic for JavaScript, and use it to reason about JavaScript clients that call the DOM.

States We assume the underlying program states of PLLogic to be modelled as elements of a *PCM* (partial commutative monoid) $(\text{PLStates}, \circ, 0_{\text{PL}})$, where $\circ$ denotes state composition, and 0_{PL} denotes the unit set. To reason about the DOM operations, in PLDOMLogic we extend the states of PLLogic to incorporate DOM heaps; that is, we define a program state to be a pair, $(h, \mathbf{h})$, comprising a PL state $h \in \text{PLStates}$, and a DOM heap $\mathbf{h} \in \text{DOMHeaps}$.

Definition 5. *Given the PCM of* PL, *the set of* PLDOMLogic program states *is* $\Sigma \in \text{State} \triangleq \text{PLStates} \times \text{DOMHeap}$. State composition, $+ : \text{State} \times \text{State} \rightharpoonup \text{State}$, *is defined component-wise as* $+ \triangleq (\circ, \bullet)$ *and is not defined if composition on either component is undefined. The* unit set *is* $I \triangleq \{(h, \mathbf{0}) \mid h \in 0_{\text{PL}}\}$.

Assertions We assume the PLLogic assertions to include: i) standard classical assertions; ii) standard boolean assertions; iii) standard SL assertions; and iv) an assertion to describe the PL variable store as seen in §2 of the form $\mathsf{store}(\dots)$. In PLDOMLogic we extend the PLLogic assertions with those of DOM (Def. 4), *semantic implication* $\Rrightarrow$, and the *semantic magic wand* $\rightsquigarrow\!\!*$, described shortly.

Definition 6. *The set of* PLDOMLogic assertions, $P \in \textsc{Asst}$, *is defined as follows in the binding order* $*, \Rightarrow, \Rrightarrow, -\!\!*, \sim\!\!*$, *with* $\ominus \in \{\in, =, <, \leq, \subset, \subseteq\}$*:*

$$
\begin{array}{rll}
P, Q ::= & \text{false} \mid P \Rightarrow Q \mid \exists \mathrm{x}.\ P \mid E_1 \ominus E_2 & \text{Classical} \mid \text{Boolean assertions} \\
& \mid \text{emp} \mid P * Q \mid P -\!\!* Q & \text{SL assertions} \\
& \mid \mathsf{store}(\overline{\mathtt{x}_i : \mathrm{v}_i}) \mid \Lambda & \text{variable store} \mid \text{PLLogic-specific assertions} \\
& \mid \psi \mid P \Rrightarrow Q \mid P \sim\!\!* Q & \text{DOM} \mid \text{Structural assertions}
\end{array}
$$

Assertions are interpreted as sets of program states (Def. 5). Classical and boolean assertions are standard. The emp assertion describes an empty program state in the unit set I; the $P * Q$ describes a state that can be split into two substates satisfying P and Q. The $-\!\!*$ connective is the right adjunct of $*$, i.e. $P * (P -\!\!* Q) \Rightarrow Q$. Informally, a state that satisfies $P -\!\!* Q$ is one that is *missing* P, and when combined with P, it satisfies Q. The $\mathsf{store}(\overline{\mathtt{x}_i : \mathrm{v}_i})$ describes a variable store in PL where variables $\overline{\mathtt{x}_i}$ have values $\overline{\mathrm{v}_i}$, respectively. The Λ describes states of the form $(h, \mathbf{0})$ where h satisfies Λ. Dually, the ψ describes states of the form $(h, \mathbf{h})$ where $h \in 0_{\text{PL}}$ and $\mathbf{h}$ satisfies ψ. The $P \Rrightarrow Q$ assertion denotes *semantic implication* and integrates logical implication ($\Rightarrow$) with abstract (de)allocation on DOM heaps (Def. 3). The $\sim\!\!*$ connective is the *semantic right adjunct* of $*$: $P * (P \sim\!\!* Q) \Rrightarrow Q$. It is similar to $-\!\!*$ and incorporates the $\approx$ relation on DOM heaps. Intuitively, a state that satisfies $P \sim\!\!* Q$ is one that is missing P, such that when combined with P *and* undergone a number of (possibly zero) abstract (de)allocations, it satisfies Q. We write $E_1 \dot{\ominus} E_2$ for $E_1 \ominus E_2 \wedge \text{emp}$.

Programming language, proof rules and soundness We extend the programming language of PL with the operations of our DOM fragment (e.g. `getAttribute` in §2.1). The proof rules of PLDOMLogic are those of PLLogic with the exception of the rule of consequence: we generalise the premise to allow semantic implication ($\Rrightarrow$) between assertions rather than logical implication ($\Rightarrow$). We further extend the proof rules with the axioms of DOM operations, DOMAx, defined shortly in §3.4 below. The modified rule of consequence and the rule for DOM axioms are given below. We prove PLDOMLogic sound in [17].

$$
\frac{P \Rrightarrow P' \quad \{P'\}\ \mathtt{C}\ \{Q'\} \quad Q' \Rrightarrow Q}{\{P\}\ \mathtt{C}\ \{Q\}}\ (\text{Con}) \qquad \frac{(P, \mathtt{C}, Q) \in \text{DOMAx}}{\{P\}\ \mathtt{C}\ \{Q\}}\ (\text{Ax})
$$

3.4 DOM Operations and Axioms

We formally axiomatise the behaviour of a DOM operations associated with our fragment. In Fig. 3 we give a select number of axioms including those of the operations used in the examples of this paper. The behaviour of some of the operations is captured by several axioms; we have omitted analogous cases. A full list of DOM operations modelled and their axioms, DOMAx, are given in [17].

The assertions in the pre- and postconditions of axioms are of the form $\mathsf{store}(\cdots) * \psi$ where the store predicate states the value associated with each program variable, and ψ is a DOM assertion that describes the operation footprint. Since the DOM library may be called by different client programs written in

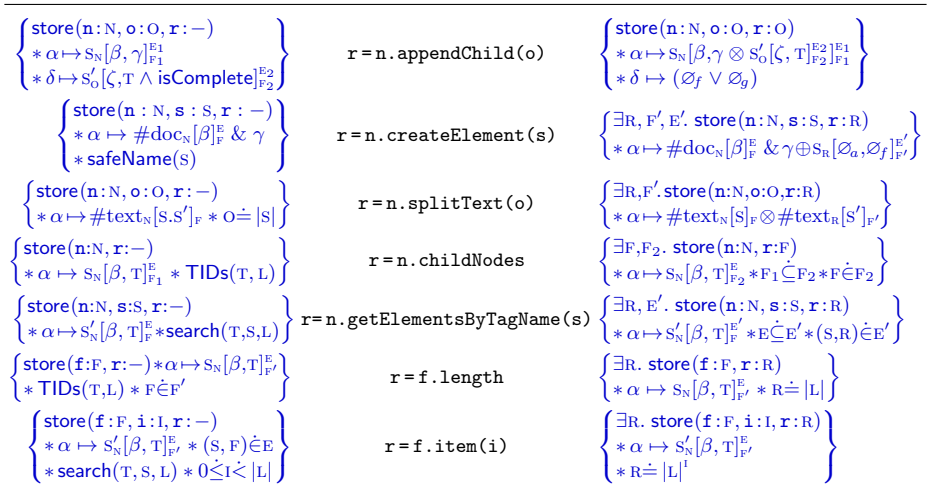


Fig. 3: DOM Core Level 1 axioms (excerpt)

different programming languages, store denotes a *black-box* predicate that can be instantiated to describe a variable store in the client programming language. In §4 we reason about JavaScript client programs that call the DOM and thus instantiate store to describe the JavaScript variable store emulated in the heap.

We now describe of the DOM operations in Fig. 3 and their axioms, delaying the description of the last four operations until §3.5.

`n.appendChild(o)`: when `n` and `o` both identify nodes, this operation appends `o` to the end of `n`'s child list and returns `o`. It fails if `o` is an ancestor of `n` (otherwise it would introduce a cycle and break the DOM structure); or if `n` is a text node or a document node with a non-empty document element; or if `o` is an attribute or a document node. Fig. 3 shows the axiom for when `o` is an element node (O). To ensure that O is not an ancestor of `n`, we require the entire subtree at `o` to be *separate* from the subtree at `n`. This is achieved by the isComplete assertion and the separating conjunction $*$. The isComplete is a derived assertion defined in Fig. 4. It describes DOM data with no context holes. The postcondition leaves $\varnothing_f \vee \varnothing_g$ in place of `o` once moved since we do not know if `o` has come from a forest or grove position. The disjunction leaves the choice to the frame.

`n.createElement(s)`: when `n` identifies a document node, it creates a new element named `s`, and returns its identifier. The new element has no attributes or children and resides in the grove. The grove in the precondition is thus extended with the new node in the postcondition. The safeName(s) assertion is defined in Fig. 4 ensures that the tag name does not contain the invalid character '#'.

`n.splitText(o)`: when `n` identifies a text node and `o` denotes an integer, it breaks the data of `n` into two text nodes at offset `o` (indexed from 0), keeping both nodes in the tree as siblings. It fails when `o` is an invalid offset (i.e. negative or greater than the length). The return value is the identifier of the new node.

$$\mathsf{isComplete} \triangleq \neg\exists\alpha.\ \Diamond\alpha \qquad \mathsf{safeName}(\mathrm{S}) \triangleq \neg\exists \mathrm{S}_1, \mathrm{S}_2.\ \mathrm{S} \doteq \mathrm{S}_1.\text{`\#'}.\mathrm{S}_2$$

$$\mathsf{val}(\mathrm{T}, \mathrm{S}) \triangleq (\mathrm{T} \doteq \varnothing_{tf} * \mathrm{S} \doteq \text{``''}) \vee (\exists \mathrm{N}, \mathrm{S}_1, \mathrm{S}_2, \mathrm{T}'.\ \mathrm{T} \doteq \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}_1]_{\_} \oslash \mathrm{T}' * \mathsf{val}(\mathrm{T}', \mathrm{S}_2) * \mathrm{S} \doteq \mathrm{S}_1.\mathrm{S}_2)$$

$$\mathsf{out}(\mathrm{A}, \mathrm{S}) \triangleq (\mathrm{A} \doteq \varnothing_a) \vee (\exists \mathrm{S}', \mathrm{N}, \mathrm{T}, \mathrm{A}'.\ \mathrm{A} \doteq \mathrm{S}'_{\mathrm{N}}[\mathrm{T}]_{\_} \odot \mathrm{A}' * \mathrm{S} \neq \mathrm{S}' * \mathsf{out}(\mathrm{A}', \mathrm{S}))$$

$$\mathsf{TIDs}(\mathrm{T}, \mathrm{L}) \triangleq (\mathrm{L} \doteq [\,] * \mathrm{T} \doteq \varnothing_f) \vee \big(\exists \mathrm{N}, \mathrm{S}, \mathrm{A}, \mathrm{F}, \mathrm{T}', \mathrm{L}'.\ \mathrm{L} \doteq \mathrm{N}{:}\mathrm{L}' * (\mathrm{T} \doteq \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}]_{\_} \otimes \mathrm{T}' \vee \mathrm{T} \doteq \mathrm{S}_{\mathrm{N}}[\mathrm{A},\mathrm{F}]^{\_} \otimes \mathrm{T}') * \mathsf{TIDs}(\mathrm{T}', \mathrm{L}')\big)$$

$$\mathsf{search}(\mathrm{T},\mathrm{S},\mathrm{L}) \triangleq (\mathrm{T} \doteq \varnothing_f * \mathrm{L} \doteq [\,]) \vee (\exists \mathrm{N}, \mathrm{S}', \mathrm{T}'.\ \mathrm{T} \doteq \#\mathrm{text}_{\mathrm{N}}[\mathrm{S}']_{\_} \otimes \mathrm{T}' * \mathsf{search}(\mathrm{T}', \mathrm{S}, \mathrm{L}))$$
$$\vee \big(\exists \mathrm{S}', \mathrm{N}, \mathrm{T}_1, \mathrm{T}_2, \mathrm{L}_1, \mathrm{L}_2.\ \mathrm{T} \doteq \mathrm{S}'_{\mathrm{N}}[-, \mathrm{T}_1]^{\_} \otimes \mathrm{T}_2 * \mathsf{search}(\mathrm{T}_1, \mathrm{S}, \mathrm{L}_1) * \mathsf{search}(\mathrm{T}_2, \mathrm{S}, \mathrm{L}_2)$$
$$* (\mathrm{S} \doteq \mathrm{S}' \vee \mathrm{S} \doteq \text{``*''} \Rightarrow \mathrm{L} \doteq \mathrm{N}{:}(\mathrm{L}_1 +\!\!+ \mathrm{L}_2)) * (\mathrm{S} \neq \mathrm{S}' \wedge \mathrm{S} \neq \text{``*''} \Rightarrow \mathrm{L} \doteq \mathrm{L}_1 +\!\!+ \mathrm{L}_2)\big)$$

Fig. 4: Derived DOM assertions

Our specifications have smaller footprints than those of [9,22]. In particular, the axiom of `appendChild` requires a substantial *overapproximation* of the footprint due to the reasons discussed in §2.1, namely the need for a *linking context* (see page 7). This axiom is given below using MCL [5] (adapted to our notation):

$$\big\{(C \bullet_\alpha \mathrm{S}_{\mathrm{N}}[\mathrm{A}, \gamma]) \bullet_\beta \mathrm{S}'_{\mathrm{O}}[\mathrm{A}', \mathrm{T}]\big\}\ \texttt{n.appendChild(o)}\ \big\{(C \bullet_\alpha \mathrm{S}_{\mathrm{N}}[\mathrm{A}, \gamma \otimes \mathrm{S}'_{\mathrm{O}}[\mathrm{A}', \mathrm{T}]]) \bullet_\beta \varnothing_f\big\}$$

This axiom is not small enough: the only parts required by `appendChild` are the tree at O being moved, and the element N whose children are extended by O. However, as before the precondition above also requires the linking context C.

3.5 Live Collections

The DOM API provides several interfaces for traversing DOM trees based on *live collections* of nodes, such as the *NodeList* interface in DOM CL1-4. DOM CL 4 also introduces the *HTMLCollection* interface for live collections of element nodes. We describe our model of live collections in terms of NodeLists. However, our model is abstract and captures the behaviour of both NodeLists and HTMLCollections.

The *NodeList* interface is an ordered collection of nodes. NodeLists are *live* in that they dynamically reflect document changes. Several DOM operations return NodeLists. For example, `n.getElementsByTagName(s)` returns a NodeList (using depth-first, left-to-right search) containing the identifiers of the elements named `s` underneath the tree rooted at `n`. Given the DOM tree of Fig. 1a, when `n`=4 and `s`="img", then `r = n.getElementsByTagName(s)` yields `r`=[3, 8, 2]. However, since NodeLists are live, if node 8 is later removed from the document, then `r`=[3, 2]. When `s`="*" denoting a wildcard, then the resulting NodeList must contain the identifiers of *all* element nodes underneath `n`. For instance, with the DOM tree of Fig. 1a, when `n`=4 and `s`="*", then `r = n.getElementsByTagName(s)` yields `r`=[9, 3, 8, 6, 2]. This operation may be called on both document and element nodes. We thus associate each such node with a *set of tag listeners*, *ts*. Each listener is of the form (s, fid) where s denotes the search string (e.g. "img" in the example above) and $\mathit{fid} \in \mathrm{ID}$ denotes the identifier of the resulting NodeList.

The `n.childNodes` operation also returns a NodeList, containing the identifiers of the immediate children of `n`. For instance, with the DOM tree of Fig. 1a,

when n=4, then r = n.childNodes returns r=[9, 6]. Again, the value of r is live and dynamically reflects the changes to the child forest of n. The n.childNodes operation may be called on *any* DOM node. We therefore associate each DOM node with a *set of forest listeners*, *fs*. Each forest listener, *fid* ∈ ID, denotes the identifier of a NodeList. Our specification is the first that faithfully models the behaviour of NodeLists. In particular, both [9] and [22] associate a single forest listener with DOM nodes and consequently admit behaviours that are not guaranteed by the standard. We proceed with the NodeList axioms in Fig. 3.

n.childNodes: when n=N, this operation returns (the identifier of) a forest listener NodeList F associated with N. Fig. 3 shows the axiom for when N is an element. When asked for a forest listener NodeList, a node may either return an existing one, or generate a fresh one and extend its set with it. This flexibility is due to an under-specification in the standard. Thus, in the postcondition the original set F_1 is extended to F_2 ($F_1 \dot{\subseteq} F_2$) with return value $F \in F_2$. The TIDs(T,L) assertion is defined in Fig. 4 and states that list L contains the top-level node identifiers (from left to right) of the forest denoted by T. For instance, TIDs(T, [9, 6]) holds in Fig. 1a when T denotes the child forest of node 4 (named "body"). As such, the TIDs(T,L) in the precondition stipulates that T contain enough resource for compiling a list of the immediate children of N (i.e. the top-level nodes in T).

n.getElementsByTagName(s): when n=N and s=S, this operation returns (the identifier of) a NodeList containing the identifiers of the elements with tag name S in the forest underneath N. The axiom in Fig. 3 describes the case when N is an element node. The original set of tag listeners E is extended to E′ with (S, R) ∈ E′ where R is the return value. The search(T, S, L) assertion is defined in Fig. 4 and describes the search result of getElementsByTagName (i.e. the list L contains the identifiers of those element nodes in the forest T whose name matches S). For instance, when T denotes the child forest of node 4 (named "body") in Fig. 1a, then both search(T, "img", [3, 8, 2]) and search(T, "*", [9, 3, 8, 6, 2]) hold. As such, the search(T, S, L) in the precondition ensures that T contains enough resource for compiling a list of elements named S.

f.length: when f=F identifies a NodeList, its length is returned. The axiom in Fig. 3 describes the case when F is a forest listener NodeList on element N; the return value is the number of N's immediate children. This is captured by TIDs(T,L) stipulating that list L contains the identifiers of those nodes at the top level of child forest T. The return value is thus the length of L (i.e. |L|).

f.item(i): this is analogous to f.length with $|L|^I$ denoting the Ith item of L. The axiom in Fig. 3 describes the case when F is a tag listener NodeList on N.

4 Verifying JavaScript Programs that Call the DOM

We instantiate the method described in §3.3 to extend the SL-based JavaScript program logic (hereafter JSLogic) in [7], to JSDOMLogic, in order to enable DOM reasoning. We then use JSDOMLogic to reason about a realistic ad blocker program in §4.1, and a further ad blocker in [17]. These examples are interesting as they combine JavaScript heap reasoning with DOM reasoning.

JSLogic States The states of JSLogic are *JavaScript heaps*. A JavaScript heap, $h \in \text{JSHEAP}$, is a partial function mapping *references*, which are pairs of memory locations and field names, to values. A heap cell is written $(l, x) \mapsto 7$, stating that the object at l has a field named x and holds value 7. An empty JavaScript heap is denoted by 0_{JS}; JavaScript heap composition, $\circ : \text{JSHEAP} \times \text{JSHEAP} \rightharpoonup \text{JSHEAP}$, is the standard disjoint function union. The PCM of JavaScript heaps is $(\text{JSHEAP}, \circ, \{0_{\text{JS}}\})$. The states of JSDOMLogic are then pairs of the form $(h, \mathbf{h})$, comprising a JavaScript heap h, and a DOM heap $\mathbf{h}$ (see Def. 5).

JSLogic Assertions, programming language and proof rules As stipulated by Def. 6, the JSLogic assertions include the standard boolean, classical and SL assertions. JSLogic further includes JavaScript heap assertions of the form $(E_1, E_2) \mapsto E_3$, describing a single-cell JavaScript heap. The variable store in JavaScript is emulated in the heap. As required by Def. 6, JSLogic introduces a derived assertion $\mathsf{store}(\overline{\mathtt{x}_i : \mathrm{V}_i})$, describing the JavaScript variable store in the heap where variables $\overline{\mathtt{x}_i}$ have values $\overline{\mathrm{V}_i}$. The programming language of JSLogic is a broad subset of the JavaScript language [7]. The JSLogic assertions, their semantics, the definition of store, and the JSLogic proof rules are given in [7].

4.1 A JavaScript Ad Blocker

We use JSDOMLogic to reason about an *ad blocker* script used for blocking the images from untrusted sources in a DOM tree. The `adBlocker1(n)` program in Fig. 5 compiles a NodeList containing all "img" elements in the tree rooted at `n` by calling the `getElementsByTagName` operation. It then iterates over this NodeList, sanitising each image by executing the `sanitiseImg` program in §2.

At each iteration I, the subtree at node n=N is described by $\mathsf{tree}(\mathrm{I}, \mathrm{E})$ where T_I denotes the child forest of N at iteration I, and E denotes the tag listener set associated with N, and L denotes the list of "img" elements below N.[3]

Since we iterate over the "img" elements in L and inspect their attributes, we need to *partition* them in into three categories: i) *empty*: without a "src" attribute; ii) *untrusted*: with a"src" attribute and a blacklisted value; iii) *trusted*: with a "src" attribute and a trusted value. At each iteration, if the node considered is untrusted, it is sanitised and removed from the untrusted category. We thus define a fourth category, *sanitised*, including those elements whose values were initially blacklisted and are later sanitised. This is captured by $\mathsf{partition}(\mathrm{I})^3$. The first part states that the list of "img" elements L can be partitioned into the three categories described above where $\mathrm{L} \equiv \mathrm{S}$ states that set S is a permutation of list L. The second part states that list L has been processed up to index I; i.e. the sanitised category S_s includes all the untrusted elements in L up to index I. The last four parts describe the "img" elements according to their category.

[3] All free logical variables on the right-hand side are parameters of the predicate on the left. We omit them for readability as they do not change throughout the execution. By contrast, the iteration number I, and the tag listeners E of node N may change (the latter may grow by `getElementsByTagName`) and are explicitly parameterised.

$\mathsf{cache}(\mathrm{C}) \triangleq \circledast_{\mathrm{F}\in\mathcal{X}} \big((\mathrm{C},\mathrm{F})\mapsto 1 \vee (\mathrm{C},\mathrm{F})\mapsto 0\big)$ $\quad \mathsf{unfld}(\mathrm{I},\mathrm{E}) \triangleq \exists \overline{\alpha,\beta,\gamma}^{\mathrm{L}}.\, \mathsf{partition}(\mathrm{I}) * (\forall \mathrm{I}.\, \mathsf{partition}(\mathrm{I}) \mathrel{-\!\!*} \mathsf{tree}(\mathrm{I},\mathrm{E}))$

$\mathsf{tree}(\mathrm{I},\mathrm{E}) \triangleq \alpha \mapsto \mathrm{S}_{\mathrm{N}}[\mathrm{A},\mathrm{T}_{\mathrm{I}}]^{\mathrm{E}}_{\mathrm{F}} * \mathsf{search}(\mathrm{T}_{\mathrm{I}},\text{“img”},\mathrm{L})$ $\quad \mathsf{fld}(\mathrm{I},\mathrm{E}) \triangleq \mathsf{tree}(\mathrm{I},\mathrm{E}) * \big((\forall \mathrm{I},\mathrm{E}.\, \mathsf{tree}(\mathrm{I},\mathrm{E}) \mathrel{-\!\!*} \mathsf{unfld}(\mathrm{I},\mathrm{E})) \wedge \mathrm{emp}\big)$

$\mathsf{rem}(\mathrm{I}) \triangleq \exists \mathrm{S}_s.\ \mathrm{S}_s \doteq \mathrm{S}_u \cap \{|\mathrm{L}|^{\mathrm{J}} \mid \mathrm{J} < \mathrm{I}\} * \beta \mapsto \varnothing_g \bigoplus_{\mathrm{J}\in\mathrm{S}_s} \mathrm{A}_{\mathrm{J}}$

$\mathsf{partition}(\mathrm{I}) \triangleq \mathrm{L} \doteq \mathrm{S}_e \uplus \mathrm{S}_u \uplus \mathrm{S}_t * \exists \mathrm{S}_s.\ \mathrm{S}_s \doteq \mathrm{S}_u \cap \{|\mathrm{L}|^{\mathrm{J}} \mid \mathrm{J} < \mathrm{I}\} * \circledast_{\mathrm{J}\in\mathrm{S}_s} \big(\alpha_{\mathrm{J}} \mapsto \mathrm{img}_{\mathrm{J}}[\beta_{\mathrm{J}} \odot \mathrm{src}_{\mathrm{M}_{\mathrm{J}}}[\#\mathrm{text}_{-}[\mathrm{S}]_{-}]_{\mathrm{F}'_{\mathrm{J}}}, \gamma_{\mathrm{J}}]^{\mathrm{E}_{\mathrm{J}}}_{\mathrm{F}_{\mathrm{J}}}\big)$
$\circledast_{\mathrm{J}\in\mathrm{S}_e} (\alpha_{\mathrm{J}} \mapsto \mathrm{img}_{\mathrm{J}}[\mathrm{A}_{\mathrm{J}}, \gamma_{\mathrm{J}}]^{\mathrm{E}_{\mathrm{J}}}_{\mathrm{F}_{\mathrm{J}}} * \mathsf{out}(\mathrm{A}_{\mathrm{J}},\mathrm{src})) * \circledast_{\mathrm{J}\in\mathrm{S}_t} \big(\alpha_{\mathrm{J}} \mapsto \mathrm{img}_{\mathrm{J}}[\beta_{\mathrm{J}} \odot \mathrm{src}_{\mathrm{M}_{\mathrm{J}}}[\mathrm{A}_{\mathrm{J}}]_{\mathrm{F}'_{\mathrm{J}}}, \gamma_{\mathrm{J}}]^{\mathrm{E}_{\mathrm{J}}}_{\mathrm{F}_{\mathrm{J}}} * \mathsf{val}(\mathrm{A}_{\mathrm{J}},\mathrm{V}_{\mathrm{J}}) * \neg\mathsf{isB}(\mathrm{V}_{\mathrm{J}})\big)$
$\circledast_{\mathrm{J}\in\mathrm{S}_u\setminus\mathrm{S}_s} \big(\alpha_{\mathrm{J}} \mapsto \mathrm{img}_{\mathrm{J}}[\beta_{\mathrm{J}} \odot \mathrm{src}_{\mathrm{M}_{\mathrm{J}}}[\mathrm{A}_{\mathrm{J}}]_{\mathrm{F}'_{\mathrm{J}}}, \gamma_{\mathrm{J}}]^{\mathrm{E}_{\mathrm{J}}}_{\mathrm{F}_{\mathrm{J}}} * \mathsf{val}(\mathrm{A}_{\mathrm{J}},\mathrm{V}_{\mathrm{J}}) * \mathsf{isB}(\mathrm{V}_{\mathrm{J}})\big)$

$\{\mathsf{store}(\mathtt{n}{:}\mathrm{N},\mathtt{cat}{:}\mathrm{S},\mathtt{cache}{:}\mathrm{C},\mathtt{imgs}{:}-,\mathtt{len}{:}-,\mathtt{i}{:}-,\mathtt{c}{:}-,\mathtt{isB}{:}-,\mathtt{url}{:}-) * \neg\mathsf{isB}(\mathrm{S}) * \mathsf{cache}(\mathrm{C}) * \mathsf{fld}(0,\mathrm{E}) * \mathsf{rem}(0)\}$

1. `adBlocker1(n)` $\triangleq$ {
2. `  imgs = n.getElementsByTagName("img");`
   $\{\exists \mathrm{R},\mathrm{E}'.\ \mathsf{store}(\mathtt{n}{:}\mathrm{N},\mathtt{cat}{:}\mathrm{S},\mathtt{cache}{:}\mathrm{C},\mathtt{imgs}{:}\mathrm{R},\mathtt{len}{:}-,\mathtt{i}{:}-,\mathtt{c}{:}-,\mathtt{isB}{:}-,\mathtt{url}{:}-) * \neg\mathsf{isB}(\mathrm{S}) * \mathsf{cache}(\mathrm{C}) * \mathsf{rem}(0) * \mathsf{fld}(0,\mathrm{E}') * \mathrm{E} \dot{\subseteq} \mathrm{E}' * (\text{“img”}, \mathrm{R}) \dot{\in} \mathrm{E}'\}$
3. `  len = imgs.length; i = 0;`
   $\{\exists \mathrm{R},\mathrm{E}'.\ \mathsf{store}(\mathtt{n}{:}\mathrm{N},\mathtt{cat}{:}\mathrm{S},\mathtt{cache}{:}\mathrm{C},\mathtt{imgs}{:}\mathrm{R},\mathtt{len}{:}|\mathrm{L}|,\mathtt{i}{:}0,\mathtt{c}{:}-,\mathtt{isB}{:}-,\mathtt{url}{:}-) * \neg\mathsf{isB}(\mathrm{S}) * \mathsf{cache}(\mathrm{C}) * \mathsf{rem}(0) * \mathsf{fld}(0,\mathrm{E}') * \mathrm{E} \dot{\subseteq} \mathrm{E}' * (\text{“img”}, \mathrm{R}) \dot{\in} \mathrm{E}'\}$
   $\{\exists \mathrm{R},\mathrm{E}',\mathrm{I}.\ \mathsf{store}(\mathtt{n}{:}\mathrm{N},\mathtt{cat}{:}\mathrm{S},\mathtt{cache}{:}\mathrm{C},\mathtt{imgs}{:}\mathrm{R},\mathtt{len}{:}|\mathrm{L}|,\mathtt{i}{:}\mathrm{I},\mathtt{c}{:}-,\mathtt{isB}{:}-,\mathtt{url}{:}-) * \neg\mathsf{isB}(\mathrm{S}) * \mathsf{cache}(\mathrm{C}) * \mathsf{rem}(\mathrm{I}) * \mathsf{fld}(\mathrm{I},\mathrm{E}') * \mathrm{E} \dot{\subseteq} \mathrm{E}' * (\text{“img”}, \mathrm{R}) \dot{\in} \mathrm{E}' * \mathrm{I} \dot{\leq} |\mathrm{L}|\}$
4. `  while(i<len){`
5. `    c = imgs.item(i);`
   $\{\exists \mathrm{R},\mathrm{E}',\mathrm{I}.\ \mathsf{store}(\mathtt{n}{:}\mathrm{N},\mathtt{cat}{:}\mathrm{S},\mathtt{cache}{:}\mathrm{C},\mathtt{imgs}{:}\mathrm{R},\mathtt{len}{:}|\mathrm{L}|,\mathtt{i}{:}\mathrm{I},\mathtt{c}{:}|\mathrm{L}|^{\mathrm{I}},\mathtt{isB}{:}-,\mathtt{url}{:}-) * \neg\mathsf{isB}(\mathrm{S}) * \mathsf{cache}(\mathrm{C}) * \mathsf{rem}(\mathrm{I}) * \mathsf{fld}(\mathrm{I},\mathrm{E}') * \mathrm{E} \dot{\subseteq} \mathrm{E}' * (\text{“img”}, \mathrm{R}) \dot{\in} \mathrm{E}' * \mathrm{I} \dot{\leq} |\mathrm{L}|\}$
   //Apply derivation steps in (10)-(12).
   $\{\exists \mathrm{R},\mathrm{E}',\mathrm{I}.\ \mathsf{store}(\mathtt{n}{:}\mathrm{N},\mathtt{cat}{:}\mathrm{S},\mathtt{cache}{:}\mathrm{C},\mathtt{imgs}{:}\mathrm{R},\mathtt{len}{:}|\mathrm{L}|,\mathtt{i}{:}\mathrm{I},\mathtt{c}{:}|\mathrm{L}|^{\mathrm{I}},\mathtt{isB}{:}-,\mathtt{url}{:}-) * \neg\mathsf{isB}(\mathrm{S}) * \mathsf{cache}(\mathrm{C}) * \mathsf{rem}(\mathrm{I}) * \mathrm{E} \dot{\subseteq} \mathrm{E}' * (\text{“img”}, \mathrm{R}) \dot{\in} \mathrm{E}' * \mathrm{I} \dot{<} |\mathrm{L}| * \mathsf{unfld}(\mathrm{I},\mathrm{E}') * ((\forall \mathrm{I},\mathrm{E}.\ \mathsf{tree}(\mathrm{I},\mathrm{E}) \mathrel{-\!\!*} \mathsf{unfld}(\mathrm{I},\mathrm{E})) \wedge \mathrm{emp})\}$
6. `    sanitiseImg(c,cat);`
   $\{\exists \mathrm{R},\mathrm{E}',\mathrm{I}.\ \mathsf{store}(\mathtt{n}{:}\mathrm{N},\mathtt{cat}{:}\mathrm{S},\mathtt{cache}{:}\mathrm{C},\mathtt{imgs}{:}\mathrm{R},\mathtt{len}{:}|\mathrm{L}|,\mathtt{i}{:}\mathrm{I},\mathtt{c}{:}|\mathrm{L}|^{\mathrm{I}},\mathtt{isB}{:}-,\mathtt{url}{:}-) * \neg\mathsf{isB}(\mathrm{S}) * \mathsf{cache}(\mathrm{C}) * \mathsf{rem}(\mathrm{I}{+}1) * \mathrm{E} \dot{\subseteq} \mathrm{E}' * (\text{“img”}, \mathrm{R}) \dot{\in} \mathrm{E}' * \mathrm{I} \dot{<} |\mathrm{L}| * \mathsf{unfld}(\mathrm{I}{+}1,\mathrm{E}') * ((\forall \mathrm{I},\mathrm{E}.\ \mathsf{tree}(\mathrm{I},\mathrm{E}) \mathrel{-\!\!*} \mathsf{unfld}(\mathrm{I},\mathrm{E})) \wedge \mathrm{emp})\}$
   // Apply derivation steps in (12)-(14).
   $\{\exists \mathrm{R},\mathrm{E}',\mathrm{I}.\ \mathsf{store}(\mathtt{n}{:}\mathrm{N},\mathtt{cat}{:}\mathrm{S},\mathtt{cache}{:}\mathrm{C},\mathtt{imgs}{:}\mathrm{R},\mathtt{len}{:}|\mathrm{L}|,\mathtt{i}{:}\mathrm{I},\mathtt{c}{:}|\mathrm{L}|^{\mathrm{I}},\mathtt{isB}{:}-,\mathtt{url}{:}-) * \neg\mathsf{isB}(\mathrm{S}) * \mathsf{cache}(\mathrm{C}) * \mathsf{rem}(\mathrm{I}{+}1) * \mathrm{E} \dot{\subseteq} \mathrm{E}' * (\text{“img”}, \mathrm{R}) \dot{\in} \mathrm{E}' * \mathrm{I} \dot{<} |\mathrm{L}| * \mathsf{fld}(\mathrm{I}{+}1, \mathrm{E}')\}$
7. `    i = i+1;`
   $\{\exists \mathrm{R},\mathrm{E}',\mathrm{I}.\ \mathsf{store}(\mathtt{n}{:}\mathrm{N},\mathtt{cat}{:}\mathrm{S},\mathtt{cache}{:}\mathrm{C},\mathtt{imgs}{:}\mathrm{R},\mathtt{len}{:}|\mathrm{L}|,\mathtt{i}{:}\mathrm{I},\mathtt{c}{:}-,\mathtt{isB}{:}-,\mathtt{url}{:}-) * \neg\mathsf{isB}(\mathrm{S}) * \mathsf{cache}(\mathrm{C}) * \mathsf{rem}(\mathrm{I}) * \mathrm{E} \dot{\subseteq} \mathrm{E}' * (\text{“img”}, \mathrm{R}) \dot{\in} \mathrm{E}' * \mathrm{I} \dot{\leq} |\mathrm{L}| * \mathsf{fld}(\mathrm{I}, \mathrm{E}')\}$
8. } } $\{\exists \mathrm{R},\mathrm{E}',\mathrm{I}.\ \mathsf{store}(\mathtt{n}{:}\mathrm{N},\mathtt{cat}{:}\mathrm{S},\mathtt{cache}{:}\mathrm{C},\mathtt{imgs}{:}-,\mathtt{len}{:}-,\mathtt{i}{:}-,\mathtt{c}{:}-,\mathtt{isB}{:}-,\mathtt{url}{:}-) * \neg\mathsf{isB}(\mathrm{S}) * \mathsf{cache}(\mathrm{C}) * \mathsf{rem}(|\mathrm{L}|) * \mathrm{E} \dot{\subseteq} \mathrm{E}' * (\text{“img”}, \mathrm{R}) \dot{\in} \mathrm{E}' * \mathrm{I} \dot{\leq} |\mathrm{L}| * \mathsf{fld}(|\mathrm{L}|, \mathrm{E}')\}$

Fig. 5: A proof sketch of the `adBlocker1` program[3]

The partition predicate describes the “img” elements in L only and does not include the *remainder* of the subtree at N. At every iteration, this remainder is untouched and the modified parts are in the partitions. We thus describe the remainder for an arbitrary iteration I as $\forall \mathrm{I}.\ \mathsf{partition}(\mathrm{I}) \mathrel{-\!\!*} \mathsf{tree}(\mathrm{I},\mathrm{E})$, i.e. the entire tree for that iteration, $\mathsf{tree}(\mathrm{I},\mathrm{E})$, *minus* its partitions. The *unfolded* tree at iteration I, $\mathsf{unfld}(\mathrm{I},\mathrm{E})$[3], consists of the partitions at I, plus the remainder.

Note that for NodeList operations such as `item` (line 5), we need the *folded* tree ($\mathsf{tree}(\mathrm{I},\mathrm{E})$) with the entire subtree containing the “img” list L, as required by their axioms (Fig. 3). Conversely, for the `sanitiseImg` call (line 6), we need the *unfolded* “img” elements ($\mathsf{partition}(\mathrm{I})$) so that we can access the relevant “img”

node at each iteration. We thus need to move between the folded and unfolded tree depending on the operation considered. The $\mathsf{fld}(\mathrm{I}, \mathrm{E})$ predicate describes the folded tree at iteration I. The first part, $\mathsf{tree}(\mathrm{I}, \mathrm{E})$, describes the resources of the folded tree at iteration I. The second part contains no resources (emp); it simply states that at any iteration I, the folded tree $\mathsf{tree}(\mathrm{I}, \mathrm{E})$, can be *exchanged* for the unfolded tree $\mathsf{unfld}(\mathrm{I}, \mathrm{E})$. As we show in the derivation below, this second part allows us to move from folded to unfolded resources (10-12) and vice versa (12-14), for any I. The bi-implication of (10) follows from the definition of fld and that empty resources (emp) can be freely duplicated. In (11) we eliminate the first universal quantifier. We then eliminate the adjunct $(P * (P \mathrel{-\!\!*} Q) \Rrightarrow Q)$ and arrive at (12). The implication of (13) follows from the definition of unfld and the elimination of the first universal quantifier. To get (14), we eliminate the adjunct, eliminate the existential quantifiers and wrap the definition of fld.

$$
\begin{aligned}
\mathsf{fld}(\mathrm{I}, \mathrm{E}) \Leftrightarrow\ & \mathsf{tree}(\mathrm{I}, \mathrm{E}) * (\forall \mathrm{I}, \mathrm{E}.\ \mathsf{tree}(\mathrm{I}, \mathrm{E}) \mathrel{-\!\!*} \mathsf{unfld}(\mathrm{I}, \mathrm{E}) \wedge \mathrm{emp}) \\
& * (\forall \mathrm{I}, \mathrm{E}.\ \mathsf{tree}(\mathrm{I}, \mathrm{E}) \mathrel{-\!\!*} \mathsf{unfld}(\mathrm{I}, \mathrm{E}) \wedge \mathrm{emp}) && (10) \\
\Rightarrow\ & \mathsf{tree}(\mathrm{I}, \mathrm{E}) * (\mathsf{tree}(\mathrm{I}, \mathrm{E}) \mathrel{-\!\!*} \mathsf{unfld}(\mathrm{I}, \mathrm{E})) * (\forall \mathrm{I}, \mathrm{E}.\, \mathsf{tree}(\mathrm{I}', \mathrm{E}) \mathrel{-\!\!*} \mathsf{unfld}(\mathrm{I}, \mathrm{E}) \wedge \mathrm{emp}) && (11) \\
\Rrightarrow\ & \mathsf{unfld}(\mathrm{I}, \mathrm{E}) * (\forall \mathrm{I}, \mathrm{E}.\, \mathsf{tree}(\mathrm{I}, \mathrm{E}) \mathrel{-\!\!*} \mathsf{unfld}(\mathrm{I}, \mathrm{E}) \wedge \mathrm{emp}) && (12) \\
\Rightarrow\ & \exists \overline{\alpha, \beta, \gamma}^{\mathrm{L}}.\ \mathsf{partition}(\mathrm{I}) * (\mathsf{partition}(\mathrm{I}) \mathrel{-\!\!*} \mathsf{tree}(\mathrm{I}, \mathrm{E})) \\
& * (\forall \mathrm{I}, \mathrm{E}.\ \mathsf{tree}(\mathrm{I}, \mathrm{E}) \mathrel{-\!\!*} \mathsf{unfld}(\mathrm{I}, \mathrm{E}) \wedge \mathrm{emp}) && (13) \\
\Rrightarrow\ & \mathsf{tree}(\mathrm{I}, \mathrm{E}) * (\forall \mathrm{I}, \mathrm{E}.\ \mathsf{tree}(\mathrm{I}, \mathrm{E}) \mathrel{-\!\!*} \mathsf{unfld}(\mathrm{I}, \mathrm{E}) \wedge \mathrm{emp}) \ \Leftrightarrow\ \mathsf{fld}(\mathrm{I}, \mathrm{E}) && (14)
\end{aligned}
$$

Recall that when the value of an attribute node is updated via the `setAttribute` operation, its text forest is replaced with a new text node containing the new value, and its old text forest is added to the grove (see axiom (3)). As such, at each iteration if we sanitise the "src" attribute of `c` (via `sanitiseImg` in line 6), then the old text forest of the "src" attribute is moved to the grove. This is described by the $\mathsf{rem}(\mathrm{I})$ assertion stating that for each attribute node sanitised so far (i.e. those in S_s), the old text forest A_J has been added to the grove.

Recall that `sanitiseImg` (Fig. 2) maintains a local cache of blacklisted URLs, implemented as an object at C with one field per URL (where $(\mathrm{C}, \mathrm{F}) \mapsto 1$ asserts the URL f is blacklisted, and $(\mathrm{C}, \mathrm{F}) \mapsto 0$ asserts that there are no cached results associated with f). We thus define the cache as the collection of all fields (denoted by $\mathcal{X}$) on C with value 1 or 0, where $\circledast$ is the iterated analogue of $*$.

We give a proof sketch of `adBlocker1` in Fig. 5. The precondition consists of the variable store, the cache and the *unprocessed* (iteration 0) tree. The postcondition comprises the store, the cache and the *fully processed* (iteration $|\mathrm{L}|$) tree with the tag listeners of N extended with a new listener for "img".

Concluding remarks We use SSL [25] to formally specify an expressive fragment of DOM Core Level 1, closely following the standard [1]. In comparison to existing work [9,22], our specification i) allows for *local* and *compositional* client specification and verification; ii) can be simply *integrated* with SL-based program logics; and iii) is *faithful* to the standard with respect to the behaviour of live collections. We demonstrate our compositional client reasoning by extending JSLogic [7] to incorporate our DOM specification and verifying functional properties of ad-blocker client programs that call the DOM.

Acknowledgements This research was supported by EPSRC programme grants EP/H008373/1, EP/K008528/1 and EP/K032089/1.

References

1. W3C DOM standard, `www.w3.org/TR/REC-DOM-Level-1/level-one-core.html`.
2. N. Biri and D. Galmiche. A Separation Logic for Resource Distribution. In *FST TCS*, 2003.
3. N. Biri and D. Galmiche. Models and separation logics for resource trees. In *Journal of Logic and Computation*, 2007.
4. M. Bodin, A. Charguéraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudžiūnienė, A. Schmitt, and G. Smith. A mechanised JavaScript specification. In *POPL*, 2014.
5. C. Calcagno, T. Dinsdale-Young, and P. Gardner. Adjunct elimination in context logic for trees. In *Programming Languages and Systems*, 2007.
6. C. Calcagno, P. Gardner, and U. Zarfaty. Context logic and tree update. In *POPL*, 2005.
7. P. Gardner, S. Maffeis, and G. Smith. Towards a program logic for JavaScript. In *POPL*, 2012.
8. P. Gardner, A. Raad, M. Wheelhouse, and A. Wright. Local reasoning for concurrent libraries: mind the gap. In *MFPS*, 2014.
9. P. Gardner, G. Smith, M. Wheelhouse, and U. Zarfaty. Local Hoare Reasoning about DOM. In *PODS*, 2008.
10. A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *OOPSLA*, 1999.
11. S. Jensen, A. Møller, and P.Thiemann. Type analysis for JavaScript. In *SAS*, 2009.
12. S.H. Jensen, M. Madsen, and A. Møller. Modeling the HTML DOM and browser API in static analysis of JavaScript Web applications. In *ESEC/FSE '11*, 2013.
13. B. S. Lerner, M. Carroll, D. P. Kimmel, H. Q. La Vallee, and S. Krishnamurthi. Modeling and reasoning about DOM events. In *WebApps*, 2012.
14. S. Maffeis, J. C. Mitchell, and A. Taly. An operational semantics for JavaScript. In *APLAS*, 2008.
15. C. Park, S. Won, J. Jin, and S. Ryu. A static analysis of JavaScript web applications in the wild via practical DOM modeling (T). In *ASE*, 2015.
16. M. Parkinson. *Local reasoning for Java.* PhD thesis, Cambridge University, 2006.
17. A. Raad. *(To appear).* PhD thesis, Imperial College, 2016.
18. V. Rajani, A. Bichhawat, D. Garg, Deepak, and C. Hammer. Information flow control for event handling and the DOM in web browsers. In *CSF*, 2015.
19. J. C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*, 2002.
20. A. Russo, A. Sabelfeld, and A. Chudnov. Tracking information flow in dynamic tree structures. In *ESORICS*, 2009.
21. A. Møller S. H. Jensen, M. Madsen. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In *FSE*, 2011.
22. G. Smith. *Local reasoning for web programs.* PhD thesis, Imperial College, 2010.
23. N. Swamy, J. Weinberger, C. Schlesinger, J. Chen, and B. Livshits. Verifying higher-order programs with the Dijkstra monad. In *PLDI*, 2013.
24. P. Thiemann. A type safe DOM API. In *DBPL*, 2005.
25. A. Wright. *Structural separation logic.* PhD thesis, Imperial College, 2013.

Verified Models and Reference Implementations for the TLS 1.3 Standard Candidate

Karthikeyan Bhargavan, Bruno Blanchet, Nadim Kobeissi
INRIA
{karthik.bhargavan,bruno.blanchet,nadim.kobeissi}@inria.fr

***Abstract*—TLS 1.3 is the next version of the Transport Layer Security (TLS) protocol. Its clean-slate design is a reaction both to the increasing demand for low-latency HTTPS connections and to a series of recent high-profile attacks on TLS. The hope is that a fresh protocol with modern cryptography will prevent legacy problems; the danger is that it will expose new kinds of attacks, or reintroduce old flaws that were fixed in previous versions of TLS. After 18 drafts, the protocol is nearing completion, and the working group has appealed to researchers to analyze the protocol before publication. This paper responds by presenting a comprehensive analysis of the TLS 1.3 Draft-18 protocol.**

We seek to answer three questions that have not been fully addressed in previous work on TLS 1.3: (1) Does TLS 1.3 prevent well-known attacks on TLS 1.2, such as Logjam or the Triple Handshake, even if it is run in parallel with TLS 1.2? (2) Can we mechanically verify the computational security of TLS 1.3 under standard (strong) assumptions on its cryptographic primitives? (3) How can we extend the guarantees of the TLS 1.3 protocol to the details of its implementations?

To answer these questions, we propose a methodology for developing verified symbolic and computational models of TLS 1.3 hand-in-hand with a high-assurance reference implementation of the protocol. We present symbolic ProVerif models for various intermediate versions of TLS 1.3 and evaluate them against a rich class of attacks to reconstruct both known and previously unpublished vulnerabilities that influenced the current design of the protocol. We present a computational CryptoVerif model for TLS 1.3 Draft-18 and prove its security. We present RefTLS, an interoperable implementation of TLS 1.0-1.3 and automatically analyze its protocol core by extracting a ProVerif model from its typed JavaScript code.

I. Introduction

The Transport Layer Security (TLS) protocol is widely used to establish secure channels on the Internet. It was first proposed under the name SSL [45] in 1994, and has undergone a series of revisions since, leading up to the standardization of TLS 1.2 [37] in 2008. Each version adds new features, deprecates obsolete constructions, and introduces countermeasures for weaknesses found in previous versions. The behavior of the protocol can be further customized via *extensions*, some of which are mandatory to prevent known attacks on the protocol.

One may expect that TLS clients and servers would use only the latest version of the protocol with all security-critical extensions enabled. In practice, however, many legacy variants of the protocol continue to be supported for backwards compatibility, and the everyday use of TLS depends crucially on clients and servers negotiating the most secure variant that they have in common. Securely composing and implementing the many different versions and features of TLS has proved to be surprisingly hard, leading to the continued discovery of high-profile vulnerabilities in the protocol.

A history of vulnerabilities. We identify four kinds of attacks that TLS has traditionally suffered from. *Downgrade* attacks enable a network adversary to fool a TLS client and server into using a weaker variant of the protocol than they would normally use with each other. In particular, version downgrade attacks were first demonstrated from SSL 3 to SSL 2 [72] and continue to be exploited in recent attacks like POODLE [60] and DROWN [7]. *Cryptographic* vulnerabilities rely on weaknesses in the protocol constructions used by TLS. Recent attacks have exploited key biases in RC4 [3], [71], padding oracles in MAC-then-Encrypt [4], [60], padding oracles in RSA PKCS#1 v1.5 [7], weak Diffie-Hellman groups [1], and weak hash functions [23]. *Protocol composition* flaws appear when multiple modes of the protocol interact in unexpected ways if enabled in parallel. For example, the renegotiation attack [65] exploits the sequential composition of two TLS handshakes, the Triple Handshake attack [15] composes three handshakes, and cross-protocol attacks [58], [72] use one kind of TLS handshake to attack another. *Implementation bugs* contribute to the fourth category of attacks on TLS, and are perhaps the hardest to avoid. They range from memory safety bugs like HeartBleed and coding errors like GotoFail to complex state machine flaws like SKIP and FREAK [12]. Such bugs can be exploited to bypass all the security guarantees of TLS, and their prevalence, even in widely-vetted code, indicates the challenges of implementing TLS securely.

Security proofs. Historically, when an attack is found on TLS, practitioners propose a temporary fix that is implemented in all mainstream TLS libraries, then a longer-term countermeasure is incorporated into a protocol extension or in the next version of the protocol. This has led to a attack-patch-attack cycle that does not provide much assurance in any single version of the protocol, let alone its implementations.

An attractive alternative would have been to develop security proofs that systematically demonstrated the absence of large classes of attacks in TLS. However, developing proofs for an existing standard that was not designed with security models in mind is exceedingly hard [63]. After years of effort, the cryptographic community only recently

published proofs for the two main components of TLS: the *record* layer that implements authenticated encryption [57], [62], and the *handshake* layer that composes negotiation and key-exchange [46], [51]. These proofs required new security definitions and custom cryptographic assumptions, and even so, they apply only to abstract models of certain modes of the protocol. For example, the proofs do not account for low-level details of message formats, downgrade attacks, or composition flaws. Since such cryptographic proofs are typically carried out by hand, extending the proofs to cover all these details would require a prohibitive amount of work, and the resulting large proofs themselves would need to be carefully checked.

A different approach taken by the protocol verification community is to *symbolically* analyze cryptographic protocols using simpler, stronger assumptions on the underlying cryptography, commonly referred to as the Dolev-Yao model [39]. Such methods are easy to automate and can tackle large protocols like TLS in all their gory detail, and even aspects of TLS implementations [31], [18]. Symbolic protocol analyzers are better at finding attacks, but since they treat cryptographic constructions as perfect black boxes, they provide weaker security guarantees than classic cryptographic proofs that account for probabilistic and computational attacks.

The most advanced example of mechanized verification for TLS is the ongoing miTLS project [21], which uses dependent types to prove both the symbolic and cryptographic security of a TLS implementation that supports TLS 1.0-1.2, multiple key exchanges and encryption modes, session resumption, and renegotiation. This effort has uncovered weaknesses in both the TLS 1.2 standard [15] and its other implementations [12], and the proof is currently being extended towards TLS 1.3.

Towards Verified Security for TLS 1.3. In 2014, the TLS working group at the IETF commenced work on TLS 1.3, with the goal of designing a faster protocol inspired by the success of Google's QUIC protocol [44]. Learning from the pitfalls of TLS 1.2, the working group invited the research community to contribute to the design of the protocol and help analyze its security even before the standard is published. A number of researchers, including the authors of this paper, responded by developing new security models and cryptographic proofs for various draft versions, and using their analyses to propose protocol changes. Cryptographic proofs were developed for Draft-5 [40], Draft-9 [52], and Draft-10 [55], which justified the core design of the protocol. A detailed symbolic model in Tamarin was developed for Draft-10 [35]. Other works studied specific aspects of TLS 1.3, such as key confirmation [41], client authentication [50], and downgrade resilience [14].

Some of these analyses also found attacks. The Tamarin analysis [35] uncovered a potential attack on the composition of pre-shared keys and certificate-based authentication, and this attack was prevented in Draft-11. A version downgrade attack was found in Draft-12 and its countermeasure in Draft-13 was proved secure [14]. A cross-protocol attack on RSA signatures was described in [47]. Even in this paper, we describe two vulnerabilities in 0-RTT client authentication that we discovered and reported, which influenced the subsequent designs of Draft-7 and -13.

After 18 drafts, TLS 1.3 is entering the final phase of standardization. Although many of its design decisions have now been vetted by multiple security analyses, several unanswered questions remain. First, the protocol has continued to evolve rapidly with every draft version, so many of the cryptographic proofs cited above are already obsolete and do not apply to Draft-18. Since many of these are manual proofs, it is not easy to update them and check all the proof steps. Second, none of these symbolic or cryptographic analyses, with the exception of [14], consider the composition of TLS 1.3 with legacy versions like TLS 1.2. Hence, they do not account for attacks like [47] that exploit weak legacy crypto in TLS 1.2 to break the modern cryptographic constructions of TLS 1.3. Third, none of these works addresses TLS 1.3 implementations. In this paper, we seek to cover these gaps with a new comprehensive analysis of TLS 1.3 Draft-18.

Our Contributions. We propose a methodology for developing mechanically verified models of TLS 1.3 alongside a high-assurance reference implementation of the protocol.

We present symbolic protocol models for TLS 1.3 written in ProVerif [27]. They incorporate a novel security model (described in §II) that accounts for all recent attacks on TLS, including those relying on weak cryptographic algorithms. In §III-V, we use ProVerif to evaluate various modes and drafts of TLS 1.3 culminating in the first symbolic analysis of Draft-18 and the first composite analysis of TLS 1.3+1.2. Our analyses uncover known and new vulnerabilities that influenced the final design of Draft-18. Some of the features we study no longer appear in the protocol, but our analysis is still useful for posterity, to warn protocol designers and developers who may be tempted to reintroduce these problematic features in the future.

In §VI, we develop the first machine-checked cryptographic proof for TLS 1.3 using the verification tool CryptoVerif [24]. Our proof reduces the security of TLS 1.3 Draft-18 to standard cryptographic assumptions over its primitives. In contrast to manual proofs, our CryptoVerif script can be more easily updated from draft-to-draft, and as the protocol evolves.

Our ProVerif and CryptoVerif models capture the protocol core of TLS 1.3, but they elide many implementation details such as the protocol API and state machine. To demonstrate that our security results apply to carefully-written implementations of TLS 1.3, we present RefTLS (§VII), the first reference implementation of TLS 1.0-1.3 whose core protocol code has been formally analyzed for security. RefTLS is written in Flow, a statically typed variant of JavaScript, and is structured so that all its protocol code is isolated in a single module that can be automatically translated to ProVerif and symbolically analyzed against our rich threat model.

The full version of this paper is published as a technical report [13], and our models and code are available at:

https://github.com/inria-prosecco/reftls

II. A Security Model for TLS

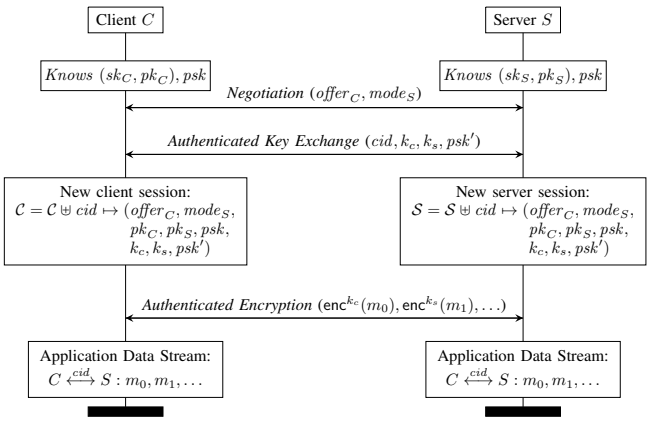


Figure 1: TLS Protocol Structure: Negotiation, then Authenticated Key Exchange (AKE), then Authenticated Encryption (AE) for application data streams.

Figure 1 depicts the progression of a typical TLS connection. Since a client and server may support different sets of features, they first *negotiate* a protocol mode that they have in common. In TLS, the client C makes an $offer_C$ and the server chooses its preferred $mode_S$, which includes the protocol version, the key exchange protocol, the authenticated encryption scheme, the Diffie-Hellman group (if applicable), and the signature and hash algorithms.

