Projet Ajacs

Deliverable WP3
Formalization of privacy properties and their enforcement by hybrid analysis

December 2018

This deliverable includes the following articles describing recent work done on WP3.

Formal Verification of Smart Contracts, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cdric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Bguelin

A Better Facet of Dynamic Information Flow Control, Minh Ngo, Nataliia Bielova, Cormac Flanagan, Tamara Rezk, Alejandro Russo, and Thomas Schmitz

Short Paper: Formal Verification of Smart Contracts

Karthikeyan Bhargavan[2] Antoine Delignat-Lavaud[1] Cédric Fournet[1]
Anitha Gollamudi[3] Georges Gonthier[1] Nadim Kobeissi[2] Aseem Rastogi[1]
Thomas Sibut-Pinote[2] Nikhil Swamy[1] Santiago Zanella-Béguelin[1]

[1]Microsoft Research [2]Inria [3]Harvard University

{antdl,fournet,gonthier,aseemr,nswamy,santiago}@microsoft.com
{karthikeyan.bhargavan,nadim.kobeissi,thomas.sibut-pinote}@inria.fr agollamudi@g.harvard.edu

Abstract

Ethereum is a cryptocurrency framework that uses blockchain technology to provide an open distributed computing platform, called the Ethereum Virtual Machine (EVM). EVM programs are written in bytecode which operates on a simple stack machine. Programmers do not usually write EVM code; instead, they can program in a JavaScript-like language called Solidity that compiles to bytecode. Since the main application of EVM programs is as *smart contracts* that manage and transfer digital assets, security is of paramount importance. However, writing trustworthy smart contracts can be extremely difficult due to the intricate semantics of EVM and its openness: both programs and pseudonymous users can call into the public methods of other programs. This problem is best illustrated by the recent attack on TheDAO contract, which allowed roughly \$50M USD worth of Ether to be transferred into the control of an attacker. Recovering the funds required a hard fork of the blockchain, contrary to the *code is law* premise of the system. In this paper, we outline a framework to analyze and verify both the runtime safety and the functional correctness of Solidity contracts in F*, a functional programming language aimed at program verification.

Categories and Subject Descriptors F.3 [*F.3.1 Specifying and Verifying and Reasoning about Programs*]

Keywords Ethereum, Solidity, EVM, smart contracts

1. Introduction

The blockchain technology, pioneered by Bitcoin [7] provides a globally-consistent append-only ledger that does not rely on a central trusted authority. In Bitcoin, this ledger records transactions of a virtual currency, which is created by a process called mining. In the *proof-of-work* mining scheme, each node of the network can earn the right to append the next block of transactions to the ledger by finding a formatted value (which includes all transactions to appear in the block) whose SHA256 digest is below some difficulty threshold. The system is designed to ensure that blocks are mined at a constant rate: when too many blocks are submitted too quickly, the difficulty increases, thus raising the computational cost of mining.

Ethereum is similarly built on a blockchain based on proof-of-work; however, its ledger is considerably more expressive than that of Bitcoin's: it stores Turing-complete programs in the form of Ethereum Virtual Machine (EVM) bytecode, while transactions are construed as function calls and can carry additional data in the form of arguments. Furthermore, contracts may also use non-volatile storage and log events, both of which are recorded in the ledger.

The initiator of a transaction pays a fee for its execution measured in units of *gas*. The miner who manages to append a block including the transaction gets to claim the fee converted to Ether at a specified gas price. Some operations are more expensive than others: for instance, writing to storage and initiating a transaction is four orders of magnitude more expensive than an arithmetic operation on stack values. Therefore, Ethereum can be thought of as a distributed computing platform where anyone can run code by paying for the associated gas charges.

The integrity of the system relies on the honesty of a majority of miners: a miner may try to cheat by not running the program, or running it incorrectly, but honest miners will reject the block and fork the chain. Since the longest chain is the one that is considered valid, miners are incentivized not to cheat and to verify that others do as well, since their block reward may be lost unless malicious miners can supply the majority of new blocks to the network.

While Ethereum's adoption has led to smart contracts managing millions of dollars in currency, the security of these contracts has become highly sensitive. For instance, a variant of a well-documented reentrancy attack was recently exploited in TheDAO [2], a contract that implements a decentralized autonomous venture capital fund, leading to the theft of more than \$50M worth of Ether, and raising the question of whether similar bugs could be found by static analysis [6].

In this paper, we outline a framework to analyze and formally verify Ethereum smart contracts using F* [9], a functional programming language aimed at program verification. Such contracts are generally written in Solidity [3], a JavaScript-like language, and compiled down to bytecode for the EVM. We consider the Solidity compiler as untrusted and develop a language-based approach for verifying smart contracts. Namely, we present two tools based on F⋆:

Solidity⋆ a tool to translate Solidity program to shallow-embedded F⋆ programs (Section 2).

EVM⋆ a decompiler for EVM bytecode that produces equivalent shallow-embedded F⋆ programs that operate on a simpler machine without stack (Section 3).

These tools enable three different forms of verification:

1. Given a Solidity program, we can use Solidity⋆ to translate it to F⋆ and verify at the source level functional correctness specifications such as contract invariants, as well as safety with respect to runtime errors.
2. Given an EVM bytecode, we can use EVM⋆ to decompile it and analyze low-level properties, such as bounds on the amount of gas consumed by calls.
3. Given a Solidity program and allegedly functionally equivalent EVM bytecode, we can verify their equivalence by translating each into F⋆. Thus, we can check the correctness of the output of the Solidity compiler on a case-by-case basis using relational reasoning [1].