Then, C and S execute the negotiated *authenticated key exchange* protocol (e.g. Ephemeral Elliptic-Curve Diffie Hellman), which may use some combination of the long-term keys (e.g. public/private key pairs, symmetric pre-shared keys) known to the client and server. The key exchange ends by computing fresh symmetric keys (k_c, k_s) for a new session (with identifier cid) between C and S, and potentially a new pre-shared key (psk') that can be used to authenticate future connections between them.

In TLS, the negotiation and key exchange phases are together called the *handshake* protocol. Once the handshake is complete, C and S can start exchanging application data, protected by an authenticated encryption scheme (e.g. AES-GCM) with the session keys (k_c, k_s). The TLS protocol layer that handles authenticated encryption for application data is called the *record* protocol.

Security Goals for TLS. Each phase of a TLS connection has its own correctness and security goals. For example, during negotiation, the server must choose a $mode_S$ that is consistent with the client's $offer_C$; the key exchange must produce a secret session key, and so on. Although these intermediate security goals are important building blocks towards the security of the full TLS protocol, they are less meaningful to applications that typically use TLS via a TCP-socket-like API and are unaware of the protocol's internal structure. Consequently, we state the security goals of TLS from the viewpoint of the application, in terms of messages it sends and receives over a protocol session.

All goals are for messages between honest and authenticated clients and servers, that is, for those whose long-term keys (sk_C, sk_S, psk) are unknown to the attacker. If only the server is authenticated, then the goals are stated solely from the viewpoint of the client, since the server does not know whether it is talking to an honest client or the attacker.

Secrecy: If an application data message m is sent over a session cid between an honest client C and honest server S, then this message is kept confidential from an attacker who cannot break the cryptographic constructions used in the session cid.

Forward Secrecy: Secrecy (above) holds even if the long-term keys of the client and server (sk_C, pk_C, psk) are given to the adversary after the session cid has been completed and the session keys k_c, k_s are deleted by C and S.

Authentication: If an application data message m is received over a session cid from an honest and authenticated peer, then the peer must have sent the same application data m in a matching session (with the same parameters $cid, offer_C, mode_S, pk_C, pk_S, psk, k_c, k_s, psk'$).

Replay Prevention: Any application data m sent over a session cid may be accepted at most once by the peer.

Unique Channel Identifier: If a client session and a server session have the same identifier cid, then all other parameters in these sessions must match (same $cid, offer_C, mode_S, pk_C, pk_S, psk, k_c, k_s, psk'$).

These security goals encompass most of the standard security goals for secure channel protocols such as TLS. For example, secrecy for application data implicitly requires that the authenticated key exchange must generate secret keys. Authentication incorporates the requirement that the client and server must have matching sessions, and in particular, that they agree on each others' identities as well as the inputs and outputs of negotiation. Hence, it prohibits client and server impersonation, and man-in-the-middle downgrade attacks.

The requirement for a unique channel identifier is a bit more unusual, but it allows multiple TLS sessions to be securely composed, for example via session resumption or renegotiation, without exposing them to credential forwarding attacks like Triple Handshake [15]. The channel identifier could itself be a session key or a value generated from it, but is more usually a public value that is derived from session data contributed by both the client and server [17].

Symbolic vs. Computational Models. Before we can model and verify TLS 1.3 against the security goals given above, we need to specify our protocol execution model. There are two different styles in which protocols have classically been modeled, and in this paper, we employ both of them. *Symbolic* models were developed by the security protocol verification community for ease of automated analysis. Cryptographers, on the other hand, prefer to use *computational* models and do their proofs by hand. A full comparison between these styles is beyond the scope of this paper (see e.g. [26]); here we briefly outline their differences in terms of the two tools we will use.

ProVerif [25], [27] analyzes symbolic protocol models, whereas CryptoVerif [24] verifies computational models.

The input languages of both tools are similar. For each protocol role (e.g. client or server) we write a *process* that can send and receive messages over public channels, trigger security events, and store messages in persistent databases.

In ProVerif, messages are modeled as abstract terms. Processes can generate new nonces and keys, which are treated as atomic opaque terms that are fresh and unguessable. Functions map terms to terms. For example, encryption constructs a complex term from its arguments (key and plaintext) that can only be deconstructed by decryption (with the same key). The attacker is an arbitrary ProVerif process running in parallel with the protocol, which can read and write messages on public channels, and can manipulate them symbolically.

In CryptoVerif, messages are concrete bitstrings. Freshly generated nonces and keys are randomly sampled bitstrings that the attacker can guess with some probability (depending on their length). Encryption and decryption are functions on bitstrings to which we may associate standard cryptographic assumptions such as IND-CCA. The attacker is a probabilistic polynomial-time CryptoVerif process running in parallel.

Authentication goals in both ProVerif and CryptoVerif are written as correspondences between events: for example, if the client triggers a certain event, then the server must have triggered a matching event in the past. Secrecy is treated differently in the two tools; in ProVerif, we typically ask whether the attacker can compute a secret, whereas in CryptoVerif, we ask whether it can distinguish a secret from a random bitstring.

The analysis techniques employed by the two tools are quite different. ProVerif searches for a protocol trace that violates the security goal, whereas CryptoVerif tries to construct a cryptographic proof that the protocol is equivalent (with high probability) to a trivially secure protocol. ProVerif is a push-button tool that may return that the security goal is true in the symbolic model, or that the goal is false with a counterexample, or that it is unable to conclude, or may fail to terminate. CryptoVerif is semi-automated, it can search for proofs but requires human guidance for non-trivial protocols.

We use both ProVerif and CryptoVerif for their complementary strengths. CryptoVerif can prove stronger security properties of the protocol under precise cryptographic assumptions, but the proofs require more work. ProVerif can quickly analyze large protocols to automatically find attacks, but a positive result does not immediately provide a cryptographic proof of security. Deriving sound cryptographic proofs using symbolic analysis is still an open problem for real-world protocols [34].

A Realistic Threat Model for TLS. We seek to analyze TLS 1.3 for the above security goals against a rich threat model that includes both classic protocol adversaries as well as new ones that apply specifically to multi-mode protocols like TLS. In particular, we model recent downgrade attacks on TLS by allowing the use of weak cryptographic algorithms in older versions of TLS. In our analyses, the attacker can use any of the following attack vectors to disrupt the protocol.

- **Network Adversary:** As usual, we assume that the attacker can intercept, modify, and send all messages sent on public network channels.
- **Compromised Principals:** The attacker can compromise any client or server principal P by asking for its long-term secrets, such as its private key (sk_P) or pre-shared key (psk). We do not restrict which principals can be compromised, but whenever such a compromise occurs, we mark it with a security event: $\texttt{Compromised}(pk_{\texttt{P}})$ or $\texttt{CompromisedPSK}(psk)$. If the compromise event occurs after a session is complete, we issue a different security event: $\texttt{PostSessionCompromise}(\texttt{cid}, pk_{\texttt{P}})$.
- **Weak Long-term Keys:** If the client or server has a weak key that the attacker may be able to break with sufficient computation, we treat such keys the same way as compromised keys and we issue a more general event:$\texttt{WeakOrCompromised}(pk_{\texttt{P}})$. This conservative model of weak keys is enough to uncover attacks like FREAK [12] that rely on the use of 512-bit RSA keys by TLS servers.
- **RSA Decryption Oracles:** TLS versions up to 1.2 use RSA PKCS#1 v1.5 encryption, which is known to be vulnerable to a form of padding oracle attack on decryption originally discovered by Bleichenbacher [28]. Although countermeasures to this attack have been incorporated into TLS, they remains hard to implement securely [59] resulting in continued attacks such as DROWN [7]. Furthermore, such padding oracles can sometimes even be converted to signature oracles for the corresponding private key [47].
  We assume that any TLS server (at any version) that enables RSA decryption may potentially be vulnerable to such attacks. We distinguish between two kinds of RSA key exchange: $\texttt{RSA}(\texttt{StrongRSADecryption})$ and $\texttt{RSA}(\texttt{WeakRSADecryption})$. In any session, if the server chooses the latter, we provide the attacker with a decryption and signature oracle for that private key.
- **Weak Diffie-Hellman Groups:** To account for attacks like Logjam [1], we allow servers to choose between strong and weak Diffie-Hellman groups (or elliptic curves), and mark the corresponding key exchange mode as $\texttt{DHE}(\texttt{StrongDH})$ or $\texttt{DHE}(\texttt{WeakDH})$. We conservatively assume that weak groups have size 1, so all Diffie-Hellman exponentiations in these groups return the same distinguished element `BadElement`.
  Even strong Diffie-Hellman groups typically have small subgroups that should be avoided. We model these subgroups by allowing a weak subgroup (of size 1) even within a strong group. A malicious client or server may choose `BadElement` as its public value, and then all exponentiations with this element as the base will also return `BadElement`. To avoid generating keys in this subgroup, clients and servers must validate the received public value.
- **Weak Hash Functions:** TLS uses hash functions for key derivation, HMAC, and for signatures. Versions up to TLS 1.2 use various combinations of MD5 and SHA-1, both of which are considered weak today, leading to exploitable attacks on TLS such as SLOTH [23].

We model both strong and weak hash functions, and the client and server get to negotiate which function they will use in signatures. Strong hash functions are treated as one-way functions in our symbolic model, whereas weak hash functions are treated as point functions that map all inputs to a constant value: `Collision`. Hence, in our model, it is trivial for the attacker to find collisions as well as second preimages for weak hash functions.

- **Weak Authenticated Encryption:** To model recent attacks on RC4 [3], [71] and TripleDES [22], we allow both weak and strong authenticated encryption schemes. For data encrypted with a weak scheme, irrespective of the key, we provide the adversary with a decryption oracle.
  A number of attacks on the TLS Record protocol stem from its use of a MAC-Encode-Encrypt construction for CBC-mode ciphersuites. This construction is known to be vulnerable to padding oracle attacks such as POODLE [60] and Lucky13 [4], and countermeasures have proved hard to implement correctly [2]. We model such attacks using a leaky decryption function. Whenever a client or server decrypts a message with this function, the function returns the right result but also leaks the plaintext to the adversary.

The series of threats described above comprise our conservative threat model for TLS 1.3, and incorporates entire classes of attacks that have been shown to be effective against older versions of the protocol, including Triple Handshake, POODLE, Lucky 13, RC4 NOMORE, FREAK, Logjam, SLOTH, DROWN. In most cases, we assume strictly stronger adversaries than have been demonstrated in practice, but since attacks only get better over time, our model seeks to be defensive against future attacks. It is worth noting that, even though TLS 1.3 does not itself support any weak ciphers, TLS 1.3 clients and servers will need to support legacy protocol versions for backwards compatibility. Our model enables a fine-grained analysis of vulnerabilities: we can ask whether TLS 1.3 connections between a client and a server are secure even if TLS 1.2 connections between them are broken.

Verifying TLS 1.2 in ProVerif. We encode our threat model as a generic ProVerif crypto library that can be used with any protocol. To evaluate this model, and in preparation for our analysis of TLS 1.3, we symbolically analyze a model of TLS 1.2 using ProVerif. Our model includes TLS 1.2 clients and servers that support both RSA and Diffie-Hellman key exchanges, and are willing to use both weak and strong cryptography. We assume that clients are unauthenticated.

We write ProVerif processes for TLS 1.2 clients and servers that exchange messages according to the protocol standard, and issue a sequence of events–`ClientOffers`, `ServerChooses`, `ClientFinished`, `ServerFinished`, `ClientSends`, `ServerReceives`–indicating their progress through the protocol. We then compose these processes with our threat model and add queries for message authenticity and secrecy. For example, a secrecy query may ask whether the attacker can learn some application data message `m` sent by the client over a TLS 1.2 session with identifier `cid`.

When we run ProVerif for this query, it finds a counter-example: the attacker can learn `m` if it can compromise server's private key ($\texttt{WeakOrCompromised}(\texttt{pk}_\texttt{S})$). To check whether this is the only case in which `m` is leaked, we refine the secrecy query and run ProVerif again. ProVerif again finds a counter-example: the attacker can learn `m` if the server chooses a weak Diffie-Hellman group ($\texttt{ServerChoosesKex}(\texttt{DHE}(\texttt{WeakDH}))$). In this way, we keep refining our queries until we obtain the strongest security properties that hold for TLS 1.2 in our model:

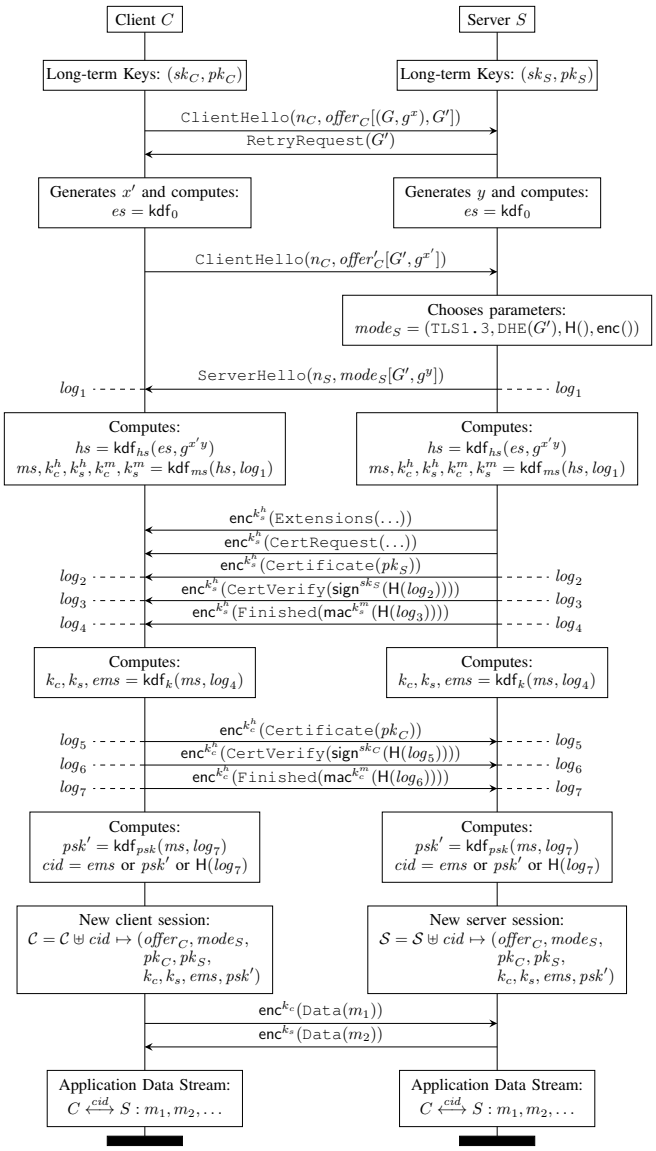
- **TLS 1.2 Secrecy:** A message m sent by an honest client in a session cid to a server S cannot be known to the adversary unless one of the following conditions holds:
  (1) the server's public key is weak or compromised, or
  (2) the session uses a weak Diffie-Hellman group, or
  (3) the session uses weak authenticated encryption, or
  (4) the server uses weak RSA decryption with the same public key (in this or any other session), or
  (5) the server uses a weak hash function for signing with the same public key (in any session).
- **TLS 1.2 Authenticity & Replay Protection:** Every message m accepted by an honest client in a session cid with some server S corresponds to a unique message sent by S on a matching session, unless one of the conditions (1)-(5) above holds.

Both these queries are verified by ProVerif in a few seconds. All the disjuncts (1)-(5) in these queries are necessary, removing any of them results in a counterexample discovered by ProVerif, corresponding to some well-known attack on badly configured TLS 1.2 connections.

Interestingly, the conditions (2) and (3) are session specific, that is, only the sessions where these weak constructions are used are affected. In contrast, (4) and (5) indicate that the use of weak RSA decryption or a weak hash function in any session affects all other sessions that use the same server public key. As we shall see, this has an impact on the security of TLS 1.3 when it is composed with TLS 1.2.

We can also verify our TLS 1.2 model for more advanced properties. Forward secrecy does not hold in general for TLS 1.2, but can be proved for DHE sessions that use strong groups. Channel identifiers like $cid = k_c$ are not unique, and ProVerif finds a variant of the Triple Handshake attack, unless we implement the recommended countermeasure [64].

Verification Effort. The work of verifying TLS 1.2 can be divided into three tasks. We first modeled the threat model as a 400 line ProVerif library, but this library can now be reused for other protocols, including TLS 1.3. We then modeled the TLS 1.2 protocol in about 200 lines of ProVerif. Finally, we wrote about 50 lines of queries, both to validate our model (e.g. checking that the protocol completes in the absence of an attacker) and to prove our desired security goals. Most of the effort is in formalizing, refining, and discovering the right security queries. Although ProVerif is fully automated, verification gets more expensive as the protocol grows more complex. So, as we extend our models to cover multiple modes of TLS 1.3 composed with TLS 1.2, we sometimes need to simplify or restructure our models to aid verification.



Key Derivation Functions:

$\mathsf{hkdf\text{-}extract}(k, s) = \mathsf{HMAC\text{-}H}^k(s)$

$\mathsf{hkdf\text{-}expand\text{-}label}_1(s, l, h) =$
$\quad \mathsf{HMAC\text{-}H}^s(len_{\mathsf{H}()} \| \text{“}\texttt{TLS 1.3,}\text{”} \| l \| h \| \texttt{0x01})$

$\mathsf{derive\text{-}secret}(s, l, m) = \mathsf{hkdf\text{-}expand\text{-}label}_1(s, l, \mathsf{H}(m))$

1-RTT Key Schedule:

$\mathsf{kdf}_0 = \mathsf{hkdf\text{-}extract}(0^{len_{\mathsf{H}()}}, 0^{len_{\mathsf{H}()}})$

$\mathsf{kdf}_{hs}(es, e) = \mathsf{hkdf\text{-}extract}(es, e)$

$\mathsf{kdf}_{ms}(hs, log_1) = ms, k_c^h, k_s^h, k_c^m, k_s^m$ where
$\quad ms = \mathsf{hkdf\text{-}extract}(hs, 0^{len_{\mathsf{H}()}})$
$\quad hts_c = \mathsf{derive\text{-}secret}(hs, \mathsf{hts}_c, log_1)$
$\quad hts_s = \mathsf{derive\text{-}secret}(hs, \mathsf{hts}_s, log_1)$
$\quad k_c^h = \mathsf{hkdf\text{-}expand\text{-}label}(hts_c, \mathsf{key}, \text{“”})$
$\quad k_c^m = \mathsf{hkdf\text{-}expand\text{-}label}(hts_c, \mathsf{finished}, \text{“”})$
$\quad k_s^h = \mathsf{hkdf\text{-}expand\text{-}label}(hts_s, \mathsf{key}, \text{“”})$
$\quad k_s^m = \mathsf{hkdf\text{-}expand\text{-}label}(hts_s, \mathsf{finished}, \text{“”})$

$\mathsf{kdf}_k(ms, log_4) = k_c, k_s, ems$ where
$\quad ats_c = \mathsf{derive\text{-}secret}(ms, \mathsf{ats}_c, log_4)$
$\quad ats_s = \mathsf{derive\text{-}secret}(ms, \mathsf{ats}_s, log_4)$
$\quad ems = \mathsf{derive\text{-}secret}(ms, \mathsf{ems}, log_4)$
$\quad k_c = \mathsf{hkdf\text{-}expand\text{-}label}(ats_c, \mathsf{key}, \text{“”})$
$\quad k_s = \mathsf{hkdf\text{-}expand\text{-}label}(ats_s, \mathsf{key}, \text{“”})$

$\mathsf{kdf}_{psk}(ms, log_7) = psk'$ where
$\quad psk' = \mathsf{derive\text{-}secret}(ms, \mathsf{rms}, log_7)$

PSK-based Key Schedule:

$\mathsf{kdf}_{es}(psk) = es, k^b$ where
$\quad es = \mathsf{hkdf\text{-}extract}(0^{len_{\mathsf{H}()}}, psk)$
$\quad k^b = \mathsf{derive\text{-}secret}(es, \mathsf{pbk}, \text{“”})$

$\mathsf{kdf}_{0RTT}(es, log_1) = k_c$ where
$\quad ets_c = \mathsf{derive\text{-}secret}(es, \mathsf{ets}_c, log_1)$
$\quad k_c = \mathsf{hkdf\text{-}expand\text{-}label}(ets_c, \mathsf{key}, \text{“”})$

Figure 2: TLS 1.3 Draft-18 1-RTT Protocol (left) and Key Schedule (right). The protocol uses an (EC)DHE key exchange with server certificate authentication: client authentication and the `RetryRequest` negotiation steps are optional. The hash function $\mathsf{H}()$ used in the key schedule is typically `SHA-256`, which has length $len_{\mathsf{H}()} = 32$ bytes. The PSK-based key derivations in the key schedule are not used in the 1-RTT protocol here; they will be used later in Figure 4.

III. TLS 1.3 1-RTT: Simpler, Faster Handshakes

In its simplest form, TLS 1.3 consists of a Diffie-Hellman handshake, typically using an elliptic curve, followed by application data encryption using an AEAD scheme like AES-GCM. The essential structure of 1-RTT has remained stable since early drafts of TLS 1.3. It departs from the TLS 1.2 handshake in two ways. First, the key exchange is executed alongside the negotiation protocol so the client can start sending application data along with its second flight of messages (after one round-trip, hence 1-RTT), unlike TLS 1.2 where the client had to wait for two message flights from the server. Second, TLS 1.3 eliminates a number of problematic features in TLS 1.2; it removes RSA key transport, weak encryption schemes (RC4, TripleDES, AES-CBC), and renegotiation; it requires group negotiation with strong standardized Diffie-Hellman groups, and it systematically binds session keys to the handshake log to prevent attacks like the Triple Handshake. In this section, we detail the protocol flow, we model it in ProVerif, and we analyze it alongside TLS 1.2 in the security model of §II.

1-RTT Protocol Flow. A typical 1-RTT connection in Draft 18 proceeds as shown in Figure 2. The first four messages form the negotiation phase. The client sends a `ClientHello` message containing a nonce n_C and an $offer_C$ that lists the versions, groups, hash functions, and authenticated encryption algorithms that it supports. For each group G that the client supports, it may include a Diffie-Hellman key share g^x. On receiving this message, the server chooses a $mode_S$ that fixes the version, group, and all other session parameters. Typically, the server chooses a group G for which the client already provided a public value, and so it can send its `ServerHello` containing a nonce n_S, $mode_S$ and g^y to the client. If none of the client's groups are acceptable, the server may ask the client (via `RetryRequest`) to resend the client hello with a key share $g^{x'}$ for the server's preferred group G'. (In this case, the handshake requires two round trips.)

Once the client receives the `ServerHello`, the negotiation is complete and both participants derive handshake encryption keys from $g^{x'y}$, one in each direction (k_c^h, k_s^h), with which they encrypt all subsequent handshake messages. The client and server also generate two MAC keys (k_c^m, k_s^m) for use in the `Finished` messages described below. The server then sends a flight of up to 5 encrypted messages: `Extensions` contains any protocol extensions that were not sent in the `ServerHello`; `CertRequest` contains an optional request for a client certificate; `Certificate` contains the server's X.509 public-key certificate; `CertVerify` contains a signature with server's private key sk_S over the log of the transcript so far (log_2); `Finished` contains a MAC with k_s^m over the current log (log_3). Then the server computes the 1-RTT traffic keys k_c, k_s and may immediately start using k_s to encrypt application data to the client.

Upon receiving the server's encrypted handshake flight, the client verifies the certificate, the signature, and the MAC, and if all verifications succeed, the client sends its own second flight consisting of an optional certificate `Certificate` and signature `CertVerify`, followed by a mandatory `Finished` with a MAC over the full handshake log. Then the client starts sending its own application data encrypted under k_c. Once the server receives the client's second flight, we consider the handshake complete and put all the session parameters into the local session databases at both client and server ($\mathcal{C}$, $\mathcal{S}$).

In addition to the traffic keys for the current session, the 1-RTT handshake generates two extra keys: ems is an exporter master secret that may be used by the application to bind authentication credentials to the TLS channel; psk' is a resumption master secret that may be used as a pre-shared key in future TLS connections between C and S.

The derivation of keys in the protocol follows a linear key schedule, as depicted on the right of Figure 2. The first version of this key schedule was inspired by OPTLS [52] and introduced into TLS 1.3 in Draft-7. The key idea in this design is to accumulate key material and handshake context into the derived keys using a series of HKDF invocations as the protocol progresses. For example, in connections that use pre-shared keys (see §V), the key schedule begins by deriving es from psk, but after the `ServerHello`, we add in $g^{x'y}$ to obtain the handshake secret hs. Whenever we extract encryption keys, we mix in the current handshake log, in order to avoid key synchronization attacks like the Triple Handshake.

Since its introduction in Draft-7, the key schedule has undergone many changes, with a significant round of simplifications in Draft-13. Since all previously published analyses of 1-RTT predate Draft-13, this leaves open the question whether the current Draft-18 1-RTT protocol is still secure.

Modeling 1-RTT in ProVerif. We write client and server processes in ProVerif that implement the message sequence and key schedule of Figure 2.

Our models are abstract with respect to the message formats, treating each message (e.g. $\texttt{ClientHello}(\cdots)$) as a symbolic constructor, with message parsing modeled as a pattern-match with this constructor. This means that our analysis assumes that message serialization and parsing is correct; it won't find any attacks that rely on parsing ambiguities or bugs. This abstract treatment of protocol messages is typical of symbolic models; the same approach is taken by Tamarin [35]. In contrast, miTLS [21] includes a fully verified parser for TLS messages.

The key schedule is written as a sequence of ProVerif functions built using an HMAC function, $\texttt{hmac}(\texttt{H}, \texttt{m})$, which takes a hash function `H` as argument and is assumed to be a one-way function as long as $\texttt{H} = \texttt{StrongHash}$. All other cryptographic functions are modeled as described in §II, with both strong and weak variants.

Persistent state is encoded using tables. To model principals and their long-term keys, we use a global private table that maps principals (A) to their key pairs ((sk_A, pk_A)). To begin with, the adversary does not know any of the private keys in this table, but it can compromise any principal and obtain her private key. As described in §II, this compromise is recorded in ProVerif by an event $\texttt{WeakOrCompromised}(pk_{\texttt{A}})$.

As the client and server proceed through the handshake they record security events indicating their progress. We treat the negotiation logic abstractly; the adversary gets to choose $offer_C$ and $mode_S$, and we record these choices as events (`ClientOffers`, `ServerChooses`) at the client and server. When the handshake is complete, the client and server issue events `ServerFinished`, `ClientFinished`, and store their newly established sessions in two private tables `clientSession` and `serverSession` (corresponding to $\mathcal{C}$ and $\mathcal{S}$). These tables are used by the record layer to retrieve the traffic keys k_c, k_s for authenticated encryption. Whenever the client or server sends or receives an application data message, it issues further events (`ClientSends`, `ServerReceives`, etc.) We use all these events along with the client and server session tables to state our security goals.

1-RTT Security Goals. We encode our security goals as ProVerif *queries* as follows:

- **Secrecy** for a message, such as m_1, is encoded using an auxiliary process that asks the adversary to guess the

value of m_1; if the adversary succeeds, the process issues an event $\texttt{MessageLeaked}(cid, \texttt{m}_1)$. We then write a query to ask ProVerif whether this event is reachable.

- **Forward Secrecy** is encoded using the same query, but we explicitly leak the client and server's long-term keys (sk_C, sk_S) at the end of the session cid. ProVerif separately analyzes pre-compromise and post-compromise sessions as different *phases*; the forward secrecy query asks that messages sent in the first phase are kept secret even from attackers who learn the long-term keys in the second phase.
- **Authentication** for a message m_1 received by the server is written as a query that states that whenever the event $\texttt{ServerReceives}(cid, \texttt{m}_1)$ occurs, it must be preceded by three matching events: $\texttt{ServerFinished}(cid, \ldots)$, $\texttt{ClientFinished}(cid, \ldots)$, and $\texttt{ClientSends}(cid, \texttt{m}_1)$, which means that some honest client must have sent m_1 on a matching session. The authentication query for messages received by clients is similar.
- **Replay protection** is written as a stronger variant of the authentication query that requires *injectivity*: each `ServerReceives` event must correspond to a unique, matching, preceding `ClientSends` event.
- **Unique Channel Identifiers** are verified using another auxiliary process that looks up sessions from the `clientSession` and `serverSession` tables and checks that if the cid in both is the same, then all other parameters match. Otherwise it raises an event, and we ask ProVerif to prove that this event is not reachable.

When we first ask ProVerif to verify these queries, it fails and provides counterexamples; for example, client message authentication does not hold if the client is compromised $\texttt{Compromised}(pk_\texttt{C})$ or unauthenticated in the session. We then refine the query by adding this failure condition as a disjunct, and run ProVerif again and repeat the process until the query is proved. Consequently, our final verification results are often stated as a long series of disjuncts listing the cases where the desired security goal does not hold.

Verifying 1-RTT in Isolation. For our model of Draft-18 1-RTT, ProVerif can prove the following secrecy query about all messages $(m_{0.5}, m_1, m_2)$:

- **1-RTT (Forward) Secrecy**: Messages m sent in a session between C and S are secret as long as the private keys of C and S are not revealed before the end of the session, and the server chooses a $mode_S$ with a strong Diffie-Hellman group, a strong hash function, and a strong authenticated encryption algorithm.

If we further assume that TLS 1.3 clients and servers only support strong algorithms, we can simplify the above query to show that all messages sent between uncompromised principals are kept secret. In the rest of this paper, we assume that TLS 1.3 only enables strong algorithms, but that earlier versions of the protocol may continue to support weak algorithms.

Messages m_1 from the client to the server enjoy strong authentication and protection from replays:

- **1-RTT Authentication (and Replay Prevention)**: If a message m is accepted by S over a session with an honest C, then this message corresponds to a unique message sent by the C over a matching session.

However the authentication guarantee for messages $m_{0.5}, m_1$ received by the client is weaker. Since the client does not know whether the server sent this data before or after receiving the client's second flight, the client and server sessions may disagree about the client's identity. Hence, for these messages, we can only verify a weaker property:

- **0.5-RTT Weak Authentication (and Replay Prevention)**: If a message m is accepted by C over a session with an honest S, then this message corresponds to a unique message sent by S over a server session that matches all values in the client session except (possibly) the client's public key pk_C, the resumption master secret psk', and the channel identifier cid.

We note that by allowing the server to send 0.5-RTT data, Draft-18 has weakened the authentication guarantees for all data received by an authenticated client. For example, if a client requests personal data from the server over a client-authenticated 1-RTT session, a network attacker could delay the client's second flight ($\texttt{Certificate}-\texttt{Finished}$) so that when the client receives the server's 0.5-RTT data, it thinks that it contains personal data, but the server actually sent data intended for an anonymous client.

Verifying TLS 1.3 1-RTT composed with TLS 1.2. We combine our model with the TLS 1.2 model described at the end of §II so that each client and server supports both versions. We then ask the same queries as above, but only for sessions where the server chooses TLS 1.3 as the version in $mode_S$. Surprisingly, ProVerif finds two counterexamples.

First, if a server supports `WeakRSADecryption` with RSA key transport in TLS 1.2, then the attacker can use the RSA decryption oracle to forge TLS 1.3 server signatures and hence break our secrecy and authentication goals. This attack found by ProVerif directly corresponds to the cross-protocol Bleichenbacher attacks described in [47], [7]. It shows that removing RSA key transport from TLS 1.3 is not enough, one must disable the use of TLS 1.2 RSA mode on any server whose certificate may be accepted by a TLS 1.3 client.

Second, if a client or server supports a weak hash function for signatures in TLS 1.2, then ProVerif shows how the attacker can exploit this weakness to forge TLS 1.3 signatures in our model, hence breaking our security goals. This attack corresponds to the SLOTH transcript collision attack on TLS 1.3 signatures described in [23]. To avoid this attack, TLS 1.3 implementations must disable weak hash functions in all supported versions, not just TLS 1.3.

After disabling these weak algorithms in TLS 1.2, we can indeed prove all our expected security goals about Draft-18 1-RTT, even when it is composed with TLS 1.2.

We may also ask whether TLS 1.3 clients and servers can be downgraded to TLS 1.2. If such a version downgrade takes place, we would end up with a TLS 1.2 session, so we need to state the query in terms of sessions where $mode_S$ contains TLS 1.2. ProVerif finds a version downgrade attack on a TLS 1.3 session, if the client and server support weak Diffie-Hellman groups in TLS 1.2. This attack closely mirrors the flaw described in [14]. Draft-13 introduced a

countermeasure in response to this attack, and we verify that by adding it to the model, the downgrade attack disappears.

Although our models of TLS 1.3 and 1.2 are individually verified in a few seconds each, their composition takes several minutes to analyze. As we add more features and modes to the protocol, ProVerif takes longer and requires more memory. Our final composite model for all modes of TLS 1.3+1.2 takes hours on a powerful workstation.

IV. 0-RTT with Semi-Static Diffie-Hellman

In earlier versions of TLS, the client would have to wait for two round-trips of handshake messages before sending its request. 1-RTT in TLS 1.3 brings this down to one round trip, but protocols like QUIC use a "zero-round-trip" (0-RTT) mode, by relying on a *semi-static* (long-term) Diffie-Hellman key. This design was adapted for TLS in the OPTLS proposal [52] and incorporated in Draft-7 (along with a fix we proposed, as described below).

Protocol Flow. The protocol is depicted in Figure 3. Each server maintains a Diffie-Hellman key pair (s, g^s) and publishes a signed server configuration containing g^s. As usual, a client initiates a connection with a `ClientHello` containing its ephemeral key g^x. If a client has already obtained and cached the server's certificate and signed configuration (in a prior exchange for example), then the client computes a shared secret g^{xs} and uses it to derive an initial set of shared keys which can then immediately be used to send encrypted data. To authenticate its 0-RTT data, the client may optionally send a certificate and a signature over the client's first flight.

The server then responds with a `ServerHello` message that contains a fresh ephemeral public key g^y. Now, the client and server can continue with a regular 1-RTT handshake using the new shared secret g^{xy} in addition to g^{xs}.

The 0-RTT protocol continued to evolve from Draft-7 to Draft-12, but in Draft-13, it was removed in favor of a PSK-based 0-RTT mode. Even though Diffie-Hellman-based 0-RTT no longer exists in Draft-18, we analyze its security in this section, both for posterity and to warn protocol designers about the problems they should watch our for if they decide to reintroduce DH-based 0-RTT in a future version of TLS.

Verification with ProVerif. We modeled the protocol in ProVerif and wrote queries to check whether the 0-RTT data m_0 is (forward) secret and authentic. ProVerif is able to prove secrecy but finds that m_0 is not forward secret if the semi-static key s is compromised once the session is over. ProVerif also finds a Key Compromise Impersonation attack on authentication: if g^s is compromised, then an attacker can forge 0-RTT messages from C to S. Furthermore, the 0-RTT flight can be replayed by an attacker and the server will process it multiple times, thinking that the client has initiated a new connection each time. In addition to these three concerns, which were documented in Draft-7, ProVerif also finds a new attack, explained below, that breaks 0-RTT authentication if the server's certificate is not included in the 0-RTT client signature.

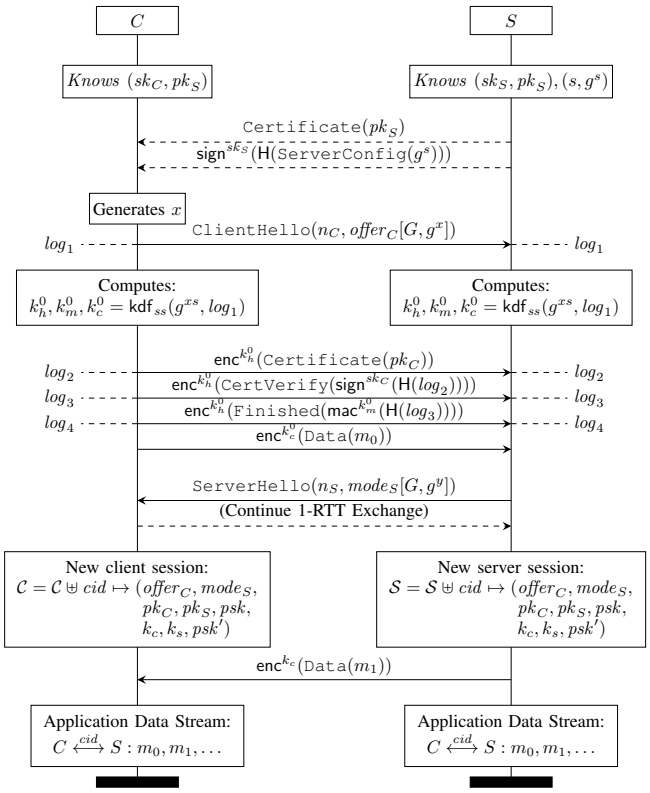


Figure 3: DH-based 0-RTT in TLS 1.3 Draft-12, inspired by QUIC and OPTLS.

Unknown Key Share Attack on DH-based 0-RTT in QUIC, OPTLS, and TLS 1.3. We observe that in the 0-RTT protocol, the client starts using g^s without having any proof that the server knows s. So a dishonest server M can claim to have the same semi-static key as S by signing g^s under its own key sk_M. Now, suppose a client connects to M and sends its client hello and 0-RTT data; M can simply forward this whole flight to S, which may accept it, because the semi-static keys match. This is an *unknown key share* (UKS) attack where C thinks it is talking to M but it is, in fact, connected to S.

In itself, the UKS attack is difficult to exploit, since M does not know g^{xs} and hence cannot decrypt or tamper with messages between C and S. However, if the client authenticates its 0-RTT flight with a certificate, then M can forward C's certificate (and C's signature) to S, resulting in a *credential forwarding* attack, which is much more serious. Suppose C is a browser that has a page open at website M; from this page M can trigger any authenticated 0-RTT HTTPS request m_0 to its own server, which then uses the credential forwarding attack to forward the request to S, who will process m_0 as if it came from C. For example, M may send a POST request that modifies C's account details at S.

The unknown key share attack described above applies to both QUIC and OPTLS, but remained undiscovered despite several security analyses of these protocols [42], [56], [52],

because these works did not consider client authentication, and hence did not formulate an authentication goal that exposed the flaw. We informed the authors of QUIC and they acknowledged our attack. They now recommend that users who need client authentication should not use QUIC, and should instead move over to TLS 1.3. We also informed the authors of the TLS 1.3 standard, and on our suggestion, Draft-7 of TLS 1.3 included a countermeasure for this attack: the client signature and 0-RTT key derivation include not just the handshake log but also the cached server certificate. With this countermeasure in place, ProVerif proves authentication for 0-RTT data.

V. Pre-Shared Keys for Resumption and 0-RTT

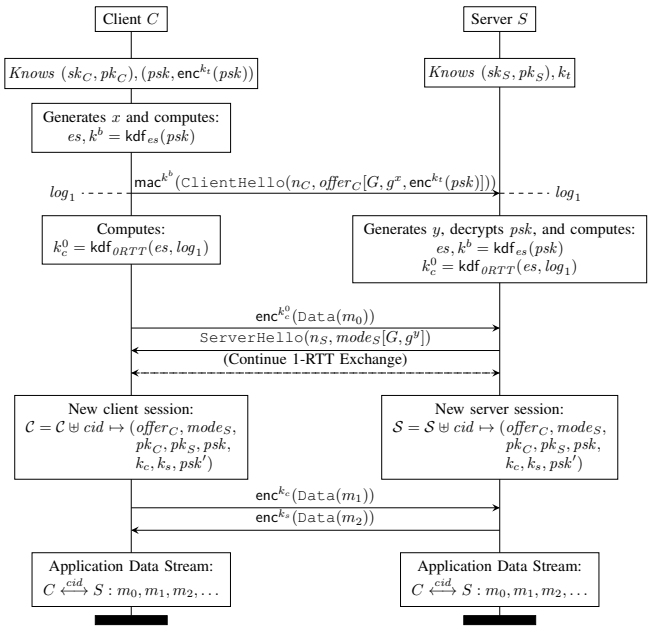


Figure 4: TLS 1.3 Draft-18 PSK-based Resumption and 0-RTT.

Aside from the number of round-trips, the main cryptographic cost of a TLS handshake is the use of public-key algorithms for signatures and Diffie-Hellman, which are still significantly slower than symmetric encryption and MACs. So, once a session has already been established between a client and server, it is tempting to reuse the symmetric session key established in this session as a pre-shared symmetric key in new connections. This mechanism is called *session resumption* in TLS 1.2 and is widely used in HTTPS where a single browser typically has many parallel and sequential connections to the same website. In TLS 1.2, pre-shared keys (PSKs) are also used instead of certificates by resource-constrained devices that cannot afford public-key encryption. TLS 1.3 combines both these use-cases in a single PSK-based handshake mode that combines resumption, PSK-only handshakes, and 0-RTT.

Protocol Flow. Figure 4 shows how this mode extends the regular 1-RTT handshake; in our analysis, we only consider PSKs that are established within TLS handshakes, but similar arguments apply to PSKs that are shared out-of-band. We assume that the client and server have established a pre-shared key psk in some earlier session. The client has cached psk, but in order to remain state-less, the server has given the client a ticket containing psk encrypted under an encryption key k_t. As usual, the client sends a `ClientHello` with its ephemeral key share g^x and indicates that it prefers to use the shared PSK psk. To prove its knowledge of psk and to avoid certain attacks (described below), it also MACs the `ClientHello` with a *binder* key k^b derived from the psk. The client can then use psk to already derive an encryption key for 0-RTT data m_0 and start sending data without waiting for the server's response. When the server receives the client's flight, it can choose to accept or reject the offered psk. Even if it accepts the psk, the server may choose to reject the 0-RTT data, it may choose to skip certificate-based authentication, and (if it does not care about forward secrecy) it may choose to skip the Diffie-Hellman exchange altogether. The recommended mode is PSK-DHE, where psk and g^{xy} are both mixed into the session keys. The server then sends back a `ServerHello` with its choice and the protocol proceeds with the appropriate 1-RTT handshake and completes the session.

Verifying PSK-based Resumption. We first model the PSK-DHE 1-RTT handshake (without certificate authentication) and verify that it still meets our usual security goals:

- **PSK-DHE 1-RTT (Forward) Secrecy** Any message m sent over a PSK-DHE session in 1-RTT is secret as long as the PSK psk and the ticket encryption key k_t are not compromised until the end of the session.
- **PSK-DHE 1-RTT Authentication and Replay Protection** Any message m received over a PSK-DHE session in 1-RTT corresponds to a unique message sent by a peer over a matching session (notably with the same psk) unless psk or k_t are compromised during the session.
- **PSK-DHE 1-RTT Unique Channel Identifier** The values psk', ems, and $\mathsf{H}(log_7)$ generated in a DHE or PSK-DHE session are all unique channel identifiers.

Notably, data sent over PSK-DHE is forward secret even if the server's long term ticket encryption key k_t is compromised after the session. In contrast, pure PSK handshakes do not provide this forward secrecy.

The authentication guarantee requires that the client and server must agree on the value of the PSK psk, and if this PSK was established in a prior session, then the unique channel identifier property says that the client and server must transitively agree on the prior session as well. An earlier analysis of Draft-10 in Tamarin [35] found a violation of the authentication goal because the 1-RTT client signature in Draft-10 did not include the server's `Finished` or any other value that was bound to the PSK. This flaw was fixed in Draft-11 and hence we are able to prove authentication for Draft-18.

Verifying PSK-based 0-RTT. We extend our model with the 0-RTT exchange and verify that m_0 is authentic and secret. The strongest queries that ProVerif can prove are the

following:

- **PSK-based 0-RTT (Forward) Secrecy** A message m_0 sent from C to S in a 0-RTT flight is secret as long as psk and k_t are never compromised.
- **PSK-based 0-RTT Authentication** A message m_0 received by S from C in a 0-RTT flight corresponds to some message sent by C with a matching `ClientHello` and matching psk, unless the psk or k_t are compromised.

In other words, PSK-based 0-RTT data is not forward secret and is vulnerable to replay attacks. As can be expected, it provides a symmetric authentication property: since both C and S know the psk, if either of them is compromised, the attacker can forge 0-RTT messages.

An Attack on 0-RTT Client Authentication. Up to Draft-12, the client could authenticate its 0-RTT data with a client certificate in addition to the PSK. This served the following use case: suppose a client and server establish an initial 1-RTT session (that outputs psk') where the client is unauthenticated. Some time later, the server asks the client to authenticate itself, and so they perform a PSK-DHE handshake (using psk') with client authentication. The use of psk' ensures continuity between the two sessions. In the new session, the client wants to start sending messages immediately, and so it would like to use client authentication in 0-RTT.

To be consistent with Draft-12, suppose we remove the outer binder MAC (using k^b) on the `ClientHello` in Figure 4, and we allow client authentication in 0-RTT. Then, if we model this protocol in ProVerif and ask the 0-RTT authentication query again, ProVerif finds a credential forwarding attack, explained next.

Suppose a client C shares psk with a malicious server M, and M shares a different psk' with an honest server S. If C sends an authenticated 0-RTT flight (certificate, signature, data m_0) to M, M can decrypt this flight using psk, re-encrypt it using psk', and forward the flight to S. S will accept the authenticated data m_0 from C as intended for itself, whereas C intended to send it only to M. In many HTTPS scenarios, as discussed in §IV, M may be able to control the contents of this data, so this attack allows M to send arbitrary requests authenticated by C to S.

This attack was not discovered in previous analyses of TLS 1.3 since many of them did not consider client authentication; the prior Tamarin analysis [35] found a similar attack on 1-RTT client authentication but did not consider 0-RTT client authentication. The attacks described here and in [35] belong to a general class of *compound authentication* vulnerabilities that appear in protocols that compose multiple authentication credentials [17]. In this case, the composition of interest is between PSK and certificate-based authentication. We found a similar attack on 1-RTT server authentication in pure PSK handshakes.

In response to our attack, Draft-13 included a `resumption_context` value derived from the psk in the handshake hash, to ensure that the client's signature over the hash cannot be forwarded on another connection (with a different psk'). This countermeasure has since evolved to the MAC-based design showed in Figure 4, which has now been verified in this paper.

The Impact of Replay on 0-RTT and 0.5-RTT. It is now widely accepted that asynchronous messaging protocols like 0-RTT cannot be easily protected from replay, since the recipient has no chance to provide a random nonce that can ensure freshness. QUIC attempted to standardize a replay-prevention mechanism but it has since abandoned this mechanism, since it cannot prevent attackers from forcing the client to resend 0-RTT data over 1-RTT [66].

Instead of preventing replays, TLS 1.3 Draft-18 advises applications that they should only send non-forward-secret and idempotent data over 0-RTT. This recommendation is hard to systematically enforce in flexible protocols like HTTPS, where all requests have secret cookies attached, and even GET requests routinely change state.

We argue that replays offer an important attack vector for 0-RTT and 0.5-RTT data. If the client authenticates its 0-RTT flight, then an attacker can replay the entire flight to mount *authenticated replay* attacks. Suppose the (client-authenticated) 0-RTT data asks the server to send a client's bank statement, and the server sends this data in a 0.5-RTT response. An attacker who observes the 0-RTT request once, can replay it any number of times to the server from anywhere in the world and the server will send it the user's (encrypted) bank statement. Although the attacker cannot complete the 1-RTT handshake or read this 0.5-RTT response, it may be able to learn a lot from this exchange, such as the length of the bank statement, and whether the client is logged in.

In response to these concerns, client authentication has now been removed from 0-RTT. However, we note that similar replay attacks apply to 0-RTT data that contains an authentication cookie or OAuth token. We highly recommend that TLS 1.3 servers should implement a replay cache (based on the client nonce n_C and the ticket age) to detect and reject replayed 0-RTT data. This is less practical in server farms, where time-based replay mitigation may be the only alternative.

VI. Computational Analysis of TLS 1.3 Draft-18

Our ProVerif analysis of TLS 1.3 Draft-18 identifies the necessary conditions under which the symbolic security guarantees of the protocol hold. We now use the tool CryptoVerif [24] to see whether these conditions are sufficient to obtain cryptographic security proofs for the protocol in a more precise computational model. In particular, under the assumption that the algorithms used in TLS 1.3 Draft-18 satisfy certain strong cryptographic assumptions, we prove that the protocol meets our security goals.

Proofs in the computational model are hard to mechanize, and CryptoVerif offers less flexibility and automation than ProVerif. To obtain manageable proofs, we focus only on TLS 1.3 (we do not consider TLS 1.2) and we ignore downgrade attacks. We split the protocol into three pieces and prove them separately using CryptoVerif, before composing them manually to obtain a proof for the full protocol.

A. Cryptographic Assumptions

We make the following assumptions about the cryptographic algorithms supported by TLS 1.3 clients and servers.

Diffie-Hellman. We assume that the Diffie-Hellman groups used in TLS 1.3 satisfy the gap Diffie-Hellman (GDH) assumption [61]. This assumption means that given g, g^a, and g^b for random a, b, the adversary has a negligible probability to compute g^{ab}, even when the adversary has access to a decisional Diffie-Hellman oracle, which tells him given G, X, Y, Z whether there exist x, y such that $X = G^x$, $Y = G^y$, and $Z = G^{xy}$.

In our proof, we require GDH rather than the weaker decisional Diffie-Hellman (DDH) assumption, in order to prove secrecy of keys on the server side as soon as the server sends its `Finished` message: at this point, if the adversary controls a certificate accepted by the client, he can send its own key share y' to the client to learn information on $g^{x'y'}$, and that would be forbidden under DDH. We also require that $x^y = x'^y$ implies $x = x'$ and that $x^y = x^{y'}$ implies $y = y'$, which holds when the considered Diffie-Hellman group is of prime order. This is true for all groups currently specified in TLS 1.3, and our proof requires it for all groups included in the future.

We also assume that all Diffie-Hellman group elements have a binary representation different from $0^{len_{\mathsf{H}()}}$. This assumption simplifies the proof by avoiding a possible confusion between handshakes with and without Diffie-Hellman exchange. Curve25519 does have a 32-byte zero element, but excluding zero Diffie-Hellman shared values is already recommended to avoid points of small order [54].

Finally, we assume that all Diffie-Hellman group elements have a binary representation different from $len_{\mathsf{H}()} \| \text{“TLS 1.3,”} \| l \| h \| \mathtt{0x01}$. This helps ease our proofs by avoiding a collision between $\mathsf{hkdf\text{-}extract}(es, e)$ and $\mathsf{derive\text{-}secret}(es, \mathsf{pbk}, \text{“”})$ or $\mathsf{derive\text{-}secret}(es, \mathsf{ets}_c, log_1)$. This assumption holds with the currently specified groups and labels, since group elements have a different length than the bitstring above. The technical problem identified by our assumption was independently discovered and discussed on the TLS mailing list [67], and has led to a change in Draft-19 which will make this assumption unnecessary.

Signatures. We assume that the function sign is unforgeable under chosen-message attacks (UF-CMA) [43]. This means that an adversary with access to a signature oracle has a negligible probability of forging a signature for a message not signed by the signature oracle. Only the oracle has access to the signing key; the adversary has the public key.

Hash Functions. We assume that the function H is collision-resistant [36]: the adversary has a negligible probability of finding two different messages with the same hash.

HMAC. We need two assumptions on HMAC-H:

We require that the functions $x \mapsto \mathsf{HMAC\text{-}H}^{0^{len_{\mathsf{H}()}}}(x)$ and $x \mapsto \mathsf{HMAC\text{-}H}^{\mathsf{kdf}_0}(x)$ are independent random oracles, in order to justify the use of HMAC-H as a randomness extractor in the HKDF construct. This assumption can itself be justified as follows. Assuming that the compression function underlying the hash function is a random oracle, Theorem 4.4 in [38] shows that HMAC is indifferentiable [33] from a random oracle, provided the MAC keys are less than the block size of the hash function minus one, which is true for HMAC-SHA-256, HMAC-SHA-384, and HMAC-SHA-512. It is then easy to show that $x \mapsto \mathsf{HMAC\text{-}H}^{0^{len_{\mathsf{H}()}}}(x)$ and $x \mapsto \mathsf{HMAC\text{-}H}^{\mathsf{kdf}_0}(x)$ are indifferentiable from independent random oracles in this case.

We assume that HMAC-H is a pseudo-random function (PRF) [9], that is, HMAC-H is indistinguishable from a random function provided its key is random and used only in HMAC-H, when the key is different from $0^{len_{\mathsf{H}()}}$ and kdf_0. We avoid these two keys to avoid confusion with the two random oracles above. Since keys are chosen randomly with uniform probability from a set key (with cardinality $|\mathsf{key}|$), the only consequence of avoiding these keys is that $\frac{2}{|\mathsf{key}|}$ is added to the probability of breaking the PRF assumption.

Authenticated Encryption. The authenticated encryption scheme is IND-CPA (indistinguishable under chosen plaintext attacks) and INT-CTXT (ciphertext integrity) [11], provided the same nonce is never used twice with the same key. IND-CPA means that the adversary has a negligible probability of distinguishing encryptions of two distinct messages of the same length that it has chosen. INT-CTXT means that an adversary with access to encryption and decryption oracles has a negligible probability of forging a ciphertext that decrypts successfully and has not been returned by the encryption oracle.

B. Verifying 1-RTT Handshakes without Pre-Shared Keys

To prove the security of TLS 1.3 in CryptoVerif, we first establish some lemmas about the primitives, as detailed in Appendix A. Then, we split the protocol into three parts, as shown in Figure 5, and verify them in sequence, before composing them by hand into a proof for the full protocol. This modular hybrid approach allows us to have proofs of manageable complexity, and to obtain results even when keys are reused many times, such as when several PSK-based resumptions are performed, which would otherwise be out of scope of CryptoVerif.

We first consider the initial 1-RTT handshake shown in Figure 2, until the new client and server session boxes. We model a honest client and a honest server, which are willing to interact with each other, but also with dishonest clients and servers included in the adversary. We do not consider details of the negotiation (or the `RetryRequest` message). We give the handshake keys (k_c^h and k_s^h) to the adversary, and let it encrypt and decrypt the handshake messages, so our security proof does not rely on the encryption of the handshake.

We assume that the server is always authenticated and consider both the handshake with and without client authentication. The honest client and server may be compromised at any time: the secret key of the compromised participant is then sent to the adversary, and the compromise is recorded by defining a variable `corruptedClient` or `corruptedServer`.

The outputs of this protocol are the application traffic secrets ats_c and ats_s (the derivation of the keys k_c and k_s from these secrets is left for the record protocol), the exporter master secret ems, and the resumption master secret psk' (later used as pre-shared key). CryptoVerif proves the following properties:

- **Key Authentication:** If the client terminates a session with the server and the server is not compromised, then the server has accepted a session with the client, and they share the same parameters: the keys ats_c, ats_s, and ems and all messages sent in the protocol until the server `Finished` message. (We can make no claim on the client `Finished` message because it has not been received by the server at this point, nor on psk' because it depends on the client `Finished` message.)
  In our CryptoVerif model, we formalize this property by adding an event $\mathsf{ClientTerm}(\dots)$ in the client, executed when the client terminates a session (that is, sends his `Finished` message) with an honest server (that is, `corruptedServer` is not defined). We similarly define an event $\mathsf{ServerAccept}(\dots)$ at the server, executed when the server accepts a session (that is, sends his `Finished` message). The arguments of these events include the session keys and all the messages sent in the protocol until the server `Finished` message. We then ask CryptoVerif or prove an authentication query that states that, with overwhelming probability, each execution of event ClientTerm corresponds to a distinct execution of event ServerAccept with the same arguments.
  Conversely, if a server terminates a session with an honest client, and either the client is authenticated and not compromised, or the client key share $g^{x'}$ accepted by the server was generated by the client, then the client must have accepted a session with the server, and they must agree on the established keys and on all messages sent in the protocol. We state this property as a CryptoVerif query and verify it.
- **Replay Prevention:** The authentication properties stated above are already injective, that is, they guarantee that each session of the client (resp. server) corresponds to a distinct session of the server (resp. client), and consequently, they forbid replay attacks.
- **(Forward) Secrecy of Keys:** The keys ats_c, ats_s, ems, and psk' exchanged in several protocol sessions are indistinguishable from independent fresh random values. This property means for instance that the keys psk' remains secret (indistinguishable from independent fresh random values) even if ats_c, ats_s, ems are given to the adversary, and similarly for the other keys. Secrecy holds on the client side when the server is not compromised before the end of the session. It holds on the server side when the client is authenticated and not compromised before the end of the session or when the key share $g^{x'}$ used by the server comes from the client. We prove secrecy of ats_c, ats_s, and ems on the server side when the key share $g^{x'}$ comes from the client as soon as the server sends its `Finished` message. This property allows us to prove security of 0.5-RTT messages by composition with the record protocol.
- **Unique Channel Identifier:** When cid is psk' or $\mathsf{H}(log_7)$, we do not use CryptoVerif as the result is immediate: if a client session and a server session have the same cid, then these sessions have the same log_7 by collision-resistance of H (which implies collision-resistance of HMAC-H), so all their parameters are equal.
  When cid is ems, collision-resistance just yields that the client and server sessions have the same log_4. CryptoVerif proves that, if a client session and a server session both terminate successfully with the same log_4, then they have the same log_7 and the same keys, so all their parameters are equal.

We need to guide CryptoVerif in order to prove these properties, with the following main steps. We first apply the security of the signature under the server key sk_S. We introduce tests to distinguish cases, depending on whether the Diffie-Hellman share received by the server is a share $g^{x'}$ from the client, and whether the Diffie-Hellman share received by the client is the share g^y generated by the server upon receipt of $g^{x'}$. Then we apply the random oracle assumption on $x \mapsto \mathsf{HMAC\text{-}H}^{\mathsf{kdf}_0}(x)$, replace variables that contain $g^{x'y}$ with their values to make equality tests $m = g^{x'y}$ appear, and apply the gap Diffie-Hellman assumption. At this point, the handshake secret hs is a fresh random value. We use the properties on the key schedule established in Appendix A to show that the other keys are fresh random values, and apply the security of the MAC and of the signature under the client key sk_C.

C. Verifying Handshakes with Pre-Shared Keys

We now analyze the handshake protocol in Figure 4, up until the new client and server sessions are established. The protocol begins with 0-RTT and continues on to 1-RTT. We consider both variants of PSK-based 1-RTT, with and without Diffie-Hellman exchange.

We ignore the ticket $\mathsf{enc}^{k_t}(psk)$ and consider a honest client and a honest server that initially share the pre-shared key psk. Dishonest clients and servers may be included in the adversary. As in the previous section, we give the handshake keys (k_c^h and k_s^h) to the adversary and ignore handshake encryption. Certificates for the client and server are optional, since they are already authenticated via the psk; we do not rely on authentication in our proofs and consider that the adversary performs the signature and verification operations on certificates if they occur.

The outputs of this protocol are the client early traffic secret ets_c (the derivation of the key k_c from ets_c is left for the record protocol), the application traffic secrets ats_c and ats_s, the exporter master secret ems, and the resumption master secret psk'. We run CryptoVerif on our model to obtain the following verification results:

- **Key Authentication:** CryptoVerif shows the same authentication properties as for the handshake without pre-shared key, assuming that both participants are uncompromised. Notably, however, CryptoVerif cannot prove authentication of ets_c. While the binder $\mathsf{mac}^{k_b}(\dots)$ authenticates most of the client `ClientHello` message, the client

may offer several pre-shared keys and send a binder for each of these keys. Only the binder for the pre-shared key selected by the server is checked. Hence the adversary may alter another of the proposed binders, yielding a different log_1 and a different ets_c on the server side. This is not a serious attack, as the record protocol will fail if ets_c does not match on the client and server sides.

- **Replay Prevention:** CryptoVerif proves that all the authentication properties shown above are injective, thus showing the absence of replays for ats_c, ats_s, and ems. However, CryptoVerif cannot prove replay protection for the 0-RTT session key ets_c, and indeed the client `ClientHello` message can be replayed, yielding the same key ets_c for several sessions of the server even though there is a single session of the client.
- **Secrecy of Keys:** The keys ets_c, ats_c, ats_s, ems, and psk' exchanged in several protocol sessions are indistinguishable from independent fresh random values. Secrecy holds both on the client side and on the server side except that, on the server side, the keys ets_c are not independent of each other since an adversary may force the server to accept several times the same key ets_c by replaying the client `ClientHello` message. We prove the secrecy of ats_c, ats_s, and ems on the server side as soon as the server sends its `Finished` message.
- **Forward Secrecy:** CryptoVerif is unable to prove secrecy of the keys when psk is compromised after the end of the session, even assuming that hkdf-extract is a random oracle. Secrecy obviously does not hold in this case for the handshake without Diffie-Hellman exchange. We believe that it still holds for the handshake with Diffie-Hellman exchange; our failure to prove it in this case is due to the current limitations of CryptoVerif.
- **Unique Channel Identifier:** We proceed as in the handshake without pre-shared key. We additionally notice that, if a client session and a server session have the same log_7, then they have the same psk. Indeed, by collision-resistance of mac = HMAC-H, they have the same k^b, so the same es, so the same psk.

D. Verifying the Record Protocol

The third component of TLS 1.3 is the record protocol that encrypts and decrypts messages after the new client and server sessions have been established in Figures 2 and 4.

In our model, we assume that the client and server share a fresh random traffic secret. We generate an encryption key and an initialization vector (IV), and send and receive encrypted messages using those key and IV, and a counter that is distinct for each message. (Our model is more detailed than the symbolic presentation given in the figures as we consider the IV and the counter.) We also generate a new traffic secret as specified in the key update mechanism of TLS 1.3 Draft-18 (Section 7.2). CryptoVerif proves the following properties automatically:

- **Key and Message Secrecy:** CryptoVerif proves that the updated traffic secret is indistinguishable from a fresh random value. It also proves that, when the adversary provides two sets of plaintexts m_i and m'_i of the same padded length, it is unable to determine which of two sets is encrypted, even when the updated traffic secret is leaked.
- **Message Authentication:** CryptoVerif proves that, if a message m is decrypted by the receiver with a counter c, then the message m has been encrypted and sent by an honest sender with the same counter c.
- **Replay Prevention:** The authentication property above is injective, that is, any sent application data may be accepted at most once by the receiver.

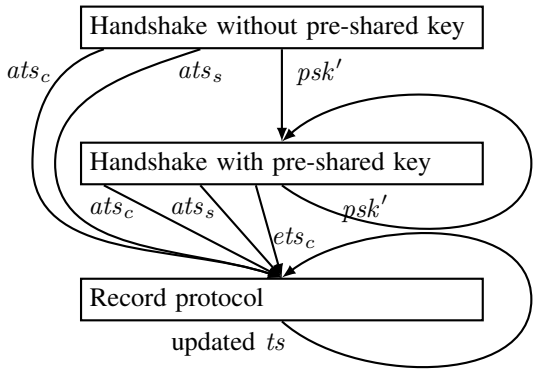


Figure 5: Structure of the CryptoVerif proof

E. A Composite Proof for TLS 1.3 Draft-18

We compose these results using a hybrid argument (as in [40]). Figure 5 summarizes the structure of the composition; more details are given in the full version [13].

First, we use the secrecy property of the initial handshake to replace all session keys with independent fresh random values. We rely on authentication and replay prevention to show that the same replacement is performed in matching sessions of the client and server.

Then, we use the security properties of the record protocol using ats_c and ats_s as traffic secrets, to obtain secrecy, forward secrecy (with respect to the compromise of sk_S and sk_C), authentication, and replay prevention for application messages in both directions. The security of the record protocol also shows that the updated traffic secrets generated during subsequent key updates preserve these properties.

Using the key psk' provided by the initial handshake, we then apply the security of the PSK-based handshake, to obtain that the keys ets_c, ats_c, ats_s, and psk' provided by this handshake are independent fresh random values. (The forward secrecy property of the initial handshake allows us to leak the keys sk_S and sk_C, so that the adversary can indeed perform the signature operations related to certificates, as we assumed in our model of handshakes with pre-shared keys.) We then apply the security of the record protocol to ats_c and ats_s, as above, for 1-RTT messages. We also apply it to ets_c for 0-RTT messages, but since the handshake does not prevent replays for this key, the composition will not prevent replays for messages sent under this key.

Finally, we apply the security of the PSK-based handshake again to the newly obtained psk', hence obtaining composite security for arbitrary sequences of PSK-based resumptions.

VII. RefTLS: A Reference TLS 1.3 Implementation with a Verified Protocol Core

In today's web ecosystem, TLS is used by wide variety of client and server applications to establish secure channels across the Internet. For example, Node.js servers are written in JavaScript and can accept HTTPS connections using a Node's builtin `https` module that calls OpenSSL. Popular desktop applications, such as WhatsApp messenger, are also written in JavaScript using the Electron framework (which combines Node.js with the Chromium rendering engine); they connect to servers using the same `https` module.

Our goal is to develop a high-assurance reference implementation of TLS 1.3, called RefTLS, that can be seamlessly used by Electron apps and Node.js servers. We want our implementation to be small, easy to read and analyze, and effective as an early experimental version of TLS 1.3 that real-world applications can use to help them transition to TLS 1.3, before it becomes available in mainstream libraries like OpenSSL. Crucially, we want to be able to verify the security of the core protocol code in RefTLS, and show that it avoids both protocol-level attacks as well as implementation bugs in its protocol state machine [12].

In this section, we describe RefTLS and evaluate its progress towards these goals. RefTLS has been used as a prototype implementation of TLS Draft-13 to Draft-18, interoperating with other early TLS 1.3 libraries. Its protocol core has been symbolically analyzed with ProVerif, and it has been successfully integrated into Electron applications.

Flow and ProScript. RefTLS is written in Flow [32], a typed variant of JavaScript. Static typing in Flow guarantees the absence of a large class of classic JavaScript bugs, such as reading a missing field in an object. Consequently, our code looks very much like a program in a typed functional language like OCaml or F#. We would like to verify the security of all our Flow code, but since Flow is a fully-fledged programming language, it has loops, mutable state, and many other features that are hard to automatically verify.

In earlier work, we developed a typed subset of JavaScript called ProScript [48] that was designed for writing cryptographic protocol code that could be compiled automatically to ProVerif. ProScript is also a subset of Flow and so we can reuse its ProVerif compiler to extract symbolic models from the core protocol code in RefTLS, if we write it carefully.

ProScript code is written defensively, in that it cannot, even accidentally, access external libraries or extensible JavaScript functionalities such as object instantiation, or redefinable properties such as Array.split. These restrictions are necessary in JavaScript where external functions can completely redefine the behavior of all libraries and object prototypes. The resulting style enforces syntactic scoping and strict type checking for all variables and functions, and disallows implicit coercions, object prototype access, and dynamic extensions of arrays and objects.

For ease of analysis, ProScript disallows loops, recursion, and only allows access mutable state through a well defined `table` interface. These are significant restrictions, but as we show, the resulting language is still expressive enough to write the core composite protocol code for TLS 1.0-1.3.

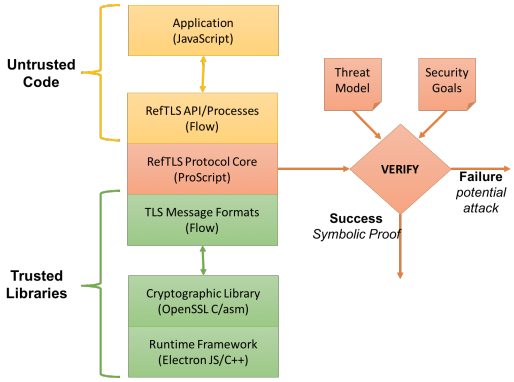


Figure 6: RefTLS Architecture. The library is written in Flow, a typed subset of JavaScript. The protocol core is verified by translation to ProVerif. The cryptographic library, message formatting and parsing, and the runtime framework are trusted. The application and parts of the RefTLS library are untrusted (assumed to be adversarial in our model).

Implementation Structure. Figure 6 depicts the architecture of RefTLS and shows how it can be safely integrated into larger, unverified and untrusted applications. At the top, we have Node.js and Electron applications written in JavaScript. RefTLS exposes an interface to these applications that exactly matches that of the default Node.js `https` module (which uses OpenSSL), allowing these applications to transparently use RefTLS instead of OpenSSL.

The RefTLS code itself is divided into untrusted Flow code that handles network connections and implements the API, a verified protocol module, written in ProScript, and some trusted but unverified Flow code for parsing and serializing TLS messages. All this code is statically typechecked in Flow. The core protocol module, called RefTLS-CORE, implements all the cryptographic operations of the protocol. It exposes an interface that allows RefTLS to drive the protocol, but hides all keying material and sensitive session state within the core module. This isolation is currently implemented via the Node module system; but we can also exploit Electron's multi-threading feature in order to provide thread-based isolation to the RefTLS-CORE module, allowing it to only be accessed through a pre-defined RPC interface. Strong isolation for RefTLS-CORE allows us to verify it without relying on the correctness of the rest of the RefTLS codebase.

However, RefTLS still relies on the security and correctness of the crypto library and the underlying Electron, Node.js, and JavaScript runtimes. In the future, we may be able to reduce this trusted computing base by relying on verified crypto [73], verified JavaScript interpreters [29], and least-privilege architectures, such as ESpectro [69], which can control access to dangerous libraries from JavaScript.

A Verified Protocol Core. In RefTLS-CORE, we develop, implement and verify (for the first time) a composite state machine for TLS 1.2 and 1.3 (shown in Appendix B). Each state transition is implemented by a ProScript function that processes a flight of incoming messages, changes the session state, and produces a flight of outgoing messages. For

TLS 1.3 clients, these functions are `get_client_hello`, `put_server_hello`, and `put_server_finished`; servers use the functions `put_client_hello`, `get_server_finished`, and `put_client_finished`.

We then use the ProScript compiler to translate this module into a ProVerif script that looks much like the protocol models described in earlier sections of this paper. (See [48] for details of the translation.) Each pure function in ProScript translates to a ProVerif function; functions that modify mutable state are translated to ProVerif processes that read and write from tables. The interface of the module is compiled to a top-level process that exposes a subset of the protocol functions to the adversary over a public channel.

The adversary can call these functions in any order and any number of times, to initiate connections in parallel, to provide incoming flights of messages, and to obtain outgoing flights of messages. The ProVerif model uses internal tables, not accessible to the attacker, to manage state updates between flights and preserve state invariants through the protocol execution.

Our approach allows us to quickly obtain verifiable ProVerif models from running RefTLS code. For example, we were able to rapidly prototype changes to the TLS 1.3 specification between Draft-13 and Draft-18, while testing for interoperability and analyzing the core protocol at the same time. In particular, we extracted a model from our Draft-18 implementation, and verified our security goals from §III and §V with ProVerif.

We engineered the ProScript compiler to generate readable ProVerif models that can be modified by a protocol analyst to experiment with different threat models. We are working towards applying the same automated translation approach towards CryptoVerif models. CryptoVerif syntax differs slightly from the ProVerif syntax, yet there is ongoing work in the CryptoVerif team to have it accept the same source syntax as ProVerif. However, the kind of models that are easy to verify using CryptoVerif differ from the models that ProVerif can automatically verify, and the assumptions on cryptographic primitives will always remain different. Therefore, even if the source syntax is the same, we may need to adapt our compiler to generate different models for ProVerif and CryptoVerif.

Evaluation: Verification, Interoperability, Performance. The full RefTLS codebase consists of about 6500 lines of Flow code, including 3000 lines of trusted libraries (mostly message parsing), 2500 lines of untrusted application code, and 1000 lines of verified protocol core. From the core, we extracted an 800 line protocol model in ProVerif and composed it with our generic library from §II. Verifying this model took several hours on a powerful workstation.

RefTLS implements TLS 1.0-1.3, and interoperates with all major TLS libraries for TLS 1.0-1.2. Fewer libraries currently implement TLS 1.3, but RefTLS participated in the IETF Hackathon and achieved interoperability with other implementations of Draft-14. It now interoperates with NSS (Firefox) and BoringSSL (Chrome) for Draft-18.

By implementing Node's `https` interface, we are able to naturally integrate RefTLS within any Node or Electron application. We demonstrate the utility of this approach by integrating RefTLS into the Brave web browser, which is written in Electron. We are able to intercept all of Brave's HTTPS requests and reliably fulfill them through RefTLS.

We benchmarked RefTLS against Node.js's default OpenSSL-based HTTPS stack when run against an OpenSSL peer over TLS 1.2. In terms of computational overhead, RefTLS is two times slower than Node's native library, which is not surprising since RefTLS is written in JavaScript, whereas OpenSSL is written in C. In exchange for speed, RefTLS offers an early implementation of TLS 1.3 and a verified protocol core. Furthermore, in many application scenarios, network latency dominates over crypto, so the performance penalty of RefTLS may not be that noticeable.

VIII. Discussion and Related Work

Symbolic Analysis of TLS 1.3. We symbolically analyzed a composite model of TLS 1.3 Draft-18 with optional client authentication, PSK-based resumption, and PSK-based 0-RTT, running alongside TLS 1.2 against a rich threat model, and we established a series of security goals. In summary, 1-RTT provides forward secrecy, authentication and unique channel identifiers, 0.5-RTT offers weaker authentication, and 0-RTT lacks forward secrecy and replay protection.

We discovered potential vulnerabilities in 0-RTT client authentication in earlier draft versions. These attacks were presented at the TLS Ready-Or-Not (TRON) workshop and contributed to the removal of certificate-based 0-RTT client authentication from TLS 1.3. The current design of PSK binders in Draft-18 is also partly inspired by these kinds of authentication attacks.

TLS 1.3 has been symbolically analyzed before, using the Tamarin prover [35]. ProVerif and Tamarin are both state-of-the-art protocol analyzers with different strengths. Tamarin can verify arbitrary compositions of protocols by relying on user-provided lemmas, whereas ProVerif is less expressive but offers more automation. In terms of protocol features, the Tamarin analysis covered PSK and ECDHE handshakes for 0-RTT and 1-RTT in Draft-10, but did not consider 0-RTT client certificate authentication or 0.5-RTT data. On the other hand, they do consider delayed (post-handshake) authentication, which we did not consider here.

The main qualitative improvement in our verification results over theirs is that we consider a richer threat model that allows for downgrade attacks, and that we analyze TLS 1.3 in composition with previous versions of the protocol, whereas they verify TLS 1.3 in isolation.

Our full ProVerif development consists of 1030 lines of ProVerif; including a generic library incorporating our threat model (400 lines), processes for TLS 1.2 (200 lines) and TLS 1.3 (250 lines), and security queries for TLS 1.2 (50 lines) and TLS 1.3 (180 lines). All proofs complete in about 70 minutes on a powerful workstation. In terms of manual effort, these models took about 3 person-weeks for a ProVerif expert.

Computational Proofs for TLS 1.3. We presented the first mechanically-checked cryptographic proof for TLS 1.3, developed using the CryptoVerif prover. We prove secrecy,

forward secrecy with respect to the compromise of long-term keys, authentication, replay prevention (except for 0-RTT data), and existence of a unique channel identifier for TLS 1.3 draft-18. Our analysis considers PSK modes with and without DHE key exchange, with and without client authentication. It includes 0-RTT and 0.5-RTT data, as well as key updates, but not post-handshake authentication.

Unlike the ProVerif analysis, our CryptoVerif model does not consider compositions of client certificates and pre-shared keys in the same handshake. It also does not account for version or ciphersuite negotiation; instead, we assume that the client and server only support TLS 1.3 with strong cryptographic algorithms. The reason we limit the model in this way is to make the proofs more tractable, since CryptoVerif is not fully automated and requires significant input from the user. With future improvements in the tool, we may be able to remove some of these restrictions.

CryptoVerif is better suited to proofs than finding attacks. Sometimes, proof failures in CryptoVerif might lead us towards computational attacks that do not appear at the symbolic level, but we did not find such attacks in our model of TLS 1.3. We failed to prove forward secrecy for handshakes that use both pre-shared keys and Diffie-Hellman, but this failure is due to limitations in our tool, not due to an attack. Our proofs required some unusual assumptions on public values in Diffie-Hellman groups to avoid confusions between different key exchange modes; these ambiguities are inherent in Draft-18 but have been fixed in Draft-19, making some of our assumptions unnecessary.

In comparison with previous cryptographic proofs of draft versions of TLS 1.3 [40], [52], [55], our cryptographic assumptions and proof structure is similar. The main difference in this work is that our proof is mechanized, so we can easily adapt and recheck our proofs as the protocol evolves.

Our full CryptoVerif development consists of 1895 lines, including new definitions and lemmas for the key schedule (570 lines), a model of the initial handshake (550 lines), a model of PSK-based handshakes (625 lines), and a model of the record protocol (150 lines). For different proofs, we sometimes wrote small variations of these files, and we do not count all those variations here. All proofs completed in about 6 minutes. The total verification effort took about 5 person-weeks for a CryptoVerif expert.

Verifying TLS Implementations. Specifications for protocols like TLS are primarily focused on interoperability; the RFC standard precisely defines message formats, cryptographic computations, and expected message sequences. However, it says little about what state machine these protocol implementations should use, or what APIs they should offer to their applications. This specification ambiguity is arguably the culprit for many implementation bugs [12] and protocol flaws [15] in TLS.

In the absence of a more explicit specification, we advocate the need for verified reference implementations of TLS that can provide exemplary code and design patterns on how to deploy the protocol securely. We proposed one such implementation, RefTLS, for use in JavaScript applications. The core protocol code in RefTLS implements both TLS 1.2 and 1.3 and has been verified using ProVerif. However, RefTLS is a work-in-progress and many of its trusted components remain to verified. For example, we did not verify our message parsing code or cryptographic libraries, and our verification results rely on the correctness of the unverified ProScript-to-ProVerif compiler [48].

The symbolic security guarantees of RefTLS are weaker than those of computationally-verified implementations like miTLS [21]. However, unlike miTLS, our analysis is fully automated and it can quickly find attacks. The type-based technique of miTLS requires significant user intervention and is better suited to building proofs than finding attacks.

Other Verification Approaches. In addition to ProVerif and CryptoVerif, there are many symbolic and computational analysis tools that have been used to verify cryptographic protocols like TLS. As discussed above, Tamarin [68] was used to symbolically analyze TLS 1.3 Draft-10 [35]. EasyCrypt [8] has been used to develop cryptographic proofs for various components used in TLS, including the MAC-Encode-Encrypt construction used in the record layer [5].

Our ProScript-to-ProVerif compiler is inspired by previous works on deriving ProVerif models from F# [20], Java [6], and JavaScript [16]. Such translations have been used to symbolically and computationally analyze TLS implementations [18]. An alternative to model extraction is to synthesize a verified implementation from a verified model; [30] shows how to compile CryptoVerif models to OCaml and uses it to derive a verified SSH implementation.

The most advanced case studies for verified protocol implementations use dependent type systems, because they scale well to large codebases. Refinement types for F# have been used to prove both symbolic [19] and cryptographic security properties, with applications to TLS [21]. The F* programming language [70] has been used to verify small protocols and cryptographic libraries [73]. Similar techniques have been applied to the cryptographic verification of Java programs [53].

IX. Conclusion and Future Work

TLS 1.3 is a social and technical experiment in the collaborative design of a practical protocol with regular input and review from the academic research community. It seeks to reverse the traditional pattern where security analyses are performed several years after standardization, when it may be too late to change how implementations work. This paper describes our contribution to this standardization effort.

We present verification results for symbolic models in ProVerif, computational models in CryptoVerif, and a reference implementation in JavaScript of TLS 1.3 Draft-18. There are still many features and aspects of the emerging protocol standard that remain to be analyzed. Furthermore, the formal connections between our ProVerif models, CryptoVerif proofs, and JavaScript code are not as strong as could be desired. We have focused on proof automation and readable models as a pragmatic first step, but we are working on formal proofs of correctness for our translations from Flow to ProVerif and CryptoVerif, so that we can obtain strong guarantees for our protocol source code.

REFERENCES

[1] D. Adrian, K. Bhargavan, Z. Durumeric, P. Gaudry, M. Green, J. A. Halderman, N. Heninger, D. Springall, E. Thomé, L. Valenta *et al.*, "Imperfect forward secrecy: How Diffie-Hellman fails in practice," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015, pp. 5–17.

[2] M. R. Albrecht and K. G. Paterson, "Lucky microseconds: A timing attack on Amazon's S2N implementation of TLS," in *EUROCRYPT*, 2016, pp. 622–643.

[3] N. AlFardan, D. J. Bernstein, K. G. Paterson, B. Poettering, and J. C. Schuldt, "On the security of RC4 in TLS," in *USENIX Security Symposium*, 2013, pp. 305–320.

[4] N. J. AlFardan and K. G. Paterson, "Lucky thirteen: Breaking the TLS and DTLS record protocols," in *2013 IEEE Symposium on Security and Privacy (SP 2013)*, 2013, pp. 526–540.

[5] J. B. Almeida, M. Barbosa, G. Barthe, and F. Dupressoir, "Verifiable Side-Channel Security of Cryptographic Implementations: Constant-Time MEE-CBC," in *Fast Software Encryption (FSE)*, 2016, pp. 163–184.

[6] M. Avalle, A. Pironti, R. Sisto, and D. Pozza, "The Java SPI framework for security protocol implementation," in *Availability, Reliability and Security (ARES), 2011 Sixth International Conference on*, Aug 2011, pp. 746–751.

[7] N. Aviram, S. Schinzel, J. Somorovsky, N. Heninger, M. Dankel, J. Steube, L. Valenta, D. Adrian, J. A. Halderman, V. Dukhovni, E. Käsper, S. Cohney, S. Engels, C. Paar, and Y. Shavitt, "DROWN: breaking TLS using SSLv2," in *USENIX Security Symposium*, 2016, pp. 689–706.

[8] G. Barthe, F. Dupressoir, B. Grégoire, C. Kunz, B. Schmidt, and P.-Y. Strub, "EasyCrypt: A tutorial," in *Foundations of Security Analysis and Design VII (FOSAD)*, ser. Lecture Notes in Computer Science. Springer, 2014, vol. 8604, pp. 146–166.

[9] M. Bellare, "New proofs for NMAC and HMAC: Security without collision-resistance," in *Advances in Cryptology (CRYPTO)*, 2006, pp. 602–619.

[10] M. Bellare, J. Kilian, and P. Rogaway, "The security of the cipher block chaining message authentication code," *Journal of Computer and System Sciences*, vol. 61, no. 3, pp. 362–399, Dec. 2000.

[11] M. Bellare and C. Namprempre, "Authenticated encryption: Relations among notions and analysis of the generic composition paradigm," in *Advances in Cryptology – ASIACRYPT'00*, 2000, pp. 531–545.

[12] B. Beurdouche, K. Bhargavan, A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, A. Pironti, P.-Y. Strub, and J. K. Zinzindohoue, "A messy state of the union: taming the composite state machines of TLS," in *IEEE Symposium on Security & Privacy (Oakland)*, 2015.

[13] K. Bhargavan, B. Blanchet, and N. Kobeissi, "Verified models and reference implementations for the TLS 1.3 standard candidate," Inria, Research report RR-9040, 2017.

[14] K. Bhargavan, C. Brzuska, C. Fournet, M. Green, M. Kohlweiss, and S. Z. Béguelin, "Downgrade resilience in key-exchange protocols," in *IEEE Symposium on Security and Privacy (Oakland)*, 2016, pp. 506–525.

[15] K. Bhargavan, A. Delignat-Lavaud, C. Fournet, A. Pironti, and P.-Y. Strub, "Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS," in *IEEE Symposium on Security & Privacy (Oakland)*, 2014, pp. 98–113.

[16] K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis, "Language-based defenses against untrusted browser origins," in *USENIX Security Symposium*, 2013, pp. 653–670.

[17] K. Bhargavan, A. Delignat-Lavaud, and A. Pironti, "Verified contributive channel bindings for compound authentication," in *Network and Distributed System Security Symposium (NDSS '15)*, 2015.

[18] K. Bhargavan, C. Fournet, R. Corin, and E. Zălinescu, "Verified cryptographic implementations for TLS," *ACM TOPLAS*, vol. 15, no. 1, pp. 3:1–3:32, 2012.

[19] K. Bhargavan, C. Fournet, and A. D. Gordon, "Modular verification of security protocol code by typing," in *ACM Symposium on Principles of Programming Languages (POPL)*, 2010, pp. 445–456.

[20] K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse, "Verified interoperable implementations of security protocols," *ACM Transactions on Programming Languages and Systems*, vol. 31, no. 1, 2008.

[21] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P.-Y. Strub, "Implementing TLS with verified cryptographic security," in *IEEE Symposium on Security & Privacy (Oakland)*, 2013. [Online]. Available: pubs/implementing-tls-with-verified-cryptographic-security-sp13.pdf

[22] K. Bhargavan and G. Leurent, "On the practical (in-)security of 64-bit block ciphers: Collision attacks on HTTP over TLS and OpenVPN," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016, pp. 456–467.

[23] ——, "Transcript collision attacks: Breaking authentication in TLS, IKE, and SSH," in *ISOC Network and Distributed System Security Symposium (NDSS)*, 2016.

[24] B. Blanchet, "A computationally sound mechanized prover for security protocols," *IEEE Transactions on Dependable and Secure Computing*, vol. 5, no. 4, pp. 193–207, 2008.

[25] ——, "Automatic verification of correspondences for security protocols," *Journal of Computer Security*, vol. 17, no. 4, pp. 363–434, 2009.

[26] ——, "Security protocol verification: Symbolic and computational models," in *Principles of Security and Trust (POST)*, 2012, pp. 3–29.

[27] ——, "Modeling and verifying security protocols with the applied pi calculus and ProVerif," *Foundations and Trends in Privacy and Security*, vol. 1, no. 1–2, pp. 1–135, Oct. 2016.

[28] D. Bleichenbacher, "Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS# 1," in *Annual International Cryptology Conference*, ser. Lecture Notes in Computer Science, vol. 1462. Springer, 1998, pp. 1–12.

[29] M. Bodin, A. Charguéraud, D. Filaretti, P. Gardner, S. Maffeis, D. Naudziuniene, A. Schmitt, and G. Smith, "A trusted mechanised javascript specification," in *ACM Symposium on the Principles of Programming Languages (POPL)*, 2014, pp. 87–100.

[30] D. Cadé and B. Blanchet, "Proved generation of implementations from computationally secure protocol specifications," *Journal of Computer Security*, vol. 23, no. 3, pp. 331–402, 2015.

[31] S. Chaki and A. Datta, "Aspier: An automated framework for verifying security protocol implementations," in *2009 22nd IEEE Computer Security Foundations Symposium*. IEEE, 2009, pp. 172–185.

[32] A. Chaudhuri, "Flow: Abstract interpretation of javascript for type checking and beyond," in *ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, 2016.

[33] J.-S. Coron, Y. Dodis, C. Malinaud, and P. Puniya, "Merkle-Damgård revisited: How to construct a hash function," in *Advances in Cryptology (CRYPTO)*, 2005, pp. 430–448.

[34] V. Cortier, S. Kremer, and B. Warinschi, "A survey of symbolic methods in computational analysis of cryptographic systems," *Journal of Automated Reasoning*, vol. 46, no. 3-4, pp. 225–259, 2011.

[35] C. Cremers, M. Horvat, S. Scott, and T. van der Merwe, "Automated analysis and verification of TLS 1.3: 0-RTT, resumption and delayed authentication," in *IEEE Symposium on Security and Privacy (Oakland)*, 2016, pp. 470–485.

[36] I. B. Damgård, "A design principle for hash functions," in *Advances in Cryptology–CRYPTO89*, 1989, pp. 416–427.

[37] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2," IETF RFC 5246, 2008.

[38] Y. Dodis, T. Ristenpart, J. Steinberger, and S. Tessaro, "To hash or not to hash again? (In)differentiability results for H^2 and HMAC," in *Advances in Cryptology (Crypto)*, 2012, pp. 348–366.

[39] D. Dolev and A. C. Yao, "On the security of public key protocols," *IEEE Transactions on Information Theory*, vol. 29, no. 2, pp. 198–207, 1983.

[40] B. Dowling, M. Fischlin, F. Günther, and D. Stebila, "A cryptographic analysis of the TLS 1.3 handshake protocol candidates," in *ACM Conference on Computer and Communications Security (CCS)*, 2015, pp. 1197–1210.

[41] M. Fischlin, F. Günther, B. Schmidt, and B. Warinschi, "Key confirmation in key exchange: A formal treatment and implications for TLS 1.3," in *IEEE Symposium on Security and Privacy (Oakland)*, 2016, pp. 452–469.

[42] M. Fischlin and F. Günther, "Multi-stage key exchange and the case of Google's QUIC protocol," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2014, pp. 1193–1204.

[43] S. Goldwasser, S. Micali, and R. Rivest, "A digital signature scheme secure against adaptive chosen-message attacks," *SIAM Journal of Computing*, vol. 17, no. 2, pp. 281–308, April 1988.
[44] R. Hamilton, J. Iyengar, I. Swett, and A. Wilk, "QUIC: A UDP-based multiplexed and secure transport," 2016, IETF Internet Draft.
[45] K. E. Hickman, "The SSL protocol," 1995, IETF Internet Draft, https://tools.ietf.org/html/draft-hickman-netscape-ssl-00.
[46] T. Jager, F. Kohlar, S. Schäge, and J. Schwenk, "On the security of TLS-DHE in the standard model," in *CRYPTO 2012*, 2012, pp. 273–293.
[47] T. Jager, J. Schwenk, and J. Somorovsky, "On the security of TLS 1.3 and QUIC against weaknesses in PKCS#1 v1.5 encryption," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015, pp. 1185–1196.
[48] N. Kobeissi, K. Bhargavan, and B. Blanchet, "Automated verification for secure messaging protocols and their implementations: A symbolic and computational approach," in *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2017.
[49] H. Krawczyk, "Cryptographic extraction and key derivation: The HKDF scheme," in *Advances in Cryptology (CRYPTO)*, 2010, pp. 631–648.
[50] ——, "A unilateral-to-mutual authentication compiler for key exchange (with applications to client authentication in tls 1.3)," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016, pp. 1438–1450.
[51] H. Krawczyk, K. G. Paterson, and H. Wee, "On the security of the TLS protocol: A systematic analysis," in *CRYPTO 2013*, 2013, pp. 429–448.
[52] H. Krawczyk and H. Wee, "The OPTLS protocol and TLS 1.3," in *IEEE European Symposium on Security & Privacy (Euro S&P)*, 2016, cryptology ePrint Archive, Report 2015/978.
[53] R. Küsters, T. Truderung, and J. Graf, "A framework for the cryptographic verification of Java-like programs," in *IEEE Computer Security Foundations Symposium (CSF)*, 2012, pp. 198–212.
[54] A. Langley, M. Hamburg, and S. Turner, "Elliptic curves for security," IRTF RFC 7748 https://tools.ietf.org/html/rfc7748, Jan. 2016.
[55] X. Li, J. Xu, Z. Zhang, D. Feng, and H. Hu, "Multiple handshakes security of TLS 1.3 candidates," in *IEEE Symposium on Security and Privacy (Oakland)*, 2016, pp. 486–505.
[56] R. Lychev, S. Jero, A. Boldyreva, and C. Nita-Rotaru, "How secure and quick is QUIC? provable security and performance analyses," in *IEEE Symposium on Security & Privacy (Oakland)*, 2015, pp. 214–231.
[57] U. Maurer and B. Tackmann, "On the soundness of authenticate-then-encrypt: formalizing the malleability of symmetric encryption," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2010, pp. 505–515.
[58] N. Mavrogiannopoulos, F. Vercauteren, V. Velichkov, and B. Preneel, "A cross-protocol attack on the TLS protocol," in *ACM CCS*, 2012.
[59] C. Meyer, J. Somorovsky, E. Weiss, J. Schwenk, S. Schinzel, and E. Tews, "Revisiting SSL/TLS implementations: New Bleichenbacher side channels and attacks," in *23rd USENIX Security Symposium*. USENIX Association, 2014, pp. 733–748.
[60] B. Möller, T. Duong, and K. Kotowicz, "This POODLE bites: exploiting the SSL 3.0 fallback," https://www.openssl.org/~bodo/ssl-poodle.pdf, 2014.
[61] T. Okamoto and D. Pointcheval, "The gap-problems: a new class of problems for the security of cryptographic schemes," in *Practice and Theory in Public Key Cryptography (PKC)*, 2001, pp. 104–118.
[62] K. G. Paterson, T. Ristenpart, and T. Shrimpton, "Tag size does matter: Attacks and proofs for the TLS record protocol," in *ASIACRYPT*, 2011, pp. 372–389.
[63] K. G. Paterson and T. van der Merwe, "Reactive and proactive standardisation of TLS," in *Security Standardisation Research (SSR)*, 2016, pp. 160–186.
[64] M. Ray, A. Pironti, A. Langley, K. Bhargavan, and A. Delignat-Lavaud, "Transport Layer Security (TLS) session hash and extended master secret extension," 2015, IETF RFC 7627.
[65] E. Rescorla, M. Ray, S. Dispensa, and N. Oskov, "TLS renegotiation indication extension," IETF RFC 5746, 2010.
[66] E. Rescorla, "0-RTT and Anti-Replay," https://www.ietf.org/mail-archive/web/tls/current/msg15594.html, Mar. 2015.
[67] ——, "[TLS] PR#875: Additional Derive-Secret stage," https://www.ietf.org/mail-archive/web/tls/current/msg22373.html, Feb. 2017.
[68] B. Schmidt, S. Meier, C. Cremers, and D. Basin, "Automated analysis of Diffie-Hellman protocols and advanced security properties," in *IEEE Computer Security Foundations Symposium (CSF)*, 2012, pp. 78–94.
[69] D. Stefan, "Espectro project description," 2016, https://cseweb.ucsd.edu/~dstefan/#projects.
[70] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoue, and S. Zanella-Béguelin, "Dependent types and multi-monadic effects in F*," in *ACM Symposium on Principles of Programming Languages (POPL)*, 2016, pp. 256–270.
[71] M. Vanhoef and F. Piessens, "All your biases belong to us: Breaking RC4 in WPA-TKIP and TLS," in *USENIX Security Symposium*, 2015, pp. 97–112.
[72] D. Wagner and B. Schneier, "Analysis of the SSL 3.0 protocol," in *USENIX Electronic Commerce*, 1996.
[73] J. K. Zinzindohoue, E. Bartzia, and K. Bhargavan, "A verified extensible library of elliptic curves," in *IEEE Computer Security Foundations Symposium (CSF)*, 2016, pp. 296–309.

APPENDIX A.
LEMMAS ON PRIMITIVES AND ON THE KEY SCHEDULE

We show the following properties:

- $\mathsf{mac}^k_{\mathsf{H}}(m) = \mathsf{mac}^k(\mathsf{H}(m))$ is an SUF-CMA (strongly unforgeable under chosen message attacks) MAC. Indeed, since $\mathsf{mac} = \mathsf{HMAC\text{-}H}$ is a PRF, it is an SUF-CMA MAC as shown in [10], and this property is preserved by composition with a collision-resistant hash function.
- $\mathsf{sign}^{sk}_{\mathsf{H}}(m) = \mathsf{sign}^{sk}(\mathsf{H}(m))$ is an UF-CMA signature. Indeed, sign is an UF-CMA signature, and this property is preserved by composition with a collision-resistant hash function.

We also prove several lemmas on the key schedule of TLS 1.3, using CryptoVerif.

- When es is a fresh random value, $e \mapsto \mathsf{hkdf\text{-}extract}(es, e)$ and $log_1 \mapsto \mathsf{derive\text{-}secret}(es, \mathsf{ets}_c, log_1)$ are indistinguishable from independent random functions, and $k^b = \mathsf{derive\text{-}secret}(es, \mathsf{pbk}, \text{“”})$ and $\mathsf{hkdf\text{-}extract}(es, 0^{len_{\mathsf{H}()}})$ are indistinguishable from independent fresh random values independent from these random functions.
- When hs is a fresh random value, $log_1 \mapsto \mathsf{derive\text{-}secret}(hs, \mathsf{hts}_c, log_1) \| \mathsf{derive\text{-}secret}(hs, \mathsf{hts}_s, log_1)$ is indistinguishable from a random function and $\mathsf{hkdf\text{-}extract}(hs, 0^{len_{\mathsf{H}()}})$ is indistinguishable from a fresh random value independent from this random function.
- When ms is a fresh random value, the functions $log_4 \mapsto \mathsf{derive\text{-}secret}(ms, \mathsf{ats}_c, log_4) \| \mathsf{derive\text{-}secret}(ms, \mathsf{ats}_s, log_4) \| \mathsf{derive\text{-}secret}(ms, \mathsf{ems}, log_4)$ and $log_7 \mapsto \mathsf{derive\text{-}secret}(ms, \mathsf{rms}, log_7)$ are indistinguishable from independent random functions.
- When l_1, l_2, l_3 are pairwise distinct labels and s is a fresh random value, $\mathsf{hkdf\text{-}expand\text{-}label}(s, l_i, \text{“”})$ for $i = 1, 2, 3$ are indistinguishable from independent fresh random values.

All random values considered above are uniformly distributed. We use these properties as assumptions in our proof of the protocol. This modular approach considerably reduces the complexity of the games that CryptoVerif has to consider.

These results suggest that the key schedule could be simplified by replacing groups of calls to derive-secret that

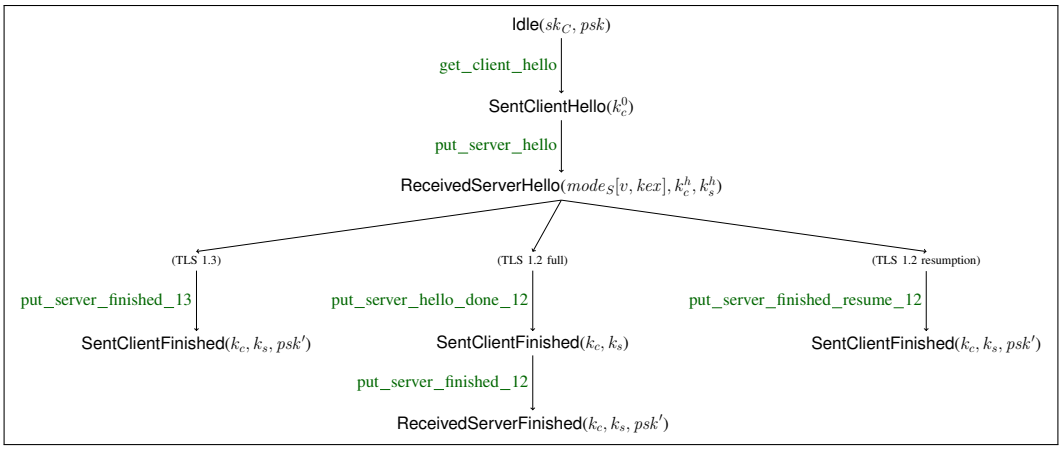


Figure 7: Client state machine

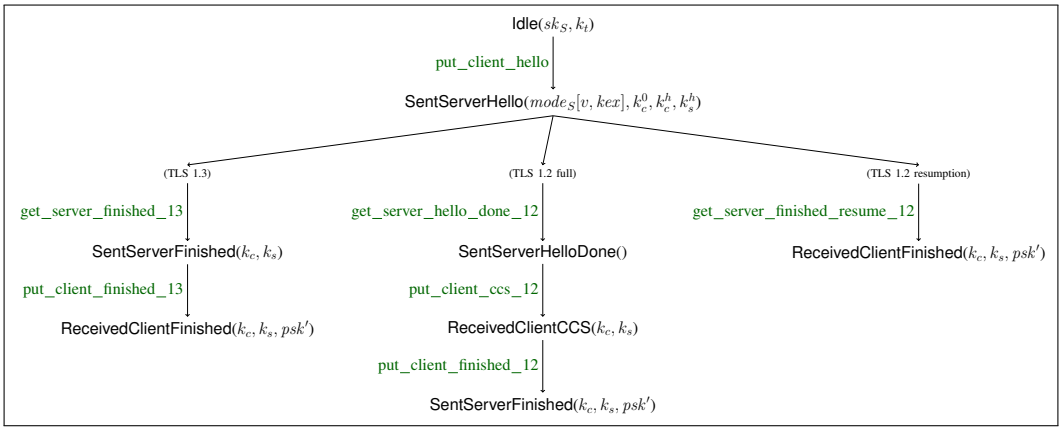


Figure 8: Server state machine

use the same key and log with a single call to derive-secret that would output the concatenation of severals keys. The same remark also holds for calls to hkdf-expand-label that use the same key. This approach corresponds to the usage of expansion recommended in the formalization of HKDF [49], and would simplify the proof: some lemmas above would no longer be needed. We would also recommend replacing $ms = \mathsf{hkdf\text{-}extract}(hs, 0^{len_{\mathsf{H}()}})$ with $ms = \mathsf{derive\text{-}secret}(hs, \mathsf{ms}, \text{“”})$: that would be more natural since we use the PRF property of HMAC-H for this computation and not the randomness extraction. If the argument $0^{len_{\mathsf{H}()}}$ may change in the future, then we would support Krawczyk's recommendation [67] of applying hkdf-extract to the result of $\mathsf{derive\text{-}secret}(hs, \mathsf{ms}, \text{“”})$.

APPENDIX B. REFTLS PROTOCOL STATE MACHINES

Client. The RefTLS client implements the composite state machine shown in Figure 7 for TLS 1.3 and TLS 1.2. Each state represents a point in the protocol where the client is either waiting for a flight of handshake messages from the server, or it has new session keys that it wishes to communicate to the record layer. Each arrow is annotated with the name of the function in RefTLS-CORE API that implements the corresponding state transition. Each transition may involve processing a flight of incoming messages, changing the session state, and producing a flight of outgoing messages.

Server. The RefTLS server implements a dual state machine for TLS 1.3 and TLS 1.2, as depicted in Figure 8. The server decides which protocol version and key exchange the handshake will use, and triggers the appropriate branch in the state machine by sending a `ServerHello`. Like the client, each of its state transition functions corresponds either to a flight of messages or to a change of keys.

A Formal Model for ACME: Analyzing Domain Validation over Insecure Channels

Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Nadim Kobeissi

A Formal Model for ACME: Analyzing Domain Validation over Insecure Channels

Karthikeyan Bhargavan[1], Antoine Delignat-Lavaud[2] and Nadim Kobeissi[1]

[1] INRIA
{karthikeyan.bhargavan, nadim.kobeissi}@inria.fr
[2] Microsoft Research
antdl@microsoft.com

Abstract. Web traffic encryption has shifted from applying only to highly sensitive websites (such as banks) to a majority of all Web requests. Until recently, one of the main limiting factors for enabling HTTPS is the requirement to obtain a valid certificate from a trusted certification authority, a tedious process that typically involves fees and ad-hoc key generation, certificate request and domain validation procedures. To remove this barrier of entry, the Internet Security Research Group created Let's Encrypt, a new non-profit certificate authority which uses a new protocol called Automatic Certificate Management Environment (ACME) to automate certificate management at all levels (request, validation, issuance, renewal, and revocation) between clients (website operators) and servers (certificate authority nodes). Let's Encrypt's success is measured by its issuance of over 12 million free certificates since its launch in April 2016.

In this paper, we survey the existing process for issuing domain-validated certificates in major certification authorities to build a security model of domain-validated certificate issuance. We then model the ACME protocol in the applied pi-calculus and verify its stated security goals against our threat model of domain validation. We compare the effective security of different domain validation methods and show that ACME can be secure under a stronger threat model than that of traditional CAs. We also uncover weaknesses in some flows of ACME 1.0 and propose verified improvements that have been adopted in the latest protocol draft submitted to the IETF.

1 Introduction

Since the dawn of HTTPS, being able to secure a public website with SSL or TLS required obtaining a signature for the website's public certificate from a certificate authority [1] (CA). These certificate authorities had to be recognized by all major operating system and browser vendors to be legitimate entities that could attest for a reasonable link between a certificate and identity of the server or domain it claims to represent.

For example, all major operating systems ship with Symantec's root certificate signing key as built-in and trusted. This symbolizes a permission by

these major players for Symantec to then act as a CA, which allows Alice, the owner of `AliceShop.com`, to ask Symantec to attest that some SSL certificate being served by this website is indeed identifying the legitimate server behind `AliceShop.com`. After Alice pays Symantec some verification fee, Symantec performs some check to verify that Alice and her web server indeed have the authority over `AliceShop.com`. If successful, Symantec then signs a certificate intended for that domain. Since the aforementioned operating systems already trust Symantec, this trust now extends towards Alice's certificate as being authentically representative of `AliceShop.com`.

The security of this trust model has always relied on the responsibility and trustworthiness of the CAs themselves, since a single malicious CA can issue arbitrary valid certificates for any website on the Internet. Each certificate authority is free to engineer different user sign-up, domain validation, certificate issuance and certificate renewal protocols of its own design. Since these ad-hoc protocols often operate over weak channels such as HTTP and DNS, with no strong notion of cryptographic client authentication, most of them can be considered secure only under relatively weak threat models, reducing user credentials to a web login, and domain validation to an email exchange.

The main guidelines controlling what type of domain validation CAs are allowed to apply are the recommendations in the CA/Browser Forum Baseline Requirements [2]. These requirements, which are adopted by ballot vote between the participating organizations, cover the definition of common notions such as domain common names (CNs), registration authorities (RAs) and differences between regular domain validation (DV) and extended validation (EV).

These guidelines have not proven sufficient for a well-regulated and well specified approach for domain validation: Mozilla was recently forced to remove WoSign [3] (and its subsidiary StartSSL, both major certificate authorities) from the certificate store of Firefox and all other Mozilla products due to a series of documented instances that range from the CA intentionally ignoring security best-practices for certificate issuance, to vulnerabilities allowing clients to obtain a signed certificate for any website of their choosing.

The lack of a standardized protocol operating under a well-defined threat model and with clear security goals for certificate issuance has so far prevented a systematic treatment of CA security using well-established formal methods.

In 2015, a consortium of high-profile organizations including Mozilla and the Electronic Frontier Foundation launched Let's Encrypt [4], a non-profit effort to specify, standardize and automate certificate issuance between web servers and certificate authorities, and to provide certificate issuance itself as a free-of-charge service. Since its launch in April 2016, Let's Encrypt has issued more than 12 million certificates [5] and has been linked to a general increase in HTTPS adoption across the Internet.

Let's Encrypt also introduces ACME [6], an automated domain validation and certificate issuance protocol that gives us for the first time a protocol that can act as a credible target for formal verification in the context of domain validation. ACME also removes almost entirely the human element from the process

of domain validation: the subsequently automated validation and issuance of millions of certificates further increases the necessity of a formal approach to the protocol involved.

In this paper, we formally specify, model and verify ACME using the automated protocol verifier ProVerif [7]. Against a classic symbolic protocol adversary, ACME achieves most of its security goals. Notably, it prevents many more attacks than traditional CAs through stronger cryptographic notions of user identity, achieved through automated client signatures and strong binding between the clients identity and the validated domain. In comparison, we show that ACME's design allows it to resist a substantially stronger threat model than the traditional ad-hoc protocols of other CAs that rely on bearer tokens (passwords, cookies, authorization strings) for authentication and domain validation.

Nevertheless, we still discover issues and weaknesses in ACME's domain validation and account recovery features, potentially amounting to user account compromise. We attempt to address in this paper what seem to be open questions regarding ACME: how does ACME compare to the existing security model of the actual top real-world certificate authorities? How can we most fruitfully illustrate and formally verify its security properties, and what can we prove about them?

Contributions We present an outline of our contributions in this paper:

- **A Survey of Existing Domain Validation Practices on Major CAs** In §2, we survey some of the most-used certificate authorities both in terms of protocols and infrastructure. We argue that all of the top 10 traditional CAs operate under an unrealistic threat model.
- **A Threat Model for Domain Validation on the Internet** In §3, we specify a high-level threat model for traditional CAs as well as ACME. We link this threat model to our network topology analysis in §2. In §4, We demonstrate that ACME resists a stronger threat model than ad-hoc protocols.
- **Formally Specifying and Verifying ACME** In §4, we formally specify the ACME protocol and verify it using ProVerif. Although ACME is shown to be more resistant to attacks than ad-hoc CAs, we also discover weaknesses in ACME's domain validation and account recovery and suggest countermeasures.

2 Current State of Domain Validation

A goal of this paper is to establish a relationship between current domain validation practices in the real world and a more formal threat model on which we base our security results. We begin by taking a closer look into the network infrastructure, user authentication and domain validation protocols currently in use by ad-hoc CAs.

Basing ourselves on earlier studies of Web PKIs [8,9], we chose to investigate the domain validation mechanisms of top traditional certificate authorities due

to their high usage and relevance for web certificate issuance. With each CA, we attempted to obtain a regular, one-year, single-domain certificate signature for a domain name that we own.

§3.2.2.4 of the CA/Browser Forum's Baseline Requirements allow for domain validation to occur in ten different ways, including over postal mail and by validating a random number over a TLS certificate. Of these methods, only three are in popular use: validation via email, the setting of an arbitrary DNS record, or serving some HTTP value on the target domain.

2.1 Domain Validation Mechanisms

With ad-hoc CAs, user C authenticates its identity C_{pk} to CA A as a simple username/password web login, with an option for account recovery via email. C can then request that A validate some domain $C_{wx} \subset C_w$. A's flow with the various domain validation channels proceeds thus:

- *HTTP Identifier* A sends to C a nonce A_{URI^C} via an HTTPS channel that C must then advertise at some agreed-upon location under C_{wx}. A then accesses C_{wx} using an unauthenticated, unencrypted HTTP connection to ensure that it can retrieve A_{URI^C}. This identifier depends on honest DNS resolution query responses and an HTTP connection that is resistant to tampering.
- *DNS Identifier* A sends to C a nonce A_{DNS^C} via an HTTPS channel that C must then advertise at some agreed-upon `TXT` record under the DNS records of C_{wx}. A then queries C_{wx}'s name servers using to ensure that it can retrieve A_{DNS^C}. This identifier is dependent on honest DNS resolution query responses.
- *Email Identifier* A sends to C a URI A_{URI^C} via email that C must then visit. Visiting this URI satisfies A, which then issues the certificate for C_{wx}. This identifier is dependent on a private email channel and honest DNS resolution query responses.

Once one of the above identifiers succeeds in validating C's ownership of C_{wx} to A, A issues the certificate and the protocol ends.

2.2 User Authentication and Domain Validation

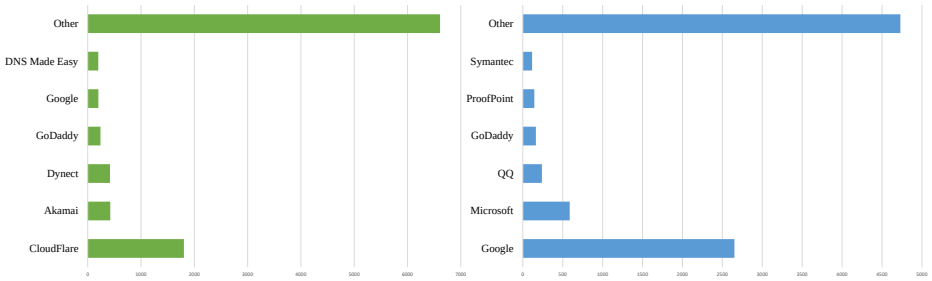
While CAs are required to document their certificate issuance policies in Certificate Practice Statements [10–17], a real-world survey found that these statements are not always accurate. Most ad-hoc CAs in our study restricted their validation capabilities to that of Email Identifiers. Unlike HTTP and DNS Identifiers, Email Identifiers effectively offer C a *read*-based challenge instead proof of some write access. In §3, we discuss how Email Identifiers are the weakest available form of identification given our threat model. In §4, we elaborate on a weakness in ACME affecting both account recovery and domain validation. While this weakness is also generalizable to traditional certificate authorities, ACME offers an opportunity for a stronger fix.

CA	Identifiers	Email Recovery	Public Key Auth.	Per-CSR Check
AlphaSSL	Email	✓	✗	N/A
Comodo PositiveSSL	Email	✓	✗	✓
DigiCert	Email	✓	✗	✗
GeoTrust QuickSSL	Email	✓	✗	✗
GlobalSign	HTTP, DNS, Email	✓	✗	✗
GoDaddy SSL	HTTP, DNS	✓	✗	✗
Let's Encrypt (ACME draft-1)	HTTP	✓	✓	✗
Network Solutions	Email	✓	✗	✗
RapidSSL	Email	✓	✗	✓
SSL.com BasicSSL	HTTP, DNS, Email	✓	✗	N/A
StartCom StartSSL	Email	✓	✓	✗

Fig. 1: Popular CAs, their validation methods, whether they permit account recovery via email, whether they allow login via a public-key based approach (such as client certificates) and whether domain validation is carried out once for every certificate request, even for already-validated domain names.

No CA we surveyed offered a login mechanism that was completely independent of email. An exception almost occurs with StartSSL, which uses browser-generated client certificates for web login, but this exception is negated by StartSSL still allowing email-based account recovery in case of a lost certificate private key. Reliance on the security of the email channel can in many cases be even more serious: in many surveyed CAs, simply being able to complete a web login will allow a user to re-issue certificates for domains they had already validated before, without any further validation.

A scan of the DNS MX and NS records of the web's top 10,000 websites (according to Alexa.com) [18] showed that roughly 45% of surveyed domain names used only six DNS providers, of which CloudFlare alone had a 18% share. Meanwhile, Let's Encrypt's infrastructure is hosted almost entirely on Akamai, rendering it a centralized point of failure for ACME and ad-hoc CAs alike. While ACME is a centralization-agnostic protocol, Let's Encrypt operates with a fully centralized infrastructure. A similar centralization of authority exists with email, where the top six providers serve more than 55% of domain names surveyed, with Google alone holding roughly 27% market share (Figure 2.)



(a) Name Servers. (b) Mail Servers.

Fig. 2: Povider repartition among the Alexa Top 10,000 global sites, as of October 2016. Notably, CloudFlare and Akamai also provide CDN services to domains under their name servers, allowing them stronger control over HTTP traffic.

These results suggest that the number of actors of which the compromise could affect traditional domain validation is significantly small. This is relevant given how top CAs allow for account recovery, certificate re-issuance and more with simple email-based validation.

3 A Security Model for Domain Validation

The protocols considered in this paper operate between a party C claiming to serve and represent one or more domain names C_w (for which it wants certificates), and it is incumbent upon a certificate issuer A to verify that all domains in C_w are indeed controlled and managed by C. User C authenticates itself to CA A using a public key C_{pk} of a private identity C_k. C can then link identifiers under C_k that prove that it manages and controls domains in C_w.

This and following sections are largely based on our full symbolic model[3] of ACME and ad-hoc CA protocol and network flow, which is written in the applied pi calculus and verified using ProVerif. Excerpts of this model are inlined throughout.

3.1 Security Goals and Threat Model

Our security goals are straightforward: for any domain in C_w, A must not sign a certificate for that domain unless C_{pk} is linked with a valid identifier that binds C as the identity that owns and manages this domain. A domain name under C_w is considered to be *validated* under C_w if an identifier for that domain can be linked to C_{pk}.

The network topology, channels and actors are essentially the same for both ACME and ad-hoc CAs. However, the manner in which these actors communi-

[3] Full models available at https://github.com/Inria-Prosecco/acme-model

cate over the channels is different, and leads to different attempts to establish the same security guarantees.

Channels Intuitively, the channels we want encapsulate the following properties:

- **HTTPS Channel** Intuitively a regular web channel, we treat it as a A-authenticated duplex channel whereupon anyone can send a request to A, only A can read this request and respond, and only the sender can read this response.
- **Strong Identifier Channels** These channels must be assumed to be writable only by C. They are therefore relevant for HTTP and DNS Identifiers.
- **Weak Identifier Channel** Anyone can write to this channel, but only C can read from it. This makes it relevant for domain validation via Email Identifiers.

A shared consideration between ACME and ad-hoc CAs involves the critical importance of DNS resolution: if the attacker can simply produce false DNS responses for A resolving a domain request for any domain in C_w, it becomes impossible to safely carry out domain validation under any circumstances. As a sidenote, this allows us to argue that since the DNS channel must be trusted, it could also be considered as the safest channel on which to carry out domain validation using DNS Identifiers since that would allow C to avoid needlessly involving other channels.

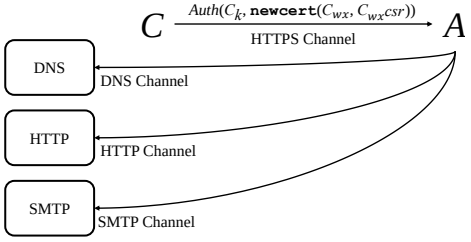


Fig. 3: Channels overview.

In formally describing our network model in ProVerif, we simulate simultaneous requests from Alice, Bob and Mallory as independent clients C. We also simulate two independent ACME CAs, which interchangeably assume the role of A. For each C, we specify a triple of distinct channels:

$$(C_{HTTP}, C_{EMAIL}, C_{DNSTXT})$$

Each channel represents access to a different domain validation mechanism. While C is given complete access over these channels, a write transformation $w(channel) \rightarrow channel$ is applied to C_{EMAIL} before it's handed to A. Similarly, a read transformation $r(channel) \rightarrow channel$ is applied to C_{HTTP} and C_{DNSTXT}.

A routing proxy is then specified in order to model the transportation across these channels by executing the following unbounded processes in parallel[4]:

[4] We also specify a fully public channel named `pub`.

$$in(w(C_{EMAIL}), x);\ out(r(C_{EMAIL}), x)$$
$$in(pub, x);\ out(r(C_{EMAIL}), x)$$
$$in(w(C_{HTTP}), x); out(pub, x); out(r(C_{HTTP}), x)$$
$$in(w(C_{DNSTXT}), x); out(pub, x); out(r(C_{DNSTXT}), x)$$

Threat Model We assume that the adversary controls parts of the network and so can intercept, tamper with and inject network messages. As such, an attacker could make requests for domains they do not own, intercept and delay legitimate certificate requests, and so on. Our adversary has full access to *pub*, $w(C_{EMAIL})$, $r(C_{HTTP})$ and $r(C_{HTTP})$. We also publish Mallory's channels and C_k over *pub*. As such, the attacker controls a set of valid participants (e.g. M) with their own valid identities (e.g. M_k, M_{pk}). The attacker may advertise any identity for its controlled principals, including false identities, and may attempt to obtain a certificate for domains not legitimately under M_w.

The adversary also has at his disposal certain special functions:

- *poisonDnsARecord*, which takes in a domain C_{wx} and allows the attacker to poison its DNS records to redirect to a server owned by M. Using this function triggers the $ActiveDnsAttack(C_{wx})$ event.
- *manInTheMiddleHttp*, which allows the attacker to write arbitrary HTTP requests as if they were emitting from C_{HTTP} by disclosing C_{HTTP} to the attacker. Using this function triggers the $ActiveHttpAttack(C_{wx})$ event.

3.2 Events and Queries

Queries under our model are constructed from sequences of the following events, each callable by a particular type of actor:

- **Client** The client is allowed to assert that they own some domain by triggering the event $Owner(M, C_{wx})$. Once C receives a certificate $C_{wx_{cert}}$ for C_{wx} from A, they also trigger $CertReceived(C_{wx}, C_{wx_{cert}}, C_{pk}, A_{pk})$
- **Server** The server (ACME instance or CA) triggers the event $HttpAuth(C_{pk}, C_{wx})$, $DnsAuth(C_{pk}, C_{wx})$ and $EmailAuth(C_{pk}, C_{wx})$ depending on the type of domain validation used. Once A issues a certificate $C_{wx_{cert}}$ for C_{wx} to C, they also trigger $CertIssued(C_{wx}, C_{wx_{cert}}, C_{pk}, A_{pk})$
- **Adversary** As noted above, the adversary may trigger the events $ActiveDnsAttack(C_{wx})$ and $ActiveHttpAttack(C_{wx})$. In addition, the adversary is allowed to masquerade as M in order assert that they own some domain by triggering the event $Owner(M_{pk}, C_{wx})$.

Queries We evaluate our model against the following queries:

Validation with DNS Identifiers We assert that if DNS validation succeeded, then A must have been able to successfully carry out DNS validation according to spec, *or* an adversary was able to instantiate an active DNS poisoning attack (with no third possible scenario):

$$DnsAuth(C_{pk}, C_{wx}) \implies (Owner(C_k, C_{wx}) \vee DnsAttack(C_{wx}))$$

Validation with HTTP Identifiers We explicitly show that HTTP authentication is weaker than DNS authentication, since it is possible under both cases of DNS poisoning *and* an HTTP man-in-the-middle attack:

$$HttpAuth(C_{pk}, C_{wx}) \implies Owner(C_k, C_{wx}) \ \vee \ (HttpAttack(C_{wx}) \vee DnsAttack(C_{wx}))$$

Predictable Certificate Issuance We attempt to verify that all received certificates were issued by the expected CA. This query fails to verify, and leads us to the attack we discuss in §5.2:

$$CertReceived(C_{wx}, C_{wx_{cert}}, C_{pk}, A_{pk}) \implies CertIssued(C_{wx}, C_{wx_{cert}}, C_{pk}, A_{pk})$$

4 Specifying and Formally Verifying ACME

In this section we provide a formal description of the ACME protocol functionality and identify three issues that affect ACME's security. We also discuss details of how we describe the ACME protocol flow in the applied pi calculus, so that we can verify for certain queries using ProVerif.

4.1 ACME Network Flow

Unlike ad-hoc CAs which are limited to a web login, ACME's authentication depends on C generating a private value C_k and a public signing key C_{pk}, which are used to generate automated client signatures throughout the protocol.

HTTP Identifier A sends to C a nonce A_{URI^C} via the HTTPS channel. C must then advertise, at an agreed-upon location under C_{wx}, the value (C_pk, A_{URI^C}). A then accesses C_{wx} using an unauthenticated, unencrypted HTTP connection to ensure that it can retrieve the intended value.

DNS Identifier Since ACME is designed to take advantage of domain validation methods that can be automated and since DNS record management depends on a series of ad-hoc protocols of its own between C and DNS service providers, it is not used by ACME.

Email Identifier This identifier is only used for account recovery, and a resulting weakness is discussed in §5.2.

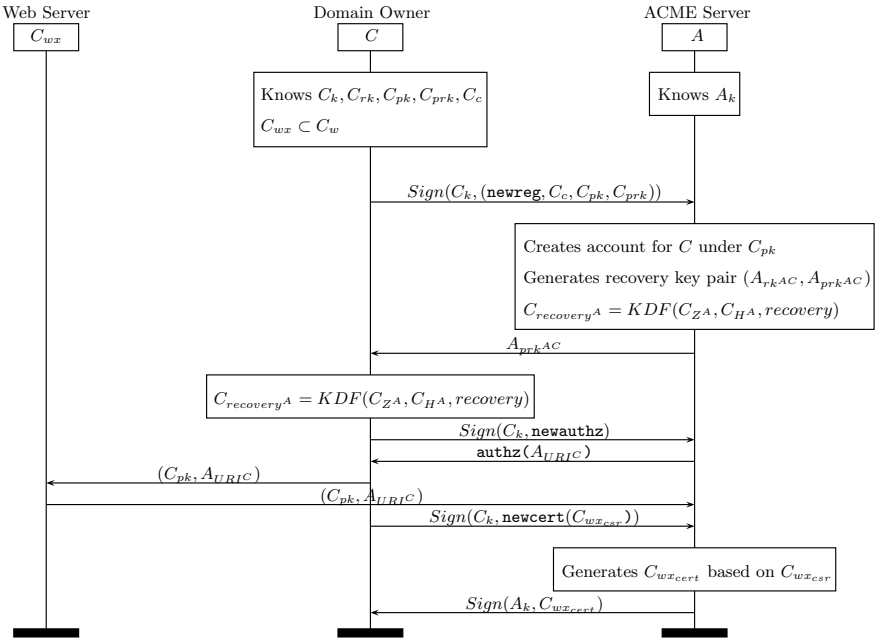


Fig. 4: ACME draft-1 protocol functionality for C account registration, recovery key generation, and validation with certificate issuance for C_{wx}. This chart demonstrates validation via an HTTP identifier. In draft-3 and above, the HTTP challenge (C_{pk}, A_{URI^C}) is replaced with $Sign(C_k, (C_{pk}, A_{URI^C}))$.

4.2 ACME Protocol Functionality

In this paper we focus on draft-1 of the IETF specification for the ACME protocol. As of October 2016, the draft specification deployed in official Let's Encrypt client and server implementations is somewhere between draft-2 and draft-3[5]. However, draft-1 was adopted after Let's Encrypt's launch and for a majority of 2016. In part due to the issues we discuss in the paper and have communicated with the ACME team, draft-3 (and subsequently draft-4) does away with some features, most notably Account Recovery, and generally is resistant to the issues discussed here.

Preliminaries In some parts of ACME's protocol flow, C and A will need to establish a number of shared secrets, each bound to a strict protocol context, over their public keys. In ACME, this is accomplished using ANSI-X9.63-KDF:

[5] https://github.com/letsencrypt/boulder/blob/master/docs/acme-divergences.md

1. C and A agree on a ECDH shared secret C_{Z^A} using their respective key pairs (C_k, C_{pk}) and (A_k, A_{pk}).
2. A hashing function C_{H^A} is chosen according to the elliptic curve used to calculate C_{Z^A}: $SHA256$ for $P256$, $SHA384$ for $P384$ and $SHA512$ for $P521$.
3. $C_{label^A} = KDF(C_{Z^A}, C_{H^A}, label)$, with *label* indicating the chosen context for this particular key's usage.

As a protocol, ACME provides the following seven certificate management functionalities (illustrated in Figure 4) between web server C and certificate management authority A:

- *Account Key Registration* In this step, C specifies her contact information (email address, phone number, etc.) as C_c and generates a random private signing key C_k with (over a safe elliptic curve) a public key C_{pk}. A `POST` request is sent to A containing $Sign(C_k, (\texttt{newreg}, C_c, C_{pk}))$. The `newreg` header indicates to A that this is an account registration request. If A has no prior record of C_{pk} being used for an account, and if the message's signature is valid under C_{pk}, A creates a new account for C using C_{pk} as the identifier and responds with a success message.
- *MAC-Based Account Recovery* C may choose to identify an account recovery secret with A. In order to do this, C generates an account recovery key pair (C_{rk}, C_{prk}) and simply includes C_{prk} in an optional `recoverykey` field in its initial `newreg` message to A. A generates the complementary $(A_{rk^{AC}}, A_{prk^{AC}})$ and both parties calculate $C_{recovery^A}$ using their recovery key pairs. A communicates $A_{prk^{AC}}$ in its response to C. Later, if C loses C_k, she can ask A to re-assign her account to a new identity $(C_{k'}, C_{pk'})$ by using $C_{recovery^A}$ as a key to generate a MAC of some value chosen by A.
- *Contact-Based Account Recovery* C can request that A send a verification token to one of the contact methods previously specified in C_c. For example, this could be a URI sent to an email in C_c. If C successfully opens this URI, she becomes free to replace C_{pk} with a $C_{pk'}$ for some arbitrary $C_{k'}$ at A.
- *Identifier Authorization* C can validate its ownership of a domain C_{wx} in C_w by providing one of the identifiers discussed in §3 to A. C must first request authorization for C_{wx} by sending a `newauthz` message. A then responds with the types of identifiers it is willing to accept in a `authz` message. C is then free to use any one of the permitted identifiers to validate its ownership of C_{wx} and allow A to sign certificates for it issued to C and tied to the identity C_{pk}.
- *Certificate Issuance and Renewal* After C ties an identifier to C_{wx} under C_{pk}, it may request that a certificate be issued for C_{wx} simply by requesting one from A. Generally, A will send the signed certificate with no further steps required. The renewal procedure is similarly straightforward.
- *Certificate Revocation* C may ask A to revoke the certificate for C_{wx} by sending a `POST` message containing the certificate in question, signed under either C_{pk} or the key pair for the certificate itself. C may choose which key to use for this signature. A verifies that the public key of the key pair

signing the request matches the public key in the certificate, and that the key pair signing the request is an account key, and the corresponding account is authorized to act for all of the identifier(s) in the certificate.

Given this description of the ACME protocol and the threat model defined in §3, we modeled ACME using the automated verification tool ProVerif [19]. In our model, we involve three different candidates for C: Alice, Bob and Mallory, and two CA candidates as A.

As a result of our automated verification process which an active attacker over the three channels specified in §3, we were able to find the issues discussed in §5.2. The first two are relatively minor; however, the third could lead to account compromise in the case of contact-based account recovery, and potentially to the issuance of false certificate signatures if email-based domain validation were to be implemented in ACME. Furthermore, this third issue is also generalizable to affect traditional certificate authorities, as described in §2.

4.3 Model Processes

Using the modeling conventions we established in §3 which include channels, adversaries, actors and events, we instantiate in our ProVerif model of ACME a top-level process that executes the following processes in parallel:

- *clientAuth* Run simultaneously by Alice, Bob and Mallory assuming the role of C (with a compromised Mallory), this process registers a new account with A and sends the queries illustrated in Figure 4. The events $Owner$ and $CertReceived$ are triggered as part of this process.
- *serverAuth* Run simultaneously by two independent CAs assuming the role of A, this process accepts registrations from C and follows the protocol illustrated in Figure 4. The events $HttpAuthenticated$ and $CertIssued$ are triggered as part of this process.

The processes $routingProxy$, $poisonDnsARecord$ and $manInTheMiddleHttp$, all described in §3, are also run in parallel with the above.

5 Analysis Results

5.1 Weaknesses in Traditional CAs

Traditional CA dependence on weak channels gives us a threat model where real-world attacks can have a small cost and come with severe consequences.

Email Validation In ad-hoc CAs, C is generally simply sent an email containing a URI to their email inbox, which they're supposed to click in order to validate for their chosen domain. Figure 5 shows an attack rendered possible by this mechanism. A could instead, upon a validation request, redirect C's browser to

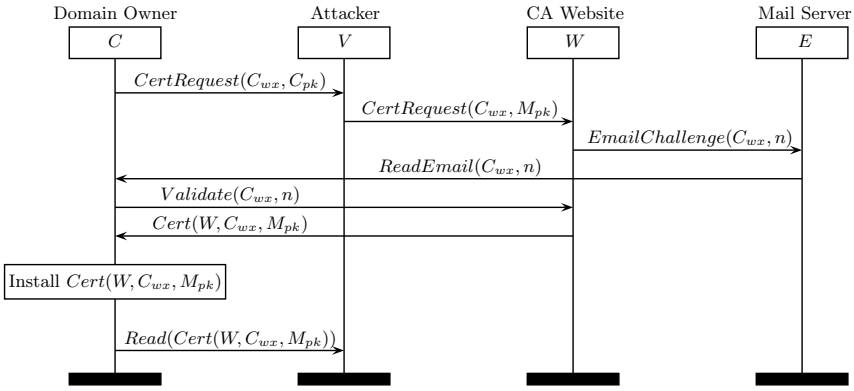


Fig. 5: Attack on Email Validation: Concurrent Request by Active Adversary.

a secret, nonce-based URI A_{URI^C} served to C over the HTTPS channel, and independently mail C the value $HMAC(A_{hk}, A_{URI^C})$ for some secret A_{hk} held by A. C would need to retrieve this second value and enter it inside the page at A_{URI^C}. This approach would largely negate the weakness discussed in §5.2, since an attacker-induced validation email would result in an email that does not include a value matching the URI given by A to C at the beginning of the validation process.

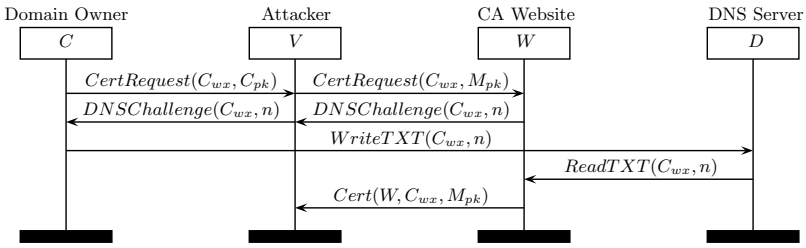


Fig. 6: Active attack on DNS/HTTP/Email Validation when using just nonces.

Usage of Nonces Traditional CAs use random nonces with no special cryptographic properties as the values that they then verify over HTTP, email or DNS.

In addition to this helping caused the attack described above, another more general attack on nonces is shown in Figure 6 in the case of an active attacker. For example, this attack can be used by a compromised CA website to get certificates issued for domain C_{wx} by another (more reputable) CA, hence amplifying the compromise across CAs. None of these attacks would be effective if nonces were tied to some cryptographic properties, such as MACs or even just by deriving them from a hash of the certificate request's public key.

In order to avoid a similar attack, ACME draft-3 and draft-4 require that HTTP identifiers be validated by broadcasting $Sign(C_k, A_{URIC})$ via the web server instead of ACME draft-1's (C_{pk}, A_{URIC}).

5.2 Weaknesses in ACME

Cross-CA Attacks on Certificate Issuance Suppose an ACME client C requests a certificate from A, and suppose that As HTTPS private key is compromised (or that A is malicious, or a man-in-the-middle has compromised the ACME channel). Now, the attacker can intercept authorization and certificate requests from C to A, and instead forward them to another ACME server A'. If A' requests domain validation with a token T, the attacker forwards the token to the client, who will dutifully place its account key K and token T on its validation channel. A' will check this token and accept the authorization and issue a certificate that the attacker can forward to C.

This means that C asked for a certificate from A, but instead received a certificate from A'. Moreover, it may have paid C for the service, but A' might have done it for free. This issue, while not critical, can be prevented if C checks the certificate it gets to make sure it was issued by the expected CA. An alternative, and possibly stronger, mitigation would be for ACME to extend the Key Authorization string to include the CAs identifier.

More generally, this issue reveals that ACME does not provide channel binding, and this appears as soon as we model the ACME HTTPS Channel. We would have expected to model this as a mutually-authenticated channel since the client always signs its messages with the account key. However, although the clients signature is tunnelled inside HTTPS, the signature itself is not bound to the HTTPS channel. This means that a message from an ACME client C to A can be forwarded by A to a different A' (as long as C supports both A and A'). This kind of credential forwarding attack can be easily mitigated by channel binding. For example, ACME could rely on the Token Binding specifications to securely bind the client signature to the underlying channel. Alternatively, ACME could extend the signed request format to always include the servers name or certificate-hash, to ensure that the message cannot be forwarded to other servers.

Contact-Based Recovery Hijacking While the use of sender-authenticated channels in ACME seems to be relatively secure, more attention needs to be paid to the receiver-authenticated channels. For example, if the ACME server uses

the website administrator's email address to send the domain validation token, a naïve implementation of this kind of challenge would be vulnerable to attack.

In the current specification, the contact channel (typically email) is used for account recovery when the ACME client has forgotten its account key. We show how the careless use of this channel can be vulnerable to attack, and propose a countermeasure. Suppose an ACME client C issues an account recovery request for an account under C_{pk} with a new key $C_{k'}$ to the ACME server A. A network attacker M blocks this request and instead sends his own account recovery request for the account under C_{pk} (pretending to be C) with his own key $M_{k'}$. A will then send C an email asking to click on a link. C will think this is a request in response to its own account recovery request and will click on it. Similarly to the (slightly different) flow described in Figure 5, A will think that C has confirmed account recovery and will transfer the account under C_{pk} to the attackers key $M_{k'}$. In the above attack, the attacker did not need to compromise the contact channel (or for that matter, the ACME channel).

The key observation here is that on receiver-authenticated channels (e.g. email) the receiver does not get to bind the token provided by A with its own account key. Consequently, we need to add a further check. The email sent from A to C should contain a fresh token in addition to Cs new account key. Instead of clicking on the link (out-of-band), C should cut and paste the token into the ACME client which can first check that the account key provided by A matches the one in the ACME client and only then does it send the token back to A, or alternatively that the email recipient at C visually confirms that the account key (thumbprint) provided by A matches the one displayed in the ACME client.

The attack described here is on account recovery, but a similar attack appears if we allow email-based domain validation. A malicious ACME server or man-in-the-middle can then get certificate issued for C's domains with its own public key, without compromising the contact/validation channel. The mitigation for that attack would be very similar to the one proposed above.

6 Conclusion

In this paper, we have provided the results of an empirical case study that allowed us to describe a real-world threat model governing both traditional certificate authorities and ACME in terms of user authentication and domain validation. We formally modeled these protocols and provided the results of security queries under our threat model, using automated verification. As a result of our disclosures to the ACME team, the latest ACME protocol version (draft-4) has been designed to avoid the pitfalls that make these attacks possible.

Given the weak threat model that ad-hoc CAs are expecting to survive under and the subsequent weaknesses that we describe, we believe that there is a strong need for either larger ACME adoption, or the improvement of the current state of the art for ad-hoc CA domain validation mechanisms. We hope that this work will help act as a bedrock for future formal description of domain validation systems. We also hope to see a wider deployment of related technologies, such

as DNSSEC [20], DANE [21] and SMTPS, that help strengthen the channels involved in domain validation protocols.

References

1. S. Chokhani, W. Ford, R. Sabett, C. Merrill, and S. Wu. Internet X.509 Public Key Infrastructure: Certificate Policy and Certification Practices Framework. RFC 3647, Internet Engineering Task Force, November 2003.
2. CA/Browser Forum. Baseline requirements for the issuance and management of policy-trusted certificates, v.1.1.5, May 2013.
3. Gervase Markham, Ryan Sleevi, Richard Barnes, and Kathleen Wilson. Wosign and startcom.
4. Internet Security Research Group. Let's encrypt overview, 2016.
5. Internet Security Research Group. Let's encrypt statistics, 2016.
6. Richard Barnes, Jacob Hoffman-Andrews, and James Kasten. Automatic certificate management environment (acme), Jul 2016.
7. Bruno Blanchet, Ben Smyth, and Vincent Cheval. Proverif 1.90: Automatic cryptographic protocol verifier, user manual and tutorial, 2014.
8. Antoine Delignat-Lavaud, Martín Abadi, Andrew Birrell, Ilya Mironov, Ted Wobber, Yinglian Xie, and Microsoft Research. Web pki: Closing the gap between guidelines and practices. In *NDSS*, 2014.
9. Olivier Levillain, Arnaud Ébalard, Benjamin Morin, and Hervé Debar. One year of SSL Internet measurement. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 11–20, New York, NY, USA, 2012. ACM.
10. Comodo CA Ltd. Comodo Certification Practice Statement. Technical report, Comodo CA Ltd., August 2015.
11. DigiCert. DigiCert Certification Practices Statement. Technical report, DigiCert, September 2016.
12. GeoTrust Inc. GeoTrust Certification Practice Statement. Technical report, GeoTrust, September 2016.
13. GlobalSign CA. GlobalSign CA Certification Practice Statement. Technical report, GlobalSign CA, August 2016.
14. Internet Security Research Group. Certification Practice Statement. Technical report, Internet Security Research Group, October 2016.
15. Symantec Corporation. Symantec Trust Network (STN) Certification Practice Statement. Technical report, Symantec Corporation, September 2016.
16. StartCom CA Ltd. StartCom Certificate Policy and Practice Statements. Technical report, StartCom CA Ltd., September 2016.
17. LLC Network Solutions. Network Solutions Certification Practice Statement. Technical report, Network Solutions, LLC, September 2016.
18. Alexa Internet Inc. Top 1,000,000 sites (updated daily), 2013.
19. R. K usters and T. Truderung. Using ProVerif to analyze protocols with Diffie-Hellman exponentiation. In *IEEE Computer Security Foundations Symposium (CSF)*, pages 157–171, 2009.
20. Giuseppe Ateniese and Stefan Mangard. A new approach to dns security (dnssec). In *Proceedings of the 8th ACM conference on Computer and Communications Security*, pages 86–95. ACM, 2001.
21. Paul Hoffman and Jakob Schlyter. The dns-based authentication of named entities (dane) transport layer security (tls) protocol: Tlsa. Technical report, 2012.

Automated Verification for Secure Messaging Protocols and their Implementations: A Symbolic and Computational Approach

Nadim Kobeissi
INRIA Paris

Karthikeyan Bhargavan
INRIA Paris

Bruno Blanchet
INRIA Paris

***Abstract*—Many popular web applications incorporate end-to-end secure messaging protocols, which seek to ensure that messages sent between users are kept confidential and authenticated, even if the web application's servers are broken into or otherwise compelled into releasing all their data. Protocols that promise such strong security guarantees should be held up to rigorous analysis, since protocol flaws and implementations bugs can easily lead to real-world attacks.**

We propose a novel methodology that allows protocol designers, implementers, and security analysts to collaboratively verify a protocol using automated tools. The protocol is implemented in ProScript, a new domain-specific language that is designed for writing cryptographic protocol code that can both be executed within JavaScript programs and automatically translated to a readable model in the applied pi calculus. This model can then be analyzed symbolically using ProVerif to find attacks in a variety of threat models. The model can also be used as the basis of a computational proof using CryptoVerif, which reduces the security of the protocol to standard cryptographic assumptions. If ProVerif finds an attack, or if the CryptoVerif proof reveals a weakness, the protocol designer modifies the ProScript protocol code and regenerates the model to enable a new analysis.

We demonstrate our methodology by implementing and analyzing a variant of the popular Signal Protocol with only minor differences. We use ProVerif and CryptoVerif to find new and previously-known weaknesses in the protocol and suggest practical countermeasures. Our ProScript protocol code is incorporated within the current release of Cryptocat, a desktop secure messenger application written in JavaScript. Our results indicate that, with disciplined programming and some verification expertise, the systematic analysis of complex cryptographic web applications is now becoming practical.

1. Introduction

Designing new cryptographic protocols is highly error-prone; even well-studied protocols, such as Transport Layer Security (TLS), have been shown to contain serious protocol flaws found years after their deployment (see e.g. [1]). Despite these dangers, modern web applications often embed custom cryptographic protocols that evolve with each release. The design goal of these protocols is typically to protect user data as it is exchanged over the web and synchronised across devices, while optimizing performance for application-specific messaging patterns and deployment constraints. Such custom protocols deserve close scrutiny, but their formal security analysis faces several challenges.

First, web applications often evolve incrementally in an ad hoc manner, so the embedded cryptographic protocol is only ever fully documented in source code. Even when protocol designers or security researchers take the time to create a clear specification or formal model for the protocol, these documents are typically incomplete and quickly go out-of-date. Finally, even if the protocol itself is proved to be secure, bugs in its implementation can often bypass the intended security guarantees. Hence, it is not only important to extract a model of the protocol from the source code and analyze its security, it is essential to do so in a way that the model can evolve as the application is modified.

In this paper, we study the protocol underlying the Signal messaging application developed by Open Whisper Systems. Variants of this protocol have also been deployed within WhatsApp, Facebook Messenger, Viber, and many other popular applications, reaching over a billion devices. The protocol has been known by other names in the past, including Axolotl, TextSecure (versions 1, 2, and 3), and it continues to evolve within the Signal application under the name Signal Protocol. Until recently, the main documentation for the protocol was its source code, but new specifications for key components of the protocol have now been publicly released.[1]

Signal Protocol has ambitious security goals; it enables asynchronous (zero round-trip) authenticated messaging between users with end-to-end confidentiality. Each message is kept secret even if the messaging server is compromised, and even if the user's long term keys are compromised, as long as these keys are not used by the attacker before the target message is sent (forward and future secrecy.) To achieve these goals, Signal uses a novel authenticated key exchange protocol (based on mixing multiple Diffie-Hellman shared secrets) and a key refresh mechanism (called double ratcheting). The design of these core mechanisms in TextSecure version 2 was cryptographically analyzed in [2] but the protocol has evolved since then and the security of Signal as it is currently implemented and deployed remains an open question.

In fact, although they all implement the same core protocol, different implementations of the Signal protocol

1. https://whispersystems.org/docs/specifications/x3dh/

vary in important details, such as how users are identified and authenticated, how messages are synchronised across devices, etc. We seek to develop and analyze one such variant that was recently incorporated into Cryptocat, a desktop messaging application developed by one of the current authors. We call this variant SP in the rest of this paper. We develop a detailed model for SP in the applied pi calculus and verify it using the ProVerif protocol analyzer [3] for these security goals against adversaries in a classic Dolev-Yao model [4]. We also develop a computational proof for SP using the CryptoVerif prover [5]. There remains the challenge of keeping our models up-to-date as the protocol code evolves within Cryptocat. To this end, we design a model extraction tool that can compile the protocol source code to the applied pi calculus.

Signal has been implemented in various programming languages, but most desktop implementations of Signal, including Cryptocat, are written in JavaScript. Although JavaScript is convenient for widespread deployability, it is not an ideal language for writing security-critical applications. Its permissive, loose typing allows for dangerous implementation bugs and provides little isolation between verified cryptographic protocol code and unverified third-party components. Rather than trying to verify general JavaScript programs, we advocate that security-critical components like SP should be written in a well-behaved subset that enables formal analysis.

We introduce ProScript (short for *"Protocol Script"*), a programming and verification framework tailored specifically for the implementation of cryptographic protocols. ProScript extends Defensive JavaScript (DJS) [6], [7], a static type system for JavaScript that was originally developed to protect security-critical code against untrusted code running in the same origin. ProScript is syntactically a subset of JavaScript, but it imposes a strong coding discipline that ensures that the resulting code is amenable to formal analysis. ProScript programs are mostly self-contained; they cannot call arbitrary third-party libraries, but are given access to carefully implemented (well-typed) libraries such as the ProScript cryptographic library (PSCL). Protocols written in ProScript can be type-checked and then automatically translated into an applied pi calculus [8] model using the ProScript compiler. The resulting model can be analyzed directly through ProVerif and can be adapted and extended to a proof in CryptoVerif. As the code evolves, this model can be automatically refreshed to enable new analyses and proofs, if necessary.

Contributions. We present an outline of our contributions in this paper:

A Security Model and New Attacks. We present security goals and a threat model for secure messaging (§ 2). As a motivation for our verification approach, we discuss protocol weaknesses and implementation bugs in the messaging protocol underlying the popular Telegram application.

Automated Model Extraction from JavaScript. We present the ProScript compiler, which allows for the compilation from a subset of JavaScript into a readable protocol model in the applied pi calculus (§4). Model extraction enables formal verification to keep up with rapidly changing source code. Readable models allow the protocol analyst to experiment with different threat models and security goals and to test new features before including them in the implementation.

A Symbolic Security Analysis of SP. We formalize and analyze a variant of Signal Protocol for a series of security goals, including confidentiality, authenticity, forward secrecy and future secrecy, against a classic symbolic adversary (§5). Our analysis uncovers several weaknesses, including previously unreported replay and key compromise impersonation attacks, and we propose and implement fixes which we then also verify.

A Computational Cryptographic Proof for SP. We present proofs of message authenticity, secrecy and forward secrecy for SP obtained using the CryptoVerif computational model prover [5]. (§6)

A Verified Protocol Core for Cryptocat. We integrate our verified protocol code into the latest version of Cryptocat[2] (§7), a popular open source messaging client with thousands of users that is developed and maintained by one of the authors of this paper. We show how the new architecture of Cryptocat serves to protect the verified protocol code from bugs in the rest of the application.

2. A Security Model for Encrypted Messaging

We consider a simple messaging API as depicted below. An initiator A can start a conversation with B by calling `startSession` with long-term secrets for A and any identity credentials it has for B. This function returns the initial conversation state T_0. Thereafter, the initiator can call `send` with a plaintext message M_1 to obtain the encrypted message E_1 that it needs to send on the network. Or it can call `recv` with an encrypted message E_2 it received (supposedly from B) to obtain the plaintext message M_2.

```
T_0^ab = startSession(secrets_A, identity_B)
T_1^ab, E_1 = send(T_0^ab, M_1)
T_2^ab, M_2 = recv(T_1^ab, E_2)
```

The responder B uses a similar API to accept sessions and receive and send messages:

```
T_0^ba = acceptSession(secrets_B, identity_A)
T_1^ba, M_1 = recv(T_0^ba, E_1)
T_2^ba, E_2 = send(T_1^ba, M_2)
```

We deliberately chose a functional state-passing API with no side-effects in order to focus on cryptographic protocol computations, rather than the concrete details of how these messages are sent over the network.

2.1. Threat Model

While threat models vary for different protocols, we consider the following threats in this paper:

2. https://crypto.cat

- **Untrusted Network** We assume that the attacker controls the network and so can intercept, tamper with and inject network messages (e.g. E_1, E_2). Moreover, if two messaging clients communicate via a server, we typically treat that server as untrusted.
- **Malicious Principals** The attacker controls a set of valid protocol participants (e.g. M), for whom it knows the long-term secrets. The attacker may advertise any identity key for its controlled principals; it may even pretend to own someone else's identity keys.
- **Long-term Key Compromise** The attacker may compromise a particular principal (e.g. A) during or after the protocol, to obtain her long-term secrets.
- **Session State Compromise** The attacker may compromise a principal to obtain the full session state at some intermediate stage of the protocol (e.g. T_1^{ab}).

2.2. Cryptographic Models

Traditionally, symbolic cryptographic models have been particularly suitable for automated protocol analysis. They ignore attacks with negligible probability and assume that each cryptographic function is a perfect black-box. For example, in such models, hash functions never collide and encryption is a message constructor that can only be reversed by decryption. In the *computational model*, cryptographic primitives are functions over bitstrings and their security is specified in terms of probabilities. These models are more precise and closer to those used by cryptographers, but usually do not lend themselves to fully automated proofs. Generally, we will use symbolic models when we are trying to find attacks that rely on logical flaws in the protocol and in its use of cryptographic primitives. We will use computational models when we want to build a cryptographic proof of security, starting from standard cryptographic assumptions.

2.3. Security Goals

We state a series of semi-formal security goals in terms of our generic messaging API. We use the phrase "A sends a message M to B" to mean that A calls $\mathsf{Send}(T, M)$ with a session state T that represents a conversation between A and B. Similarly, we say that "B receives a message M from A" to mean that B obtained M as a result of calling $\mathsf{Recv}(T, E)$ with a session T with A.

Unless otherwise specified, the following security properties assume that both A and B are honest, that is, their long-term secrets have not been compromised. We begin with several variants of authenticity goals:

- **Message Authenticity** If B receives a message M from A, then A must have sent M to B.
- **No Replays** Each message received by B from A corresponds to a unique message sent by A. That is, the attacker must not be able to get a single message sent by A to be accepted twice at B.
- **No Key Compromise Impersonation** Even if the long-term secrets of B are compromised, message authenticity must hold at B. That is, the attacker must not be able to forge a message from A to B.

Our definition of message authenticity covers integrity as well as sender and recipient authentication. Obtaining message authenticity also helps prevent *unknown key share* attacks, where B receives a message M from A, but A sent that message to a different intended recipient C. We define four confidentiality goals:

- **Secrecy** If A sends some secret message M to B, then nobody except A and B can obtain M.
- **Indistinguishability** If A randomly chooses between two messages M_0, M_1 (of the same size) and sends one of them to B, the attacker cannot distinguish (within the constraints of the cryptographic model) which message was sent.
- **Forward Secrecy** If A sends a secret message M to B and if A's and B's long-term secrets are subsequently compromised, the message M remains secret.
- **Future Secrecy** Suppose A sends M in a session state T, then receives N, then sends M'. If the session state T is subsequently compromised, the message M' remains secret.

Some protocols satisfy a weaker notion of forward secrecy, sometimes called *weak* forward secrecy, where an attacker is not allowed to actively tamper with the protocol until they have compromised the long-term keys [9]. Some messaging protocols also seek to satisfy more specific authenticity and confidentiality goals, such as non-repudiation and plausible deniability. We will ignore them in this paper.

In the next section, we evaluate two secure messaging applications against these goals, we find that they fail some of these goals due to subtle implementation bugs and protocol flaws. Hence, we advocate the use of automated verification tools to find such attacks and to prevent their occurrence.

3. Analyzing Real-World Messaging Protocols

Modern messaging and transport protocols share several distinctive features [10]: for example, Signal Protocol, SCIMP, QUIC and TLS 1.3 share a strong focus on asynchronous key agreement with a minimum of round trips. Some also guarantee new security goals such as future secrecy. The protocols also assume non-standard (but arguably more user-friendly) authentication infrastructures such as Trust-on-First-Use (TOFU). Modern messaging protocols have several interesting features and properties that set them apart from classic cryptographic protocols and upon which we focus our formal verification efforts:

New Messaging Patterns. In contrast to connection-oriented protocols, modern cryptographic protocols are constrained by new communication flows such as zero-round-trip connections and asynchronous messaging, where the peer may not even be present.

Confidentiality Against Strong Adversaries. Web messaging protocols need to be robust against server compromise

and device theft and so seek to provide strong and novel forward secrecy guarantees.

Weak Authentication Frameworks. Many of these protocols do not rely on public key infrastructures. Instead they may authenticate peers on a TOFU basis or even let peers remain anonymous, authenticating only the shared connection parameters.

Code First, Specify Later. Unlike Internet protocols, which are designed in committee, these protocols are first deployed in code and hand-tuned for performance on a particular platform. The code often remains the definitive protocol specification.

Before outlining our verification approach for such protocols, we take a closer look at two messaging applications: Telegram and Cryptocat.

3.1. Secret Chats in Telegram

Our first example is the "MTProto" [11] secure messaging protocol used in the Telegram messaging application. We focus on the "secret chat" feature that allows two Telegram clients who have already authenticated themselves to the server to start an encrypted conversation with each other. Although all messages pass through the Telegram server, the server is untrusted and should not be able to decrypt these messages. The two clients A and B download Diffie-Hellman parameters from the Telegram server and then generate and send their public values to each other.

The key exchange is not authenticated with long-term credentials. Instead, the two clients are expected to communicate out-of-band and compare a SHA-1 hash (truncated to 128-bits) of the Diffie-Hellman shared secret. If two users perform this authentication step, the protocol promises that messages between them are authentic, confidential, and forward secret, even if the Telegram server is compromised. However this guarantee crucially relies on several cryptographic assumptions, which may be broken either due to implementation bugs or computationally powerful adversaries, as we describe below.

Malicious Primes. MTProto relies on clients checking that the Diffie-Hellman configuration (p, g) that they received from the server is suitable for cryptographic use. The specification requires that p be a large safe prime; hence the client must check that it has exactly 2048 bits and that both p and $(p-1)/2$ are prime, using 15 rounds of the Miller-Rabin primality test. There are several problems with this check. First, the server may be able to carefully craft a non-prime that passes 15 rounds of Miller-Rabin. Second, checking primality is not enough to guarantee that the discrete log problem will be hard. If the prime is chosen such that it has "low weight", the SNFS algorithm applies, making discrete logs significantly more practical [12]. Even if we accept that primality checking may be adequate, it is unnecessary for an application like Telegram, which could simply mandate the use of well-known large primes instead [13].

Public Values in Small Subgroups. A man-in-the-middle can send to both A and B public Diffie-Hellman values g^b and g^a equal to 1 (resp. 0, resp. $p-1$). Both A and B would then compute the shared secret as $g^{ab} = 1$ (resp. 0, resp. 1 or -1). Since their key hashes match, A and B think they have a confidential channel. However, the attacker can read and tamper with all of their messages. More generally, MTProto relies on both peers verifying that the received Diffie-Hellman public values do not fall in small subgroups. This check is adequate to prevent the above attack but could be made unnecessary if the two public values were to be authenticated along with the shared secret in the hash compared by the two peers.

Implementation Bugs in Telegram for Windows. The above two weaknesses, reported for the first time in this paper, can result in attacks if the protocol is not implemented correctly. We inspected the source code for Telegram on different platforms; while most versions perform the required checks, we found that the source code for Telegram for Windows Phone did not check the size of the received prime, nor did it validate the received Diffie-Hellman values against 1, 0 or $p-1$. We reported both bugs to the developers, who acknowledged them and awarded us a bug bounty.

Such bugs and their consequent attacks are due to missed security-relevant checks, and they can be found automatically by symbolic analysis. For example, [14] shows how to model unsafe (malicious) primes and invalid public keys in ProVerif and uses this model to find vulnerabilities in several protocols that fail to validate Diffie-Hellman groups or public keys.

MTProto is also known to other cryptographic weaknesses [15], [16]. How can we be sure that there are no other protocol flaws or implementation bugs hiding in MTProto? Any such guarantee would require a systematic security analysis of both the protocol and the source code against both symbolic and computational adversaries.

3.2. A New Protocol for Cryptocat

Cryptocat is a secure messaging application that is written in JavaScript and deployed as a desktop web application. Earlier versions of Cryptocat implement a variant of the OTR (Off-The-Record) messaging protocol [17] which suffers from several shortcomings. It does not support asynchronous messaging, so both peers have to be online to be able to message each other. It does not support multiple devices or encrypted file transfer. OTR also uses legacy cryptographic constructions like DSA signatures and prime-field Diffie-Hellman, which are slower and less secure than more modern alternatives based on elliptic curves. Furthermore, Cryptocat peers did not have long-term identities and so the authentication guarantees are weak. Early version of Cryptocat suffered from many high-profile implementation bugs, including the reuse of initialization vectors for file encryption [18], bad random number generation, and a classic JavaScript type flaw that resulted in a private key of 255 bits being coerced into a string that held only 55 bits. Some of these implementation flaws would have been found using a static type checker, others required deeper analysis.

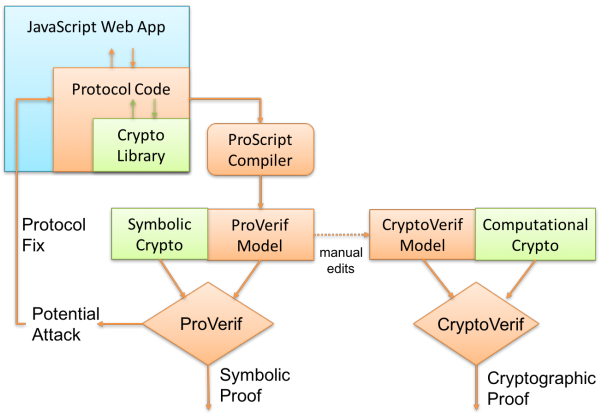


Figure 1: Verification Approach. A ProVerif model is automatically extracted from ProScript protocol code and analyzed for its security goals against a symbolic attacker. The model is then edited by hand and extended with cryptographic assumptions and intermediate lemmas to build a computational proof that is verified by CryptoVerif.

Cryptocat was recently rewritten from scratch to upgrade both its messaging protocol and its implementation. The goal of this redesign was to isolate its protocol core and replace it with a verified messaging protocol written in a statically typed subset of JavaScript.

3.3. Towards Automated Verification

The innovative designs and unusual security guarantees of secure messaging protocols demand formal security analysis. Hand-written models with detailed cryptographic proofs can be useful as a reference, but we observe that the most recent analysis of Signal Protocol [2] is already out of date, as the protocols have moved on to new versions. Furthermore, manual cryptographic proofs often leave out details of the protocol for simplicity and some of these details (e.g. client authentication) may lead to new attacks. In this paper, we advocate the use of automated verification tools to enable the analysis of complex protocols as they evolve and incorporate new features. Moreover, we would also like to find protocol implementation bugs (like the ones in previous versions of Telegram and Cryptocat) automatically.

We advocate the verification approach depicted in Figure 1. The messaging application is written in JavaScript and is broken down into a cryptographic protocol core and untrusted application code that interact through a small well-typed API that hides all protocol secrets within the protocol core and only offers a simple send/receive functionality to the application. Notably, the protocol core is written in a domain-specific language and does not rely on any external libraries except for a well-vetted cryptographic library. The protocol code can be translated to an applied pi calculus model and symbolically analyzed in ProVerif to find protocol flaws and attacks. The model can also be used as the starting point for a cryptographic proof for the protocol developed using CryptoVerif.

In the rest of this paper, we show how we applied this verification methodology to systematically analyze a variant of the Signal protocol, called SP, that is implemented in the new version of Cryptocat.

4. ProScript: A Language for Protocol Implementation

ProScript aims to be an ideal language for reliably implementing cryptographic protocols for web applications. Using ProScript, a protocol designer or implementer can implement a protocol, automatically extract a formal model from the code, verify the model using ProVerif, and then run the protocol code within a JavaScript web application. The ProScript framework does not target general JavaScript code, however existing applications can be adapted to use ProScript for their security-critical protocol components.

Our goal is to allow the developer to go back and forth between their protocol implementation and the ProVerif model, in order to help understand the behavior being illustrated, the properties being verified and how detected attacks, if any, relate to their source code. For these reasons, we pay special attention to generating models that are optimized both for verifiability as well as readability. This increases their utility to a human examiner who may wish to independently expand the model to include more specific process flows or to explore variations of the protocol against a manually defined adversary.

Syntactically, ProScript is a subset of JavaScript that can be naturally translated to the applied pi calculus. This restriction produces casualties, including recursion, `for` loops and extensible objects. A closer look at the ProScript syntax shows JavaScript employed in a particular style to bring out useful features:

Isolation. ProScript is based on Defensive JavaScript (DJS) [6], [7], a typed subset of JavaScript which focuses on protecting security-critical components from malicious JavaScript code running in the same environment. DJS imposes a strict typing discipline in order to eliminate language-based attacks like prototype poisoning. In particular, it forbids the use of unknown external libraries as well as calls to tamperable object methods such as `.toString()`. It also forbids extensible objects and arrays and prevents any access to object prototypes. These restrictions result in protocol implementations that are more robust and less influenced by the runtime environment. The ProScript type-checker builds on and extends DJS and hence, inherits both its language restrictions and isolation guarantees.

Type Declarations and Inference. ProScript requires all variables and functions to be declared before they are used, hence imposing a strict scoping discipline. For example, an expression $v.x$ is well-typed if and only if v has been defined, as a local or global variable, to be an object with a property x. As an extension to the builtin types of DJS, ProScript allows type declarations for commonly used protocol

data structures. For example, an array of 32 hexadecimal integers can be declared as a key type. The ProScript compiler recognizes such type declarations and uses them to translate the code into more concise and informative ProVerif models. Moreover, the typechecker can automatically infer fine-grained sub-types. For example, ProScript differentiates between numbers declared using decimal literals (ex. `128`) and hexadecimal literals (ex. `0x80`). Numbers defined using hexadecimal are sub-typed as bytes. This feature allows us to track how numerical values are employed in the protocol, and prevents type coercion bugs similar to an actual bug that we describe in §3.2, where a significant loss of entropy was caused by a byte being coerced into a decimal value.

State-Passing Functional Style. ProScript's syntax takes advantage of JavaScript's functional programming features in order to encourage and facilitate purely functional protocol descriptions, which the compiler can translate into symbolically verifiable, human-readable models in the applied pi calculus. The functional style encourages the construction of state-passing functions, leaving state modification up to the unverified application outside of the ProScript code. The majority of a ProScript implementation tends to be a series of pure function declarations. A small subset of these functions is exposed to a global namespace for access by the verified application while most remain hidden as utility functions for purposes such as key derivation, decryption and so on. This state-passing style is in contrast to DJS that allows direct modification of heap data structures. The functional style of ProScript allows protocol data structures, including objects and arrays, to be translated to simple terms in ProVerif built using constructors and destructors, hence avoiding the state-space explosion inherent in the heap-based approach that is needed to translate DJS to ProVerif [7].

4.1. ProScript Syntax

A ProScript implementation consists of a series of *modules*, each containing a sequence of type declarations (containing constructors, assertion utilities, and type converters), constant declarations and function declarations.

ProScript

$v ::=$	values
x	variables
n	numbers
s	strings
`true, false`	booleans
`undefined, null`	predefined constants
$e ::=$	expressions
v	values
$\{x_1 : v_1, \ldots, x_n : v_n\}$	object literals
$v.x$	field access
$[v_1, \ldots, v_n]$	array literals
$v[n]$	array access
`Lib.`$l(v_1, \ldots, v_n)$	library call
$f(v_1, \ldots, v_n)$	function call
$\sigma ::=$	statements
`var` $x; \sigma$	variable declaration
$x = e; \sigma$	variable assignment
`const` $x = e; \sigma$	constant declaration
`if` $(v_1 === v_2)$ $\{\sigma_1\}$ `else` $\{\sigma_2\}$	if-then-else
`return` e	return
$\gamma ::=$	globals
`const` $x = e$	constants
`const` $f =$ `function`$(x_1, \ldots, x_n)\{\sigma\}$	functions
`const Type_`$x = \{\ldots\}$	user types
$\mu ::= \gamma_0; \ldots; \gamma_n$	modules

Note that we will use the defined Lib.l notation to access the ProScript Cryptography Library.

Operational Semantics. ProScript's operational semantics is a subset of JavaScript, and both run on JavaScript interpreters. It is tooled based on the formal semantics of Maffeis et al. [19] and is superficially adapted for our language subset.

4.2. ProVerif Syntax

A ProVerif script Σ is divided into two major parts:

1) $\Delta_1 \ldots \Delta_n$, a sequence of declarations which encapsulates all types, free names, queries, constructors, destructors, equations, pure functions and processes. Queries define the security properties to prove. Destructors and equations define the properties of cryptographic primitives.
2) P, the top-level process which then effectively employs $\Delta_1 \ldots \Delta_n$ as its toolkit for constructing a process flow for the protocol.

In processes, the replication $!P$ represents an unbounded number of copies of P in parallel. Tables store persistent state: The process `insert` $a(M_1, \ldots, M_n); P$ inserts the entry $(M_1, \ldots, M_n)$ in table a, and runs P. The process `get` $a(=M_1, x_2, \ldots, x_n)$ `in` P looks for an entry $(N_1, \ldots, N_n)$ in table a such that $N_1 = M_1$. When such an entry is found, it binds $x_2, \ldots, x_n$ to $N_2, \ldots, N_n$ respectively and runs P. Events are used for recording that certain actions happen (e.g. a message was sent or received), in order to use that information for defining security properties. Phases model a global synchronization: processes initially run in phase 0; then at some point processes of phase 0 stop and processes of phase 1 run and so on. For instance, the protocol may run in phase 0 and some keys may be compromised after the protocol run by giving them to the adversary in phase 1.

ProVerif

$M ::=$	terms
v	values
a	names
$f(M_1, \ldots, M_n)$	function application
$E ::=$	enriched terms
M	return value
`new` $a : \tau; E$	new name a of type τ
`let` $x = M$ `in` E	variable definition
`if` $M = N$ `then` E_1 `else` E_2	if-then-else

$P, Q ::=$ processes
- 0 — null process
- $\texttt{in}(M, x : \tau); P$ — input x from channel M
- $\texttt{out}(M, N); P$ — output N on channel M
- $\texttt{let}\ x = M\ \texttt{in}\ P$ — variable definition
- $P \mid Q$ — parallel composition
- $!\,P$ — replication of P
- $\texttt{insert}\ a(M_1, \ldots, M_n); P$ — insert into table a
- $\texttt{get}\ a(=M_1, x_2, \ldots, x_n)\ \texttt{in}\ P$ — get table entry specified by M_1
- $\texttt{event}\ M; P$ — event M
- $\texttt{phase}\ n; P$ — enter phase n

$\Delta ::=$ declaration
- $\texttt{type}\ \tau$ — type τ
- $\texttt{free}\ a : \tau$ — name a
- $\texttt{query}\ q$ — query q
- $\texttt{table}\ a(\tau_1, \ldots, \tau_n)$ — table a
- $\texttt{fun}\ C(\tau_1, \ldots, \tau_n) : \tau$ — constructor
- $\texttt{reduc forall}\ x_1 : \tau_1, \ldots, x_n : \tau_n; f(M_1, \ldots, M_n) = M$ — destructor
- $\texttt{equation forall}\ x_1 : \tau_1, \ldots, x_n : \tau_n; M = M'$ — equation
- $\texttt{letfun}\ f(x_1 : \tau_1, \ldots, x_n : \tau_n) = E$ — pure function
- $\texttt{let}\ p(x_1 : \tau_1, \ldots, x_n : \tau_n) = P$ — process

$\Sigma ::= \Delta_1 \ldots \Delta_n.\texttt{process}\ P$ script

4.3. Translation

Within Σ, ProScript functions are translated into ProVerif pure functions. Type declarations are translated into ProVerif type declarations. Individual values, such as strings and numbers, are declared as global constants at the top-level scope of the ProVerif model with identifiers that are then employed throughout the model when appropriate. Objects and Arrays are instantiated in the model using functions, with destructors automatically generated in order to act as getters.

Translation Rules

$M_v ::= v \mid \{x_1 : v_1, \ldots, x_n : v_n\} \mid [v_1, \ldots, v_n]$

$\mathcal{V}[\![M_v]\!] \to M$ Values to Terms

$\mathcal{V}[\![v]\!] = v$

$\mathcal{V}[\![\{x_1 : v_1, \ldots, x_n : v_n\}]\!] = \texttt{Obj\_t}(v_1, \ldots, v_n)$

$\mathcal{V}[\![[v_1, \ldots, v_n]]\!] = \texttt{Arr\_t}(v_1, \ldots, v_n)$

$\mathcal{E}[\![e]\!] \to M$ Expressions to Terms

$\mathcal{E}[\![M_v]\!] = \mathcal{V}[\![M_v]\!]$

$\mathcal{E}[\![v.x]\!] = \texttt{get\_x}(v)$

$\mathcal{E}[\![v[i]]\!] = \texttt{get\_i}(v)$

$\mathcal{E}[\![\texttt{Lib}.l(v_1, \ldots, v_n)]\!] = \texttt{Lib\_l}(\mathcal{V}[\![v_1]\!], \ldots, \mathcal{V}[\![v_n]\!])$

$\mathcal{E}[\![f(v_1, \ldots, v_n)]\!] = f(\mathcal{V}[\![v_1]\!], \ldots, \mathcal{V}[\![v_n]\!])$

$\mathcal{S}[\![\sigma]\!] \to E$ Statements to Enriched Terms

$\mathcal{S}[\![\texttt{var}\ x; \sigma]\!] = \mathcal{S}[\![\sigma]\!]$

$\mathcal{S}[\![x = e; \sigma]\!] = \texttt{let}\ x = \mathcal{E}[\![e]\!]\ \texttt{in}\ \mathcal{S}[\![\sigma]\!]$

$\mathcal{S}[\![\texttt{const}\ x = e; \sigma]\!] = \texttt{let}\ x = \mathcal{E}[\![e]\!]\ \texttt{in}\ \mathcal{S}[\![\sigma]\!]$

$\mathcal{S}[\![\texttt{return}\ v]\!] = \mathcal{V}[\![v]\!]$

$\mathcal{S}[\![\texttt{if}\ (v_1 === v_2)\ \{\sigma_1\}\ \texttt{else}\ \{\sigma_2\}]\!] =$
$\quad \texttt{if}\ \mathcal{V}[\![v_1]\!] = \mathcal{V}[\![v_2]\!]\ \texttt{then}\ \mathcal{S}[\![\sigma_1]\!]\ \texttt{else}\ \mathcal{S}[\![\sigma_2]\!]$

$\mathcal{F}[\![\gamma]\!] \to \Delta$ Types and Functions to Declarations

$\mathcal{F}[\![\texttt{const}\ f = \texttt{function}(x_1, \ldots, x_n)\{\sigma\}]\!] =$
$\quad \texttt{letfun}\ f(x_1, \ldots, x_n) = \mathcal{S}[\![\sigma]\!]$

$\mathcal{F}[\![\texttt{const Type\_}t = \{\ldots\}]\!] = \texttt{type}\ t$

$\mathcal{C}[\![\mu]\!](P) \to P$ Constants to Top-level Process

$\mathcal{C}[\![\epsilon]\!](P) = P$

$\mathcal{C}[\![\texttt{const}\ x = e; \mu]\!](P) = \texttt{let}\ x = \mathcal{E}[\![e]\!]\ \texttt{in}\ \mathcal{C}[\![\mu]\!](P)$

$\mathcal{M}[\![\mu]\!](P) \to \Sigma$ Modules to Scripts

$\mathcal{M}[\![\mu]\!](P) = \mathcal{F}[\![\gamma_1]\!] \ldots . \mathcal{F}[\![\gamma_n]\!] . \mathcal{C}[\![\mu_c]\!](P)$
where μ_c contains all globals $\texttt{const}\ x = e$ in μ
and $\gamma_1, \ldots, \gamma_n$ are the other globals of μ.

Translation Soundness. We currently do not formally prove translation soundness, so proofs of the resulting ProVerif model do not necessarily imply proof of the source code. Instead, we use model translation as a pragmatic tool to automatically generate readable protocol models faithful to the implementation, and to find bugs in the implementation. We have experimented with multiple protocols written in ProScript, including OTR, SP, and TLS 1.3, and by carefully inspecting the source code and target models, we find that the compiler is quite reliable and that it generates models that are not so far from what one would want to write directly in ProVerif. In future work, we plan to prove the soundness of this translation to get stronger positive guarantees from the verification. To this end, we observe that our source language is a simply-typed functional programming language, and hence we should be able to closely follow the methodology of [20].

Generating Top-Level Processes. We are also able to automatically generate top-level ProVerif processes. Aiming to implement this in a way that allows us to easily integrate ProScript code into existing codebases, we decided to describe top-level functions inside `module.exports`, the export namespace used by modules for Node.js [21], a popular client/server run-time for JavaScript applications (based on the V8 engine). This makes intuitive sense: `module.exports` is used specifically in order to define the functions of a Node.js module that should be available to the external namespace once that module is loaded, and executing all this functionality in parallel can give us a reasonable model of a potential attacker process. Therefore, functions declared in this namespace will be translated into top-level processes executed in parallel. We use ProVerif `tables` in order to manage persistent state between these parallel processes: each process fetches the current state from a table, runs a top-level function in that state, and stores the updated state returned by the function in the table.

4.4. Trusted Libraries for ProScript

Protocol implementations in ProScript rely on a few trusted libraries, in particular, for cryptographic primitives and for encoding and decoding protocol messages.

When deployed in Node.js or within a browser, the protocol code may have access to native cryptographic APIs. However, these APIs do not typically provide all modern cryptographic primitives; for example, the W3C Web Cryptography API does not support Curve25519, which is needed in Signal. Consequently, implementations like Signal Messenger end up compiling cryptographic primitives from C to JavaScript. Even if the desired primitives were available in the underlying platform, accessing them in a hostile environment is unsafe, since an attacker may have redefined them. Consequently, we developed our own libraries for cryptography and message encoding.

The ProScript Cryptography Library (PSCL) is a trusted cryptographic library implementing a variety of modern cryptographic primitives such as X25519, AES-CCM and BLAKE2. All of its primitives are fully type-checked without this affecting speed: in the majority of our benchmarks, PSCL is as fast as or faster than popular JavaScript cryptographic libraries like SJCL and MSR JavaScript Crypto, which do not even benefit from defensive type checking

More crucially, PSCL functions used in ProScript code are detected by the ProScript compiler as it produces the applied pi model of the implementation, giving it the ability to convert each call to a cryptographic primitive to a call to the corresponding symbolic function in ProVerif. For example, if the ProScript compiler sees a call to PSCL's X25519 implementation, it will automatically translate it to a standard Diffie-Hellman construction in ProVerif.

5. Implementing and Verifying SP

We describe SP, a variant of Signal Protocol that closely follows TextSecure version 3. We show how we implement and verify this protocol in our framework.

5.1. Protocol Overview

In SP, as illustrated in Figure 2, each client publishes a long-term Diffie-Hellman public key and a set of ephemeral Diffie-Hellman public keys (called *"pre-keys"*). These keys include both signed pre-keys, which can be reused for some period of time, and non-signed, one-time pre-keys, which are fresh at each session. To send a message to Bob, Alice retrieves Bob's long-term keys $(g^{b_{3dh}}, g^{b_{sig}})$, a signed pre-key g^{b_s} and a one-time pre-key g^{b_o}. She then chooses her own ephemeral g^{a_e}. A four-way Diffie-Hellman handshake is accomplished using Alice and Bob's long-term identity keys and their short-term ephemeral keys in order to derive the session secret S. The one-time pre-key is optional: when there remains no available one-time pre-key, the exchange is performed with a triple Diffie-Hellman handshake. An encryption key, k_{enc}, is then derived from S by Hash-Based Key Derivation (HKDF) [22] and the message M_0 is sent encrypted under the authenticated encryption scheme AES-GCM, with public and ephemeral keys as associated data: $\mathsf{ENC}(k, m, ad)$ means that m is encrypted with k and both the message m and the associated data ad are authenticated. Subsequent messages in the conversation obtain authentication by chaining to S via a forward-ratcheting construction that also employs HKDF. Each sent message includes its own newly generated ephemeral public key and the protocol's double ratchet key refresh mechanism manages the key state by advancing key chaining with every message.

SP's forward and future secrecy goals are intended to make it so that the compromise of Alice or Bob's long-term keys allows for their impersonation but not for the decryption of their messages. The use of a signed initial ephemeral pre-key results in weaker forward secrecy guarantees for the first flight of messages from A to B: no forward secrecy is provided if both the long-term keys and pre-keys are leaked, although the guarantees for subsequent flights remain strong. If pre-keys are not signed, then the protocol only offers weak forward secrecy with respect to long-term key leakage. We note that the term "forward secrecy" can be confusing in a protocol like Signal, because a number of keys are at stake: long-term keys $((a_{3dh}, a_{sig}), (b_{3dh}, b_{sig}))$, signed pre-key b_s, one-time pre-key b_o, ephemeral keys $(a_e, a_{e'}, b_{e'})$, root keys $(rk_{ab}, ck_{ab}, rk_{ba}, ck_{ba})$ and message keys (k_{enc}). Any formal analysis of the protocol must precisely state which of these keys can be compromised and when.

Differences from other versions of Signal. An earlier version of the protocol, TextSecure Version 2, was cryptographically analyzed in previous work [2]. There are two key differences between SP and TextSecure Version 2.

Signed, Time-Based Pre-Keys. Version 2 uses a triple Diffie-Hellman handshake with one of a hundred pre-keys that Bob stores on the server (including a "last-resort" pre-key). TextSecure Version 3 and all subsequent versions of Signal, including SP, use a signed time-based pre-key, used in conjunction with an unsigned one-time pre-key in case it is available. Bob periodically replaces his signed pre-key (for example, once every week), which may be re-used until its replacement and refreshes his collection of unsigned one-time pre-keys. In SP, when one-time pre-keys are exhausted, no "last-resort" pre-key is used.

Stronger Identity Protection. Since Version 3, tag_n is expanded to include the long-term identities of the sender and recipient, which is not the case in Version 2. This provides a slightly stronger authentication guarantee in the rare case that the encryption keys for different pairs of users turns out to be the same.

In addition to these differences with Version 2, SP also differs from other variants of Signal in one key aspect. In SP, long-term identities are split into one Diffie-Hellman key pair and one signing key pair. In Signal, the same key pair is used for both operations, by applying an elliptic curve conversion from the Montgomery curve-based Diffie-Hellman key pair to obtain its twisted Edwards curve-based

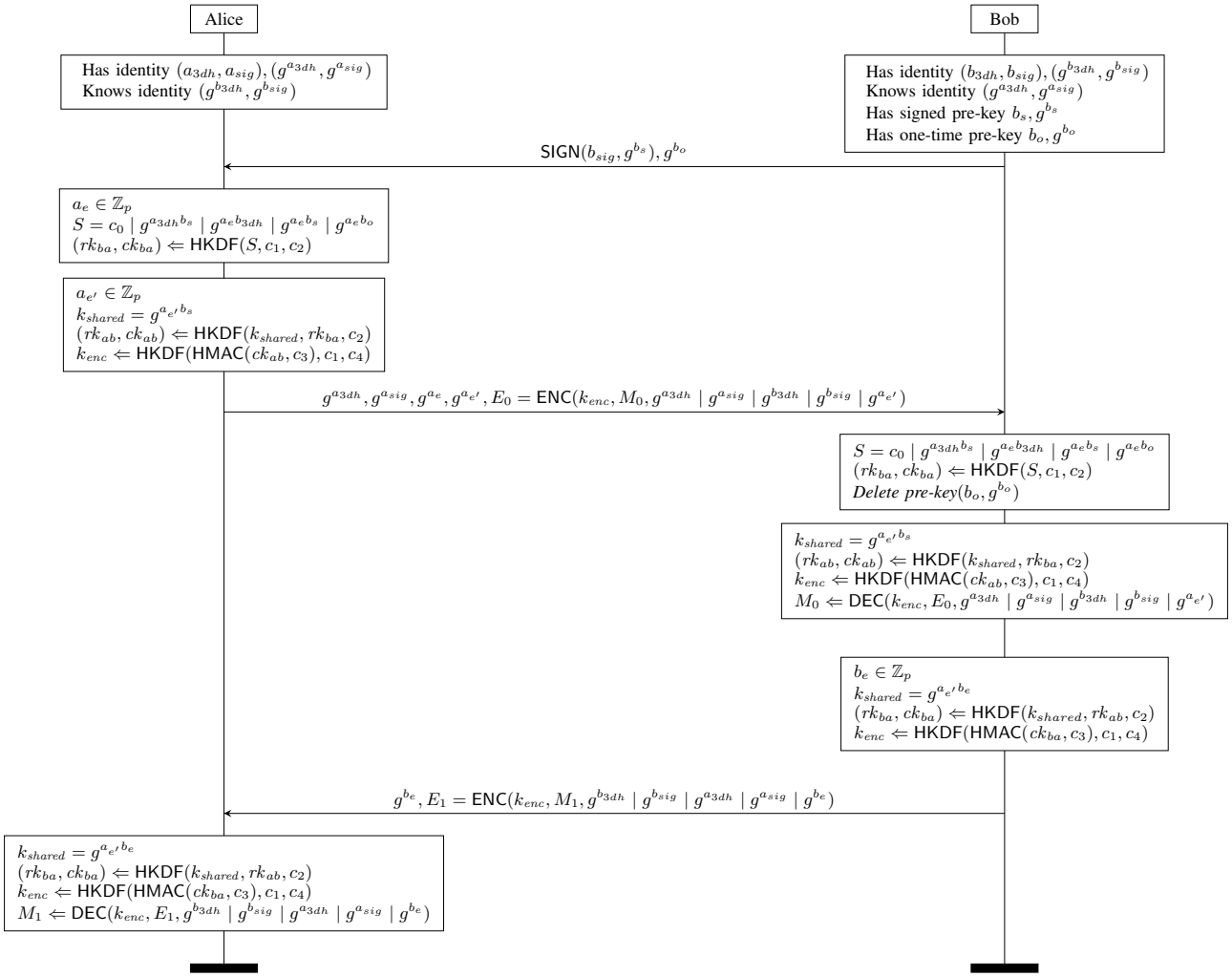


Figure 2: SP, a variant of Signal with minor differences. Alice requests a signed pre-key from Bob (via the server) and sends an initial message M_0. Bob accomplishes his side of the key exchange and obtains M_0. Bob later sends his reply M_1, illustrating the Axolotl ratchet post-AKE. We ignore the hash-based ratchet that occurs when two consecutive messages are sent in the same direction. c_i refers to various constants found throughout the protocol.

signing key pair equivalent. We choose to use separate keys instead, because in our cryptographic proof, we do not want to add a non-standard cryptographic assumption about the use of the same key in two independent cryptographic operations. In exchange for using standard cryptographic assumptions, we consider the cost of adding an extra 32 byte key to the protocol to be acceptable.

5.2. Protocol Implementation

To implement SP in ProScript, we must first deconstruct it into various protocol components: structures for managing keys and user states, messaging functions, APIs and top-level processes. ProScript is well-equipped to handle these protocol components in a way that lends itself to model extraction and verification. We break down our ProScript SP implementation into:

Types for State and Key Management. ProScript's type declaration syntax can be used to declare types for individual elements such as encryption keys but also for collections of elements such as a conversation party's state. These declarations allow for the construction of common data structures used in the protocol and also makes their management and modification easier in the extracted ProVerif models.

Messaging Interface. The ProScript implementation exposes the generic messaging API in a single global object. All interface access provides purely state-passing functionality.

Goals	Messages	Parties	Roles	Time
Secrecy	1	A, B	One	00h.04m.07s.
Secrecy	1	A, B	Two	00h.11m.17s.
Indist.	1	A, B	One	02h.06m.15s.
Authen.	1	A, B, M	One	00h.58m.19s.
Authen.	1	A, B, M	Two	29h.17m.39s.
Fo. Se.	1	A, B	One	00h.04m.14s.
KCI	1	A, B	One	00h.19m.20s.

Figure 3: Verification times for SP ProVerif models.

Long-Term and Session States. Protocol functions take long-term and session states (S^a, T_n^{ab}) as input and return T_{n+1}^{ab}. S^a contains long-term values such identity keys, while T includes more session-dependent values such as `ephemerals`, containing the current ephemeral and chaining keys for the session context and `status`, indicating whether the application layer should perform a state update.

Internal Functions. Utility functionality, such as key derivation, can also be described as a series of pure functions that are not included in the globally accessible interface.

Top-Level Process. A top-level process can serve as a harness for testing the proper functioning of the protocol in the application layer. Afterwards, when this top-level process is described in the extracted ProVerif model, the implementer will be able to use it to define which events and security properties to query for.

Inferred Types in ProScript. ProScript type declarations allow for the easier maintenance of a type-checkable protocol implementation, while also allowing the ProScript compiler to translate declared types into the extracted ProVerif model. Defining a `key` as an array of 32 bytes will allow the ProScript compiler to detect all 32 byte arrays in the implementation as keys and type their usage accordingly.

5.3. Protocol Verification

We use ProVerif to verify the security goals of our extracted model by defining defining queries that accurately test the resilience of security properties against an active adversary. Under an active Dolev-Yao adversary, ProVerif was able to verify confidentiality, authenticity, forward secrecy and future secrecy for Alice and Bob initializing a session and exchanging two secret messages, with a compromised participant, Mallory, also being allowed to initialize sessions and exchange non-secret messages with Alice and Bob. Our analysis revealed two novel attacks: a key compromise impersonation attack and a replay attack, for which we propose a fix. Aside from these attacks, we were also able to model the previously documented Unknown Keyshare Attack [2].

Extracts of our compiled SP implementation are available online [23]. Models begin with type declarations followed by public constant declarations, equational relationships for cryptographic primitives, protocol functions, queries and relevant names and finally the top-level process with its associated queries.

The top-level process queries for security properties such as confidentiality, authenticity and forward secrecy between two roles: an initiator (e.g. Alice) who sends an initial message and thereby initializes an authenticated key exchange, and a responder (e.g. Bob) who receives the message and who may send a response. Some models include a third compromised identity, Mallory, who also communicates with Alice and Bob but while leaking her private keys to the attacker beforehand. In some instances, we also model parallel process executions where each identity (Alice, Bob and optionally Mallory) assumes both the role of the initiator and the responder. We informally call this latter scenario a "two-way role" model.

Secrecy and Indistinguishability. For every message considered in our protocol model, we define a secret constant M_n where M_1 is the initial message in a session. These secret values are then used as the plaintext for the encrypted messages sent by the principals. We show that an active attacker cannot retrieve a message's plaintext M_n using the query:

$$\mathsf{query}(\mathsf{attacker}(M_n)) \tag{1}$$

Similarly, we show indistinguishability using the query $\mathsf{query}(\mathsf{noninterf}(M_n))$.

Forward and Future Secrecy. We examine forward and future secrecy in Signal Protocol in multiple scenarios: the compromise of long-term keys and the compromise of message keys in two different types of message flights. In these scenarios, we need to model that keys are leaked after sending or receiving certain messages. We rely on ProVerif *phases* for that: intuitively, t represents a global clock, and processes occurring after the declaration of a phase t are active only during this phase.

We show that message M_1 remains secret by query (1) even if the long-term keys $(a_{3dh}, a_{sig}, b_{3dh}, b_{sig})$ are leaked after sending M_1. Furthermore, we can modify our ProVerif model to produce a sanity check: if responder Bob skips the signature check on g^{a_s}, ProVerif shows that an active attacker becomes capable of violating this forward secrecy property.

Next, we examine two different messaging patterns in the Double Ratchet algorithm and find that they approach forward and future secrecy differently:

- **Single-Flight Pattern.** In this scenario, Alice sends Bob a number of messages M_n and M_{n+1} where $n > 1$ and does not receive a response. In this scenario, Bob's lack of response does not allow Alice to obtain a fresh ephemeral key share g^{b_e} required to establish a new k_{shared} in T_{n+1}^{ab} to be used for M_{n+1}, so Alice just updates the key ck_{ab} by hashing it. If Alice's session state T_{n+1}^{ab}, (which, recall, contains ${a_e}^{n+1}$ and (rk_{ab}, ck_{ab}) for M_{n+1}), is leaked, then M_n remains secret (forward secrecy). Obviously, to take advantage of this property in case of compromise,

the keys (rk_{ab}, ck_{ab}) for M_n must have been appropriately deleted, which is delicate when messages are received out-of-order: if $M_{n_1}, \ldots, M_{n_k}$ $(n_1 < \ldots < n_k)$ have been received, the receiver should keep the chaining key ck_{ab} for M_{n_k+1} and the encryption keys k_{enc} for the messages M_i not received yet with $i < n_k$. If T_n^{ab} is leaked, then M_{n+1} is not secret, so no future secrecy is obtained.

- **Message-Response Pattern.** In this scenario, Alice sends Bob a single message M_n where $n > 1$ and receives a response M_{n+1} before sending M_{n+2}. Upon receiving M_{n+1}, Alice will be able to derive a fresh $k_{shared} = g^{{a_e}^{n+2} {b_e}^{n+1}}$. As a result, if T_{n+2}^{ab} is leaked, then M_n remains secret (forward secrecy) and if T_n^{ab} is leaked after M_{n+1} is received, then M_{n+2} remains secret (future secrecy).

Message Authenticity. Signal Protocol relies on a Trust-on-First-Use (TOFU) authentication model: Alice assumes that Bob's advertised identity key is authenticated and untampered with and employs it as such until an event causes the trust of the key to be put in question, such as a sudden identity key change or an out of band verification failure. We model TOFU by embedding Alice and Bob's identity keys into each other's initial states. We are then free to model for message authenticity: informally, if B receives a message M from A, we want A to have sent M to B. In ProVerif, we can specify two events: $\mathsf{Send}(A, B, M)$, which means that A sends M to B and $\mathsf{Recv}(A, B, M)$, which means that B receives M from A. We can then formalize the correspondence

$$\mathsf{event}(\mathsf{Recv}(A, B, M)) \Longrightarrow \mathsf{event}(\mathsf{Send}(A, B, M)) \quad (2)$$

which checks if for all $\mathsf{Recv}(A, B, M)$ events, it must be the case that a $\mathsf{Send}(A, B, M)$ event has also been executed.

ProVerif succeeds in proving correspondence (2) using public keys A and B. While this implies the desired property when the relation between the public keys and the identity of the principals is bijective, a limitation of this approach is that the identities of the principals are only expressed in terms of keys and not as a more personally-linked element, such as for example a phone number. Therefore, we cannot formally express stronger identity binding as part of the protocol model. This point leads to the Unknown Key Share Attack first reported for Signal Protocol Version 2 [2]: if an adversary can register the public keys $(g^{b_{3dh}}, g^{b_{sig}})$ of B as public keys of C and A sends a message to C, then C can forward this message to B and B will accept it as coming from A, since B and C have the same public keys.

No Replays. This property is similar to message authenticity, but uses an injective correspondence instead, which means that each execution of $\mathsf{Recv}(A, B, M)$ corresponds to a distinct execution of $\mathsf{Send}(A, B, M)$:

$$\mathsf{inj\text{-}event}(\mathsf{Recv}(A, B, M)) \Longrightarrow \mathsf{inj\text{-}event}(\mathsf{Send}(A, B, M))$$

When a optional one-time pre-key is involved in the initial session handshake, ProVerif shows that the injective correspondence holds for the first message in the conversation. However, when this optional one-time pre-key is not used, a replay attack is detected. Signal Protocol Version 3 will accept a Diffie-Hellman handshake that only employs identity keys and signed pre-keys, both of which are allowed to be reused across sessions. This reuse is what makes a replay attack possible. We propose a fix for this issue by having clients keep a cache of the ephemeral keys used by the sender of received messages, associated with that sender's identity key. We are able to expand our event queries in ProVerif to account for this fix by showing the non-injective correspondence of the Send and Recv events with added ephemeral keys. Coupled with a caching of ephemeral keys, we can ensure that the Recv event is only executed once per ephemeral key. Hence, the injective correspondence is implied by the non-injective correspondence.

Key Compromise Impersonation (KCI). We present a novel key compromise impersonation attack: to detect KCI, we consider a scenario in which Alice or Bob's keys are compromised and test again for authenticity of messages received by the compromised principal. When Alice or Bob's long-term secret key is compromised, ProVerif shows that message authenticity still holds. However, when Bob's signed pre-key is also compromised, ProVerif finds an attack against message authenticity. This is a novel key compromise impersonation attack: when the adversary has Bob's signed pre-key s, he can choose x and x' and compute the session keys using Alice's and Bob' public keys $(g^{a_{3dh}}, g^{a_{sig}})$ and $(g^{b_{3dh}}, g^{b_{sig}})$ and Bob's one time pre-key g^o and send his own message in Alice's name. This message is accepted by Bob as if it came from Alice: the event $\mathsf{Recv}(A, B, M)$ is executed without having executed $\mathsf{Send}(A, B, M)$.

Integrating Symbolic Verification into the Development Cycle. Human-readability of the automatically compiled ProVerif model is key to our verification methodology. In the case of a query failure, users can opt to modify their implementation and recompile into a new model, or they can immediately modify the model itself and re-test for security queries within reasonable model verification times. For example, if an implementer wants to test the robustness of a passing forward secrecy query, they can disable the signature verification of signed pre-keys by changing a single line in the model, causing the client to accept any pre-key signature.

6. Cryptographic Proofs with CryptoVerif

To complement the results obtained in the symbolic model using ProVerif, we use the tool CryptoVerif [5] in order to obtain security proofs in the computational model. This model is much more realistic: messages are bitstrings; cryptographic primitives are functions from bitstrings to bitstrings; the adversary is a probabilistic Turing machine. CryptoVerif generates proofs by sequences of games [24], [25], like those written manually by cryptographers, automatically or with guidance of the user.

The computational model is more realistic, but it also makes it more difficult to mechanize proofs. For this reason,

CryptoVerif is less flexible and more difficult to use than ProVerif, and our results in the computational model are more limited. We model only one message of the protocol (in addition to the pre-keys), so we do not prove properties of the ratcheting algorithm. Considering several data messages exceeds the current capabilities of CryptoVerif—the games become too big.

Rather than directly using models generated from our ProScript code, we manually rewrite the input scripts of CryptoVerif, for two main reasons:

- The syntax of the protocol language of CryptoVerif differs slightly from that of ProVerif. We plan to overcome this difficulty in the future by modifying the syntax of CryptoVerif so that it is compatible with ProVerif.
- The kinds of models that are easy to verify using CryptoVerif differ from those that are easy for ProVerif; therefore, even if the source syntax were the same, we would still need to adapt our compiler to generate specialized models that would be more conducive to CryptoVerif's game-based proofs.

6.1. Assumptions

We make the following assumptions on the cryptographic primitives:

- The elliptic curve Ec25519 satisfies the gap Diffie-Hellman (GDH) assumption [26]. This assumption means that given g, g^a, and g^b for random a, b, the adversary has a negligible probability to compute g^{ab} (computational Diffie-Hellman assumption), even when the adversary has access to a decisional Diffie-Hellman oracle, which tells him given G, X, Y, Z whether there exist x, y such that $X = G^x$, $Y = G^y$, and $Z = G^{xy}$. When we consider sessions between a participant and himself, we need the square gap Diffie-Hellman variant, which additionally says that given g and g^a for random a, the adversary has a negligible probability to compute g^{a^2}. This assumption is equivalent to the GDH assumption when the group has prime order [27], which is true for Ec25519 [28]. We also added that $x^y = x'^y$ implies $x = x'$ and that $x^y = x^{y'}$ implies $y = y'$, which hold when the considered Diffie-Hellman group is of prime order.
- Ed25519 signatures, used for signing pre-keys, are unforgeable under chosen-message attacks (UF-CMA) [29].
- The functions

$$x_1, x_2, x_3, x_4 \mapsto \mathsf{HKDF}(x_1 \| x_2 \| x_3 \| x_4, c_1, c_2)$$
$$x_1, x_2, x_3 \mapsto \mathsf{HKDF}(x_1 \| x_2 \| x_3, c_1, c_2)$$
$$x, y \mapsto \mathsf{HKDF}(x, y, c_2)$$
$$x \mapsto \mathsf{HKDF}(x, c_1, c_4)$$

  are independent random oracles, where x, y, x_1, x_2, x_3, x_4, and c_1 are 256-bit long. We further justify this assumption in the full version of this work [23]: there, we show that these functions are indifferentiable [30] from independent random oracles, assuming that the compression function underlying SHA256 is a random oracle. (The considered HKDF function [22] is defined from HMAC-SHA256, which is itself defined from SHA256.)
- HMAC-SHA256 is a pseudo-random function (PRF) [31]. This assumption is used for $\mathsf{HMAC}(ck_{ab}, \cdot)$ and $\mathsf{HMAC}(ck_{ba}, \cdot)$.
- The encryption scheme ENC, which is AES-GCM, is a secure authenticated encryption with associated data (AEAD). More precisely, it is indistinguishable under chosen plaintext attacks (IND-CPA) and satisfies ciphertext integrity (INT-CTXT) [32], [33].

CryptoVerif provides a library that predefines the most common cryptographic assumptions, so that the user does not have to write them for each protocol. In our work, we had to adapt these predefined assumptions to our specific needs: the GDH assumption is predefined, but the square GDH variant is not; unary random oracles are predefined, but we also needed binary, ternary, and 4-ary ones; predefined PRFs, SUF-CMA MACs, and IND-CPA encryption schemes use a key generation function, while in our schemes the key is a plain random bitstring, without a key generation function. Adapting the definition of primitives did not present any major difficulty. As mentioned in § 5.1, we had to make one modification to the original Signal Protocol, for it to be provable in the computational model: we use different keys for the elliptic curve Diffie-Hellman and elliptic curve signatures. It is well-known that using the same keys for several cryptographic primitives is undesirable, as proving security requires a joint security assumption on the two primitives in this case. Therefore, we assume each protocol participant to have two key pairs, one for Diffie-Hellman and one for signatures. This problem remains undetected in a symbolic analysis of the protocol.

6.2. Protocol Model

We model SP as a process in the input language of CryptoVerif, which is similar to the one of ProVerif. We consider simultaneously the protocol of Figure 2 and the version without the optional one-time pre-key b_o. As mentioned above, we consider only one message in each session. Our threat model includes an untrusted network, malicious principals, and long-term key compromise, as mentioned in § 2. It does not include session state compromise, which is less useful with a single message.

At a high level, we use the same messaging API as in § 2. However, to make verification easier for CryptoVerif, we specify a lower-level interface. We consider two honest principals Alice and Bob, and define separate processes for Alice interacting with Bob, with herself, or with a malicious participant, Bob interacting with Alice, and Bob interacting with himself or a malicious participant, as well as similar processes with the roles of Alice and Bob reversed. The adversary can then implement the high-level interface of § 2 from this lower-level interface: the adversary is supposed to implement the malicious principals (including defining keys for them) and to call the low-level interface processes to run sessions that involve the honest principals Alice and Bob.

We make two separate proofs: In the first one, we prove the security properties for sessions in which Bob generates pre-keys and runs the protocol with Alice. (Other protocol sessions exist in parallel as described above; we do not prove security properties for them. For sessions for which we do not prove security properties, we give to the adversary the ephemeral a'_e and the key rk_{ba} or rk_{ba} and let the adversary encrypt and MAC the message himself, to reduce the size of our processes.) In the second one, we prove the security properties for sessions in which Alice generates pre-keys and runs the protocol with herself. Bob is included in the adversary in this proof. The security for sessions in which Alice generates pre-keys and runs the protocol with Bob follows from the first proof by symmetry. The security for sessions in which Bob generates pre-keys and runs the protocol with himself follows from the second proof. The other sessions do not satisfy security properties since they involve the adversary. (They must still be modeled, as they could break the protocol if it were badly designed.) Therefore, these two proofs provide all desired security properties.

6.3. Security Goals

We consider the following security goals from § 2:

Message Authenticity, No Replays, and Key Compromise Impersonation (KCI). These properties are modeled by correspondences as in ProVerif (§ 5.3). For key compromise impersonation, we consider the compromise of the long-term Diffie-Hellman and signature keys of Bob, and prove again message authenticity. We do not consider the compromise of the signed pre-key since we already know from the symbolic analysis that there is an attack in this case.

Computational Indistinguishability. If A randomly chooses between two messages M_0, M_1 of the same length and sends one of them to B, then the adversary has a negligible probability of guessing which of the two messages was sent. In our model, this is formalized by choosing a random bit $secb \in \{0, 1\}$; then A sends message M_b to B, and we show that the bit $secb$ remains secret, with the query **secret** $secb$.

Forward Secrecy. This is proved exactly like indistinguishability, but with an additional oracle that allows the adversary to obtain the secret keys of the principals, thus compromising them.

We do not consider future secrecy since we have a single message. We do not consider secrecy since we directly deal with the stronger property of indistinguishability.

6.4. Results

CryptoVerif proves message authenticity, absence of key compromise impersonation attacks (when the long-term keys of Bob are compromised), indistinguishability, and forward secrecy, but cannot prove absence of replays. This is due to the replay attack mentioned in § 5.3. Since this attack appears only when the optional one-time pre-key is omitted, we separate our property into two: we use

Goals	Parties	Running Time
Forward Secrecy	A, B, M	3 min. 58 sec.
Forward Secrecy	A, M	7 min. 04 sec.
KCI	A, B, M	3 min. 15 sec.
Others	A, B, M	4 min. 15 sec.
Others	A, M	3 min. 35 sec.

Figure 4: Verification times for SP CryptoVerif models, without anti-replay countermeasure. The runtimes with the anti-replay countermeasure are of the same order of magnitude. Tested using CryptoVerif 1.24.

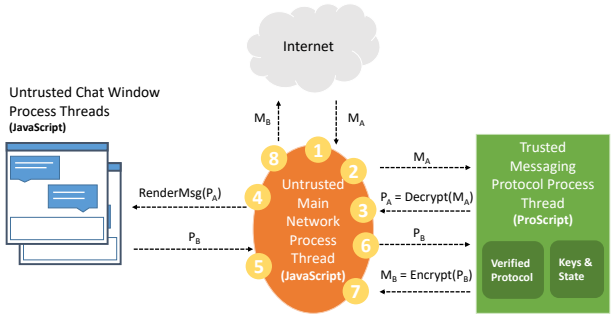


Figure 5: Cryptocat Architecture: isolating verified and untrusted components in Electron apps within separate processes.

events $\mathsf{Send}(A, B, M)$ and $\mathsf{Recv}(A, B, M)$ for the protocol with optional pre-key and events $\mathsf{Send3}(A, B, M)$ and $\mathsf{Recv3}(A, B, M)$ for the protocol without optional pre-key. CryptoVerif then proves

$$\mathsf{inj\text{-}event}(\mathsf{Recv}(A, B, M)) \Longrightarrow \mathsf{inj\text{-}event}(\mathsf{Send}(A, B, M))$$
$$\mathsf{event}(\mathsf{Recv3}(A, B, M)) \Longrightarrow \mathsf{event}(\mathsf{Send3}(A, B, M))$$

which proves message authenticity and no replays when the one-time pre-key is present and only message authenticity when it is absent. This is the strongest we can hope for the protocol without anti-replay countermeasure.

With our anti-replay countermeasure (§5.3), CryptoVerif can prove the absence of replays, thanks to a recent extension that allows CryptoVerif to take into account the replay cache in the proof of injective correspondences, implemented in CryptoVerif version 1.24. Our CryptoVerif proofs have been obtained with some manual guidance: we indicated the main security assumptions to apply, instructed CryptoVerif to simplify the games or to replace some variables with their values, to make terms such as $m^a = m^b$ appear. The proofs were similar for all properties.

7. A Verified Protocol Core for Cryptocat

We now describe how we can rewrite Cryptocat to incorporate our ProScript implementation of SP. We deconstruct the Cryptocat JavaScript code into the following components, as advocated in Figure 1.

1) **Unverified JavaScript Application** This component, which comprises the majority of the code, manages the user's state, settings, notifications, graphical interface and so on. It is connected to the protocol only via the ability to call exposed protocol functions (as documented in § 2.1). We adopt certain assumptions regarding the unverified JavaScript application, for example that it will not modify the protocol state outside of passing it through the protocol implementation interface.
2) **Verified Protocol Implementation** This component is written in ProScript and resides in a separate namespace, functioning in a purely state-passing fashion. Namely, it does not store any internal state or make direct network calls. This implementation is type-checked, and automatically verified every time it is modified.
3) **Trusted Library** This component provides cryptographic functionality. A goal is to include modern cryptographic primitives (X25519, AES-CCM) and provide type-checking assurances without affecting speed or performance.

This layered architecture is essential for our verification methodology, but is quite different from other messaging applications. For example, the Signal Desktop application is a Chrome browser application also written in JavaScript [34]. Parts of the protocol library are compiled from C using Emscripten, presumably for performance, parts are taken from third-party libraries, and other protocol-specific code is written in JavaScript. The resulting code (1.5MB, 39Kloc) is quite hard to separate into components, let alone verify for security. We hope that our layered approach can lead to verified security guarantees without sacrificing performance or maintainability.

7.1. Isolating Verified Code

We build Cryptocat using Electron [35], a framework for JavaScript desktop applications. Electron is built on top of the Node.js JavaScript runtime and the Chromium web renderer. By default, Electron allows applications to load any Node.js low-level module, which can in turn perform dangerous operations like accessing the file system and exfiltrate data over the network. Since all Node.js modules in a single process run within the same JavaScript environment, malicious or buggy modules can tamper with other modules via prototype poisoning or other known JavaScript attack vectors. Consequently, all Electron apps, including other desktop Signal implementations like WhatsApp and Signal messenger effectively include all of Electron and Node.js into their trusted computing base (TCB).

We propose a two-pronged approach to reduce this TCB.

Language-Based Isolation. Since ProScript is a subset of Defensive JavaScript, ProScript protocol code is isolated at the language level from other JavaScript code running within the same process, even if this code uses dangerous JavaScript features such as prototype access and modification. To ensure this isolation, ProScript code must not call any external (untyped) libraries.

Process Thread Isolation. We exploit features of Electron in order to isolate components of our application in different CPU threads as seen in Figure 5. When a message arrives on the network (1), the main network thread can only communicate with our Protocol TCB using a restrictive inter-process communication API. The TCB then uses its internal verified protocol functionality and state management to return a decryption of the message (3), which is then forwarded again via IPC to the chat window (4), a third separate CPU thread which handles message rendering. Furthermore, the TCB process is disallowed from loading any Node.js modules.

In particular, the network process is isolated from the chat media rendering process; neither ever obtain access to the key state or protocol functionality, which are all isolated in the ProScript protocol process. When Bob responds, a similar IPC chain of calls occurs in order to send his reply back to Alice (5, 6, 7, 8). Even if an error in the rendering code or in the XML parser escalated into a remote takeover of the entire web renderer, the calls to the protocol TCB would be restricted to those exposed by the IPC API. However, these isolation techniques only protect the ProScript code within our application when executed within a correct runtime framework. None of these techniques can guard against bugs in V8, Node.js, or Electron, or against malicious or buggy Node.js or Electron modules loaded by the application.

7.2. Performance and Limitations

Although we have verified the core protocol code in Cryptocat and tried to isolate this code from unverified code, the following limitations still apply: we have not formally verified the soundness of the cryptographic primitives themselves, although writing them in Defensive JavaScript does provide type safety. We have also not formally verified the Electron framework's isolation code. Similarly, we do not claim any formal verification results on the V8 JavaScript runtime or on the Node.js runtime. Therefore, we rely on a number of basic assumptions regarding the soundness of these underlying components. Cryptocat's successful deployment provides a general guideline for building, formally verifying, and isolating cryptographic protocol logic from the rest of the desktop runtime. Designing better methods for implementing and isolating security-critical components within Electron apps with a minimal TCB remains an open problem.

8. Related Work

Extracting Protocol Models from Running Code. There have been previous attempts [36] to extract ProVerif models from typed JavaScript, such as *DJS2PV* [7]. However, DJS2PV was only tested on small code examples: attempting to translate a complete implementation such as Signal Protocol resulted in a 3,800 line model that attempts to precisely account for the heap, but could not verify due to an exploding state space. Previous efforts such as FS2PV [20]

avoided this problem by choosing a purely functional source language that translated to simpler pi calculus scripts. We adopt their approach in ProScript to generate briefer, more readable models.

Type Systems for JavaScript. TypeScript [37], Flow [38], Defensive JavaScript and TS* [39] all define type systems that can improve the security of JavaScript programs. The type system in ProScript primarily serves to isolate protocol code from untrusted application and to identify a subset of JavaScript that can be translated to verifiable models.

Formal Analysis of Web Security Protocols. Tools like WebSpi [40] and AuthScan [41] have been used to verify the security of web security protocols such as OAuth. An expressive web security model has also been used to build manual proofs for cryptographic web protocols such as BrowserID [42]. These works are orthogonal to ProScript and their ideas can potentially be used to improve our target ProVerif models.

Analysis of Secure Messaging Protocols. Unger et al. survey previous work on secure messaging [10]. We discuss three recent closely-related works here.

Future secrecy was formalized by Cohn-Gordon et al. as "post-compromise security" [43]. Our symbolic formulation is slightly different since it relies on the definition of protocol phases in ProVerif.

Cryptographic security theorems and potential unknown key-share attacks on TextSecure Version 2 were presented by Frosch et al. [2]. In comparison to that work, our analysis covers a variant of TextSecure Version 3, our analysis is fully mechanized, and we address implementation details. Our CryptoVerif model only covers a single message, but we consider the whole protocol at once, while they prove pieces of the protocol separately. Like we do, they consider that HKDF is a random oracle. We further justify this assumption by an indifferentiability proof.

More recently and in parallel with this work, Cohn-Gordon et al. [44] prove, by hand, that the message encryption keys of Signal are secret in the computational model, in a rich compromise scenario, under assumptions similar to ours. Thereby, they provide a detailed proof of the properties of the double ratcheting mechanism. However, they do not model the signatures of the signed pre-keys, and they do not consider key compromise impersonation attacks or replay attacks or other implementation-level details. In contrast to their work, our computational proof is mechanized, but limited to only one message.

9. Conclusion and Future Work

Drawing from existing design trends in modern cryptographic web application, we have presented a framework that supports the incremental development of custom cryptographic protocols hand-in-hand with formal security analysis. By leveraging state-of-the-art protocol verification tools and building new tools, we showed how many routine tasks can be automated, allowing the protocol designer to focus on the important task of analyzing her protocol for sophisticated security goals against powerful adversaries.

We plan to continue to develop and refine ProScript by evaluating how it is used by protocol designers, in the spirit of an open source project. All the code and models presented in this paper, and a full version of this paper are available online [23]. Proving the soundness of translation from ProScript to ProVerif, by relating the source JavaScript semantics to the applied pi calculus, remains future work. The process of transforming the compiled model to a verified CryptoVerif script remains a manual task, but we hope to automate this step further, based on new and upcoming developments in CryptoVerif.

Finally, a word of caution: a protocol written in ProScript and verified with ProVerif or CryptoVerif does not immediately benefit from assurance against all possible attacks. Programming in ProScript imposes a strict discipline by requiring defensive self-contained code that is statically typed and can be translated to a verifiable model and subsequent verification can be used to eliminate certain well-defined classes of attacks. We believe these checks can add confidence to the correctness of a web application, but they do not imply the absence of security bugs, since we still have a large trusted computing base. Consequently, improving the robustness and security guarantees of runtime frameworks such as Electron, Node.js, and Chromium, remains an important area of future research.

Acknowledgments. This work was funded by the following grants: ERC CIRCUS, EU NEXTLEAP, and ANR AJACS.

References

[1] K. Bhargavan, A. Lavaud, C. Fournet, A. Pironti, and P. Strub, "Triple handshakes and cookie cutters: Breaking and fixing authentication over TLS," in *IEEE Symposium on Security & Privacy (Oakland)*, 2014, pp. 98–113.

[2] T. Frosch, C. Mainka, C. Bader, F. Bergsma, J. Schwenk, and T. Holz, "How secure is TextSecure?" in *IEEE European Symposium on Security and Privacy (Euro S&P)*, 2016.

[3] B. Blanchet, "Modeling and verifying security protocols with the applied pi calculus and ProVerif," *Foundations and Trends in Privacy and Security*, vol. 1, no. 1–2, pp. 1–135, Oct. 2016.

[4] D. Dolev and A. C. Yao, "On the security of public key protocols," *IEEE Transactions on Information Theory*, vol. 29, no. 2, pp. 198–207, 1983.

[5] B. Blanchet, "A computationally sound mechanized prover for security protocols," *IEEE Transactions on Dependable and Secure Computing*, vol. 5, no. 4, pp. 193–207, 2008.

[6] K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis, "Language-based defenses against untrusted browser origins," in *USENIX Security Symposium*, 2013, pp. 653–670.

[7] K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis, "Defensive JavaScript - building and verifying secure web components," in *Foundations of Security Analysis and Design (FOSAD VII)*, 2013, pp. 88–123.

[8] M. Abadi and C. Fournet, "Mobile values, new names, and secure communication," in *28th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'01)*. London, United Kingdom: ACM Press, Jan. 2001, pp. 104–115.

[9] H. Krawczyk, "HMQV: A High-performance Secure Diffie-Hellman Protocol," in *International Conference on Advances in Cryptology (CRYPTO)*, ser. Lecture Notes in Computer Science, V. Shoup, Ed., vol. 3621. Springer, 2005, pp. 546–566.

[10] N. Unger, S. Dechand, J. Bonneau, S. Fahl, H. Perl, I. Goldberg, and M. Smith, "SoK: Secure Messaging," in *IEEE Symposium on Security & Privacy (Oakland)*, 2015.

[11] N. Durov, "Telegram MTProto protocol," 2015, https://core.telegram.org/mtproto.

[12] O. Schirokauer, "The number field sieve for integers of low weight," *Mathematics of Computation*, vol. 79, no. 269, pp. 583–602, 2010.

[13] D. Gillmor, "Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for Transport Layer Security (TLS)," 2016, IETF RFC 7919.

[14] K. Bhargavan, A. Delignat-Lavaud, and A. Pironti, "Verified contributive channel bindings for compound authentication," in *Proceedings of the ISOC Network and Distributed System Security Symposium (NDSS '15)*, Feb 2015. [Online]. Available: http://antoine.delignat-lavaud.fr/doc/ndss15.pdf

[15] A. Rad and J. Rizzo, "A 2^64 attack on Telegram, and why a super villain doesn't need it to read your telegram chats." 2015.

[16] J. Jakobsen and C. Orlandi, "On the cca (in)security of mtproto," Cryptology ePrint Archive, Report 2015/1177, 2015, http://eprint.iacr.org/2015/1177.

[17] N. Borisov, I. Goldberg, and E. A. Brewer, "Off-the-record communication, or, why not to use PGP," in *Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society, WPES 2004, Washington, DC, USA, October 28, 2004*, V. Atluri, P. F. Syverson, and S. D. C. di Vimercati, Eds. ACM, 2004, pp. 77–84. [Online]. Available: http://doi.acm.org/10.1145/1029179.1029200

[18] N. Wilcox, Z. Wilcox-O'Hearn, D. Hopwood, and D. Bacon, "Report of Security Audit of Cryptocat," 2014, https://leastauthority.com/blog/least_authority_performs_security_audit_for_cryptocat.html.

[19] P. A. Gardner, S. Maffeis, and G. D. Smith, "Towards a program logic for JavaScript," *SIGPLAN Not.*, vol. 47, no. 1, pp. 31–44, Jan. 2012. [Online]. Available: http://doi.acm.org/10.1145/2103621.2103663

[20] K. Bhargavan, C. Fournet, A. D. Gordon, and S. Tse, "Verified interoperable implementations of security protocols," *ACM Transactions on Programming Languages and Systems*, vol. 31, no. 1, 2008.

[21] Joyent Inc. and the Linux Foundation, "Node.js," 2016, https://nodejs.org/en/.

[22] H. Krawczyk, "Cryptographic extraction and key derivation: The HKDF scheme," in *Advances in Cryptology (CRYPTO)*, ser. Lecture Notes in Computer Science. Springer, 2010, vol. 6223, pp. 631–648.

[23] N. Kobeissi, "SP code repository," https://github.com/inria-prosecco/proscript-messaging, February 2017.

[24] V. Shoup, "Sequences of games: a tool for taming complexity in security proofs," IACR Cryptology ePrint Archive, 2004, http://eprint.iacr.org/2004/332.

[25] M. Bellare and P. Rogaway, "The security of triple encryption and a framework for code-based game-playing proofs," in *Advances in Cryptology (Eurocrypt)*, ser. Lecture Notes in Computer Science, S. Vaudenay, Ed., vol. 4004. Springer, May 2006, pp. 409–426.

[26] T. Okamoto and D. Pointcheval, "The gap-problems: a new class of problems for the security of cryptographic schemes," in *Practice and Theory in Public Key Cryptography (PKC)*, ser. Lecture Notes in Computer Science, K. Kim, Ed., vol. 1992. Springer, 2001, pp. 104–118.

[27] A. Fujioka and K. Suzuki, "Designing efficient authenticated key exchange resilient to leakage of ephemeral secret keys," in *Topics in Cryptology (CT-RSA)*, ser. Lecture Notes in Computer Science, A. Kiayias, Ed., vol. 6558. Springer, 2011, pp. 121–141.

[28] D. J. Bernstein, "Curve25519: New Diffie-Hellman speed records," in *Public Key Cryptography (PKC)*, 2006, pp. 207–228.

[29] S. Goldwasser, S. Micali, and R. Rivest, "A digital signature scheme secure against adaptive chosen-message attacks," *SIAM Journal of Computing*, vol. 17, no. 2, pp. 281–308, April 1988.

[30] J.-S. Coron, Y. Dodis, C. Malinaud, and P. Puniya, "Merkle-Damgård revisited: How to construct a hash function," in *Advances in Cryptology (CRYPTO)*, ser. Lecture Notes in Computer Science, vol. 3621. Springer, 2005, pp. 430–448.

[31] M. Bellare, "New proofs for NMAC and HMAC: Security without collision-resistance," in *Advances in Cryptology (CRYPTO)*, ser. Lecture Notes in Computer Science, C. Dwork, Ed., vol. 4117. Springer, 2006, pp. 602–619.

[32] D. A. McGrew and J. Viega, "The security and performance of the Galois/Counter Mode (GCM) of operation," in *Progress in Cryptology - INDOCRYPT 2004*, ser. Lecture Notes in Computer Science, A. Canteaut and K. Viswanathan, Eds., vol. 3348. Chennai, India: Springer, Dec. 2004, pp. 343–355.

[33] P. Rogaway, "Authenticated-encryption with associated-data," in *Ninth ACM Conference on Computer and Communications Security (CCS-9)*. Washington, DC: ACM Press, Nov. 2002, pp. 98–107.

[34] Open Whisper Systems, "Signal for the browser," 2015, https://github.com/WhisperSystems/Signal-Browser.

[35] GitHub, "Electron framework," 2016, http://electron.atom.io/.

[36] M. Avalle, A. Pironti, R. Sisto, and D. Pozza, "The Java SPI framework for security protocol implementation," in *Availability, Reliability and Security (ARES), 2011 Sixth International Conference on*, Aug 2011, pp. 746–751.

[37] G. Bierman, M. Abadi, and M. Torgersen, "Understanding TypeScript," in *ECOOP 2014 Object-Oriented Programming*, ser. Lecture Notes in Computer Science, R. Jones, Ed., vol. 8586. Springer, 2014, pp. 257–281.

[38] Facebook Inc., "Flow, a static type checker for JavaScript," http://flowtype.org/docs/about-flow.html.

[39] C. Fournet, N. Swamy, J. Chen, P.-E. Dagand, P.-Y. Strub, and B. Livshits, "Fully abstract compilation to JavaScript," *SIGPLAN Not.*, vol. 48, no. 1, pp. 371–384, Jan. 2013. [Online]. Available: http://doi.acm.org/10.1145/2480359.2429114

[40] C. Bansal, K. Bhargavan, A. Delignat-Lavaud, and S. Maffeis, "Discovering concrete attacks on website authorization by formal analysis," *Journal of Computer Security*, vol. 22, no. 4, pp. 601–657, 2014.

[41] G. Bai, J. Lei, G. Meng, S. S. Venkatraman, P. Saxena, J. Sun, Y. Liu, and J. S. Dong, "AUTHSCAN: automatic extraction of web authentication protocols from implementations," in *Network and Distributed System Security Symposium (NDSS)*, 2013.

[42] D. Fett, R. Küsters, and G. Schmitz, "An Expressive Model for the Web Infrastructure: Definition and Application to the BrowserID SSO System," in *35th IEEE Symposium on Security and Privacy (S&P 2014)*. IEEE Computer Society, 2014, pp. 673–688.

[43] K. Cohn-Gordon, C. Cremers, and L. Garratt, "On post-compromise security," in *IEEE Computer Security Foundations Symposium (CSF)*, 2016, pp. 164–178.

[44] K. Cohn-Gordon, C. Cremers, B. Dowling, L. Garratt, and D. Stebila, "A formal security analysis of the signal messaging protocol," in *IEEE European Symposium on Security and Privacy (Euro S&P)*, 2017.

On the Content Security Policy Violations due to the Same-Origin Policy

Dolière Francis Some
Université Côte d'Azur
Inria
France
doliere.some@inria.fr

Nataliiia Bielova
Université Côte d'Azur
Inria
France
nataliia.bielova@inria.fr

Tamara Rezk
Université Côte d'Azur
Inria
France
tamara.rezk@inria.fr

ABSTRACT

Modern browsers implement different security policies such as the Content Security Policy (CSP), a mechanism designed to mitigate popular web vulnerabilities, and the Same Origin Policy (SOP), a mechanism that governs interactions between resources of web pages.

In this work, we describe how CSP may be violated due to the SOP when a page contains an embedded iframe from the same origin. We analyse 1 million pages from 10,000 top Alexa sites and report that at least 31.1% of current CSP-enabled pages are potentially vulnerable to CSP violations. Further considering real-world situations where those pages are involved in same-origin nested browsing contexts, we found that in at least 23.5% of the cases, CSP violations are possible.

During our study, we also identified a divergence among browsers implementations in the enforcement of CSP in srcdoc sandboxed iframes, which actually reveals a problem in Gecko-based browsers CSP implementation. To ameliorate the problematic conflicts of the security mechanisms, we discuss measures to avoid CSP violations.

CCS CONCEPTS

•Security and privacy →Web application security;

ACM Reference format:
Dolière Francis Some, Nataliiia Bielova, and Tamara Rezk. 2016. On the Content Security Policy Violations due to the Same-Origin Policy. In *Proceedings of WWW '17, Perth, Western Australia, April 3–7, 2017,* 9 pages.
DOI: 10.1145/1235

1 INTRODUCTION

Modern browsers implement different specifications to securely fetch and integrate content. One widely used specification to protect content is the Same Origin Policy (SOP) [**?**]. SOP allows developers to isolate untrusted content from a different origin. An origin here is defined as scheme, host, and port number. If an iframe's content is loaded from a different origin, SOP controls the access to the embedder resources. In particular, no script inside the iframe can access content of the embedder page. However, if the iframe's content is loaded from the same origin as the embedder page, there are no privilege restrictions w.r.t. the embedder resources. In such a case, a script executing inside the iframe can access content of the embedder webpage. Scripts are considered trusted and the *iframe becomes transparent* from a developer view point. A more recent specification to protect content in webpages is the Content Security Policy (CSP) [**?**]. The primary goal of CSP is to mitigate cross site scripting attacks (XSS), data leaks attacks, and other types of attacks. CSP allows developers to specify, among other features, trusted domain sources from which to fetch content. One of the most important features of CSP, is to allow a web application developer to specify trusted JavaScript sources. This kind of restriction is meant to permit execution of only trusted code and thus prevent untrusted code to access content of the page.

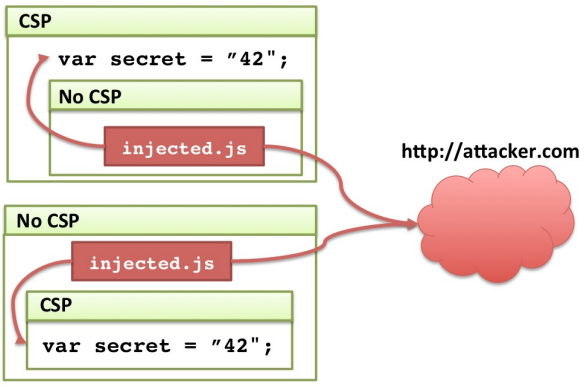


Figure 1: An XSS attack despite CSP.

In this work, we report on a fundamental problem of CSP. CSP[**?**] defines how to protect content in an isolated page. However, it does not take into consideration the page's context, that is its embedder or embedded iframes. In particular, CSP is unable to protect content of its corresponding page if the page embeds (using the *src* attribute) an iframe of the same origin. The CSP policy of a page will not be applied to an embedded iframe. However, due to SOP, the iframe has complete access to the content of its embedder. Because same origin iframes are transparent due to SOP, this opens loopholes to attackers whenever the CSP policy of an iframe and that of its embedder page are not compatible (see Fig. 1).

We analysed 1 million pages from the top 10,000 Alexa sites and found that 5.29% of sites contain some pages with CSPs (as opposed to 2% of home pages in previous studies [**?**]). We have identified that in 94% of cases, CSP may be violated in presence

WWW '17, Perth, Western Australia

978-1-4503-2138-9…$$15.00
DOI: 10.1145/1235

of the document.domain API and in 23.5% of cases CSP may be violated without any assumptions (see Table 3).

During our study, we also identified a divergence among browsers implementations in the enforcement of CSP [?] in sandboxed iframes embedded with *srcdoc*, which actually reveals an inconsistency between the CSP and HTML5 sandbox attribute specification for iframes.

We identify and discuss possible solutions from the developer point of view as well as new security specifications that can help prevent this kind of CSP violations. We have made publicly available the dataset that we used for our results in[?]. We have installed an automatic crawler to recover the same dataset every month to repeat the experiment taking into account the time variable. An accompanying technical report with a complete account of our analyses can be found at [?].

In summary, our contributions are: (i) We describe a new class of vulnerabilities that lead to CSP violations. (Section 2). (ii) We perform a large and depth scale crawl of top sites, highlighting CSP adoption at sites-level, as well as sites origins levels. Using this dataset, we report on the possibilities of CSP violations between the SOP and CSP in the wild. (Section 3). (iii) We propose guidelines in the design and deployment of CSP. (Section 4). (iv) We reveal an inconsistency between the CSP specification and HTML5 sandbox attribute specification for iframes. Different browsers choose to follow different specifications, and we explain how any of these choices can lead to new vulnerabilities. (Section 5).

2 CONTENT SECURITY POLICY AND SOP

The Content Security Policy (CSP) [?] is a mechanism that allows programmers to control which client-side resources can be loaded and executed by the browser. CSP (version 2) is an official W3C candidate recommendation [?], and is currently supported by major web browsers. CSP is delivered in the `Content-Security-Policy` HTTP response header, or in a `<meta>` element of HTML.

CSP applicability A CSP delivered with a page controls the resources of the page. However it does not apply to the page's embedding resources [?]. As such, CSP does not control the content of the iframes even if the iframe is from the same origin as the main page according to SOP. Instead, the content of the iframe is controlled by the CSP delivered with it, that can be different from the CSP of the main page.

CSP directives CSP allows a programmer to specify which resources are allowed to be loaded and executed in the page. These resources are defined as a set of origins and known as a *source list*. Additionally to controlling resources, CSP allows to specify allowed destinations of the AJAX requests by the `connect-src` directive. A special header `Content-Security-Policy-Report-Only` configures a CSP in a report-only mode: violations are recorded, but not enforced. The directive `default-src` is a special fallback directive that is used when some directive is not defined. The directive `frame-ancestors` (meant to supplant the HTTP `X-Frame-Options` header[?]), controls in which pages the current page may be included as an iframe, to prevent clickjacking attacks [?]. See Table 1 for the most commonly used CSP directives [?].

Source lists CSP source list is traditionally defined as a *whitelist* indicating which domains are trusted to load the content, or to communicate. For example, a CSP from Listing 1 allows to include scripts only from `third.com`, requires to load frames only over HTTPS, while other resource types can only be loaded from the same hosting domain.

Directive	Controlled content
`script-src`	Scripts
`default-src`	All resources (fallback)
`style-src`	Stylesheets
`img-src`	Images
`font-src`	Fonts
`connect-src`	XMLHttpRequest, WebSocket or EventSource
`object-src`	Plug-in formats (object, embed)
`report-uri`	URL where to report CSP violations
`media-src`	Media (audio, video)
`child-src`	Documents (frames), [Shared] Workers
`frame-ancestors`	Embedding context

Table 1: Most common CSP directives [?].

```
1 Content-Security-Policy: default-src 'self';
2 script-src third.com; child-src https:
```

Listing 1: Example of a CSP policy.

A whitelist can be composed of concrete hostnames (`third.com`), may include a wildcard `*` to extend the policy to subdomains (`*.third.com`), a special keyword `'self'` for the same hosting domain, or `'none'` to prohibit any resource loading.

Restrictions on scripts Directive `script-src` is the most used feature of CSP in today's web applications [?]. It allows a programmer to control the origin of scripts in his application using source lists. When the `script-src` directive is present in CSP, it blocks an execution of any inline script, JavaScript event handlers and APIs that execute string data code, such as `eval()` and other related APIs. To relax the CSP, by allowing the execution of inline `<script>` and JavaScript event handlers, a `script-src` whitelist should contain a keyword `'unsafe-inline'`. To allow `eval()`-like APIs, the CSP should contain a `'unsafe-eval'` keyword. Because `'unsafe-inline'` allows execution of *any* inlined script, it effectively removes any protection against XSS. Therefore, nonces and hashes were introduced in CSP version 2 [?], allowing to control which inline scripts can be loaded and executed.

Sandboxing iframes Directive `sandbox` allows to load resources but execute them in a separate environment. It applies to all the iframes and other content present on the page. An empty `sandbox` value creates completely isolated iframes. One can selectively enable specific features via `allow-*` flags in the directive's value. For example, `allow-scripts` will allow executions of scripts in an iframe, and `allow-same-origin` will allow iframes to be treated as being from their normal origins.

Same-Site and Same-Origin Definitions. In our terminology, we distinguish the web pages that belong to the same site from the pages that belong to the same origin. By *page* we refer to any HTML document – for example, the content of an iframe we call *iframe page*. In this case, the page that embeds an iframe is called a *parent page* or *embedder*.

By *site* we refer to the highest level domain that we extract from Alexa top 10,000 sites, usually containing the domain name and a TLD, for example `main.com`. All the pages that belong to a site, and to any of its subdomains as `sub.main.com`, are considered *same-site* pages.

According to the Same Origin Policy, an *origin* of a page is scheme, host and port of its URL. For example, in http://main.com:81/dir/p.html, the scheme is "http", the host is "main.com" and the port is 81.

2.1 CSP violations due to SOP

Consider a web application, where the main page `A.html` and its iframe `B.html` are located at `http://main.com`, and therefore belong to the same origin according to the same-origin policy. `A.html`, shown in Listing 2, contains a script and an iframe from `main.com`. The local script `secret.js` contains sensitive information given in Listing 3. To protect against XSS, the developer have installed the CSP for its main page `A.html`, shown in Listing 4.

```
<html>
  <script src="secret.js"></script>
   ...
  <iframe src="B.html"></iframe>
</html>
```

Listing 2: Source code of `http://main.com/A.html`.

```
var secret = "42";
```

Listing 3: Source code of `secret.js`.

```
Content-Security-Policy: default-src 'none';
script-src 'self'; child-src 'self'
```

Listing 4: CSP of `http://main.com/A.html`.

This CSP provides an effective protection against XSS:

2.1.1 Only parent page has CSP. According to the latest version of CSP[1], only the CSP of the iframe applies to its content, and it ignores completely the CSP of the including page. In our case, if there is no CSP in `B.html` then its resource loading is not restricted. As a result, an iframe `B.html` without CSP is potentially vulnerable to XSS, since any injected code may be executed within `B.html` with no restrictions. Assume `B.html` was exploited by an attacker injecting a script `injected.js`. Besides taking control over `B.html`, this attack now propagates to the including page `A.html`, as we show in Fig. 1. The XSS attack extends to the including parent page because of the inconsistency between the CSP and SOP. When a parent page and an iframe are from the same origin according to SOP, a parent and an iframe share the same privileges and can access each other's code and resources.

For our example, `injected.js` is shown in Listing 5.

This script executed in `B.html` retrieves the secret value from its parent page (`parent.secret`) and transmits it to an attacker's server `http://attacker.com` via XMLHttpRequest[2].

```
function sendData(obj, url){
  var req = new XMLHttpRequest();
  req.open('POST', url, true);
  req.send(JSON.stringify(obj));
}
sendData({secret: parent.secret}, 'http://
    attacker.com/send.php');
```

Listing 5: Source code of `injected.js`.

A straightforward solution to this problem is to ensure that the protection mechanism for the parent page also propagates to the iframes from the same domain. Technically, it means that the CSP of the iframe should be the same or more restrictive than the CSP of the parent. In the next example we show that this requirement does not necessarily prevent possible CSP violations due to SOP.

2.1.2 Only iframe page has CSP. Consider a different web application, where the including parent page `A.html` does not have a CSP, while its iframe `B.html` contains a CSP from Listing 4. In this example, `B.html`, shown in Listing 6 now contains some sensitive information stored in `secret.js` (see Listing 3).

```
<html>
   ...
  <script src="secret.js"></script>
</html>
```

Listing 6: Source code of `http://main.com/B.html`.

Since the including page `A.html` now has no CSP, it is potentially vulnerable to XSS, and therefore may have a malicious script `injected.js`. The iframe `B.html` has a restrictive CSP, that effectively contributes to protection against XSS. Since `A.html` and `B.html` are from the same origin, the malicious injected script can profit from this and steal sensitive information from `B.html`. For example, the script may call the `sendData` function with the secret information:

```
sendData({secret: children[0].secret}, 'http:
    //attacker.com/send.php');
```

Thanks to SOP, the script `injected.js` fetches the secret from it's child iframe `B.html` and sends it to `http://attacker.com`.

2.1.3 CSP violations due to origin relaxation. A page may change its own origin with some limitations. By using the `document.domain` API, the script can change its current domain to a superdomain. As a result, a shorter domain is used for the subsequent origin checks[3].

Consider a slightly modified scenario, where the main page `A.html` from `http://main.com` includes an iframe `B.html` from its sub-domain `http://sub.main.com`. Any script in `B.html` is able to change the origin to `http://main.com` by executing the following line:

```
document.domain = "main.com";
```

If `A.com` is willing to communicate with this iframe, it should also execute the above-written code so that the communication with `B.html` will be possible. The content of `B.html` is now treated by the web browser as the same-origin content with `A.html`, and therefore any of the previously described attacks become possible.

[1] https://www.w3.org/TR/CSP2/#which-policy-applies

[2] The XMLHttpRequest is not forbidden by the SOP for `B.html` because an attacker has activated the Cross-Origin Resource Sharing mechanism [?] on her server `http://attacker.com`.

[3] https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy#Changing_origin

2.1.4 Categories of CSP violations due to SOP. We distinguish three different cases when the CSP violation might occur because of SOP:

Only parent page or only iframe has CSP A parent page and an iframe page are from the same origin, but only one of them contains a CSP. The CSP may be violated due to the unrestricted access of a page without CSP to the content of the page with CSP. We demonstrated this example in Sections 2.1.1 and 2.1.2.

Parent and iframe have different CSPs A parent page and an iframe page are from the same origin, but they have different CSPs. Due to SOP, the scripts from one page can interfere with the content of another page thus violating the CSP.

CSP violation due to origin relaxation A parent page and an iframe page have the same higher level domain, port and scheme, but however they are not from the same origin. Either CSP is absent in one of them, or they have different CSPs – in both cases CSP may be violated because the pages can relax their origin to the high level domain by using `document.domain` API, as we have shown in Section 2.1.3.

3 EMPIRICAL STUDY OF CSP VIOLATIONS

We have performed a large-scale study on the top 10,000 Alexa sites to detect whether CSP may be violated due to an inconsistency between CSP and SOP. For collecting the data, we have used CasperJS [**?**] on top of PhantomJS headless browser [**?**]. The User-Agent HTTP header was instantiated as a recent Google Chrome browser.

3.1 Methodology

The overview of our data collection and CSP comparison process is given in Figure 2. The main difference in our data collection process from previous works on CSP measurements in the wild [**? ?**] is that we crawl not only the main pages of each site, but also other pages. First, we collect pages accessible through links of the main page and pointing to the same site. Second, to detect possible CSP violations due to SOP, we have collected all the iframes present on the home pages and linked pages.

3.1.1 Data Collection. **Home Page Crawler** For each site in top 10,000 Alexa list, we crawl the home page, parse its source code and extract three elements: (1) a CSP of the site's home page stored in HTTP header as well as in `<meta>` HTML tag; we denote the CSPs of the home page by C; (2) to extract more pages from the same site, we analyse the source of the links via `<a href=...>` tag and extract URLs that point to the same site, we denote this list by L. (3) we collect URLs of iframes present on the home page via `<iframe src=...>` tag and record only those belonging to the same site, we denote this set by $\mathcal{F}$.

Page Crawler We crawl all the URLs from the list of pages L, and for each page we repeat the process of extraction of CSP and relevant iframes, similar to the steps (1) and (3) of the home page crawler. As a result, we get a set of CSPs of linked pages C_L and a set of iframes URLs $\mathcal{F}_L$ that we have extracted from the linked pages in L.

Iframe Crawler

For every iframe URL present in the list of home page iframes $\mathcal{F}_H$, and in the list of linked pages iframes $\mathcal{F}_L$, we extract their corresponding CSPs and store in two sets: C_F for home page iframes and C_{LF} for linked page iframes.

3.1.2 CSP adoption analysis. Since CSP is considered an effective countermeasure for a number of web attacks, programmers often use it to mitigate such attacks on the main pages of their sites. However, if CSP is not installed on some pages of the same site, this can potentially leak to CSP violations due to the inconsistency with SOP when another page from the same origin is included as an iframe (see Figure 1). In our database, for each site, we recorded its home page, a number of linked pages and iframes from the same site. This allows us to analyse how CSP is adopted at every popular site by checking the presence of CSP on every crawled page and iframe of each site. To do so, we analyse the extracted CSPs: C for the home page, C_L for linked pages, C_F for home page iframes, and C_{LF} for linked pages iframes.

3.1.3 CSP violations detection. To detect possible CSP violations due to SOP, we have analysed home pages and linked pages from the same site, as well as iframes embedded into them.

CSP Selection

To detect CSP violations, we first remove all the sites where no parent page and no iframe page contains a CSP. For the remaining sites, we pointwise compare (1) the CSPs of the home pages C and CSPs of iframes present on these pages C_F; (2) the CSPs of the linked pages C_L and CSPs of their iframes C_{LF}. To check whether a parent page CSP and an iframe CSP are equivalent, we have applied the CSP comparison algorithm (Figure 2)

CSP Preprocessing We first normalise each CSP policy, by splitting it into its directives.

- If **default-src** directive is present (**default-src** is a fallback for most of the other directives), then we extract the source list s of **default-src**. We analyse which directives are missing in the CSP, and explicitly add them with the source list s.
- If **default-src** directive is absent, we extract missing directives from the CSP. In this case, there are no restrictions in CSP for every absent directive. We therefore explicitly add them with the most permissive source list. A missing **script-src** is assigned * **'unsafe-inline' 'unsafe-eval'** as the most permissive source list [**?**].
- In each source list, we modify the special keywords: (i) **'self'** is replaced with the origin of the page containing the CSP; (ii) in case of **'unsafe-inline'** with hash or nonce, we remove **'unsafe-inline'** from the directive since it will be ignored by the CSP2. (iii) **'none'** keywords are removed from all the directives; (iv) nonces and hashes are removed from all the directives since they cannot be compared; (iv) each whitelisted domain is extended with a list of schemes and port numbers from the URL of the page includes the CSP[4].

[4]For example, according to CSP2, if the page scheme is `https`, and a CSP contains a source `example.com`, then the user agent should allow content only from `https://example.com`, while if the current scheme is `http`, it would allow both `http://example.com` and `https://example.com`.

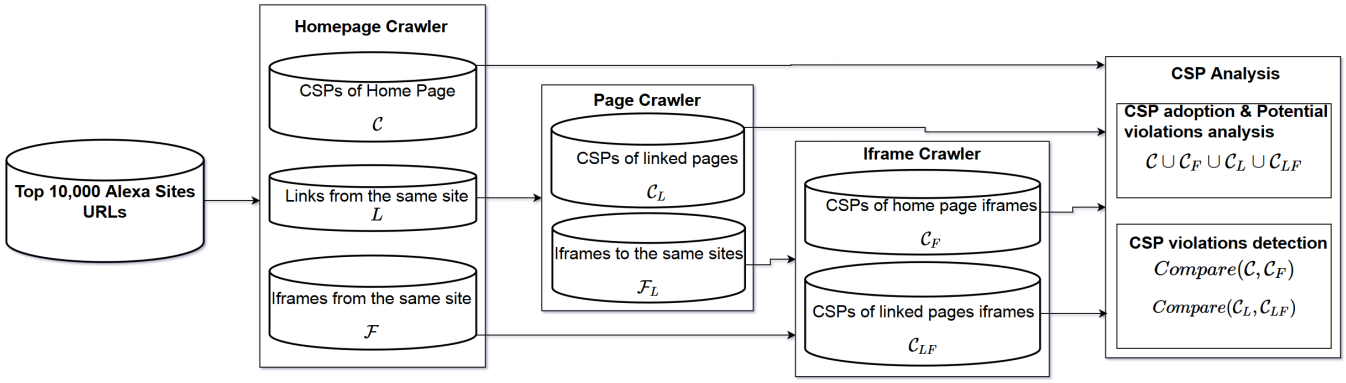


Figure 2: Data Collection and Analysis Process

Sites successfully crawled	9,885
Pages visited	1,090,226
Pages with iframe(s) from the same site	648,324
Pages with same-origin iframe(s)	92,430
Pages with same-origin iframe(s) where page and/or iframe has CSP	692
Pages with CSP	21,961 (2.00%)
Sites with CSP on home page	228 (2.3%)
Sites with CSP on some pages	523 (5.29%)

Table 2: Crawling statistics

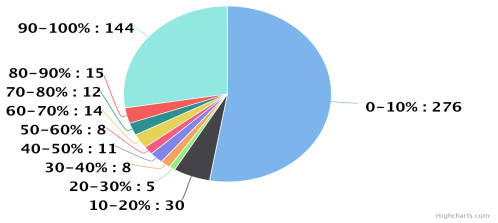


Figure 3: Percentage of pages with CSP per site

CSP Comparison We compare all the directives present in the two CSPs to identify whether the two policies require the same restrictions. Whenever the two CSPs are different, our algorithm returns the names of directives that do not match. The demonstration of the comparison is accessible on[**?**]. For each directive in the policies we compare the source lists and the algorithm proceeds if the elements of the lists are identical in the normalised CSPs.

3.1.4 Limitations. Our methodology and results have two(2) limitations that we explain here.

User interactions The automatic crawling process did not include any real-user-like interactions with top sites. As such the set of iframes and links URLs we have analysed is an underestimate of all links and iframes a site may contain.

Pairs of (parent-iframe) In this study, we consider CSP violations in same origin (parent, iframe) couples only. Their are though further combinations such as couples of sibling iframes in a parent page that we could have considered. Overall, our results are conservative, since the problem might have been worst without those limitations.

3.2 Results on CSP Adoption

The crawling of Alexa top 10,000 sites was performed in the end of August, 2016. To extract several pages from the same site, we have also crawled all the links and iframes on a page that point to the same site. In total, we have gathered 1,090,226 from 9,885 different sites. On median, from each site we extracted 45 pages, with a maximum number of 9,055 pages found on tuberel.com. Our crawling statistics is presented in Table 2. More than half of the pages contain an iframe, and 13% of pages do contain an iframe from the same site. This indicates the potential surface for the CSP violations, when at least one page on the site has a CSP installed. We discuss such potential CSP violation in details in Section 3.3.3. Similarly to previous works on CSP adoption [**? ?**], we have found that CSP is present on only 228 out of 9,885 home pages (2.31%). While extending this analysis to almost a million pages, we have found a similar rate of CSP adoption (2.00%).

Differently from previous studies that anlaysed only home pages, or only pages in separation, we have analysed how many sites have at least some pages that adopted CSP. We have grouped all pages by sites, and found that 5.29% of sites contain some pages with CSPs. It means that CSP is more known by the website developers, but for some reason is not widely adopted on all the pages of the site.

We have then analysed how many pages on each site have adopted CSPs. For each of 523 sites, we have counted how many pages (including home page, linked pages and iframes) have CSPs. Figure 3 shows that more than half of the sites have a very low CSP adoption on their pages: on 276 sites out of 529, CSP is installed on only 0-10% of their pages. This becomes problematic if other pages without CSP are not XSS-free. However, it is interesting that around a quarter of sites do profit from CSP by installing it on 90-100% of their pages.

3.3 Results on CSP violations due to SOP

As described in Section 2.1.4, we distinguish several categories of CSP violations when a parent page and an iframe on this page are from the same origin according to SOP. To account for possible CSP violations, we only consider cases when either parent, or iframe, or both have a CSP installed. From all the 21,961 pages that have CSP

	Same-origin parent-iframe	Possible to relax origin	Total
Only parent page CSP	**83**	**1388**	**1471**
Only iframe CSP	**16**	**240**	**256**
Different CSP	**70**	**44**	**114**
No CSP violations	551 (76.5%)	109 (6%)	660
CSP violations total	**169 (23.5%)**	**1672 (94%)**	**1841**

Table 3: Statistics CSP violations due to Same-Origin Policy

	Same-origin parent-iframe	Possible to relax origin
Only parent page CSP	yandex.ru	twitter.com, yandex.ru, mail.ru
Only iframe CSP	amazon.com, imdb.com	–*
Different CSP	twitter.com	–*

*Not found in top 100 Alexa sites.

Table 4: Sample of sites with CSP violations due to Same-Origin Policy

installed, we have removed the pages, where CSPs are in report-only mode, having left 18,035 pages with CSPs in enforcement mode.

Table 3 presents possible CSP violations due to SOP.

We have extracted the parent-iframe couples that might cause a CSP violation because either (1) only parent or only iframe installed a CSP, or (2) both installed different CSPs. First, to account for direct violations because of SOP, we distinguish couples where parent and iframe are from the same origin (columns 2,3), we have found 720 cases of such couples. Second, we analyse possible CSP violations due to origin relaxation: we have collected 1781 couples that are from different origins but their origins can be relaxed by `document.domain` API (see more in Section 2.1.3) – these results are shown in columns 4 and 5.

In Table 4 we present the names of the domains out of top 100 Alexa sites, where we have found different CSP violations. Each company in this table have been notified about the possible CSP violation. Concrete examples of the page and iframe URLs and their corresponding CSPs for each such violation can be found in the corresponding technical report [**?**]. All the collected data is available online[**?**].

CSP violations in presence of `document.domain` According to our results, in presence of `document.domain`, 94% of (parent, iframe) pages can have their CSP violated. Those violations can occur only if both parent and iframes pages execute `document.domain` to the same top level domain. Thus, our result is an over-approximation, assuming that `document.domain` is used in all of those pages and iframes. According to[**?**], `document.domain` is used in less than 3% of web pages.

3.3.1 Only parent page or only iframe has CSP. We first consider a scenario when a parent page and an iframe are from the same origin, but only one of them contains a CSP. Intuitively, if only a parent page has CSP, then an iframe can violate CSP by executing any code and accessing the parent page's DOM, inserting content, access cookies etc. Among 720 parent-iframe couples from *the same origin*, we have found 83 cases (11.5%) when only parent has a CSP, and 16 cases (2.2%) when only iframe has a CSP. These CSP violations originate from 13 (for parent) and 4 (for iframe) sites.

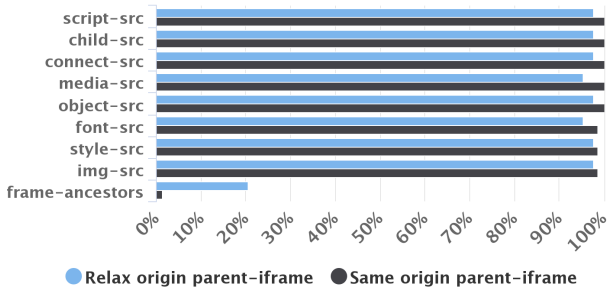


Figure 4: Differences in CSP directives for parent and iframe pages

For example, such possible violations are found on some pages of amazon.com, yandex.ru and imdb.com (see Table 4). CSP of a parent or iframe may also be violated because of *origin relaxation*. We have identified 1388 cases (78%) of parent-iframe couples where such violation may occur because CSP is present only in the parent page. This was observed on 20 different sites, including twitter.com, yandex.ru and others. Finally, in 240 cases (13.5%) only iframe has CSP installed, which was found on 11 different sites. We manually checked the parent and iframes involved in CSP violations for sites in Table 4. In all of those sites, either the parent or the iframe page is login page[**?**]. We furthermore checked how effective are the CSP of those pages, using CSPEvaluator[5], proposed by Lukas et al.[**?**]. and found out that the CSP policies involved in these are moreover all bypassable.

3.3.2 Parent and iframe have different CSPs. In a case when a page and iframe are from the same origin, but their corresponding CSPs are different, may also cause a violation of CSP. From the 720 *same-origin* parent-iframe couples, we have found 70 cases (9.7%) (from 3 sites) when their CSPs differ, and for *an origin relaxation* (from 6 sites) case, we have identified only 44 such cases (2.5%). This setting was found on some pages of twitter.com for instance.

We have further analysed the differences in CSPs found on parent and iframe pages. For all the 114 pairs of parent-iframe (either

[5]https://csp-evaluator.withgoogle.com/

	Pages	Origins	Sites
A same origin page has no CSP	4381	197	197
A same origin page has a different CSP	1223	23	23
A same origin (after relaxation) page has no CSP	4728	340	183
A same origin (after relaxation) has a different CSP	2567	135	44
Potential violations total	12899 (72%)	591 (81%)	379 (52%)

Table 5: Potential CSP violations in pages with CSP

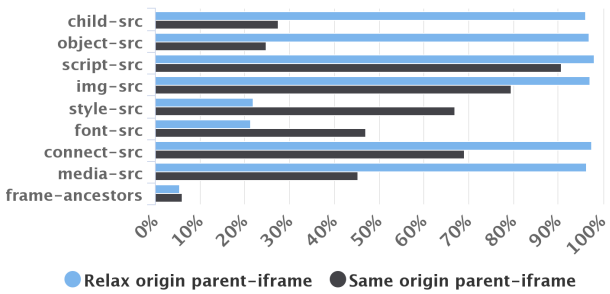


Figure 5: Differences in CSP directives for same-origin and relaxed origin pages

same-origin or possible origin relaxation), we have compared CSPs they installed, directive-by-directive. Figure 4 shows that every parent CSP and iframe CSP differ on almost every directive – between 90% and 100%. The only exception is **frame-ancestors** directive, which is almost the same in different parent pages and iframes. If properly set, this directive gives a strong protection against clickjacking attacks, therefore all the pages of the same origin are equally protected.

3.3.3 Potential CSP violations. A *potential CSP violation* may happen when in a site, either some pages have CSP and some others do not, or pages have different CSP. When those pages get nested as parent-iframe, we can run into CSP violations, just like in the direct CSP violations cases we have just reported above. To analyse how often such violations may occur, we have analysed the 18,035 pages that have CSP in enforcement mode. These pages originate from 729 different origins spread over 442 sites. Table 5 shows that 72% of CSPs (12,899 pages) are potentially violated, and these CSPs originate from pages of 379 different sites (85.75%). To detect these violations, for each page with a CSP in our database, we have analysed whether there exists another page from the same origin, that does not have CSP. This page could embed the page with CSP and violate it because of SOP. We have detected 4381 such pages (24%) from 197 origins. Similarly, we detected 1223 pages (7%) when there are same-origin pages with a different CSP. Similarly, we have analysed when potential CSP violations may happen due to origin relaxation. We have detected 4728 pages (26%), whose CSP may be violated because of other pages with no CSP, and 2567 pages (14%), whose CSP may be violated because of different CSP on other relaxed-origin pages.

For the pages that have different CSPs, we have compared how much CSPs differ. Figure 5 shows that CSPs mostly differ in `script-src` directive, which protects pages from XSS attacks. This means, that if one page in the origin does whitelist an attacker's domain or an insecure endpoints [**?**], all the other pages in the same origin become vulnerable because they may be inserted as an iframe to the vulnerable page and their CSPs can be easily violated.

3.4 Responses of websites owners

We have reported those issues to a sample of sites owners, using either HackerOne[6], or contact forms when available. Here are some selected quotes from our discussions with them.

> "*Yes, of course we understand the risk that under some circumstances XSS on one domain can be used to bypass CSP on another domain, but it's simply impossible to implement CSP across all (few hundreds) domains at once on the same level. We are implementing strongest CSP currently possible for different pages on different domains and keep going with this process to protect all pages, after that we will strengthen the CSP. We believe it's better to have stronger CSP policy where possible rather than have same weak CSP on all pages or not having CSP at all. Having in mind there are hundreds of domains within mail.ru, at least few years are required before all pages on all domains can have strong CSP.*" – `Mail.ru`

> "*[...]the sandbox is a defense in depth mitigation[...]We definitely don't allow relaxing document.domain on www.dropbox.com[...]*" – `Dropbox.com`

> "*While this is an interesting area of research, are you able to demonstrate that this behavior is currently exploitable on Twitter? It appears that the behavior you have described can increase the severity of other vulnerabilities but does not pose a security risk by itself. Is our understanding correct? [...]We consider this to be more of a defensive in depth and will take into account with our continual effort to improve our CSP policy*" – `Twitter.com`

> "*I believe we understand the risk as you've described it.*" – `Imdb.com`

4 AVOIDING CSP VIOLATIONS

Preventing CSP violations due to SOP can be achieved by having the **same** effective CSP for **all** same-origin pages in a site, and prevent origin relaxation.

Origin-wide CSP: Using CSP for all same-origin pages can be manually done but this solution is error-prone. A more effective solution is the use of a specification such as Origin Policy [**?**] in order to set a header for the whole origin.

Preventing Origin Relaxation: Having an origin-wide CSP is not enough to prevent CSP violations. By using origin relaxation, pages from different origins can bypass the SOP [**?**]. Many authors

[6]https://hackerone.com

provide guidelines on how to design an effective CSP [**?**]. Nonetheless, even with an effective CSP, an embedded page from a different origin in the same site can use `document.domain` to relax its origin. Preventing origin relaxation is trickier.

Programmatically, one could prevent other scripts from modifying `document.domain` by making a script run first in a page [**?**]. The first script that runs on the page would be:

```
Object.defineProperty(document, "domain", {
    __proto__: null, writable: false,
    configurable: false});
```

A parent page can also indirectly disable origin relaxation in iframes by sandboxing them. This can be achieved by using **sandbox** as an attribute for iframes or as directive for the parent page CSP. Unfortunately, an iframe cannot indirectly disable origin relaxation in the page that embeds it. However, the **frame-ancestors** directive of CSP gives an iframe control over the hosts that can embed it. Finally, a more robust solution is the use of a policy to deprecate `document.domain` as proposed in the draft of Feature policy [**?**]. The feature policy defines a mechanism that allows developers to selectively enable and disable the use of various browser features and APIs.

Iframe sandboxing: Combining attribute **allow-scripts** and **allow-same-origin** as values for **sandbox** successfully disables `document.domain` in an iframe[7]. We recommend the use of **sandbox** as a CSP directive, instead of an HTML iframe attribute. The first reason is that **sandbox** as a CSP directive, automatically applies to all iframes that are in a page, avoiding the need to manually modify all HTML iframe tags. Second, the **sandbox** directive is not programmatically accessible to potentially malicious scripts in the page, as is the case for the **sandbox** attribute (which can be removed from an iframe programmatically, replacing the sandboxed iframe with another identical iframe but without the **sandbox** attribute).

Limitations An origin-wide CSP (the same CSP for all same origin pages) can become very liberal if all same origin pages do not require the same restrictions. In order to implement the solution we propose, one needs to consider the intended relation between a parent page and an iframe page, in presence of CSP. In the case where the two(2) pages should be allowed direct access to each other content, then, since same origin pages can bypass page-specific security characteristics [**?**], the solution is to have the same CSP for both the page and the iframe. However, if direct access to each other content is not a required feature, one can keep different CSPs in parent and iframe, or have no CSP at all in one of the parties, but their contents should be isolated from each other. The solution here is to use sandboxing. Nonetheless, there are other means (such as `postMessage`) by which one can securely achieve communication between the pages.

5 INCONSISTENT IMPLEMENTATIONS

Combining origin-wide CSP with **allow-scripts sandbox** directive would have been sufficient at preventing the inconsistencies between CSP and the same origin policy. Unfortunately, we have discovered that for some browsers, this solution is not sufficient. Starting from HTML5, major browsers, apart from Internet Explorer, supports the new **srcdoc** attribute for iframes. Instead of providing a URL which content will be loaded in an iframe, one provides directly the HTML content of the iframe in the **srcdoc** attribute. According to CSP2 [**?**], §5.2, the CSP of a page should apply to an iframe which content is supplied in a **srcdoc** attribute. This is actually the case for all majors browsers, which support the **srcdoc** attribute. However, there is a problem when the **sandbox** attribute is set to an **srcdoc** iframe.

Webkit-based[8] and **Blink**-based[9] browsers (Chrome, Chromium, Opera) always comply with CSP. The CSP of a page will apply to all **srcdoc** iframes, even in those iframes which have a different origin than that of the page, because they are sandboxed without **allow-same-origin** .

In contrast, we noticed that in Gecko-based browsers (Mozilla Firefox), the CSP of the page applies to that of the srcdoc iframe if and only if allow-same-origin is present as value for the attribute. Otherwise it does not apply. The problem with this choice is the following. A third party script, whitelisted by the CSP of the page, can create a **srcdoc** iframe, sandboxing it with **allow-scripts** only, and load any resource that would normally be blocked by the CSP of the page if applied in this iframe. This way, the third party script successfully bypasses the restrictions of the CSP of the page. Even though loading additional scripts is considered harmless in the upcoming version 3 [**?** **?**] of CSP, this specification says nothing about violations that could occur due to the loading of other resources inside a **srcdoc** sandboxed iframe, like resources whitelisted by **object-src** directive for instance, additional iframes etc.

We have notified the W3C, and the Mozilla Security Group. Daniel Veditz, a lead at Mozilla Security Group, recognises this as a bug and explains:

> "*Our internal model only inherits CSP into same-origin frames (because in theory you're otherwise leaking info across origin boundaries) and iframe sandbox creates a unique origin. Obviously we need to make an exception here (I think we manage to do the same thing for src=data: sandboxed frames).*"

CSP specification and srcdoc iframes The problem of imposing a CSP to an unknown page is illustrated by the following example [**?**]. If a trusted third party library, whitelisted by the CSP of the page, uses security libraries inside an isolated context (by sandboxing them in a **srcdoc** iframe, setting **allow-scripts** as sole value for the **sandbox**) then, the page's CSP will block the security libraries and possibly introduce new vulnerabilities. Because of this, it was unclear to us the intent of CSP designers regarding srcdoc iframes. Mike West, one of the CSP editors at the W3C and also Developper Advocate in Google Chrome's team, clarified this to us:

> "*I think your objection rests on the notion of the same-origin policy preventing the top-level document from reaching into it's sandboxed child. That seems accurate, but it neglects the bigger picture:* ***srcdoc*** *documents are produced entirely from the*

[7] We found out that dropbox.com actually puts **sandbox** attribute for all its iframes, and therefore avoids the possible CSP violations. We have had a very interesting discussion on Hackerone.com with Devdatta Akhawe, a Security Engineer at Dropbox, who told us more about their security practices regarding CSP in particular.

[8] https://en.wikipedia.org/wiki/WebKit

[9] https://en.wikipedia.org/wiki/Blink_(web_engine)

> *top-level document context. Since those kinds of documents are not delivered over the network, they don't have the opportunity to deliver headers which might configure their settings. We impose the parent's policy in these cases, because for all intents and purposes, the* ***srcdoc*** *document* is *the parent document.*"

6 RELATED WORK

CSP has been proposed by Stamm et al.[**?**] as a refinement of SOP[**?**], in order to help mitigate Cross-Site-Scripting[**?**] and data exfiltration attacks. The second version[**?**] of the specification is supported by all major browsers, and the third version [**?**] is under active development. Even though CSP is well supported [**?**], its endorsement by web sites is rather slow. Weissbacher et al.[**?**] performed the first large scale study of CSP deployment in top Alexa sites, and found that around 1% of sites were using CSP at the time. A more recent study by Calzavara et al.[**?**], show that nearly 8% of Alexa top sites now have CSP deployed in their front pages. Another recent study, by Weichselbaum et al.[**?**] come with similar results to the study of Weissbacher et al.[**?**]. Our work extends previous results by analysing the adoption of CSP by site not only considering front pages but all the pages in a site. Almost all authors agree that CSP adoption is not a straightforward task, and lots of (manual) effort are needed in order to reorganize and modify web pages to support CSP.

Therefore, in order to help web sites developers in adopting CSP, Javed proposed CSP Aider, [**?**] that automatically crawl a set of pages from a site and propose a site-wide CSP. Patil and Frederik[**?**] proposed UserCSP, a framework that monitors the browser internal events in order to automatically infer a CSP for a web page based on the loaded resources. Pan et al.[**?**] propose CSPAutoGen, to enforce CSP in real-time on web pages, by rewriting them on the fly client-side. Weissbacher et al.[**?**] have evaluated the feasibility of using CSP in report-only mode in order to generate a CSP based on reported violations, or semi-automatically inferring a CSP policy based on the resources that are loaded in web pages. They concluded that automatically generating a CSP is ineffective. A difficulty which remains is the use of inline scripts in many pages. The first solution is to externalize inline scripts, as can be done by systems like deDacota[**?**]. Kerschbaumer et al.[**?**] find that too many pages are still using **'unsafe-inline'** in their CSPs. They propose a system to automatically identify legitimate inline scripts in a page, thereby whitelisting them in the CSP of the underlying page, using script hashes.

Another direction of research on CSP, has been evaluating its effectiveness at successfully preventing content injection attacks. Calzavara et al.[**?**] found out that many CSP policies in real web sites have errors including typos, ill-formed or harsh policies. Even when the policies are well formed, they have found that almost all currently deployed CSP policies are bypassable because of a misunderstanding of the CSP language itself. Patil and Frederik found similar errors in their study[**?**]. Hausknecht et al.[**?**] found that some browser extensions, modified the CSP policy headers, in order to whitelist more resources and origins. Van Acker et al.[**?**] have shown that CSP fails at preventing data exfiltration specially when resources are prefetched, or in presence of a CSP policy in the HTML meta tag, because the order in which resources are loaded in a web application is hard to predict. Johns[**?**] proposed hashes for static scripts, and PreparedJS, an extension for CSP, in order to securely handle server-side dynamically generated scripts based on user input. Weichselbaum et al.[**?**] have extended nonces and hashes, introduced in CSP level 2[**?**], to remote scripts URLs, specially to tackle the high prevalence of insecure hosts in current CSP policies. Furthermore, they have introduced **strict-dynamic**. This new keyword states that any additional script loaded by a whitelisted remote script URL is considered a trusted script as well. They also provide guidelines on how to build an effective CSP. Jackson and Barth[**?**] have shown that same origin pages can bypass page-specific policies, like CSP. Though, their work predates CSP. To the best of our knowledge, we are the first to explore the interactions between CSP and SOP and report possible CSP violations.

7 CONCLUSIONS

In this work, we have revealed a new problem that can lead to violations of CSP. We have performed an in-depth analysis of the inconsistency that arises due to CSP and SOP and identified three cases when *CSP may be violated.*

To evaluate how often such violations happen, we performed a large-scale analysis of more than 1 million pages from 10,000 Alexa top sites. We have found that 5.29% of sites contain pages with CSPs (as opposed to 2% of home pages in previous studies).

We have also found out that 72% of current web pages with CSP, are potentially vulnerable to CSP violations. This concerns 379 (72.46%) sites that deploy CSP. Further analysing the contexts in which those web pages are used, our results show that when a parent page includes an iframe from the same origin according to SOP, in 23.5% of cases their CSPs may be violated. And in the cases where `document.domain` is required in both parent and iframes, we identified that such violations may occur in 94% of the cases.

We discussed measures to avoid CSP violations in web applications by installing an origin-wide CSP and using sandboxed iframes. Finally, our study reveals an inconsistency in browsers implementation of CSP for `srcdoc` iframes, that appeared to be a bug in Mozilla Firefox browsers.

ACKNOWLEDGMENTS

The authors would like to thank the WebAppSec W3C Working Group for useful pointers to related resources at the early stage of this work, Mike West for very insightful discussions that considerably helped improve this work, Devdatta Akhawe for discussing some security practices at *Dropbox*, and anonymous reviewers and Stefano Calzavara for their valuable comments and suggestions.

REFERENCES

[1] Chrome Platform Status. https://www.chromestatus.com/metrics/feature/popularity#DocumentSetDomain.

[2] CSP violations online. https://webstats.inria.fr?cspviolations.

[3] Same Origin Policy. https://www.w3.org/Security/wiki/Same_Origin_Policy.

[4] S. V. Acker, D. Hausknecht, and A. Sabelfeld. Data Exfiltration in the Face of CSP. In X. Chen, X. Wang, and X. Huang, editors, *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security, AsiaCCS 2016, Xi'an, China, May 30 - June 3, 2016*, pages 853–864. ACM, 2016.

[5] S. Calzavara, A. Rabitti, and M. Bugliesi. Content security problems?: Evaluating the effectiveness of content security policy in the wild. In Weippl et al. [**?**], pages 1365–1375.

[6] A. Doupé, W. Cui, M. H. Jakubowski, M. Peinado, C. Kruegel, and G. Vigna. deDacota: toward preventing server-side XSS via automatic code and data separation. In A. Sadeghi, V. D. Gligor, and M. Yung, editors, *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 1205–1216. ACM, 2013.

[7] D. Hausknecht, J. Magazinius, and A. Sabelfeld. May I? - Content Security Policy Endorsement for Browser Extensions. In M. Almgren, V. Gulisano, and F. Maggi, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment - 12th International Conference, DIMVA 2015, Milan, Italy, July 9-10, 2015, Proceedings*, volume 9148 of *Lecture Notes in Computer Science*, pages 261–281. Springer, 2015.

[8] A. Hidayat. PhantomJS Headless Browser, 2010-2016.

[9] C. Jackson and A. Barth. Beware of finer-grained origins. In *Web 2.0 Security and Privacy (W2SP 2008)*, 2008.

[10] A. Javed. CSP Aider: An Automated Recommendation of Content Security Policy for Web Applications. In *IEEE Oakland Web 2.0 Security and Privacy (W2SP'12)*, 2012.

[11] M. Johns. PreparedJS: Secure Script-Templates for JavaScript. In K. Rieck, P. Stewin, and J. Seifert, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment - 10th International Conference, DIMVA 2013, Berlin, Germany, July 18-19, 2013. Proceedings*, volume 7967 of *Lecture Notes in Computer Science*, pages 102–121. Springer, 2013.

[12] C. Kerschbaumer, S. Stamm, and S. Brunthaler. Injecting CSP for Fun and Security. In O. Camp, S. Furnell, and P. Mori, editors, *Proceedings of the 2nd International Conference on Information Systems Security and Privacy (ICISSP 2016), Rome, Italy, February 19-21, 2016.*, pages 15–25. SciTePress, 2016.

[13] X. Pan, Y. Cao, S. Liu, Y. Zhou, Y. Chen, and T. Zhou. Cspautogen: Black-box enforcement of content security policy upon real-world websites. In Weippl et al. [**?**], pages 653–665.

[14] K. Patil and B. Frederik. A measurement study of the content security policy on real-world applications. *I. J. Network Security*, 18(2):383–392, 2016.

[15] N. Perriault. CasperJS navigation and scripting tool for PhantomJS, 2011-2016.

[16] G. Rydstedt, E. Bursztein, D. Boneh, and C. Jackson. Busting frame busting: a study of clickjacking vulnerabilities at popular sites. In *in IEEE Oakland Web 2.0 Security and Privacy (W2SP 2010)*, 2010.

[17] K. Singh, A. Moshchuk, H. J. Wang, and W. Lee. On the incoherencies in web browser access control policies. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pages 463–478, 2010.

[18] D. F. Some, N. Bielova, and T. Rezk. On the Content Security Policy violations due to the Same-Origin Policy. Technical report. http://www-sop.inria.fr/members/Nataliia.Bielova/papers/CSP-SOP.pdf.

[19] S. Stamm, B. Sterne, and G. Markham. Reining in the web with content security policy. In M. Rappa, P. Jones, J. Freire, and S. Chakrabarti, editors, *Proceedings of the 19th International Conference on World Wide Web, WWW 2010, Raleigh, North Carolina, USA, April 26-30, 2010*, pages 921–930. ACM, 2010.

[20] N. Swamy, C. Fournet, A. Rastogi, K. Bhargavan, J. Chen, P. Strub, and G. M. Bierman. Gradual typing embedded securely in JavaScript. In S. Jagannathan and P. Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 425–438. ACM, 2014.

[21] A. van Kesteren. Cross Origin Resource Sharing. W3C Recommendation, 2014.

[22] L. Weichselbaum, M. Spagnuolo, S. Lekies, and A. Janc. CSP is dead, long live csp! on the insecurity of whitelists and the future of content security policy. In Weippl et al. [**?**], pages 1376–1387.

[23] E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, editors. *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. ACM, 2016.

[24] M. Weissbacher, T. Lauinger, and W. K. Robertson. Why Is CSP Failing? Trends and Challenges in CSP Adoption. In A. Stavrou, H. Bos, and G. Portokalidis, editors, *Research in Attacks, Intrusions and Defenses - 17th International Symposium, RAID 2014, Gothenburg, Sweden, September 17-19, 2014. Proceedings*, volume 8688 of *Lecture Notes in Computer Science*, pages 212–233. Springer, 2014.

[25] M. West. Content Security Policy: Embedded Enforcement, 2016.

[26] M. West. Content Security Policy Level 3. W3C Working Draft, 2016.

[27] M. West. Origin Policy. A Collection of Interesting Ideas, 2016.

[28] M. West, A. Barth, and D. Veditz. Content Security Policy Level 2. W3C Candidate Recommendation, 2015.

[29] M. West and I. Grigorik. Feature Policy. W3C Draft Community Group Report, 2016.

[30] I. Yusof and A. K. Pathan. Mitigating Cross-Site Scripting Attacks with a Content Security Policy. *IEEE Computer*, 49(3):56–63, 2016.

Type Abstraction for Relaxed Noninterference*

Raimil Cruz[1], Tamara Rezk[2], Bernard Serpette[2], and Éric Tanter[1]

1 **PLEIAD Lab, Computer Science Department (DCC), University of Chile**
{racruz,etanter}@dcc.uchile.cl
2 **INRIA - Indes Project-Team, Sophia Antipolis, France**
first.last@inria.fr

Abstract

Information-flow security typing statically prevents confidential information to leak to public channels. The fundamental information flow property, known as *noninterference*, states that a public observer cannot learn anything from private data. As attractive as it is from a theoretical viewpoint, noninterference is impractical: real systems need to intentionally declassify some information, selectively. Among the different information flow approaches to declassification, a particularly expressive approach was proposed by Li and Zdancewic, enforcing a notion of *relaxed noninterference* by allowing programmers to specify *declassification policies* that capture the intended manner in which public information can be computed from private data. This paper shows how we can exploit the familiar notion of type abstraction to support expressive declassification policies in a simpler, yet more expressive manner. In particular, the type-based approach to declassification—which we develop in an object-oriented setting—addresses several issues and challenges with respect to prior work, including a simple notion of label ordering based on subtyping, support for recursive declassification policies, and a local, modular reasoning principle for relaxed noninterference. This work paves the way for integrating declassification policies in practical security-typed languages.

1998 ACM Subject Classification D.4.6 Security and Protection: Information flow controls, D.3.2 Language Classifications: Object-oriented languages

Keywords and phrases type abstraction, relaxed noninterference, information flow control

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2017.53

1 Introduction

Information-flow security typing enables statically classifying program entities with respect to their confidentiality levels, expressed via a lattice of security labels [17]. For instance, a two-level lattice $\mathsf{L} \preccurlyeq \mathsf{H}$ allows distinguishing public or low data (*e.g.* Int_L) from confidential or high data (*e.g.* Int_H). An information-flow security type system statically ensures *noninterference*, *i.e.* that high-confidentiality data may not flow directly or indirectly to lower-confidentiality channels [35]. To do so, the security type system tracks the confidentiality level of computation based on the confidentiality of the data involved.

As attractive as it is, noninterference is too strict to be useful in practice, as it prevents confidential data to have *any* influence whatsoever on observable, public output. Indeed, even a simple password checker function violates noninterference. Consider the following:

* This work was partially funded by Project Conicyt REDES 140219 "CEV: Challenges in Practical Electronic Voting". Raimil Cruz is funded by CONICYT-PCHA/Doctorado Nacional/2014-63140148.

```
String login(String guess, String password)
  if(password == guess)
    return "Login Successful"
  else
     return "Login failed"
}
```

By definition, a *public observer* that tries to log in *should* be able to "learn something" about the confidential input (here, password), thereby violating the confidentiality restriction imposed by noninterference.

This problem with noninterference has long been recognized. Supporting such intentional downward information flows is called *declassification*, which can be supported in many different ways [28]. For example, Jif [23] supports an explicit `declassify` operator to allow downward flows to be accepted by the security type system. In the above example, one can use `declassify(password == guess)` to state that the returned value is public knowledge. However, arbitrary uses of a `declassify` operator may lead to serious information flow leaks; for instance `declassify(password)` simply makes the password publicly available. One solution adopted by Jif is to control declassification using principals with privileges, as in the Decentralized Label Model (DLM) [24]. Trusted declassification [20] restricts Jif's mechanism to specify authorization in a global policy file and formulate *noninterference modulo trusted methods*. Robust declassification [39] relies on integrity to ensure that low integrity flows do not influence high confidentiality data that will later be declassified.

To capture the essence of expressive declassification without appealing to additional mechanisms like integrity or authority, Li and Zdancewic proposed an expressive mechanism for declassification policies that supports the extensional specification of secrets and their intended declassification [21]. A declassification policy is a function that captures *what* information on a confidential value can be *declassified* to eventually produce a public value. For the password checker example, if the declassification policy for password is $\lambda x.\lambda y.x\texttt{==}y$, then an equality comparison with password can be declassified (and thus be made public). However, this declassification policy for password disallows arbitrary declassifications such as revealing the password. Furthermore, declassification can be *progressive*, requiring several operations to be performed in order to obtain public data: *e.g.* $\lambda x.\lambda y.\texttt{hash}(x)\texttt{==}y$ specifies that only the result of comparing the hash of the password for equality can be made public.

The formal security property, called *relaxed noninterference*, states that a secure program can be rewritten into an equivalent program without any variable containing confidential data but whose inputs are confidential and declassified. For the password checker example with $p \triangleq \lambda x.\lambda y.x\texttt{==}y$ as the declassification policy for password, the program login(guess,password) can be rewritten to the equivalent program login'(guess,p(password)) where login' is:

```
String login'(String guess, String→Bool eq){
  if(eq(guess)) ...
}
```

Note that p(password) is a closure that strongly encapsulates the secret value, and only allows equality comparisons.

While the proposal of Li and Zdancewic elegantly and formally captures the essence of flexible declassification while retaining a way to state a clear and extensional security property of interest, it suffers from a number of limitations that jeopardize its practical adoption. First, security labels are sets of functions that form a security lattice whose ordering, based on a semantic interpretation of these sets of functions, is far from trivial [21]:

it relies on a general notion of program equivalences that would be both hard to implement and to comprehend. Second, Li and Zdancewic explicitly rule out *recursive* declassification policies, which are however natural when expressing declassification of recursive data structures. Finally, the rewriting-based definition of relaxed noninterference is unsatisfying for practical software development, as it rigidly requires all secrets to be both *global* and *external*, thereby losing modular reasoning; as recognized by the authors, local language constructs for introducing secrets and their policies are lacking [21].

In this work, we exploit the familiar notion of *type abstraction* to capture declassification policies in a simpler, yet more expressive manner. Type abstraction in programming languages manifests in different ways [25]; here, we specifically adopt the setting of object-oriented programming, where object types are *interfaces*, *i.e.* the set of methods available to the client of an object, and type abstraction is driven by subtyping. For instance, the empty interface type—the root of the subtyping hierarchy—denotes an object that hides all its attributes, which intuitively coincides with secret data, while the interface that coincides with the implementation type of an object exposes all of them, which coincides with public data. Our initial observation is that any interface in between these two extremes denotes *declassification* opportunities. Additionally, choosing objects, as opposed to records, allows us to explore recursive declassification policies from the start, given that the essence of data abstraction in OOP are recursive types [16].

The type-based approach to confidentiality is very intuitive as it only relies on concepts that are readily available in object-oriented languages: a declassification policy is simply a *method signature*, a security label is an *object interface*, and label ordering boils down to *subtyping*. Progressive declassification occurs through chaining of *method invocations*. In fact, the only extension to the standard programming model is that a security type has two facets, each representing the view available to a private and public observer, respectively. In addition to being intuitive, the type-based approach addresses the issues and challenges of the downgrading policies of Li and Zdancewic: *a)* there is no need to rely on general program equivalences to define and decide label ordering, which is just standard, syntactic subtyping; *b)* declassification naturally scales to recursive policies over recursive data structures; and *c)* type-based relaxed noninterference is formulated as a *modular* reasoning principle, and local secrets can be introduced with standard type annotations.

This work makes the following contributions:

- We develop a novel type-based approach to declassification policies, which supports interesting scenarios while appealing to standard programming concepts such as interface types and subtyping (Section 2).
- We capture the essence of type-based declassification in a core object-oriented language, $\mathsf{Ob}_{\mathsf{SEC}}$, in which a security type is a pair of (recursive) object types (Section 3). We describe the static and dynamic semantics of $\mathsf{Ob}_{\mathsf{SEC}}$ and prove type *safety*.
- We specify the formal semantic notion of *type-based relaxed noninterference*, which accounts for type-based declassification policies, independently of any enforcement mechanism (Section 4). We then prove type *soundness* of $\mathsf{Ob}_{\mathsf{SEC}}$: a well-typed program satisfies type-based relaxed noninterference.
- We informally explore how the expressiveness of declassification policies scales with the expressiveness of types (Section 5), identifying interesting venues for extensions.

Section 6 discusses related work and Section 7 concludes. Auxiliary definitions are provided in Appendix.

2 Type-Based Declassification Policies

We now progressively and informally introduce the type-based approach to declassification policies, appealing first to a simple intuitive connection with type abstraction. We then explain why this first intuition is insufficient, and refine it in order to support the key features of a security-typed language with expressive declassification. We end by discussing the security guarantee supported by the approach.

Type abstraction and confidentiality. It is well-known that type abstraction can capture the need to expose only a subset of the operations of an object. For instance, if the password secret is made available using the interface type StringEq $\triangleq$ [eq : String $\to$ Bool], the `login` function from Section 1 can be rewritten as follows:

```
String login(String guess, StringEq password){
  if(password.eq(guess)) ...
}
```

Because password has type StringEq, the login function cannot accidentally leak information about the password. In particular, note that the function cannot even return the password because StringEq is a *supertype* of String, not a subtype. Therefore, the standard substitutability expressed by subtyping seems to align well with the valid information flows permitted in a confidentiality type system: a (public) string value at type String can be used freely, and passed as argument expecting a (mostly) private StringEq, which only exposes equality comparison. Similarly, any value can flow to a private variable, characterized by the empty interface type, $\top \triangleq [\,]$.[1]

Progressive declassification policies can be expressive with *nested* interface types. For instance, assume that String objects have a hash method, of type Unit $\to$ Int. To specify that only the hash of the password can be compared for equality, it suffices to expose the password at type StringHashEq $\triangleq$ [hash : Unit $\to$ IntEq], where IntEq $\triangleq$ [eq : Int $\to$ Bool]:

```
String login(Int guess, StringHashEq password){
  if(password.hash().eq(guess)) ...
}
```

In the code above, the only available operation on password is hash(), which in turn returns an integer that only exposes an equality comparison. Note that here again, StringHashEq $>:$ String and IntEq $>:$ Int.

Recursive declassification. The informal presentation of type-based declassification so far has exemplified two of the main advantages of our approach: security label ordering is syntactic subtyping, and secrets and their declassification policies can be declared locally, by standard type annotations. We now illustrate recursive declassification policies.

Recursive declassification policies are desirable to express interesting declassification of either inductive data structures or object interfaces (whose essence are recursive types [16]). Consider for instance a secret list of strings, for which we want to allow traversal of the

[1] The reader might wonder at this point about the effect of using arbitrary downcasts, as supported in Java. Indeed, downcasts are a way to violate type abstraction, and therefore to violate the type-based security guarantees. For instance, the login function could return (String)password, thereby returning the password for public consumption. Fortunately, there is a simple solution to this issue, which we discuss in Section 5.

structure and comparison of its elements with a given string. This can be captured by the recursive type StrEqList defined as:

$$\mathsf{StrEqList} \triangleq [\mathsf{isEmpty} : \mathsf{Unit} \to \mathsf{Bool},\ \mathsf{head} : \mathsf{Unit} \to \mathsf{StringEq},\ \mathsf{tail} : \mathsf{Unit} \to \mathsf{StrEqList}]$$

To allow traversal, the declassification policy exposes the methods isEmpty, head and tail, with the specific constraints that *a)* accessing an element through head yields a StringEq, not a full String, and *b)* the tail method returns the tail of the list *with the same* declassification policy. Type-based declassification policies can therefore naturally be recursive, as long as the underlying type language allows (some form of) recursive types.

Facets of computation. With the standard programming approach described so far, a program that attempts to violate the declassification protocol of an object is rejected by the (standard) type system because it is ill-typed. For instance:

```
String login(Int guess, StringEq password){
  if(password.length().eq(guess)) ...
}
```

is rejected because length is not part of the exposed interface of password.

However, security-typed languages typically are more flexible than this: they allow computation to proceed with private information, but ensure the result of such computation is itself private [37]. For instance, adding a public integer and a private integer yields a private result. Li and Zdancewic follow the same approach with declassification policies: using a secret in a way that does not follow its declassification policy yields a private result [21]. The justification of these approaches is that computation with private data *is* relevant, but only visible to a high security, private observer; noninterference only dictates that a low security, public observer should not be able to deduce information about private data by observing public outputs.

This means that security-typed languages inherently adopt a multi-faceted view of computation, where each observation level corresponds to a different facet. Sticking to a two-facet, private/public model, the definition of login above is well-typed if one "knows" that password is in fact a String object. In this case using length is valid: it just yields a private result. Flow-sensitivity then ensures that the result of login, which follows from a conditional branching computed based on a private value, is also private.

Faceted types. To accommodate the possibility of computing with private data, we extend standard types to *faceted types*. A security type S, noted $T \triangleleft U$, consists of two standard types: type T for the private interface, and type U for the public interface.[2]. In this paper, we often use the notation T_{L} as a shortcut for the lowest-confidentiality security type $T \triangleleft T$, in which the public facet exposes the same interface as the private facet, and T_{H} for the fully-confidential security type $T \triangleleft \top$ in which the public facet is empty.

To express that password is a private string that can only be declassified through equality comparison, we can use the following signature for login:

```
String_L login(Int_L guess, String◁StringEq password)
```

[2] Similarly to multi-faceted execution [**?**], one can extend the model to support n levels of observations, by introducing security types with n facets.

With this signature the previous definition of login, which invokes length, is still ill-typed. Indeed, the body of the function now has type $\mathsf{String}_\mathsf{L}$, capturing the fact that the resulting string is private, but the signature pretends that the result of login is public, which violates noninterference. For login to be well-typed, either the declared return type should be changed to $\mathsf{String}_\mathsf{H}$, or the conditional should adhere to the public facet StringEq.

Note that subtyping naturally extends *covariantly* to faceted types, *i.e.* $T_1 \triangleleft U_1 <: T_2 \triangleleft U_2$ iff both $T_1 <: T_2$ and $U_1 <: U_2$. Therefore, it is invalid to pass a private string of type $\mathsf{String} \triangleleft \top$ to a function expecting a declassifiable string of type $\mathsf{String} \triangleleft \mathsf{StringEq}$, because $\top$ is not a subtype of StringEq. Subtyping on the public facet corresponds to security label ordering; compared to the semantic, equivalence-based interpretation of labels of Li and Zdancewic, here label ordering is just standard syntactic subtyping.

Object types directly support the possibility to offer different declassification paths for the same secret. For instance, the security type $\mathsf{String} \triangleleft [\mathsf{hash} : \mathsf{Unit}_\mathsf{L} \to \mathsf{Int}_\mathsf{L}, \mathsf{length} : \mathsf{Unit}_\mathsf{L} \to \mathsf{Int}_\mathsf{L}]$ allows a client to obtain a public integer from a string by using either its hash or its length. Naturally, by breadth subtyping, such a secret with two possible declassification paths can also be used as a more restricted secret, *e.g.* one that only exposes its hash publicly.

Type-based relaxed noninterference. The security property we establish in this work is a particular form of termination insensitive noninterference, called *typed-based relaxed noninterference* (TRNI for short). Like the relaxed noninterference result of Li and Zdancewic [21], TRNI accounts for declassification policies.

To understand the intuition behind TRNI, we must first establish a notion of type-based observational equivalence between objects. The starting point of the notion of equivalence is that an object is defined by the observations that can be made on it, that is, by invoking its methods [16]. More precisely, two objects o_1 and o_2 are said to be observationally equivalent at type S, with $S \triangleq T \triangleleft U$, if for each method $m : S_1 \to S_2$ of the *public* facet U, invoking m on o_1 and o_2 with equivalent arguments at type S_1, yields equivalent results at type S_2. Crucially, the definition of equivalence uses the *public* facet of the type, thereby accounting for observational equivalence only up to declassified information.

For example, the strings `"john"` and `"mary"` are not equivalent at type $\mathsf{String} \triangleleft \mathsf{String}$, because a public observer can observe the first character of each string and realize they are different. However, these strings are equivalent when observed at $\mathsf{String} \triangleleft \mathsf{StringLen}$, where $\mathsf{StringLen} \triangleq [\mathsf{length} : \mathsf{Unit}_\mathsf{L} \to \mathsf{Int}_\mathsf{L}]$, because the only declassified information about the strings is their length, which is here equal. This also means that `"john"` and `"james"` are equivalent when are observed at type $\mathsf{String}_\mathsf{H}$ (*i.e.* $\mathsf{String} \triangleleft \top$) since there are no observations available to distinguish them. In fact, any two objects of type T are equivalent at type T_H.

Given this notion of equivalence, a program satisfies TRNI at type S_{out}, if given two inputs that are equivalent at type S_{in}, it produces two results that are equivalent at type S_{out}. Intuitively, the types S_{in} and S_{out} capture the *knowledge* of public observers. Another way to understand TRNI is that, if the initial knowledge *implies* the final knowledge, then the program is secure for the public observer.

For instance, consider a program with an input x of type $\mathsf{String} \triangleleft \mathsf{StringLen}$. The program x.length satisfies TRNI at type $\mathsf{Int} \triangleleft \mathsf{Int}$: two executions of the program with related inputs at $\mathsf{String} \triangleleft \mathsf{StringLen}$, such as `"john"` and `"mary"`, yields two identical results at type $\mathsf{Int} \triangleleft \mathsf{Int}$ (*i.e.* 4 in both cases). However, the program if(x.eq("mary")) return 1 else 2 does not satisfy TRNI at type $\mathsf{Int} \triangleleft \mathsf{Int}$ because there are equivalent inputs at type $\mathsf{String} \triangleleft \mathsf{StringLen}$ (`"john"` and `"mary"`) that yield different outputs at type $\mathsf{Int} \triangleleft \mathsf{Int}$ (1 and 2). For this program, the only secure observation level is $\mathsf{Int} \triangleleft \top$.

$$
\begin{array}{rcll}
e & ::= & v \mid e.m(e) \mid x & \text{(terms)} \\
v & ::= & \left[z : S \Rightarrow \overline{m\,(x)\,e}\right] & \text{(values)} \\
T, U & ::= & O \mid \alpha & \text{(types)} \\
O & ::= & \mathbf{Obj}(\alpha).\,\left[\overline{m : S \to S}\right] & \text{(object type)} \\
S & ::= & T \triangleleft U & \text{(security type)}
\end{array}
\qquad
\begin{array}{rl}
x, y, z & \text{(variables)} \\
\alpha, \beta & \text{(type variables)} \\
m & \text{(method labels)}
\end{array}
$$

$$T_{\mathsf{L}} \triangleq T \triangleleft T \qquad T_{\mathsf{H}} \triangleq T \triangleleft \top$$

Figure 1 $\mathsf{Ob}_{\mathsf{SEC}}$: Syntax

We formally define these notions, and prove that the type system we propose enforces TRNI, in Section 4.

3 An Object Language for Type-Based Declassification

We develop type-based declassification and relaxed noninterference using a core object-oriented language, $\mathsf{Ob}_{\mathsf{SEC}}$, whose syntax is presented in Figure 1. The syntax of $\mathsf{Ob}_{\mathsf{SEC}}$ is similar to that of the object calculi of Abadi and Cardelli [2]. It includes three kinds of expressions: variables, objects and method invocations. Note that we do not include method updates or classes, both unnecessary to formulate our proposal. An object $\left[z : S \Rightarrow \overline{m\,(x)\,e}\right]$ is a collection of method definitions, where method names are unique. The object definition explicitly binds the self variable z in method bodies, with ascribed security type S. The distinguishing feature of $\mathsf{Ob}_{\mathsf{SEC}}$ are security types: as introduced in Section 2, a security type S is a two-faceted type $T \triangleleft U$, where T (resp. U) is the private (resp. public) facet. The public facet corresponds to the declassification policy of an object. A fully opaque secret has type $T \triangleleft \top$ (also noted T_{H}), exposing no method at all, while a low-confidentiality object has type $T \triangleleft T$ (also noted T_{L}), publicly exposing its full interface. A type T or U is either a (recursive) object type $\mathbf{Obj}(\alpha).\,\left[\overline{m : S \to S}\right]$, where method types can use the self type variable α, or a type variable. Note that we do not model parametric polymorphism in this core calculus, so type variables are only used for self types. Following the tradition of Abadi and Cardelli [2], $\mathsf{Ob}_{\mathsf{SEC}}$ does not include base (non-object) types, however they can be easily added or encoded.

Subtyping. The $\mathsf{Ob}_{\mathsf{SEC}}$ subtyping judgment $\Phi \vdash T <: U$ is presented in Figure 2. The subtyping environment Φ is a set of subtyping assumptions between type variables, *i.e.* $\Phi ::= \cdot \mid \Phi, \alpha <: \beta$.[3] For all judgments in this work, we often omit the empty environment, *e.g.* we write $\vdash T <: U$ for $\cdot \vdash T <: U$.

Rule (SObj) accounts for subtyping between object types. Object type T_1 is a subtype of object type T_2 if T_1 has at least the same methods as T_2, possibly more specialized. For this, the rule checks subtyping between method types under a subtyping assumption between the self type variable of T_1 and that of T_2. For instance, consider the following object types:

$$
\begin{array}{rcl}
\mathsf{Counter} & \triangleq & \mathbf{Obj}(\alpha).\,[\mathsf{get} : \mathsf{Unit}_{\mathsf{L}} \to \mathsf{Int}_{\mathsf{L}},\ \mathsf{inc} : \mathsf{Unit}_{\mathsf{L}} \to \alpha_{\mathsf{L}},\ \mathsf{dec} : \mathsf{Unit}_{\mathsf{L}} \to \alpha_{\mathsf{L}}] \\
\mathsf{IncCounter} & \triangleq & \mathbf{Obj}(\beta).\,[\mathsf{get} : \mathsf{Unit}_{\mathsf{L}} \to \mathsf{Int}_{\mathsf{L}},\ \mathsf{inc} : \mathsf{Unit}_{\mathsf{L}} \to \beta_{\mathsf{L}}]\,.
\end{array}
$$

To establish that Counter is a subtype of IncCounter, the covariance between the return types of the `inc` method requires a subtyping assumption between type variables, here $\alpha <: \beta$. Rule (SVar) specifies subtyping between type variables, which only holds if the relation is

[3] Type variables must appear at most once in the subtyping environment.

$\boxed{\Phi \vdash T <: T}$

$$(\text{SObj})\ \frac{O_1 \triangleq \mathbf{Obj}(\alpha).\ [\overline{m : S_1 \to S_2}] \quad O_2 \triangleq \mathbf{Obj}(\beta).\ [\overline{m' : S'_1 \to S'_2}] \quad \overline{m'} \subseteq \overline{m} \quad m_i = m'_j \implies (\Phi, \alpha <: \beta \vdash S'_{1j} <: S_{1i} \quad \Phi, \alpha <: \beta \vdash S_{2i} <: S'_{2j})}{\Phi \vdash O_1 <: O_2}$$

$$(\text{SVar})\ \frac{\alpha <: \beta \in \Phi}{\Phi \vdash \alpha <: \beta} \qquad (\text{SSubEq})\ \frac{O_1 \equiv O_2}{\Phi \vdash O_1 <: O_2} \qquad (\text{STrans})\ \frac{\Phi \vdash T_1 <: T_2 \quad \Phi \vdash T_2 <: T_3}{\Phi \vdash T_1 <: T_3}$$

$\boxed{\Phi \vdash S <: S}$

$$(\text{TSubST})\ \frac{\Phi \vdash T_1 <: T_2 \quad \Phi \vdash U_1 <: U_2}{\Phi \vdash T_1 \triangleleft U_1 <: T_2 \triangleleft U_2}$$

Figure 2 $\mathsf{Ob}_{\mathsf{SEC}}$: Subtyping rules

$\boxed{\mathsf{methsig}(O, m) = S \to S}$

$$\frac{O \triangleq \mathbf{Obj}(\alpha).\ [\overline{m : S_1 \to S_2}] \quad S \triangleq S_{1i}\,[O/\alpha] \quad S' \triangleq S_{2i}\,[O/\alpha]}{\mathsf{methsig}(O, m_i) = S \to S'}$$

$\boxed{m \in O}$

$$\frac{O \triangleq \mathbf{Obj}(\alpha).\ [\overline{m : S_1 \to S_2}]}{m_i \in O}$$

$\boxed{\mathsf{methimpl}(o, m) = x.e}$

$$\frac{o \triangleq [z : S \Rightarrow \overline{m\,(x)\,e}]}{\mathsf{methimpl}(o, m_i) = x.e_i}$$

Figure 3 $\mathsf{Ob}_{\mathsf{SEC}}$: Some auxiliary definitions

in the subtyping environment. Rule (SSubEq) justifies subtyping between *equivalent types*. We consider type equivalence up to renaming and folding/unfolding of self type variables; for instance:

$\mathbf{Obj}(\alpha).\ [\mathsf{m} : \alpha_{\mathsf{L}} \to \alpha_{\mathsf{L}}] \equiv \mathbf{Obj}(\beta).\ [\mathsf{m} : \beta_{\mathsf{L}} \to \beta_{\mathsf{L}}]$ (alpha equivalence)
$\mathbf{Obj}(\alpha).\ [\mathsf{m} : S \to \alpha_{\mathsf{L}}] \equiv \mathbf{Obj}(\alpha).\ [\mathsf{m} : S \to \mathbf{Obj}(\beta).\ [\mathsf{m} : S \to \beta_{\mathsf{L}}]_{\mathsf{L}}]$ (fold/unfold equivalence)

(Appendix A.4 provides the complete definition of type equivalence.)

Rule (STrans) is standard. Rule (TSubST) justifies subtyping between security types, which is covariant in both facets.

Figure 3 presents auxiliary functions used to test method membership in a type ($m \in T$), to get the type of a method in an object type (methsig) and to get the implementation of a method (methimpl). These operations are standard; the only interesting thing to note is that in methsig we close the types in the method signature, by replacing type variables with their object types.

Static semantics. Figure 4 shows the typing rules of $\mathsf{Ob}_{\mathsf{SEC}}$. The type judgment $\Gamma \vdash e : S$ gives a security type to an expression under a type environment Γ that binds variables to types ($\Gamma ::= \cdot \mid \Gamma, x : S$). In what follows, we assume well-formedness of types and environments: informally, an environment is well-formed if all security types are closed and well-formed; a well-formed security type satisfies the requirement that the private type is a subtype of the public type. We further discuss well-formedness at the end of this section.

Rules (TVar) and (TSub) are standard. The (TObj) rule accounts for objects. It requires

$\boxed{\Gamma \vdash e : S}$

$$(\text{TVar})\frac{x \in dom(\Gamma)}{\Gamma \vdash x : \Gamma(x)} \qquad (\text{TSub})\frac{\Gamma \vdash e : S' \quad \vdash S' <: S}{\Gamma \vdash e : S}$$

$$(\text{TObj})\frac{S \triangleq T \triangleleft U \quad \mathsf{methsig}(T, m_i) = S_i' \to S_i'' \quad \Gamma, z : S, x : S_i' \vdash e_i : S_i''}{\Gamma \vdash \left[z : S \Rightarrow \overline{m\,(x)\,e}\right] : S}$$

$$(\text{TmD})\frac{\Gamma \vdash e_1 : T \triangleleft U \qquad m \in U \qquad \mathsf{methsig}(U, m) = S_1 \to S_2 \qquad \Gamma \vdash e_2 : S_1}{\Gamma \vdash e_1.m(e_2) : S_2}$$

$$(\text{TmH})\frac{\Gamma \vdash e_1 : T \triangleleft U \qquad m \notin U \qquad \mathsf{methsig}(T, m) = S_1 \to T_2 \triangleleft U_2 \qquad \Gamma \vdash e_2 : S_1}{\Gamma \vdash e_1.m(e_2) : T_2 \triangleleft \top}$$

Figure 4 $\mathsf{Ob}_{\mathsf{SEC}}$: Static semantics

each method body to be well-typed with respect to the private facet of the object. In particular, the method body must match the return type of the method signature in the private facet of the self type S.

From a security point of view, the interesting rules are the ones for method invocation. Rule (TmD) applies when the invoked method is part of the *public* facet of the receiver. In this case, because the method invocation respects the declassification policy, the overall type of the invocation is the return type of the method in the public facet. This expresses that the invocation advances a step in the progressive declassification of the object. For instance, if the expression e_1 has the public type $\mathsf{StringHashEq} \triangleq [\mathsf{hash} : \mathsf{Unit}_{\mathsf{L}} \to \mathsf{Int} \triangleleft \mathsf{IntEq}]$, the invocation $e_1.\mathsf{hash}()$ has type $\mathsf{Int} \triangleleft \mathsf{IntEq}$, expressing that the returned value is a secret that can further be declassified by calling the method eq from IntEq.

Rule (TmH) applies when the method is not in the public type U, but only in the private type T (if the method is not in T, the expression is ill typed). In this case, the method call is accessing the "secret" part of the object: the result of the method invocation must therefore be protected by changing its public facet to $\top$. This rule captures the design decision that using a secret beyond its declassification policy is allowed, but the result must be secret. In other words, only a private observer can use objects beyond their declassification policies; to a public observer, the results of these interactions are unobservable.[4]

Dynamic semantics. We define a standard call-by-value small-step semantics for $\mathsf{Ob}_{\mathsf{SEC}}$, based on evaluation contexts $E ::= [\,] \mid E.m(e) \mid v.m(E)$.

The language includes a single reduction rule, for method invocation, which is standard:

$$(\text{EMInv})\frac{o \triangleq [z : \_ \Rightarrow \_] \qquad \mathsf{methimpl}(o, m) = x.e}{E[o.m(v)] \longmapsto E[e\,[o/z]\,[v/x]]}$$

Type safety. We now establish that well-typed $\mathsf{Ob}_{\mathsf{SEC}}$ programs are safe. Note that type safety does not provide any *security guarantees* for $\mathsf{Ob}_{\mathsf{SEC}}$. (Security guarantees will be

[4] Access modifiers in object-oriented languages, such as private and public in Java are a really different mechanism. Such modifiers are about *encapsulation*, not about *information flow*. The essential difference can be observed in rule (TmH), which propagates privacy on return values.

$$\begin{array}{rcl}
\mathcal{V}_k[\![S]\!] & = & \{v = [z : S_1 \Rightarrow _] \mid S \triangleq T \triangleleft U \quad \vdash S_1 <: S \wedge \\
& & \quad (\forall j < k.\ v \in \mathcal{V}_j[\![S_1]\!] \wedge \\
& & \quad\quad (\forall m \in T, v'.\ \mathsf{methsig}(T, m) = S' \to S'' \quad \mathsf{methimpl}(v, m) = x.e \\
& & \quad\quad\ v' \in \mathcal{V}_j[\![S']\!] \implies e\,[v/z]\,[v'/x] \in \mathcal{C}_j[\![S'']\!]))\} \\
\\
\mathcal{C}_k[\![S]\!] & = & \{e \mid \forall j < k.\ \forall e'.(e \longmapsto^j e' \ \wedge\ \mathsf{irred}(e')) \implies e' \in \mathcal{V}_{k-j}[\![S]\!]\}
\end{array}$$

Figure 5 $\mathsf{Ob}_{\mathsf{SEC}}$: Unary logical relation for safety

addressed in Section 4.) A program e is *safe*, noted $\mathsf{safe}(e)$, if it does not get stuck, *i.e.* if it either reduces to a value or diverges.

▶ **Definition 1** (Safety). $\mathsf{safe}(e) \iff \forall e'.\ e \longmapsto^* e' \implies e' = v \text{ or } \exists e''.\ e' \longmapsto e''$

We prove type safety for $\mathsf{Ob}_{\mathsf{SEC}}$ using a semantic interpretation of types as a unary logical relation [3]. We cannot however define the logical relation based on a direct induction over the structure of types, because of recursive types, which would make such a definition ill-founded. Therefore, we use a step-indexed logical relation [4, 6]. We establish an intermediary result for a fixed number k of steps, meaning that a term is safe for k evaluation steps, and then quantify $\forall k \geq 0$ to obtain the general result. Step indexing ensures the well-foundedness of the logical relation.

Figure 5 defines the unary logical relation that captures the safety interpretation of types as values and computations, in a mutually recursive manner. The set $\mathcal{V}_k[\![S]\!]$ denotes the safe value interpretation of type S for k steps; it contains all the *values* (*i.e.* objects) for which it is safe (for any $j < k$ number of steps) to invoke methods of the private type T of the security type $S \triangleq T \triangleleft U$. Note that the definition needs to assume that the self object is in the value interpretation of S, for $j < k$ steps; without step-indexing, this relation would be ill-founded due to the recursive nature of objects through their self variables. The set $\mathcal{C}_k[\![S]\!]$ contains all the *expressions* that can be safely executed for k steps at the security type S. In the definition, the $\mathsf{irred}(e)$ predicate denotes irreducible expressions, *i.e.* expressions e such that $\nexists e'.e \longmapsto e'$.

We define *semantic typing*, written $\models e : S$, to denote that a closed expression e executes safely for any fixed number of steps:

▶ **Definition 2** (Semantic typing). $\models e : S \iff \forall k \geq 0.\ e \in \mathcal{C}_k[\![S]\!]$.

We then first prove that semantic typing does imply safety as per Definition 1.

▶ **Lemma 3** (Semantic type safety). $\models e : S \implies \mathsf{safe}(e)$

Proof. To show $\mathsf{safe}(e)$ we need to consider an arbitrary e' such that $e \longmapsto^* e'$ and then show that either $e' = v$ or $\exists e''.\ e' \longmapsto e''$
Let us consider an arbitrary j_1 to count the step that takes $e \longmapsto^* e'$. Let us denote $l = j_1 + 1$
By expanding the definition of $\models e : S$ we have $\forall k \geq 0.\ e \in \mathcal{C}_k[\![S]\!]$. We instantiate this with $k = l$ to obtain $e \in \mathcal{C}_l[\![S]\!]$. By expanding this we have:
$\forall j < l.\ \forall e_1.(e \longmapsto^j e_1 \ \wedge\ \mathsf{irred}(e_1)) \implies e_1 \in \mathcal{V}_{k-j}[\![S]\!]$. We instantiate $e \in \mathcal{C}_l[\![S]\!]$ with j_1 and e' and we obtain: $(e \longmapsto^{j_1} e' \ \wedge\ \mathsf{irred}(e')) \implies e' \in \mathcal{V}_{k-j_1}[\![S]\!]$.
There are two cases to consider: $\neg\mathsf{irred}(e')$ and $\mathsf{irred}(e')$. If $\neg\mathsf{irred}(e')$, then by definition $\exists e''.\ e' \longmapsto e''$. If $\mathsf{irred}(e')$, we have that $e' \in \mathcal{V}_{k-j}[\![S]\!]$, so e' is a value. ◀

Second, we prove that syntactic typing (Figure 4) implies semantic typing.

▶ **Lemma 4** (Syntactic typing implies semantic typing). $\vdash e : S \implies \models e : S$

Proof. The result follows from a similar lemma on open terms: $\Gamma \vdash e : S \implies \Gamma \models e : S$. We define a standard notion of safe value substitutions [3], *i.e.* partial maps from variables to safe values, $\gamma \in \mathcal{G}_k[\![\Gamma]\!]$ and $\Gamma \models e : S$ as follows:
$\gamma \in \mathcal{G}_k[\![\Gamma]\!] \iff dom(\gamma) = dom(\Gamma)$ and $\forall x \in dom(\Gamma).\gamma(x) \in \mathcal{V}_k[\![\Gamma(x)]\!]$
$\Gamma \models e : S \iff \forall k \geq 0,\ \forall \gamma.\ \gamma \in \mathcal{G}_k[\![\Gamma]\!] \implies \gamma(e) \in \mathcal{C}_k[\![S]\!]$.
Then we prove that $\Gamma \vdash e : S \implies \Gamma \models e : S$ by induction on the typing derivation of e. The case (TVar) is direct from the definition of $\gamma \in \mathcal{G}_k[\![\Gamma]\!]$. The case (TSub) follows directly from a subsumption lemma ($e \in \mathcal{C}_k[\![S]\!] \wedge \vdash S <: S' \implies e \in \mathcal{C}_k[\![S']\!]$). Cases (TObj), (TmD) and (TmH) are proven by unfolding the definitions of $\mathcal{C}_k[\![S]\!]$ and $\mathcal{V}_k[\![S]\!]$, and applying the induction hypotheses for smaller indexes. For these cases, we use mainly a monotonicity lemma for the value interpretation of a type regarding the index, *i.e.* $e \in \mathcal{V}_k[\![S]\!] \wedge j \leq k \implies v \in \mathcal{V}_j[\![S]\!]$. ◀

Together, Lemmas 3 and 4 imply that well-typed programs are safe.

▶ **Theorem 5** (Syntactic type safety). $\vdash e : S \implies \mathsf{safe}(e)$

Now that we have established that $\mathsf{Ob}_{\mathsf{SEC}}$ is a well-defined, type-safe language, Section 4 will develop its security guarantees.

A note on well-formedness. Before we proceed, however, we need to mention a technical yet important issue that we overlooked so far. For the main results of Section 4 to hold, we need to ensure that we work with *well-formed* security types, *i.e.* that the private facet type is a subtype of the public facet type. In a language with simple, non-recursive types, defining such subtyping constraints is straightforward. However, in the presence of recursive (object) types, defining the rules for the subtyping constraint of security types is rather subtle and involved. The subtlety with type variables is that, at some point, we might have to check well-formedness of a security type with a type variable in one of its facets, *e.g.* $\alpha \triangleleft T$, without knowing any relation between α and T. To address this, we need *to remember* the surrounding recursive object type O that binds α, and to transform the check $\vdash \alpha <: T$ to $\vdash O <: T$. For conciseness, we leave out the well-formedness rules from the main body of the paper; they are fully described in Appendix A.2. In what follows, we systematically assume that security types (and by extension, type environments) are well-formed.

4 Type-Based Relaxed Noninterference

Faceted security types support information-flow security with declassification. The security property that type-based declassification supports is a form of relaxed noninterference [21], which we informally explained in Section 2. This section formally defines the notion of type-based relaxed noninterference (TRNI) *independently of any enforcement mechanism*. Then, we prove that the type system of $\mathsf{Ob}_{\mathsf{SEC}}$ is sound with respect to this property.

Type-based equivalence. As introduced in Section 2, TRNI is defined in terms of a notion of type-based equivalence between objects: a program satisfies TRNI at type S_{out}, if given two inputs at type S_{in}, it produces two equivalent results at type S_{out}. Equivalence at a type accounts for the possible observations (*i.e.* method invocations) that one is allowed to make on an object. We define this equivalence as a step-indexed logical relation [4], in Figure 6.

$$
\begin{array}{lcl}
v_1 \approx_k v_2 : \mathcal{V}[\![S]\!] & \iff & S \triangleq T \triangleleft U \quad v_i \triangleq [z : \_ \Rightarrow \_] \\
& & \vdash_1 v_i : T \wedge (\forall m \in U.\quad \mathsf{methsig}(U, m) = S' \to S'' \quad \mathsf{methimpl}(v_i, m) = x.e_i \\
& & \forall j < k, v_1', v_2'.\ \ v_1 \approx_j v_2 : \mathcal{V}[\![S]\!] \wedge \\
& & \quad (v_1' \approx_j v_2' : \mathcal{V}[\![S']\!] \implies e_1\,[v_1/z]\,[v_1'/x] \approx_j e_2\,[v_2/z]\,[v_2'/x] : \mathcal{C}[\![S'']\!])) \\
\\
e_1 \approx_k e_2 : \mathcal{C}[\![S]\!] & \iff & S \triangleq T \triangleleft U \\
& & \vdash_1 e_i : T \wedge (\forall j < k.(e_1 \longmapsto^{\leq j} v_1 \wedge e_2 \longmapsto^{\leq j} v_2) \implies v_1 \approx_{k-j} v_2 : \mathcal{V}[\![S]\!])
\end{array}
$$

Figure 6 Step-indexed logical relation for type-based equivalence

We define how to relate values (*i.e.* objects) as well as computations (*i.e.* expressions). Step indexing is required due to the recursive nature of object types, as explained below.

Note that the definitions use a simple typing judgment that does not account for security typing at all; its sole purpose is to ensure safety. This is crucial: the public facets of security types only play the role of *specifications* of declassification policies, and the logical relation specifies the *meaning* of these specifications, without any consideration for an enforcement mechanism. In particular, observe that the definitions in Figure 6 do *not* appeal to security type judgments ($\vdash$), but only to simple type judgments ($\vdash_1$).

▶ **Definition 6** (Simple typing judgment)**.** Based on the security typing judgment $\Gamma \vdash e : S$, we define the simple typing judgment $\Gamma \vdash_1 e : T$ by focusing only on the private facet of security types. Formally: $\Gamma \vdash_1 e : T \iff \Gamma \vdash e : T \triangleleft U$ for some U.
(The inductive definition of simple typing is in Appendix A.5.)

Intuitively, two objects v_1 and v_2 are equivalent at type $S \triangleq T \triangleleft U$ for k steps, noted $v_1 \approx_k v_2 : \mathcal{V}[\![S]\!]$, when one cannot distinguish them by invoking any method m of U. More precisely, to ensure safety, we first demand that both values are well-typed at T with the simple type system. Then, for each method $m \in U$ and every $j < k$, the invocations of m on v_1 and v_2 with related arguments at the argument type S' of m must be equivalent computations at the return type S'' for j steps, as defined below. Finally, note that the definition also requires that v_1 and v_2 are related self objects, for $j < k$ steps; this is necessary for the relation to be well-founded. (Observe that two simply well-typed objects are vacuously equivalent for zero steps.)

Two expressions e_1 and e_2 are equivalent at security type $S \triangleq T \triangleleft U$ for k steps, noted $e_1 \approx_k e_2 : \mathcal{C}[\![S]\!]$, if they are both (simply) well-typed at T and, provided that they both reduce to values in *at most* $j < k$ steps (noted $e \longmapsto^{\leq j} v$), then both values are equivalent at type S for the remaining $k - j$ steps. Note that this definition is termination insensitive: if one expression does not terminate in less than k steps, then both expressions are deemed equivalent.

Defining TRNI. The type-based approach to declassification policies allows us to formulate the corresponding relaxed noninterference property as a *modular* reasoning principle, similarly to the common formulation of noninterference in languages without declassification [37], thereby avoiding the global and external formulation of the transformation approach [21].

Standard noninterference is usually stated as a modular reasoning principle on open terms [37]: given a well-typed open term, which depends on some private variables, closing the term with private inputs yields equivalent programs when observed by a low-confidentiality observer. This statement can be generalized using the notion of *value substitutions*, *i.e.* partial maps from variables to values: given an open term that typechecks in

a given environment Γ, applying two *related* substitutions yields equivalent computations. Applying a substitution, noted $\gamma(e)$, substitutes the free variables of e with their values in γ.

▶ **Definition 7** (Satisfactory substitution)**.** A substitution γ satisfies type environment Γ, noted $\gamma \models \Gamma$, iff $dom(\gamma) = dom(\Gamma) \wedge \forall x \in dom(\Gamma).\ \vdash_1 \gamma(x) : T$ where $\Gamma(x) \triangleq T \triangleleft U$

▶ **Definition 8** (Related substitutions)**.** Two substitutions γ_1 and γ_2 are equivalent for k steps with respect to a type environment Γ, noted $\gamma_1 \approx_k \gamma_2 : \mathcal{G}[\![\Gamma]\!]$, if $\gamma_i \models \Gamma$ and

$$\forall x \in dom(\Gamma).\gamma_1(x) \approx_k \gamma_2(x) : \mathcal{V}[\![\Gamma(x)]\!]$$

The statement of type-based relaxed noninterference is a direct generalization of standard noninterference: an open term e, simply well-typed in environment Γ, satisfies type-based relaxed noninterference at security type S, noted $\mathsf{TRNI}(\Gamma, e, S)$, if two executions of e with related substitutions with respect to Γ produce equivalent computational expressions at type S, for any number of steps.

▶ **Definition 9** (Type-based relaxed noninterference)**.**

$$\begin{aligned}\mathsf{TRNI}(\Gamma, e, S) \iff\ & S \triangleq T \triangleleft U \quad \Gamma \vdash_1 e : T \wedge \\ & \forall k \geq 0.\ \forall \gamma_1, \gamma_2.\ \gamma_1 \approx_k \gamma_2 : \mathcal{G}[\![\Gamma]\!] \implies \gamma_1(e) \approx_k \gamma_2(e) : \mathcal{C}[\![S]\!]\end{aligned}$$

This definition captures the semantic characterization of TRNI-secure expressions, independently of any enforcement mechanism (recall that, in Figure 6, the public facets of security types only play the role of *specifications* of declassification policies). The $\mathsf{Ob}_{\mathsf{SEC}}$ type system is a sound, conservative enforcement mechanism for TRNI.

Security type soundness. To establish that well-typed $\mathsf{Ob}_{\mathsf{SEC}}$ programs satisfy TRNI, we first introduce a general notion of type-based equivalence between open expressions. Two open expressions, well-typed under a type environment Γ, are equivalent at a security type $S \triangleq T \triangleleft U$, if both expressions have simple type T, and given two related value substitutions for Γ, closing each expression with a satisfactory substitution yields equivalent expressions at type S.

▶ **Definition 10** (Equivalence of open terms)**.**

$$\begin{aligned}\Gamma \vdash e_1 \approx e_2 : S \iff\ & S \triangleq T \triangleleft U \quad \Gamma \vdash_1 e_i : T \wedge \\ & \forall k \geq 0.\ \forall \gamma_1, \gamma_2.\ \gamma_1 \approx_k \gamma_2 : \mathcal{G}[\![\Gamma]\!] \implies \gamma_1(e_1) \approx_k \gamma_2(e_2) : \mathcal{C}[\![S]\!]\end{aligned}$$

As is clear from the definitions, if a term is equivalent to itself at type S, then it satisfies TRNI at S.

▶ **Lemma 11** (Self-equivalence)**.** $\Gamma \vdash e \approx e : S \implies \mathsf{TRNI}(\Gamma, e, S)$

Type soundness of $\mathsf{Ob}_{\mathsf{SEC}}$ follows from the fact that the $\mathsf{Ob}_{\mathsf{SEC}}$ type system enforces such a self-equivalence.

▶ **Lemma 12** (Fundamental property)**.** $\Gamma \vdash e : S \implies \Gamma \vdash e \approx e : S$

Proof. The proof is by induction on the typing derivation of e. The (TVar) case follows directly from Definition 8 and the (TSub) case follows from a subtyping lemma: if $e_1 \approx_k e_2 : \mathcal{C}[\![S]\!]$ and $\vdash S <: S'$ then $e_1 \approx_k e_2 : \mathcal{C}[\![S']\!]$. The (TObj) case applies the induction hypothesis (IH) on method bodies. To use the IH results, we need to show that the value

substitutions that result from extending the current substitutions with both self and actual arguments are also related. This step requires auxiliary lemmas of monotonicity of the logical relations regarding smaller indexes. The (TmD) case follows from applying the IH over both subexpressions, selecting adequate indexes. The (TmH) case is simpler because there is no method to invoke in the public type $\top$. ◀

Finally, type soundness for $\mathsf{Ob}_{\mathsf{SEC}}$ follows directly from Lemmas 11 and 12.

▶ **Theorem 13** (Security type soundness). $\Gamma \vdash e : S \implies \mathsf{TRNI}(\Gamma, e, S)$

Illustration. We now illustrate the relation between the security typing and the definition of TRNI. In the examples we use some standard constructs like conditionals, not included in $\mathsf{Ob}_{\mathsf{SEC}}$, but easily encodable.

As introduced in Section 2, the property $\mathsf{TRNI}(\Gamma, e, T \triangleleft U)$ can be intuitively understood as: the initial knowledge of a public observer in Γ (*i.e.* the declassification policies) implies the final knowledge (*i.e.* the resulting public type U) that the observer has at hand to distinguish the results of two arbitrary executions of the *secure* program e of simple type T.

Let us recall the type $\mathsf{StringLen} \triangleq [\mathsf{length} : \mathsf{Unit_L} \to \mathsf{Int_L}]$ from the end of Section 2. Consider the open term $e \triangleq \mathsf{x.length}$ under the type environment $\Gamma \triangleq x : \mathsf{String} \triangleleft \mathsf{StringLen}$. The judgment $\Gamma \vdash e : \mathsf{Int_L}$ ensures that $\mathsf{TRNI}(\Gamma, e, \mathsf{Int_L})$ holds. It says that executing e, with two different strings v_1 and v_2 *of the same length* is secure because the observer *does not learn anything new* by exploiting the knowledge of distinguishing the resulting integers with any method of Int. In fact, if we use the definition of TRNI, for any equivalent substitutions γ_1 and γ_2 such that $\gamma_1 \approx_k \gamma_2 : \mathcal{G}[\![\Gamma]\!]$, such as $\gamma_i \triangleq x \mapsto v_i$, we need to show $\gamma_1(x).\mathsf{length}() \approx_k \gamma_2(x).\mathsf{length}() : \mathcal{C}[\![\mathsf{Int_L}]\!]$. It is easy to see that this result follows from the assumption that v_1 and v_2 have the same length (*i.e.* are equivalent at $\mathsf{String} \triangleleft \mathsf{StringLen}$).

We have a different situation if we consider $e' \triangleq \mathsf{if(x.eq("mary"))\ return\ 1\ else\ 2}$, with the same type environment Γ. We cannot prove that $\mathsf{TRNI}(\Gamma, e', \mathsf{Int_L})$ holds, meaning this program is *not* secure at type $\mathsf{Int_L}$. Indeed, take $\gamma_1 \triangleq x \mapsto \texttt{"mary"}$ and $\gamma_2 \triangleq x \mapsto \texttt{"john"}$. Because both strings have the same length, we have $\texttt{"mary"} \approx_k \texttt{"john"} : \mathcal{V}[\![\mathsf{String} \triangleleft \mathsf{StringLen}]\!]$, so the two substitutions are equivalent. However, we cannot show that $\gamma_1(e') \approx_k \gamma_2(e') : \mathcal{C}[\![\mathsf{Int_L}]\!]$, because this requires to show that $1 \approx_k 2 : \mathcal{V}[\![\mathsf{Int_L}]\!]$, which is obviously false.

The type system of $\mathsf{Ob}_{\mathsf{SEC}}$ indeed rejects the judgment $\Gamma \vdash e : \mathsf{Int_L}$. It does accept the judgment $\Gamma \vdash e : \mathsf{Int_H}$, meaning that e' *is* secure at type $\mathsf{Int_H}$. This is correct because then the public observer has no ability to compare the resulting values of e'. Note in fact that any simply well-typed expression of type T is secure at type T_{H}. Such expressions are opaque to a public observer, but are observable by a private observer.

Principles of declassification. Our approach to type-based declassification satisfies the declassification principles stated by Sabelfeld and Sands [28].[5] We now briefly introduce each principle and informally argue why it is respected.

- *Conservativity—i.e.* "*Security for programs with no declassification is equivalent to noninterference*". It is easy to see that if a program satisfies $\mathsf{TRNI}(\Gamma, e, T_{\mathsf{L}})$, for some T, and all security types in both Γ and e are either highly confidential (T_{H}) or not confidential

[5] Sabelfeld and Sands mention a fourth principle, *non-occlusion*, which addresses the interaction between declassification and covert channels, such as heap assignments, exceptions or termination behavior. $\mathsf{Ob}_{\mathsf{SEC}}$ has neither mutation nor control operators, and termination is not considered a covert channel because we only deal with termination-insensitive noninterference.

at all (T_{L}), then the definition of TRNI coincides exactly with the definition of pure noninterference [37]. Therefore type-based relaxed noninterference is a generalization of pure noninterference.

- *Monotonicity of Release—i.e.* "*Adding further declassifications to a secure program cannot render it insecure*". This lemma follows from subtyping naturally. Recall that in our approach, in the judgment $\mathsf{TRNI}(\Gamma, e, S)$, declassification policies come from types ascribed in both Γ and e. "Adding further declassification" in the inputs means in our context replacing security types in Γ with subtypes, more precisely, where the public facets are subtypes of the original types. The security typing judgment also holds in this scenario of additional declassification in the inputs. Similarly, adding declassification in the expression e means specializing the public facets of types in object type declarations. Again, this does not affect the semantic TRNI judgment. Note, however, that if argument types are specialized, the program might not be typable anymore with the security type system, as such a change breaks the contravariance of subtyping for argument method types.

- *Semantic Consistency—i.e.* "*The (in)security of a program is invariant under semantics-preserving transformations of declassification-free subprograms.*". The principle says that it is possible to replace an expression that does not use declassification with another semantically-equivalent expression, without affecting security. As observed by Sabelfeld and Sands, the approach to declassification policies of Li and Zdancewic [21] violates this principle, because they rely on a restricted, mostly-syntactic form of program equivalence to decide label ordering. Therefore, many semantically-equivalent programs are not deemed equivalent, hence affecting their (in)security. In contrast, our notion of type-based equivalence (Figure 6) is semantic, not syntactic.

Limitations of security typing. The $\mathsf{Ob}_{\mathsf{SEC}}$ type system is a *static* enforcement mechanism for type-based relaxed noninterference. As such, it is inherently conservative. This has two implications regarding Theorem 12.

First, the type system can reject some programs that are in fact secure. For example, consider the following definitions:

$T \triangleq \mathbf{Obj}(\alpha).\,[n : \mathsf{String}_{\mathsf{L}} \to \mathsf{String}_{\mathsf{L}}]$
$T' \triangleq \mathbf{Obj}(\alpha).\,[m : \mathsf{String}_{\mathsf{H}} \to \mathsf{String}_{\mathsf{H}}]$
$v \triangleq [z : T_{\mathsf{L}} \Rightarrow n\,(x)\,\texttt{"hello"}]$
$v' \triangleq [z : T'_{\mathsf{L}} \Rightarrow m\,(x)\,v.n(x)]$

Here, v' is not well-typed using the security type system, because of the call $v.n(x)$ ($\vdash \mathsf{String}_{\mathsf{H}} \not<: \mathsf{String}_{\mathsf{L}}$). However, we can show that v' does satisfy $\mathsf{TRNI}(\cdot, v', T'_{\mathsf{L}})$, because a public observer always obtains the same result (*i.e.* "hello") for any two secrets passed to method m; the program is not leaking any information.

Second, the type system can assign the security type $T \triangleleft \top$ to an expression, despite the fact that $\top$ is not the tighter secure type for TRNI to hold. For instance, let us assume that Int has built-in methods mod2 and mod4 with the standard mathematical meaning, and we define the type $\mathsf{IntMod4} \triangleq [\mathsf{mod4} : \mathsf{Unit}_{\mathsf{L}} \to \mathsf{Int}_{\mathsf{L}}]$. Consider $\Gamma \triangleq v : \mathsf{Int} \triangleleft \mathsf{IntMod4}$ and $e \triangleq \mathsf{v.mod2()}$. The type system admits $\Gamma \vdash e : \mathsf{Int}_{\mathsf{H}}$, which implies $\mathrm{TRNI}(\Gamma, e, \mathsf{Int}_{\mathsf{H}})$, but it does not admit $\Gamma \vdash e : \mathsf{Int}_{\mathsf{L}}$; despite the fact that $\mathrm{TRNI}(\Gamma, e, \mathsf{Int}_{\mathsf{L}})$ also holds—because if a and b are equivalent modulo 4, then they are also equivalent modulo 2.

5 Expressiveness of Declassification Policies

Our approach to type-based declassification policies builds upon an underlying type system. While we have chosen a simple model of recursive object types to develop the approach in the previous sections, it is interesting to explore how the expressiveness of the underlying type discipline affects the range of declassification policies that can be defined.

Recursive types. It is possible to exploit the idea of type-based declassification policies without recursive object types. We only need a type abstraction mechanism, such as that enabled by subtyping. In fact, with only record types and subtyping, we can already capture a set of interesting policies, such as those mentioned at the begin of Section 2 (*e.g.* StringEq, StringHashEq). TRNI depends on the notion of equivalence between values and computations, which can be easily simplified for the non-recursive setting; in particular, we can get rid of step-indexing in the logical relations.

Of course, without recursive object types in the core formalism, we lose the ability to express recursive declassification policies (which are useful to declassify recursive data structures, as illustrated in Section 2). With records but without objects, we can add general recursive types of the form $\mu X.T$ to support recursive declassification policies. Note however that combining general recursive types and subtyping is challenging, and there are different definitions that may not be complete (*i.e.* unable to establish a subtyping relation that indeed holds); in particular, our subtyping rules are not complete regarding subtyping between infinite trees [5]. This challenge solely affects the kinds of security types that can be defined and deemed well-formed.

Finally, one characteristic of recursive declassification policies is that they potentially allow to chain arbitrarily many invocations of a declassification method. For instance, consider an infinite stream of strings, and a declassification that allows equality comparisons on its elements:

$$\mathsf{StrEqStream} \triangleq [\mathsf{head} : \mathsf{Unit}_\mathsf{L} \to \mathsf{StringEq}_\mathsf{L},\ \mathsf{tail} : \mathsf{Unit}_\mathsf{L} \to \mathsf{StrEqStream}_\mathsf{L}]$$

In case tolerating an unbounded number of observations would represent an unacceptable accumulated leak, the programmer can define a more restrictive declassification policy that restricts the number of tolerated calls by explicitly nesting interface types instead of defining a fully recursive one. Obviously, to be practical, one would need to define a convenient surface syntax such as:

$$\mathsf{StrEqStream} \triangleq [\mathsf{head} : \mathsf{Unit}_\mathsf{L} \to \mathsf{StringEq}_\mathsf{L},\ \mathsf{tail} : \mathsf{Unit}_\mathsf{L} \to \mathsf{StrEqStream}_\mathsf{L}@k]$$

to specify that the declassification policy only supports at most k unfoldings of StrEqStream through tail, and to desugar it to a finite nesting of interface types.

Universal types. Universal types allow programmers to define programs that are parameterized by types. This can be used to define generic data structures, such as lists:

$$\mathsf{List}\,[X] \triangleq \{\mathsf{isEmpty} : \mathsf{Unit}_\mathsf{L} \to \mathsf{Bool}_\mathsf{L},\ \mathsf{head} : \mathsf{Unit}_\mathsf{L} \to X_\mathsf{L},\ \mathsf{tail} : \mathsf{Unit}_\mathsf{L} \to \mathsf{List}[X]_\mathsf{L}\}$$

If we add parametric polymorphism to $\mathsf{Ob}_{\mathsf{SEC}}$, then in addition to get polymorphism over implementation types, we naturally get a general form of *security label polymorphism*, which is very useful (and supported in Jif [23]). For example, we can define generic data structures that are polymorphic with respect to the security labels of their inner data; the list structure defined above is a specific example.

Similarly, a declassification policy can exploit parametric polymorphism. Recall the recursive declassification example of Section 2, in which we allow traversing a list and only comparing its elements with a given public element. We can express a generic version of this declassification policy with the following type:

$$\mathsf{ListEq}\,[X] \triangleq [\mathsf{isEmpty} : \mathsf{Unit_L} \to \mathsf{Bool_L},\ \mathsf{head} : \mathsf{Unit_L} \to X \triangleleft \mathsf{Eq}[X],\ \mathsf{tail} : \mathsf{Unit_L} \to \mathsf{ListEq}[X]_\mathsf{L}]$$
$$\mathsf{Eq}\,[X] \triangleq [\mathsf{eq} : X_\mathsf{L} \to \mathsf{Bool_L}]$$

Note that the above definition is however invalid, because ListEq is not well-formed: in order to satisfy the subtyping constraint between the facets of a security type such as $X \triangleleft \mathsf{Eq}[X]$, we need to bound the type variable X, which leads us to *bounded* parametric polymorphism. Then, the type ListEq can be correctly defined as follows:

$$\mathsf{ListEq}\,[X <: \mathsf{Eq}\,[X]] \triangleq$$
$$[\mathsf{isEmpty} : \mathsf{Unit_L} \to \mathsf{Bool_L},\ \mathsf{head} : \mathsf{Unit_L} \to X \triangleleft \mathsf{Eq}[X],\ \mathsf{tail} : \mathsf{Unit_L} \to \mathsf{ListEq}[X]_\mathsf{L}]$$

Refinement types. Refinement types, as found in *e.g.* LiquidHaskell [34], enrich standard types with predicates over a decidable logic. For instance, the type $\{x : \mathsf{Int} \mid x \geq 0\}$ denotes natural numbers. Additionally, refinement types usually support a form of dependent types, allowing refinements to refer to variables in scope as well as function arguments. Combining such expressive types with our approach allows interesting declassification policies to be defined, such as restricting successive arguments of a progressive declassification.

As an example, consider the following policy:

$$\mathsf{IntModProd} \triangleq [\mathsf{mod} : \{x : \mathsf{Int_L}\} \to [\mathsf{mult} : \{y : \mathsf{Int_L} \mid x = y\} \to \mathsf{Int_L}]_\mathsf{L}]$$

This progressive declassification allows revealing the result of the chain of invocations mod then mult, *only if the argument to both invocations is the same*. Note that IntModProd is a proper supertype of Int, since $\{y : \mathsf{Int} \mid x = y\}$ is a subtype of Int.

More advanced scenarios. There are other interesting declassification policies that seem more challenging to support with our type-based approach. An interesting example is specifying that a string secret can be leaked only after it has been encrypted; it is highly unlikely that the standard String class exposes an encryption method. However, our approach does appeal to the actual interface of an object in order to define its declassification. Hicks *et al.* [20] introduce special *declassifier* functions to express arbitrary declassification that can involve operations that are not defined on the declassified object itself. Therefore a possible solution to address this example in our setting would be to rely on an external method specification mechanism, such as open classes or mixin-based composition of traits in Scala.

Nevertheless, the above approach would still fall short of expressing *global* declassification policies, as described by Li and Zdancewic [21], which can relate the declassification of different secrets at once. While the value dependencies can be expressed using, *e.g.* refinement types, the challenge is to ensure that the obtained security types are still well-formed (*i.e.* the public facet must be a supertype of the private facet). These are interesting challenges for future development of the approach.

A note about casts. In Section 2 we alluded to the challenge of integrating explicit downcasts in a language that adopts type-based declassification policies. Casts can be soundly incorporated in such a language provided that we only allow casting values from a security

type to another one that has the *same public type*, *i.e.* casts cannot affect the declassification policy. Therefore the interesting typing rule for a cast expression $\langle T \rangle\, e$ is:

$$(\text{TCast})\ \frac{\Gamma \vdash e : T' \triangleleft U \qquad \vdash T <: T'}{\Gamma \vdash \langle T \rangle\, e : T \triangleleft U}$$

As usual in security languages with casts, cast errors are seen as a non-termination channel, hence not affecting the security definitions.

6 Related work

Information flow security in general, and declassification in particular, are very active areas of research. We now discuss the most salient proposals related to this work.

Secure information flow and type abstraction. Our work shows a connection between type abstraction and declassification policies for secure information flow. Previous works also attempt to connect type abstraction and secure information flow.

Tse and Zdancewic [31] encode the Dependency Core Calculus (DCC) [**?**] in System F. The correctness theorem of their translation aims at showing that the parametricity theorem of System F implies the noninterference property. Unfortunately, Shikuma and Igarashi identify a mistake in the proof of their main result [29]; they also gave a noninterference-preserving translation for a version of DCC to the simply-typed lambda calculus. However, this translation left open the connection between parametricity and noninterference, initially aimed by Tse and Zdancewic.

Recently, Bowman and Ahmed [**?**] provide a translation from DCC to System F_ω, successfully demonstrating that noninterference can be encoded via parametricity. Our work generalizes this by showing that type abstraction implies *relaxed* noninterference. Information flow analyses have been proposed to generalize parametricity in the presence of runtime type analysis [36]. Using security labels, a programmer can specify data structures that should remain confidential in order to hide implementation details and rely on type abstraction for abstract datatypes.

An interesting research direction is to investigate whether our proposal of solving information flow problems via type abstraction, here through subtyping, can be used to generalize parametricity as proposed by Washburn and Weirich [36].

Declassification. As extensively discussed, our policies and security property are based on the work of Li and Zdancewic [21], which proposes two kinds of downgrading policies (which we call here declassification policies, since they only relate to confidentiality): local and global policies. The declassification policies in this paper directly correspond to local policies, as discussed in the introduction. Global policies refer to declassifications that involve more than one secret simultaneously. As discussed in Section 5, it is unclear if and how global policies can be supported using our type-driven approach; further exploration is necessary to settle this issue. Additionally, in contrast to the definition of relaxed noninterference of Li and Zdancewic [21], our definition is independent from the security enforcement mechanism. This allows us to distinguish programs that are not secure from programs that are not typable due to a necessarily conservative static security mechanism (see Section 4). Also, our definition of relaxed noninterference is formulated as a generalization of the semantic characterization of pure noninterference [37], providing a modular reasoning principle, as opposed to the global translation approach of Li and Zdancewic.

In the following, we focus on the closest related work on declassification policies starting from 2005 and refer the reader to [28] for a survey prior to 2005.

Typing declassification in object-oriented languages. Since 2005, several works have studied static enforcement of declassification in object-oriented languages [9, 20, 10, 15].

Banerjee and Naumann [9] study the interaction between security typing for noninterference and access control in a Java-like language. Security levels are not fixed but rather depend on access permissions. In contrast to our work, security levels are independent of method signatures or types and thus their typing does not relate to type abstraction.

Hicks *et al.* [20] propose trusted declassification for an object calculus. Principals in a program have access to specified trusted *declassifier* functions or methods. Typeable programs are secure for noninterference modulo trusted methods, in the same spirit as typing of noninterference of programs with cryptographic functions [19]. In contrast to relaxed noninterference, trusted declassification does not consider declassifiers as part of security levels. Instead, declassifiers need to be associated by a policy to different principals (security labels in our setting) in the lattice.

Barthe *et al.* [10] propose a modular method to extend type systems and proofs for noninterference to declassification and discuss how the method extends to object-oriented languages. The declassification property called delimited non-disclosure [22] does not support fine-grained specification of how to declassify a given secret, as supported by relaxed noninterference.

Tse and Zdancewic [32] propose a security-typed language for robust declassification: declassification cannot be triggered unless there is a digital certificate to assert the proper authority. Their language inherits many features from System $F_{<:}$ and uses monadic labels as in DCC [**?**]. The monadic style allows them to integrate computational effects, which we do not support. In contrast to our work, security labels are based on the Decentralized Label Model (DLM) [24], and are not semantically unified with the standard safety types of the language.

Chong and Myers [15] propose hybrid typing to enforce declassification and erasure policies and implement it in Jif [23]. Their language features a special declassification function that takes as input the expression and levels to declassify and also the conditions under which declassification can occur. Security policies are specified by means of security levels and conditions to downgrade them. This resembles our declassification policies, which specify the methods that can be applied in order to (partially) declassify; at a more abstract level, the interface types of the public facet can be seen as "conditions" for declassifying. The type system developed by Chong and Myers statically checks that conditions in declassification commands comply with the specified security policies. A dynamic mechanism enforces this, or returns a dummy value (instead of the declassified value) at runtime. In contrast to our work, their type system significantly departs from standard typing rules, and dynamic checks are required for guaranteeing security.

Extensional specification of declassification policies. The language Air [30] expresses declassification policies as security automata. The policies, seen as automata, transition when a release obligation is satisfied. When an accepting state is reached, declassification is performed. These policies resemble relaxed noninterference and our own declassification policies but they require very specific typing rules.

Banerjee et al. [**?**] study declassification properties using ideas from epistemic logic can capture global policies (as in the original work of relaxed noninterference) with an extensional

property. Their policies are not expressed using standard types as in our work.

The language Paralocks [14] supports declassification policies represented as Horn clauses, whose antecedents are conditions that should be satisfied for a flow to occur. There is a natural order between declassification policies that correspond to the logical entailment when viewing policies as Horn clauses. The policies together with the logical entailment order define a lattice that supports an extensional specification of secrets and their intended declassification, as in our work. However, declassification policies in Paralocks are not specified by using the standard types of the language, and thus their enforcement requires specific typing rules.

Multiple facets for dynamic enforcement of declassification Austin and Flanagan introduce Multiple Facets [**?**] as a dynamic mechanism to enforce secure information flow. The main idea behind multiple facets is to execute a program using multiple values, one value or facet for each security level of observation. A value considered confidential will only flow to a public facet by facet declassification, based on robust declassification [39]. Robust declassification requires the decision to declassify to be trusted according to integrity labels used to model trust. In our work, we do not consider integrity labels or robust declassification. However, the idea of multiple facets (having a facet for each observer at a given security level) is similar to our faceted types. Just as Austin and Flanagan can run a program for different facets simultaneously, we type check programs providing different views to observers with different security clearances.

Multiple facets are also inspired by Secure Multi Execution (SME) [18, 11], a dynamic mechanism that roughly executes a program multiple times in order to enforce noninterference. Hence, observers with different security clearances will potentially observe different values during the execution of a program. Several works have studied declassification in the context of SME [26, 33, 12]. Rafnsson and Sabelfeld [26] propose declassification in SME based on the gradual release property [7]. This property differs from the property we consider in our work in that it is not possible to extensionally specify what is being released or declassified. The latest works on SME declassification [33, 12] generalize security levels as declassifier functions, resembling declassification policies of both Li and Zdancewic and ours. Since SME is a dynamic enforcement mechanism, these declassification policies are not used for relating declassification and type abstraction.

7 Conclusion

One of the open challenges in the area of information flow security is integrating information flow mechanisms with existing infrastructures [38]. Our work partially addresses this challenge by showing a connection between type abstraction, more precisely that induced by the the subtyping relation in an object-oriented language, and the order relation in security lattices. In particular, we exploit an intuitive connection between object interfaces and declassification policies: an object interface already gives a way to control the exposed behavior of an object. These connections imply that standard type systems can be used as a direct means to enforce secure information flow, when types express security policies. It is left to explore how this connection scales in practice, but we expect the economy of concepts to be an important asset for adoption.

We plan to study the impact of more advanced typing disciplines on the expressiveness of type-based declassification, especially dependent object types [27] and refinement types [34]. It remains to be seen whether global policies can be expressed, and how. Another venue

for future work is to develop our approach in a setting that relies on other forms of type abstraction, such as existential types. Finally, we intend to explore how to *infer* the minimal knowledge that has to be exposed to a public observer in order to guarantee a relaxed noninterference guarantee at a given type. Inferring the minimal input declassifications of a secure program can for instance be useful to assess the impact some refactoring or extensions of that program have on security.

References

1 Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. A core calculus of dependency. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages (POPL 99)*, pages 147–160, San Antonio, TX, USA, January 1999. ACM Press.

2 Martin Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

3 Amal Ahmed. *Semantics of Types for Mutable State*. PhD thesis, Princeton University, 2004.

4 Amal Ahmed. Step-indexed syntactic logical relations for recursive and quantified types. In Peter Sestoft, editor, *Proceedings of the 15th European Symposium on Programming (ESOP 2006)*, volume 3924 of *Lecture Notes in Computer Science*, pages 69–83. Springer-Verlag, 2006.

5 Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. In David S. Wise, editor, *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages (POPL 91)*, pages 104–118. ACM Press, 1991.

6 Andrew W. Appel and David McAllester. An indexed model of recursive types for foundational proof-carrying code. *ACM Transactions on Programming Languages and Systems*, 23(5):657–683, September 2001.

7 Aslan Askarov and Andrei Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *Proceedings of the 27th IEEE Symposium on Security and Privacy (S&P 2007)*, pages 207–221. IEEE Computer Society Press, May 2007.

8 Thomas H. Austin and Cormac Flanagan. Multiple facets for dynamic information flow. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2012)*, pages 165–178. ACM Press, January 2012.

9 Anindya Banerjee and David A. Naumann. Stack-based access control and secure information flow. *Journal of Functional Programmming*, 15(2):131–177, September 2005.

10 Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Expressive declassification policies and modular static enforcement. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P 2008)*, pages 339–353. IEEE Computer Society Press, May 2008.

11 Gilles Barthe, Salvador Cavadini, and Tamara Rezk. Tractable enforcement of declassification policies. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF 2008)*, pages 83–97. IEEE Computer Society Press, June 2008.

12 Natalia Bielova and Tamara Rezk. Spot the difference: Secure multi-execution and multiple facets. In *Proceedings of the 21st European Symposium on Research in Computer Security (ESORICS 2016)*, pages 501– 519, 2016.

13 Iulia Bolosteanu and Deepak Garg. Asymmetric secure multi-execution with declassification. In *Proceedings of the 5th International Conference on Principles of Security and Trust (POST 2016)*, pages 24–45. Springer-Verlag, April 2016.

14 William J. Bowman and Amal Ahmed. Noninterference for free. In *Proceedings of the 20th ACM SIGPLAN Conference on Functional Programming (ICFP 2015)*, pages 101–113. ACM Press, August 2015.

15 Niklas Broberg and David Sands. Paralocks: role-based information flow control and beyond. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2010)*, pages 431–444. ACM Press, January 2010.

16 Stephen Chong and Andrew C. Myers. End-to-end enforcement of erasure and declassification. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium (CSF 2008)*, pages 98–111. IEEE Computer Society Press, June 2008.

17 William R. Cook. On understanding data abstraction, revisited. *ACM SIGPLAN Notices*, 44(10):557–572, 2009.

18 Dorothy E. Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, May 1976.

19 Dominique Devriese and Frank Piessens. Noninterference through Secure Multi-execution. In *Proceedings of the 31st IEEE Symposium on Security and Privacy (S&P 2010)*, pages 109–124. IEEE Computer Society Press, May 2010.

20 Cédric Fournet, Jérémy Planul, and Tamara Rezk. Information-flow types for homomorphic encryptions. In *Proceedings of the Conference on Computer and Communications Security (CCS 2011)*, pages 351–360. ACM Press, October 2011.

21 Boniface Hicks, Dave King, Patrick McDaniel, and Michael Hicks. Trusted declassification: high-level policy for a security-typed language. In *Proceedings of the workshop on Programming Languages and Analysis for Security (PLAS 2006)*, pages 65–74. ACM Press, June 2006.

22 Peng Li and Steve Zdancewic. Downgrading policies and relaxed noninterference. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2005)*, pages 158–170. ACM Press, January 2005.

23 Ana Almeida Matos and Gérard Boudol. On declassification and the non-disclosure policy. In *Proceedings of the IEEE Computer Security Foundations Workshop (CSFW 2005)*, pages 549–597. IEEE Computer Society Press, October 2005.

24 Andrew C. Myers. Jif homepage. http://www.cs.cornell.edu/jif/, accessed May 2017.

25 Andrew C. Myers and Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9:410–442, October 2000.

26 Benjamin C. Pierce. *Types and programming languages*. MIT Press, Cambridge, MA, USA, 2002.

27 Willard Rafnsson and Andrei Sabelfeld. Secure multi-execution: Fine-grained, declassification-aware, and transparent. In *Proceedings of the 26th IEEE Computer Security Foundations Symposium (CSF 2013)*, pages 33–48. IEEE Computer Society Press, June 2013.

28 Tiark Rompf and Nada Amin. Type soundness for dependent object types (DOT). In Eelco Visser and Yannis Smaragdakis, editors, *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*, pages 624–641. ACM Press, November 2016.

29 Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, 17(5):517–548, 2009.

30 Naokata Shikuma and Atsushi Igarashi. Proving noninterference by a fully complete translation to the simply typed lambda-calculus. In Mitsu Okada and Ichiro Satoh, editors, *Proceedings of the 11th Asian Computing Science Conference (ASIAN 2006)*, volume 4435 of *Lecture Notes in Computer Science*, pages 301–315. Springer-Verlag, 2006.

31 Nikhil Swamy and Michael Hicks. Verified enforcement of stateful information release policies. In Úlfar Erlingsson and Marco Pistoia, editors, *Proceedings of the Workshop on Programming Languages and Analysis for Security (PLAS 2008)*, pages 21–32. ACM Press, December 2008.

32 Stephen Tse and Steve Zdancewic. Translating dependency into parametricity. In *Proceedings of the 7th ACM SIGPLAN Conference on Functional Programming (ICFP 2004)*, pages 115–125, Snowbird, Utah, USA, September 2004. ACM Press.

33 Stephen Tse and Steve Zdancewic. A design for a security-typed language with certificate-based declassification. In *Proceedings of the 14th European Symposium on Programming Languages and Systems (ESOP 2005)*, volume 2986 of *Lecture Notes in Computer Science*, pages 279–294. Springer-Verlag, 2005.

34 Mathy Vanhoef, Willem De Groef, Dominique Devriese, Frank Piessens, and Tamara Rezk. Stateful declassification policies for event-driven programs. In *Proceedings of the 27th IEEE Computer Security Foundations Symposium (CSF 2014)*. IEEE Computer Society Press, 2014.

35 Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. Refinement types for haskell. In *Proceedings of the 19th ACM SIGPLAN Conference on Functional Programming (ICFP 2014)*, pages 269–282. ACM Press, August 2014.

36 Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2-3):167–187, January 1996.

37 Geoffrey Washburn and Stephanie Weirich. Generalizing parametricity using information-flow. In *Proceedings of the 20th IEEE Symposium on Logic in Computer Science (LICS 2005)*, pages 62–71. IEEE Computer Society Press, June 2005.

38 Steve Zdancewic. *Programming Languages for Information Security*. PhD thesis, Cornell University, August 2002.

39 Steve Zdancewic. Challenges for information-flow security. In *Proceedings of Programming Language Interference and Dependence*, 2004.

40 Steve Zdancewic and Andrew C. Myers. Robust declassification. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop (CSFW-14)*, pages 15–23. IEEE Computer Society Press, June 2001.

A Auxiliary Definitions

A.1 Environments

$$
\begin{array}{lcll}
\Gamma & ::= & \cdot \mid \Gamma, x : S & \text{(type environment)} \\
\Phi & ::= & \cdot \mid \Phi, \alpha <: \beta & \text{(subtyping environment)} \\
\Delta & ::= & \cdot \mid \Delta, \alpha & \text{(type variable environment)} \\
\Sigma & ::= & \cdot \mid \Sigma, \alpha \triangleq O & \text{(type definition environment)}
\end{array}
$$

- Γ is a finite map from variables to closed and well-formed security types. Σ is a finite map from type variables to object types. Φ is a set of subtyping relations between type variables. Δ is a set of type variables.
- $dom(Env)$ (where Env could be Γ, Σ or Φ) is the set of variables for which the finite map Env is defined. In the case of $dom(\Phi)$, it is the set of the type variables in the left part of the subtyping relation.
- We also use the notations $\Gamma, x : S$ or $\Sigma, \alpha \triangleq O$ or $\Phi, \alpha <: \beta$ to extend the environments Γ, Σ , Φ with a new binding or relation, respectively. If $x \in dom(\Gamma)$, $\alpha \in dom(\Sigma)$ or either α or $\beta \in dom(\Phi) \cup cod(\Phi)$ the extension operation is not defined for the respective environment.
- The notation Δ, α extends the set Δ with a new type variable. If $\alpha \in \Delta$ the operation is not defined.

We use the following functions to access to the elements of the environments:

- $\Gamma(x)$ returns the security type associated to x in Γ. If $x \notin dom(\Gamma)$, then $\Gamma(x)$ is undefined.
- $\Sigma(\alpha)$ returns the type associated to α in Σ. If $\alpha \notin dom(\Sigma)$, then $\Sigma(\alpha)$ is undefined.
- $\alpha <: \beta \in \Phi$ is true if $\Phi(\alpha) = \beta$, false otherwise. $\Phi(\alpha)$ returns the type variable in the right part of the subtyping relation with α in Φ. If $\alpha \notin dom(\Phi)$, then $\Phi(\alpha)$ is undefined.

A.2 Well-formedness of types and environments

For the main results of the Section 4 to hold we need to ensure we work with well-formed security types.

Well formed types. We use the predicate $\mathsf{valid}(S)$ to denote that a security type S is closed and that the object types that S contains have unique method members. The definition of $\mathsf{valid}(S)$ is based on a standard notion well-formedness of object types [2] (Figure 7).

To check for *well-formed security types*, *i.e.* that the private type is a subtype of the public type we define the judgment $\Sigma \vdash_s S$ (Figure 8). The (WFS-ST) rule is the most important. For this rule to hold, the subtyping relation between both facets must hold and also the same principle must hold for the all the security types in each facet.

The presence of type variables in the facets of a security type and the corresponding subtyping constraint introduces subtle cases to manage before using the subtyping judgment. Consider the following object type: $O \triangleq \mathbf{Obj}(\alpha).\,[\mathsf{m} : S \to \alpha \triangleleft \mathbf{Obj}(\beta).\,[\mathsf{m} : S \to \alpha \triangleleft \beta]]$. For $\vdash_s O$ to hold, $\alpha \triangleq O \vdash_s \alpha \triangleleft \mathbf{Obj}(\beta).\,[\mathsf{m} : S \to \alpha \triangleleft \beta]$ must hold. It implies to check $\vdash \alpha <: \mathbf{Obj}(\beta).\,[\mathsf{m} : S \to \alpha \triangleleft \beta]$. Note that, we can not justify that subtyping judgment, because we do not have a subtyping premise involving the type variable α. To address this, we need to remember (in Σ) the surrounding recursive object type O that binds α, and to transform the check $\alpha \triangleq O \vdash_s \alpha \triangleleft \mathbf{Obj}(\beta).\,[\mathsf{m} : S \to \alpha \triangleleft \beta]$ to $\vdash O <: \mathbf{Obj}(\beta).\,[\mathsf{m} : S \to O \triangleleft \beta]$ by closing α with the mappings in Σ (*i.e.* O). We use the notation $\Sigma\,[T]$ to substitute the free variables in type T according to the bindings in Σ.

$\boxed{\Delta \vdash_t T}$

$$\text{(WF-V)}\frac{\alpha \in \Delta}{\Delta \vdash_t \alpha} \qquad \text{(WF-O)}\frac{T \equiv \mathbf{Obj}(\alpha).\ \boxed{m : S_1 \to S_2} \quad (i \neq j \implies m_i \neq m_j) \quad \Delta, \alpha \vdash_t S_{1i} \quad \Delta, \alpha \vdash_t S_{2i}}{\Delta \vdash_t T}$$

$\boxed{\Delta \vdash_t S}$

$$\text{(WF-ST)}\frac{\Delta \vdash_t T \quad \Delta \vdash_t U}{\Delta \vdash_t T \triangleleft U} \qquad \frac{\cdot \vdash_t S}{\mathsf{valid}(S)}$$

Figure 7 Standard well-formedness of object types and type variables, and its lifting to security types.

$\boxed{\Sigma \vdash_s T}$

$$\frac{T \equiv \mathbf{Obj}(\alpha).\ \boxed{m : S_1 \to S_2} \quad \Sigma, \alpha : T \vdash_s S_{1i} \quad \Sigma, \alpha : T \vdash_s S_{2i}}{\Sigma \vdash_s T} \qquad \text{(WFS-V)}\frac{}{\Sigma \vdash_s \alpha}$$

$\boxed{\Sigma \vdash_s S}$ $\qquad$ $\boxed{\vdash S}$

$$\text{(WFS-ST)}\frac{\Sigma \vdash_s T \quad \Sigma \vdash_s U \quad \cdot \vdash \Sigma\,[T] <: \Sigma\,[U]}{\Sigma \vdash_s T \triangleleft U} \qquad \text{(WF)}\frac{\mathsf{valid}(S) \quad \cdot \vdash_s S}{\vdash S}$$

Figure 8 Well-formedness of security types

Finally, we say that a security type S is well-formed (notation $\vdash S$) if the type is valid and the subtyping constraints for S hold ($\cdot \vdash_s S$)

Well-formedness of a type environment. A type environment is well formed, noted $\Gamma \vdash \diamond$, if all types in the environment are well-formed:

$$\text{(EEnvOk)}\frac{}{\cdot \vdash \diamond} \qquad \text{(EnvOk)}\frac{\Gamma \vdash \diamond \quad \vdash S \quad x \notin dom(\Gamma)}{\Gamma, x : S \vdash \diamond}$$

A.3 Subtyping

The gray parts in the subtyping rules of the Figure 9 were not included in the Figure 2 of the main document. They prevent justifying inconsistent subtyping judgments by controlling the uses of type variables.

For example, consider the following types:
$T_1 \triangleq \mathbf{Obj}(\alpha).\,[\mathsf{n} : S \to \mathbf{Obj}(\beta).\,[\mathsf{m}_1 : \beta_{\mathsf{L}} \to S' \quad \mathsf{m}_2 : S_1 \to S_2]_{\mathsf{L}}]$
$T_2 \triangleq \mathbf{Obj}(\beta).\,[\mathsf{n} : S \to \mathbf{Obj}(\alpha).\,[\mathsf{m}_1 : \alpha_{\mathsf{L}} \to S']_{\mathsf{L}}]$
For $\vdash T_1 <: T_2$ to hold, after using the rule (SObj) twice, the contravariance of m_1 parameters $\cdot, \alpha <: \beta, \beta <: \alpha \vdash \alpha <: \beta$ must hold. We can justify this by applying the rule (SVar) because we have the assumption $\alpha <: \beta$ in the subtyping environment. So, we justify $\vdash T_1 <: T_2$ and it is not the case that T_1 is subtype of T_2. The problem is the occurrence of the variables α and β in both types, that creates subtyping assumptions in both directions and it allows to justify subtyping between type variables that represent unrelated types (by subtyping). The well-formedness condition of the subtyping environment Φ prevents this kind of cases,

$\boxed{\Phi \vdash T <: T}$

$$(\text{SObj})\ \frac{\begin{array}{c} O_1 \triangleq \mathbf{Obj}(\alpha).\ \overline{[m : S_1 \to S_2]} \quad O_2 \triangleq \mathbf{Obj}(\beta).\ \overline{[m' : S'_1 \to S'_2]} \quad \overline{m'} \subseteq \overline{m} \\ m_i = m'_j \implies (\Phi, \alpha <: \beta \vdash S'_{1j} <: S_{1i} \quad \Phi, \alpha <: \beta \vdash S_{2i} <: S'_{2j}) \\ \Phi \vdash \diamond \quad dom(\Phi) \cup cod(\Phi) \vdash_t O_i \end{array}}{\Phi \vdash O_1 <: O_2}$$

$$(\text{SVar})\ \frac{\Phi \vdash \diamond \quad \alpha <: \beta \in \Phi}{\Phi \vdash \alpha <: \beta} \qquad (\text{SSubEq})\ \frac{T_1 \equiv T_2}{\Phi \vdash T_1 <: T_2} \qquad (\text{STrans})\ \frac{\Phi \vdash T_1 <: T_2 \quad \Phi \vdash T_2 <: T_3}{\Phi \vdash T_1 <: T_3}$$

$\boxed{\Phi \vdash S <: S}$

$$(\text{TSubST})\ \frac{\Phi \vdash T_1 <: T_2 \quad \Phi \vdash U_1 <: U_2}{\Phi \vdash T_1 \triangleleft U_1 <: T_2 \triangleleft U_2}$$

Figure 9 Subtyping

$\boxed{\Phi \vdash \diamond}$

$$(\text{EEnvSubOk})\ \frac{}{\cdot \vdash \diamond} \qquad (\text{EnvSubOk})\ \frac{\Phi \vdash \diamond \quad \alpha_i \notin dom(\Phi) \cup cod(\Phi)}{\Phi, \alpha_1 <: \alpha_2 \vdash \diamond}$$

Figure 10 Well-formedness of the subtyping environment

because we cannot extend the environment with a subtyping premise, where one of the involved variables is already in the environment (Figure 10).

A.4 Type equivalence

Two types are equivalent (Figure 11) if the equivalence can be derived through the congruence induced by rules (Alpha-Eq) and (Fold-Unfold). For example:
$\mathbf{Obj}(\alpha).\ [m : \alpha \to \alpha] \equiv \mathbf{Obj}(\beta).\ [m : \beta \to \beta]$
$\mathbf{Obj}(\alpha).\ [m : \top \to \alpha] \equiv \mathbf{Obj}(\alpha).\ [m : \top \to \mathbf{Obj}(\beta).\ [m : \top \to \beta]]$

A.5 Simple type system

The simple typing judgment $\Gamma \vdash_1 e : T$ is defined in terms of "single-facet typing" (Figure 12). Single-facet typing $\Gamma \vdash_{\mathsf{sf}} e : S$ is a simplification of security typing: the rules (TmD) and (TmH) are replaced by a single rule (T1mI) that simply ignores the public type. Furthermore, the subtyping judgment $\Phi \vdash S_1 <: S_2$ is replaced by the simple subtyping judgment $\Phi \vdash_{\mathsf{sf}} S_1 <: S_2$ that only takes care of subtyping between the private facets of the security types. Its definition is direct and omitted here.

▶ **Lemma 14.** $\Gamma \vdash \diamond \ \wedge\ \Gamma \vdash e : T \triangleleft U$ *then* $\Gamma \vdash_1 e : T$

Proof. Trivial induction on typing derivations of e. ◀

▶ **Lemma 15.**

$$\Gamma \vdash \diamond \ \wedge\ \Gamma \vdash_1 e : T \implies \exists U.\ \Gamma \vdash e : T \triangleleft U$$

$\boxed{T \equiv T}$

$$\text{(Sym)}\frac{}{T \equiv T} \qquad \text{(Refl)}\frac{T_1 \equiv T_2}{T_2 \equiv T_1} \qquad \text{(Trans)}\frac{T_1 \equiv T_2 \quad T_2 \equiv T_3}{T_1 \equiv T_3}$$

$$\text{(O-Congr)}\frac{S_{1i} \equiv S'_{1i} \quad S_{2i} \equiv S'_{2i}}{\mathbf{Obj}(\alpha).\,[\overline{m : S_1 \to S_2}] \equiv \mathbf{Obj}(\alpha).\,[\overline{m : S'_1 \to S'_2}]}$$

$$\text{(Alpha-Eq)}\frac{O \triangleq \mathbf{Obj}(\alpha).\,[\overline{m : S_1 \to S_2}] \quad \beta \text{ fresh}}{O \equiv O\,[\beta/\alpha]} \qquad \text{(Fold-Unfold)}\frac{}{O \equiv O\,[O/\alpha]}$$

$\boxed{S \equiv S}$

$$\frac{T_1 \equiv T_2 \quad U_1 \equiv U_2}{T_1 \triangleleft U_1 \equiv T_2 \triangleleft U_2}$$

Figure 11 Type equivalence

$\boxed{\Gamma \vdash_{\mathsf{sf}} e : S}$

$$\text{(T1Var)}\frac{x \in dom(\Gamma)}{\Gamma \vdash_{\mathsf{sf}} x : \Gamma(x)} \qquad \text{(T1Sub)}\frac{\Gamma \vdash_{\mathsf{sf}} e : S' \quad \vdash_{\mathsf{sf}} S' <: S \quad \vdash S}{\Gamma \vdash_{\mathsf{sf}} e : S}$$

$$\text{(T1Obj)}\frac{\vdash S \quad S \triangleq T \triangleleft U \quad \mathsf{methsig}(T, m_i) = S'_i \to S''_i \quad \Gamma, z : S, x_i : S'_i \vdash_{\mathsf{sf}} e_i : S''_i}{\Gamma \vdash_{\mathsf{sf}} \left[z : S \Rightarrow \overline{m\,(x)\,e}\right] : S}$$

$$\text{(T1mI)}\frac{\Gamma \vdash_{\mathsf{sf}} e_1 : T \triangleleft U \quad \mathsf{methsig}(T, m) = S_1 \to S_2 \quad \Gamma \vdash_{\mathsf{sf}} e_2 : S_1}{\Gamma \vdash_{\mathsf{sf}} e_1.m(e_2) : S_2}$$

$\boxed{\Gamma \vdash_1 e : T}$

$$\frac{\Gamma \vdash_{\mathsf{sf}} e : T \triangleleft U}{\Gamma \vdash_1 e : T}$$

Figure 12 Simple typing, defined in terms of single-facet typing

Proof. By induction of the typing derivation of $\Gamma \vdash_1 e : T$. In all the cases, we simply choose U to be the private type T. ◀

Control What You Include!

Server-Side Protection against Third Party Web Tracking

Dolière Francis Somé, Nataliia Bielova, and Tamara Rezk

Université Côte d'Azur, Inria
{doliere.some,tamara.rezk,nataliia.bielova}@inria.fr

Abstract. Third party tracking is the practice by which third parties recognize users accross different websites as they browse the web. Recent studies show that 90% of websites contain third party content that is tracking its users across the web. Website developers often need to include third party content in order to provide basic functionality. However, when a developer includes a third party content, she cannot know whether the third party contains tracking mechanisms. If a website developer wants to protect her users from being tracked, the only solution is to exclude any third-party content, thus trading functionality for privacy.
We describe and implement a privacy-preserving web architecture that gives website developers a control over third party tracking: developers are able to include functionally useful third party content, the same time ensuring that the end users are not tracked by the third parties.

1 Introduction

Third party tracking is the practice by which third parties recognize users accross different websites as they browser the web. In recent years, tracking technologies have been extensively studied and measured [25, 28, 31, 36, 24, 33] – researchers have found that third parties embedded in websites use numerous technologies, such as third-party cookies, HTML5 local storage, browser cache and device fingerprinting that allow the third party to recognize users across websites [37] and build browsing history profiles. Researchers found that more than 90% of Alexa top 500 websites [36] contain third party web tracking content, while some sites include as much as 34 distinct third party contents [30].

But why do website developers include so many third party contents (that may track their users)? Though some third party content, such as images and CSS [3] files can be copied to the main (first-party) site, such an approach has a number of disadvantages for other kinds of content. *Advertisement* is the base of the economic model in the web – without advertisements many website providers will not be able to financially support their website maintenance. *Third party JavaScript libraries* offer extra functionality: though copies of such libraries can be stored on the main first party site, this solution will sacrifice maintenance of these libraries when new versions are released. The developer would need to manually check the new versions. *Web mashups*, as for example applications that

use hotel searching together with maps, are actually based on reusing third-party content, as well as maps, and would not be able to provide their basic functionality without including the third-party content.

Including JavaScript libraries, content for mashups or advertisements means that the web developers cannot provide to the users the guarantee of non-tracking. Hence, the promise to provide privacy has a very high cost because there are no existing automatic tools to maintain control of third party tracking on the website. To keep a promise of non-tracking, the only solution today is to exclude any third-party content[1], thus trading functionality for privacy.

In this paper, we present a new Web application architecture that allows web developers to gain control over certain types of third party content. Our solution is based on the automatic rewriting of the web application in such a way that the third party requests are redirected to a trusted web server, with a different domain than the main site. This trusted web server may be either controlled by a trusted party, or by a main site owner – it is enough that the trusted web server has a different domain. A trusted server is needed so that the user's browser will treat all redirected requests as third party requests, like in the original web application. The trusted server automatically eliminates third-party tracking cookies and other technologies.

In summary our contributions are:

- A classification of third party contents that can and cannot be controlled by the website developer.
- An analysis of third party tracking capabilities – we analyse two mechanisms: recognition of a web user, and identification of the website she is visiting [2].
- A new architecture that allows to include third party content in web applications and eliminate stateful tracking.
- An implementation of our architecture, demonstrating its effectiveness at preventing stateful third party tracking in several websites.

2 Background and Motivation

Third party web tracking is the ability of a third party to re-identify users as they browse the web and record their browsing history [31]. Tracking is often done with the purpose of web analytics, targeted advertisement, or other forms of personalization. The more a third party is prevalent among the websites a user interacts with, the more precise is the browsing history collected by the tracker. Tracking has often been conceived as the ability of a third party to recognize the web user. However, for successful tracking, each user request should contain two components:

User recognition is the information that allows tracker to recognize the user;
Website identification is the website which the user is visiting.

[1] For example, see `https://duckduckgo.com/`. [2] Tracking is often defined as the ability of a third party to recognize a user through different websites. However, being able to identify the websites a user is interacting with is equally crucial for the effectiveness of tracking.

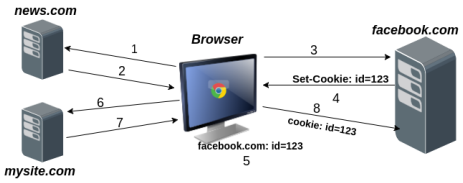


Fig. 1. Third Party Tracking

For example, when a user visits `news.com`, the browser may make additional requests to `facebook.com`, as a result, Facebook learns about the user's visit to `news.com`. Figure 1 shows a hypothetical example of such tracking where `facebook.com` is the third party.

Consider that a third party server, such as `facebook.com` hosts different contents, and some of them are useful for the website developers. The web developer of another website, say `mysite.com`, would like to include such *functional* content from Facebook, such as Facebook "Like" button, an image, or a useful JavaScript library, but the developer does not want its users to be tracked by Facebook. If the web developer simply includes third party Facebook content in his application, all its users are likely to be tracked by cookie-based tracking. Notice that each request to `facebook.com` also contains an HTTP Referrer header, automatically attached by the browser. This header contains the website URL that the user is visiting, which allows Facebook to build user's browsing history profile.

The example demonstrates cookie-based tracking, which is extremely common [36]. Other types of third party tracking, that use other client-side storage mechanisms, such as HTML5 LocalStorage, or cache, and device fingerprinting that do not require any storage capabilities, are also becoming more popular [25].

Web developer perspective A web developer may include third party content in her webpages, either because this content *intentionally* tracks users (for example, for targeted advertising), or because this content is important for the functioning of the web application. We therefore distinguish two kinds of third party contents from a web developer perspective: *tracking* and *functional*. *Tracking* content is *intentionally* embedded by website owner for tracking purposes. *Functional* content is embedded in a webpage for other purposes than tracking: for example, JavaScript libraries that provide additional functionality, such as jQuery, or other components, such as maps. In this work, we focus on *functional* content and investigate the following questions:

- What kind of third content is possible to control from a server-side (web developer) perspective?
- How to eliminate the two components of tracking (user recognition and website identification) from the functional third party that the website embeds?

2.1 Browsing Context

Browsers implement different specifications to securely fetch and aggregate third party content. One widely used approach is the the *Same Origin Policy (SOP)* [14], a security mechanism designed for developers to isolate legacy content from potentially untrusted third party content. An origin is defined as scheme, host and port number, of the URL[3] of the third party content.

When a browser renders a webpage delivered by a first party, the page is placed within a *browsing context* [2]. A browsing context represents an instance of the browser in which a document such as a webpage is displayed to a user, for instance browser tabs, and popup windows. Each browsing context contains 1) a copy of the browser properties (such as browser name, version, device screen etc), stored in a specific object; 2) other objects that depend on the origin of the document according to SOP. For instance, the object `document.cookie` gives the cookies related to the origin of the current context.

In-context and cross-context content Certain types of content embedded in a webpage, such as images, links, and scripts, are associated with the context of the webpage, and we call them *in-context* content. Other types of content, such as `<iframe>`, `<embed>`, and `<object>` tags are associated with their own browsing context, and we call them *cross-context* content. Usually, cross-context content, such as `<iframe>` elements, cannot be visually distinguished from the webpage in which they are embedded, however they are as autonomous as other browsing contexts, such as tabs or windows. Table 1 shows different third party contents and their execution contexts.

	HTML Tags	Third Party Content
in-context	`<link>`	Stylesheets
	`<img>`	Images
	`<audio>`	Audios
	`<video>`	Videos
	`<form>`	Forms
	`<script>`	Scripts
cross-context	`<(i)frame>, <frameset>, <a>`	Web pages
	`<object>, <embed>, <applet>`	Plugins and Web pages

Table 1. Third party content and execution context.

The Same Origin Policy manages the interactions between different browsing contexts. In particular, it prevents in-context scripts from interacting with the content from a cross-context content in case their origins are different. To communicate, both contexts should rely on inter-frame communications APIs such as `postMessage` [12].

[3] `https://www.w3.org/TR/url/`

2.2 Third Party Tracking

In this work, we consider only stateful tracking technologies – they require an identifier be stored client-side, the most common storage mechanism is cookies, but others, such as HTML5 LocalStorage and browser cache are also stateful tracking mechanisms. Figure 2 presents the well-known stateful tracking mechanisms. We distinguish two components necessary for successful tracking: user recognition and website identification. For each component, we describe the capabilities of in-context and cross-context. We also distinguish *passive tracking* (done through HTTP headers) and *active tracking* (through JavaScript or plugin script execution).

	User Recognition		**Website Identification**	
	Passive	Active	Passive	Active

in-context	HTTP cookies
	Cache-Control
	Etag
	Last-Modified
		-	Referer
			Origin
				`document.URL`
				`document.location`
				`window.location`

cross-context		Flash LSOs
		`document.cookie`
		`window.localStorage`
		`window.indexedDB`
			Referer	`document.referrer`

Fig. 2. Stateful tracking mechanisms

In-context tracking In-context third party content is associated with the browsing context of the webpage that embeds it (see Table 1).

Passively, such content may use HTTP header to recognize the user and identify the visited website. When a webpage is rendered, the browser sends a request to fetch all third party contents embedded in the page. The response from the third party with the requested content may contain HTTP headers that may be used for tracking. For example, *Set-cookie* HTTP header tells the browser to save the third party cookies, that will be later automatically attached to every request to this third party in the Cookie header. Etag HTTP header and other cache mechanisms like *Last-Modified* and *Cache-Control* HTTP headers may also be used to store user identifier [37]. To identify the visited website, a third party can either check the *Referer* HTTP header, automatically attached by the browser, or an *Origin* header[4].

Actively, in-context third party content cannot use browser storage mechanisms, such as cookies or HTML5 Local Storage associated to the third party

[4] Origin header is also automatically generated by the browser when the third party content is trying to access data using Cross-Origin Resource Sharing [4] mechanism.

because of the limitations imposed by the SOP (see Section 2.1). For example, if a third party script uses `document.cookie` API, it is able only to read the cookies of the main website, but not those associated to the third party. This allows tracking within the main website but does not allow tracking cross-sites [36]. For website identification, third party active content, such as JavaScript, can use several APIs, such as `document.location` and others.

Cross-context tracking Cross-context third party content, such as `iframe`, is associated with the browsing context of the third party that provided this content.

Passively, the browser may transmit HTTP headers used for user recognition and website identification, just like with the in-context third party content. Every third-party request for cross-context content will contain the URL of the embedding webpage in its *Referer* header. Note that this is true only for the cross-context content, say an `<iframe>`, directly embedded in the webpage. Within the iframe, there may be additional third party contents. Since they are not embedded directly in the webpage, and because the iframe is an autonomous though nested browsing context, requests to fetch contents embedded within this context will carry, not the URL of the webpage, but that of the iframe in their *Referer* header, and the origin of the iframe in their CORS requests *Origin* header.

Actively, cross-context third party content can use a number of APIs to store user identifier in the browser. These APIs include cookies (`document.cookie`), HTML5 LocalStorage (`document.localStorage`), IndexedDB, and Flash Local Stored Objects (LSOs). For website identification, `document.referrer` API can be used – it returns the value of HTTP Referrer header transmitted to the third party when the third party content was fetched. Because cross-context third party is associated with its own browsing context, it is able to embed even more third party contents within this cross-context.

Combining in-context and cross-context tracking Imagine a third party script from `third.com` embedded in a webpage – according to the context and to the SOP, it is in-context. If the same webpage embeds another third party content from `third.com`, which is cross-context, then because of SOP, such script and iframe cannot interact directly. However, script and iframe can still communicate through inter-frame communication APIs such `postMessage` [12].

This communication between different contexts allow them to exchange the user identifiers and the website that the user visits. Efficient implementation of such combination of tracking may profit from easily implementable user recognition by cross-context code using, say `document.cookie`, and website identification by in-context through various APIs such as `document.location`. For example, social widgets, such as Facebook "Like" button, or Google "+1" button, may be included in the webpages as a script. When the social widget script is executed on the client-side, it loads additional scripts, and new browsing contexts (iframes) allowing the third party to benefit from both in-context and cross-context capabilities to track users.

3 Privacy-preserving Web Architecture

For third party tracking to be effective, it is necessary that it has two capabilities: 1) it is able to identify the website in which it is embedded, and 2) to recognize the user interacting with that website. Disabling only one of these two capabilities for a given third party already prevents tracking. In order to mitigate the stateful tracking (see Section 2), we make the following design choices in our architecture:

1. **In-context content: prevent only user recognition**. Preventing passive user recognition for in-context content, such as images, forms and scripts is possible by removing HTTP headers such as *Set-cookie, ETag* and others. However, it is particularly difficult to remove active website identification because trying to alter or redefine `document.location` and `window.location` APIs, will cause the main page to reload.
2. **Cross-context content: prevent only website identification**. We prevent passive website identification by instructing the browser not to send HTTP *Referer* header along with requests to fetch a cross-context content. Therefore, when the cross-context gets loaded, active website identification is impossible. Indeed, executing `document.referrer` returns not the URL of the embedding page, but an empty string. Because of the limitations of the SOP, a website owner has no control over the cross-context third party content, such as iframes. Therefore, it is not possible to modify the results of storage access APIs, such as `document.cookie`. We discuss other possibilities to block such APIs in Section 4.3.
3. **Prevent communication between in-context and cross-context contents**. Our architecture proposes a way to block such communications that can be done by `postMessage` API. We discuss the limitations of this approach in Section 4.3.

To help web developers keep their promises of non-tracking and still include third-party content in their web applications, we propose a new Web application architecture. This architecture has the capability to 1) automatically rewrite all the third party *in-context* content of a Web application, 2) redirect the third party HTTP requests issued by the *in-context* content, and 3) remove/disable known *stateful* tracking mechanisms (see Section 2) for such third party content and requests. 4) It also rewrites and redirects cross-context requests so as to prevent website identification and communication with in-context scripts.

Figure 3 provides an overview of our web application architecture, that introduces two new components that are fully controlled by the website owner:

Rewrite Server (Section 3.1) acts like a reverse proxy[5] for the original web server. It rewrites the web pages in such a way that all the third party requests are redirected through the Middle Party Server before reaching the intended third party server.

[5] `https://en.wikipedia.org/wiki/Reverse_proxy`

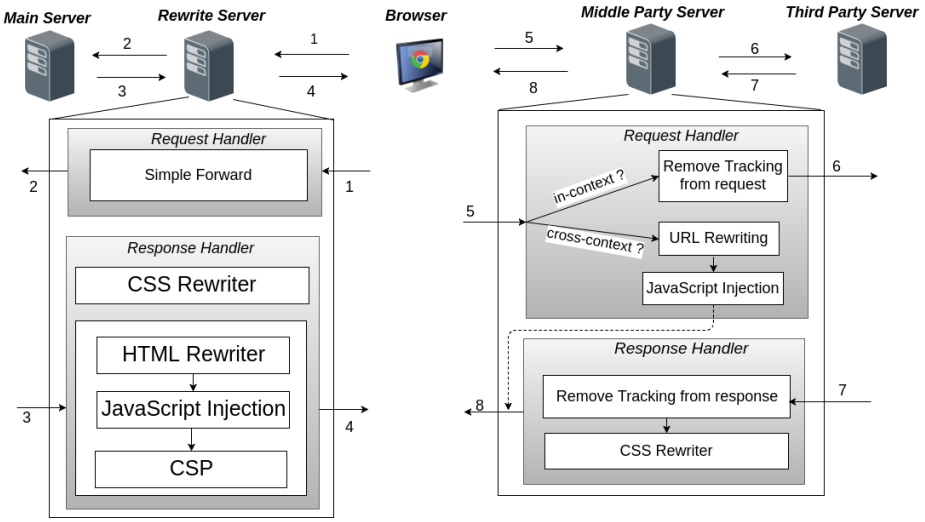


Fig. 3. Privacy-Preserving Web Architecture

Middle Party Server (Section 3.1) is at the core of our solution since it intercepts all browser third party requests, removes tracking, then forwards them to the intended third parties. When they reply, it also removes tracking information and forwards the responses back to the browser. On one hand, it hides the third party destination from the browser, and therefore prevents the browser from attaching third party HTTP cookies to such requests. Because the browser will still attach some tracking information to the requests, such as *ETag*, and *Referer* headers, Middle Party Server will also remove this information when forwarding the requests to the third party. This prevents passive user identification for in-context third party contents.

On the other hand, the Middle Party Server prevents website identification for cross-context contents and communication with in-context scripts. This is done by placing the cross-context within another cross-context controlled by the Middle Party server as illustrated by Figure 4. For instance, if an iframe was to be embedded within a webpage, it is placed within another iframe that belongs to the Middle Party. The Middle Party then instructs the browser not to send *Referer* header while loading the iframe, which prevents passive and active website identification once it is loaded. Since the iframe is nested within a iframe that belongs to Middle Party, this hides its reference to in-context scripts (see Figure 4). Therefore, it is prevented from communicating with in-context scripts in the main webpage.

3.1 Rewrite Server

The goal of the Rewrite Server is to rewrite the original content of the requested webpages in such a way that all third party requests will be redirected to the

API	Content
`document.createElement`	inject contents from Table 1
`document.write`	any content
`window.open`	Web pages(popups)
Image	images
XMLHttpRequest	any data
Fetch, Request	any content
Event Source	stream data
WebSocket	websocket data

Table 2. Embedding Dynamic Third Party Contents

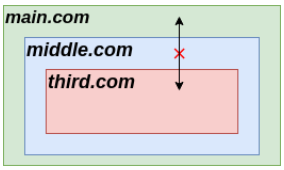


Fig. 4. Prevent Combining in-context and cross-context tracking

Middle Party Server. It consists of three main components: static HTML rewriter for HTML pages, static CSS rewriter and JavaScript injection component. Into each webpage, we inject a JavaScript code that insures that all the dynamically generated third party content is redirected to the Middle Party Server.

HTML and CSS Rewriter rewrites the URLs of static third party contents embedded in original web pages and CSS files in order to redirect them to the Middle Party Server. For example, the URL of a third-party script source `http://third.com/script.js` is written so that it is instead fetched through the Middle Party Server: `http://middle.com/?src=http://third.com/script.js`.

JavaScript Injection. The Rewrite Server also injects a script in an original webpage, that controls APIs used to inject dynamic contents. This injected script rewrites third party contents which are dynamically injected in webpages after they are rendered on the client-side. Table 2 shows APIs that can be used to dynamically inject third party content within a webpage that we control using the injected script.

A **Content Security Policy (CSP)** [41] is injected in the response header for each webpage in order to prevent third parties from bypassing the rewriting and redirection to the Middle Party Server. A CSP delivered with the webpage controls the resources of the page. It allows to specify which resources are allowed to be loaded and executed in the page. By limiting the resource origins to only those from the Middle Party Server and the website own domain, we prevent third parties from bypassing our redirection to the Middle Party Server.

3.2 Middle Party

The main goal of the Middle Party is to proxy the requests and responses between browsers and third parties in order to remove tracking information exchanged

between them. For in-context contents, it removes any user recognition as well as website identification information. For cross-context contents, it takes care of preventing website identification and communication with in-context scripts.

In-Context Contents are scripts, images, etc. (see Table 1). Since a third party script from `http://third.com/script.js` is rewritten by the Rewrite Server to `http://middle.com/?src=http://third.com/script.js`, it is fetched through the Middle Party Server. When the middle party receives such a request URL from the browser, it takes the following steps. **Remove Tracking from request** that are set by the browser as HTTP headers. Among those headers are *Cookie, Etag, If-Modified-Since, Cache-Control, Referer*. Next, it makes a request to the third party in order to get the content of the script `http://third.com/script.js`. **Remove Tracking from response** returned by the third party. The headers that the third party may send are *Set-Cookie, Etag, Last-Modified, Cache-Control*. **CSS Rewriter** rewrites the response if the content is a CSS file. Finally, the response is returned back to the browser.

Cross-context contents are iframes, links, popups, etc. (see Table 1). For instance, a third party iframe from `http://third.com/page.html` is rewritten to `http:// middle.com/?emb=http://third.com/page.html`. When the Middle Party Server receives such a request URL from the browser, it takes the following actions: **URL Rewriting**: instead of fetching directly the content of `http://third.com/page.html`, the Middle Party Server generates a content in which it puts the URL of the third party content as a hyperlink. `<a href = "http://third.com/page.html" rel = "noreferrer noopener"></a>`. The most important part of this content is in the `rel` attribute value. Therefore, `noreferrer noopener` instructs the browser not the send the *Referer* header when the link `http://third.com/page.html` is followed client-side. **JavaScript injection** module adds a script to the content so that the link gets automatically followed once the response is rendered by the browser. Once the link is followed, the browser fetches the third party content directly on the third party server, without going through the Middle Party server anymore. Nonetheless, it does not include the *Referer* header for identifying the website. Therefore, the `document.referrer` API also returns an empty string inside the iframe context. This prevents it from identifying the website.

The third party server response is placed within a new iframe nested within a context that belongs to the Middle Party, and not directly within the site webpage. This prevents in-context scripts and the cross-context contents from exchanging tracking information as illustrated by Figure 4.

4 Implementation

We have implemented both the Rewrite Server and the Middle Party Server as full Node.js [10] web servers supporting *HTTP(S)* protocols and web sockets. Implementation details are available at `https://webstats.inria.fr/sstp/`.

4.1 Rewrite Server

Simple Forward: requests that arrive to the Rewrite server are simply forwarded to the main server.

HTML Rewriter is implemented with Jsdom HTML parser [8] and **CSS Rewriter** using a CSS parser [5] for Node.js. **JavaScript injection** is done at the end of rewriting webpages. The code script injected is available at `https://webstats.inria.fr/sstp/dynamic.js`. **CSP** set on webpages only whitelists the website own domain and the Middle Party. It also prevents third party plugins.

```
Content-Security-Policy: default-src 'self' 'middle.com';
    object-src 'self';
```

4.2 Middle Party

In-Contexts Contents. Remove Tracking from requests component removes tracking information from in-context third party requests (See Section 3). The requests are then forwarded to the original third party server, to fetch the third party content. **Remove Tracking from responses** : Tracking information that are set by third parties in the responses, are removed. See Section 3 for details about information that are removed. **CSS Rewriter**: as in the case of the Rewrite Server, this component is implemented using a a CSS parser [5] for Node.js for rewriting CSS files.

Cross-Context Contents. URL Rewriting If the cross-context URL was `http://third.com/page.html`, this URL is rewritten to

```
<a href="http://third.com/page.html" rel="noreferrer
    noopener" target=""></a>.
```

JavaScript injection : the content injected is as followed.

```
var third_party = document.getElementsByTagName("a")[0];
if(window.top == window.self){
  third_party.target = "_blank";
  third_party.click();
  window.close();
}else{
  var iframe = document.createElement("iframe");
      iframe.name = "iframetarget";
  document.body.appendChild(iframe);
  third_party.target = "iframetarget";
  third_party.click();
}
```

Both the rewritten URL and the injected script are returned as a response to the browser which renders it. The injected script gets executed within a context that belongs to the Middle Party. If the original cross-context third party content was to be loaded inside an iframe, the injected script creates an iframe in which the

original third party content is loaded. However, if it was to be opened inside a new tab, the injected script opens a new tab in which the third party content is loaded. In both cases, while the cross-context content is loaded, the browser does not sent the *Referer* header. This makes the value of `document.referer` empty inside the cross-context preventing it from identifying the website. Finally, since those cross-context are loaded by the injected script from a context that belongs to the Middle Party, in-context scripts cannot communicate with the cross-context contents to exchange tracking information.

4.3 Discussion and Limitations

Our approach suffers from the following limitations. First, our implementation prevents cross-context and in-context contents from communicating with each other using `postMessage` API. However, in-context third party script can identify the website a user visits via `document.location.href` API. Then the script can include the website URL, say `http://main.com`, as a parameter of the URL of a third party iframe, for example `http://third.com/page.html?ref=http://main.com` and dynamically embed it in the webpage. In our architecture, this URL is rewritten and routed to the Middle Party. Since, the Middle Party Server does not inspect URL parameters, this information will reach the third party even though the *Referer* is not sent with cross-context requests.

Another limitation is that of dynamic CSS changes. For instance, changing the background image style of an element in the webpage is not captured by the dynamic rewriting script injected in webpages. Therefore, if the image was a third party image, the CSP will prevent it from loading.

Performance overhead There is a performance cost associated with the Rewrite Server. Rewriting contents server-side and browser-side is also expensive in terms of performance. Middle Party Server may also lead to performance overhead especially for webpages with numerous third party contents. We believe that server-side caching mechanisms may help to speed up responsiveness.

Extension to stateless tracking Even though this work did not address stateless tracking, such as device fingerprinting, our architecture already hides several fingerprintable device properties and can be extended to several others: 1) The redirection to the Middle Party anonymizes the real IP addresses of users; 2) Some stateless tracking APIs such as `window.navigator`, `window.screen`, and `HTMLCanvasElement` can be easily removed or randomized from the context of the webpage to mitigate in-context fingerprinting.

Possibility to blocking active user recognition in cross-context With the prevalence of third party tracking on the web, we have shown the challenges that a developer will face towards mitigating that. The sandbox attribute for iframes help prevent access to security-sensitive APIs. As tracking has become a hot concern, we suggest that similar mechanisms can help first party websites tackle third party tracking. The sandbox attribute can for instance be extended with specific values to tackle tracking. Nonetheless, the sandbox attribute can be used to prevent cross-context from some stateful tracking mechanisms [9].

5 Evaluation and Case Study

Demo website We have set up a demo website that embeds a collection of third party contents, both in-context and cross-context. In-context contents include images, HTML5 audio and video, and a Google Map, which further loads dynamic contents such as images, fonts, scripts, and CSS files; a Youtube video as a cross-context content. Our demo website is accessible at `http://sstp-rewriteproxy.inria.fr`. When we deployed the Rewrite Server on `http://sstp-rewriteproxy.inria.fr`, the original server has been moved to `http://sstp-rewriteproxy.inria.fr:8080`, so that it is no longer directly accessible to users. The Middle Party server runs at `http://sstp-middleparty.inria.fr`.

Originally, when all the third parties were simply included in the main webpage, they may have also been tracking the website users (see Figure 1). After the deployment of our solution, we have been able to redirect all in-context third party contents to the Middle Party. We have been able to prevent the website identification in the cross-context Youtube video. In the Appendix, we show a screenshot of requests redirection to the Middle Party Server.

Real websites Since we did not have access to a real websites, we cannot install a Rewrite Server and to evaluate our solution. We therefore implemented a browser proxy based on a Node.js proxy [11], and included all the logic of the Rewrite Server within the proxy. The proxy is running at `http://sstp-rewriteproxy.inria.fr:5555`.

We then evaluated the solution on different kinds of websites: a news website `http://www.bbc.com`, an entertainment website `http://www.imdb.com`, and a shopping website `http://verbaudet.fr`. All three websites load content from various third party domains. In all websites, we rewrote all third party contents through the proxy (acts as Rewrite Server) and the Middle Party Server removed tracking information. Visually, we did not notice any change in the behaviors of the websites. We also interacted with them in a standard way (clicking on links on a news website, choosing products and putting them in the basket on the shopping website) and all the main functionalities of the websites was preserved.

Overall, these evaluation scenarios have helped us improve the solution, especially rewriting dynamically injected third party content. We believe that this implementation will even get better in the future when we convince to deploy it for some real websites.

6 Related Work

Many studies have demonstrated that third party tracking is very prevalent on the web today as well as the underlying tracking technologies [25, 36, 31, 28]. Lerner et al. [30] dusted the story of this practice for a period of twenty years. Trackers have been categorized according to either their business relationships with websites [31], their prominence [28, 25] or the user browsing profile that they can build [36]. Mayer and Mitchell [31] grouped tracking mechanisms in two

categories called statefull (cookie-based and super-cookies) and stateless (fingerprinting). It is rather intuitive to convince ourselves about the effectiveness of a statefull tracking, since the latter is based on unique identifiers that are set in users browsers. Nonetheless, the efficacy of stateless mechanisms has been extensively demonstrated. Since the pioneer work of Eckersley [24], new fingerprinting methods have been revealed in the literature [38, 22, 25, 19, 21, 17, 39, 33, 18]. A classification of fingerprinting techniques is provided in [40]. Those studies have contributed to raising public awareness of tracking privacy threats. Mayer and Mitchell [31] have shown that users are very sensitive to their online privacy, thus hostile to third party tracking. Englehardt et al. [26] have demonstrated that tracking can be used for surveillance purposes. The success of anti-tracking defenses is yet another illustration of users concern regarding tracking [32].

There are many defenses that try to protect users against third party tracking. First, major browser vendors do natively provide mechanisms for users to block third party cookies, browse in private mode. More and more privacy-browsers even take a step further, putting privacy as a design and implementation principle. Examples of such browsers are the Tor Browser [16], TrackingFree Browser [34] or Blink [29]. But the most popular defenses are by far browser extensions. Being tightly integrated to browsers, they provide additional privacy features that are not natively implemented in browsers. Well known extensions for privacy are Disconnect [6], Ghostery [7], AdBlock [1], ShareMeNot [36], which is now part of PrivacyBadger [13], MyTrackingChoices [20], MyAdChoices [35]. Merzdovnik et al. [32] provide a large-scale study of anti-tracking defenses. Well known trackers such as advertisers, which businesses hugely depend on tracking, have also been taking steps towards limiting their tracking capabilities [31]. The W3C is pushing forward the Do Not Tracking standard [23, 27] for users to easily express their tracking preferences so that trackers may comply with them. To the best of our knowledge, we are the first to investigate how a website owner can embed third party content while preventing them from accidentally tracking users. The idea of proxying requests within a webpage is inspired by web service workers API [15], though the latter is still a working draft which is being tested in Mozilla Firefox and Google Chrome.

7 Conclusions

Most of the previous research analysed third party tracking mechanisms, and how to block tracking from a user perspective. In this work, we classified third party tracking capabilities from a website developer perspective. We proposed a new architecture for website developers that allows to embed third party contents while preserving users privacy. We implemented our solution, and evaluated it on real websites to mitigate stateful tracking.

References

[1] AdBlock - Block Ads - Browse Safe. https://getadblock.com/.
[2] Browsing Contexts. https://www.w3.org/TR/html51/browsers.html.
[3] Cascading Style Sheets. https://www.w3.org/Style/CSS/.
[4] Cross-origin-resource sharing. https://developer.mozilla.org/en-US/docs/Web/HTTP/Access_control_CORS.
[5] CSS Parser for Node.js. https://github.com/reworkcss/css.
[6] Disconnect. https://disconnect.me/.
[7] Ghostery. https://www.ghostery.com/.
[8] HTML Parser for Node.js. https://github.com/tmpvar/jsdom.
[9] Iframe Sandbox Attribute. https://www.w3.org/TR/2011/WD-html5-20110525/the-iframe-element.html#attr-iframe-sandbox.
[10] Node.js. https://nodejs.org/en/.
[11] Node.js Proxy. https://newspaint.wordpress.com/2012/11/05/node-js-http-and-https-proxy.
[12] PostMessage - Cross-Origin Iframe Secure Communication. https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage.
[13] Privacy Badger - Electronic Frontier Foundation. https://www.eff.org/fr/privacybadger.
[14] Same Origin Policy. https://www.w3.org/Security/wiki/Same_Origin_Policy.
[15] Service Worker API. https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API.
[16] The Design and Implementation of the Tor Browser [Draft]. https://www.torproject.org/projects/torbrowser/design/.
[17] E. Abgrall, Y. L. Traon, M. Monperrus, S. Gombault, M. Heiderich, and A. Ribault. XSS-FP: browser fingerprinting using HTML parser quirks. *CoRR*, abs/1211.4812, 2012.
[18] G. Acar, C. Eubank, S. Englehardt, M. Juárez, A. Narayanan, and C. Díaz. The web never forgets: Persistent tracking mechanisms in the wild. In G. Ahn, M. Yung, and N. Li, editors, *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*, pages 674–689. ACM, 2014.
[19] G. Acar, M. Juárez, N. Nikiforakis, C. Díaz, S. F. Gürses, F. Piessens, and B. Preneel. Fpdetective: dusting the web for fingerprinters. In A. Sadeghi, V. D. Gligor, and M. Yung, editors, *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 1129–1140. ACM, 2013.
[20] J. P. Achara, J. Parra-Arnau, and C. Castelluccia. Mytrackingchoices: Pacifying the ad-block war by enforcing user privacy preferences. *CoRR*, abs/1604.04495, 2016.
[21] K. Boda, Á. M. Földes, G. G. Gulyás, and S. Imre. User tracking on the web via cross-browser fingerprinting. In P. Laud, editor, *Information Security Technology for Applications - 16th Nordic Conference on Secure IT Systems, NordSec 2011, Tallinn, Estonia, October 26-28, 2011, Revised Selected Papers*, volume 7161 of *Lecture Notes in Computer Science*, pages 31–46. Springer, 2011.
[22] Y. Cao, S. Li, and E. Wijmans. (cross-)browser fingerprinting via os and hardware level features. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, 26 February - 1 March, 2017*, 2017. To Appear.

[23] N. Doty. Tracking Compliance and Scope, 2016. `https://www.w3.org/TR/tracking-compliance/`.

[24] P. Eckersley. How unique is your web browser? In M. J. Atallah and N. J. Hopper, editors, *Privacy Enhancing Technologies, 10th International Symposium, PETS 2010, Berlin, Germany, July 21-23, 2010. Proceedings*, volume 6205 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2010.

[25] S. Englehardt and A. Narayanan. Online tracking: A 1-million-site measurement and analysis. In E. R. Weippl, S. Katzenbeisser, C. Kruegel, A. C. Myers, and S. Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 1388–1401. ACM, 2016.

[26] S. Englehardt, D. Reisman, C. Eubank, P. Zimmerman, J. Mayer, A. Narayanan, and E. W. Felten. Cookies that give you away: The surveillance implications of web tracking. In A. Gangemi, S. Leonardi, and A. Panconesi, editors, *Proceedings of the 24th International Conference on World Wide Web, WWW 2015, Florence, Italy, May 18-22, 2015*, pages 289–299. ACM, 2015.

[27] R. T. Fielding. Tracking Preference Expression (DNT), 2015. `https://www.w3.org/TR/tracking-dnt/`.

[28] B. Krishnamurthy and C. E. Wills. Privacy diffusion on the web: a longitudinal perspective. In J. Quemada, G. León, Y. S. Maarek, and W. Nejdl, editors, *Proceedings of the 18th International Conference on World Wide Web, WWW 2009, Madrid, Spain, April 20-24, 2009*, pages 541–550. ACM, 2009.

[29] P. Laperdrix, W. Rudametkin, and B. Baudry. Beauty and the beast: Diverting modern web browsers to build unique browser fingerprints. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 878–894. IEEE Computer Society, 2016.

[30] A. Lerner, A. K. Simpson, T. Kohno, and F. Roesner. Internet jones and the raiders of the lost trackers: An archaeological study of web tracking from 1996 to 2016. In *25th USENIX Security Symposium (USENIX Security 16)*, Austin, TX, 2016. USENIX Association.

[31] J. R. Mayer and J. C. Mitchell. Third-party web tracking: Policy and technology. In *IEEE Symposium on Security and Privacy, SP 2012, 21-23 May 2012, San Francisco, California, USA*, pages 413–427. IEEE Computer Society, 2012.

[32] G. Merzdovnik, M. Huber, D. Buhov, N. Nikiforakis, S. Neuner, M. Schmiedecker, and E. Weippl. Block me if you can: A large-scale study of tracker-blocking tools. In *2nd IEEE European Symposium on Security and Privacy*, Paris, France, 2017. To appear.

[33] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Piessens, and G. Vigna. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 541–555. IEEE Computer Society, 2013.

[34] X. Pan, Y. Cao, and Y. Chen. I do not know what you visited last summer: Protecting users from stateful third-party web tracking with trackingfree browser. In *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society, 2015.

[35] J. Parra-Arnau, J. P. Achara, and C. Castelluccia. Myadchoices: Bringing transparency and control to online advertising. *CoRR*, abs/1602.02046, 2016.

[36] F. Roesner, T. Kohno, and D. Wetherall. Detecting and defending against third-party tracking on the web. In S. D. Gribble and D. Katabi, editors, *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation,*

NSDI 2012, San Jose, CA, USA, April 25-27, 2012, pages 155–168. USENIX Association, 2012.

[37] A. Soltani, S. Canty, Q. Mayo, L. Thomas, and C. J. Hoofnagle. Flash cookies and privacy. In *AAAI spring symposium: intelligent information privacy management*, pages 158–163, 2010.

[38] O. Starov and N. Nikiforakis. Extended tracking powers: Measuring the privacy diffusion enabled by browser extensions. In *Proceedings of the 26th International Conference on World Wide Web, WWW 2017, Perth, Australia, April 3 - 7, 2017*, 2017. To Appear.

[39] N. Takei, T. Saito, K. Takasu, and T. Yamada. Web browser fingerprinting using only cascading style sheets. In L. Barolli, F. Xhafa, M. R. Ogiela, and L. Ogiela, editors, *10th International Conference on Broadband and Wireless Computing, Communication and Applications, BWCCA 2015, Krakow, Poland, November 4-6, 2015*, pages 57–63. IEEE Computer Society, 2015.

[40] R. Upathilake, Y. Li, and A. Matrawy. A classification of web browser fingerprinting techniques. In M. Badra, A. Boukerche, and P. Urien, editors, *7th International Conference on New Technologies, Mobility and Security, NTMS 2015, Paris, France, July 27-29, 2015*, pages 1–5. IEEE, 2015.

[41] M. West, A. Barth, and D. Veditz. Content Security Policy Level 2. W3C Candidate Recommendation, 2015.

Appendix

Screenshot of the demo website map console.

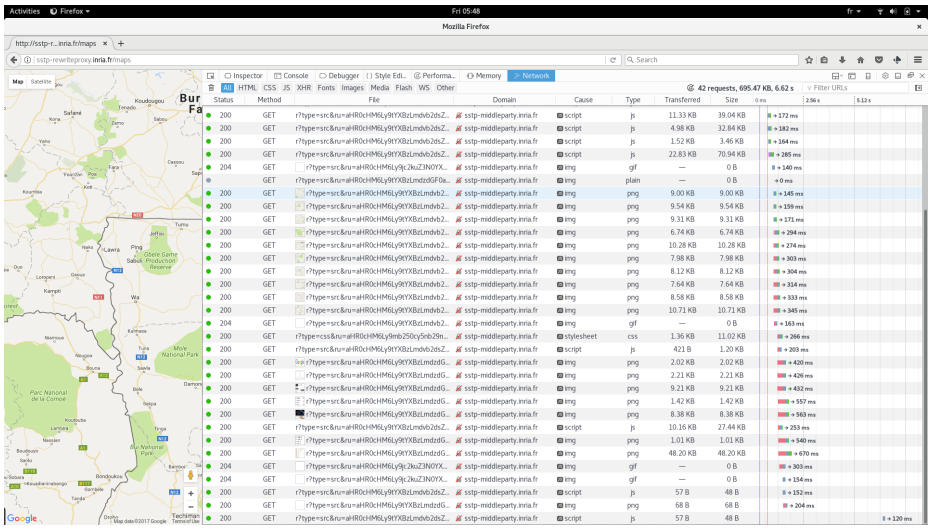


Fig. 5. Screenshot of the Browser console