1.1 Architecture of the Framework

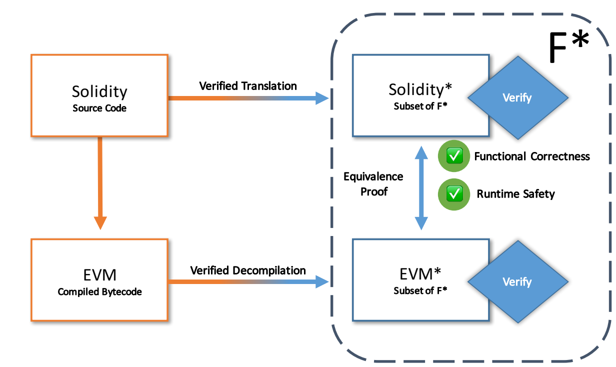

Figure 1. Overview of the architecture of our framework

Our smart contract verification framework is a two-pronged approach (Figure 1) based on F⋆. F⋆ comes with a type system that includes dependent types and monadic effects, which we apply to generate automated queries to statically verify properties on EVM bytecode and Solidity sources.

While it is clearly favorable to obtain both the Solidity source code and EVM bytecode of a target smart contract, we design our architecture with the assumption that the verifier may only have the bytecode. At the moment of this writing, only 396 out of 112,802 contracts have their source code available on `http://etherscan.io`. Therefore we provide separate tools for decompiling EVM bytecode (EVM⋆), and analyzing Solidity source code (Solidity⋆).

```
⟨solidity⟩ ::= (⟨contract⟩)*

⟨contract⟩ ::= 'contract ' @identifier '{' (⟨st⟩)*'}'

⟨st⟩ ::= ⟨typedef⟩ | ⟨statedef⟩ | ⟨method⟩

⟨typedef⟩ ::= 'struct ' @identifier ' {' (⟨type⟩ @identifier ';')* '}'

⟨type⟩ ::= 'uint' | 'address' | 'bool'
   | 'mapping (' ⟨type⟩ '=>' ⟨type⟩ ')'
   | @identifier

⟨statedef⟩ ::= ⟨type⟩ @identifier

⟨method⟩ ::= 'function' (@identifier)?'()' (⟨qualifier⟩)* '{'
      ('var' (@identifier ('=' ⟨expression⟩)? ',')+)?
      (⟨statement⟩ ';')* '}'

⟨qualifier⟩ ::= 'private' | 'public' | 'internal'
   | 'returns (' ⟨type⟩ (@identifier)? ')'

⟨statement⟩ ::= ε
   | ⟨type⟩ @identifier ('=' ⟨expression⟩)? (*decl*)
   | 'if(' ⟨expression⟩ ')' ⟨statement⟩
     ('else' ⟨statement⟩)?
   | '{' ((⟨statement⟩ ';')* '}'
   | 'return' (⟨expression⟩)?
   | 'throw'
   | ⟨expression⟩

⟨expression⟩ ::= ⟨literal⟩
   | ⟨lhs_expression⟩ '(' ((⟨expression⟩ ',')* ')'
   | ⟨expression⟩ ⟨binop⟩ ⟨expression⟩
   | ⟨unop⟩ ⟨expression⟩
   | ⟨lhs_expression⟩ '=' ⟨expression⟩
   | ⟨lhs_expression⟩

⟨lhs_expression⟩ ::=
   | @identifier
   | ⟨lhs_expression⟩ '[' ⟨lhs_expression⟩']'
   | ⟨lhs_expression⟩ '.' @identifier

⟨literal⟩ ::= ⟨function⟩
   | '{' ( @identifier ':' ⟨expression⟩ ',')* '}'
   | '[' ((⟨expression⟩ ',')* ']'
   | @number | @address | @boolean

⟨binop⟩ ::= '+' | '-' | '*' | '/' | '%'
   | '&&' | '||' | '==' | '!=' | '>' | '<' | '>=' | '<='

⟨unop⟩ ::= '+' | '-' | '!'
```

Figure 2. Syntax of the translated Solidity subset

2. Translating Solidity to F⋆

In the spirit of previous work on type-based analysis of JavaScript programs [8], we advocate an approach where the programmer can verify high-level goals of a contract using F⋆. In this section, we present a tool to translate Solidity to F⋆, and a simple automated analysis of extracted F⋆ contracts.

Solidity programs consist of a number of contract declarations. Once compiled to EVM, contracts are installed using a special kind of account-creating transaction, which allocates an address to the contract. Unlike Bitcoin, where an address is the hash of the public key of an account, Ethereum addresses can refer indistinguishably to a contract or a user public key. Similarly, there is no distinction between transactions and method calls: when sending Ether to a contract, it will implicitly call the fallback function (the unnamed method of the Solidity contract). In fact, compiled contracts in the blockchain consist of a single entry point that decides depending on the incoming transaction which method code to invoke. The methods of a Solidity contract have access to ambient global variables that contain information about the contract (such as the balance in `this.balance`), the transaction used to invoke the contract's method (such as the source address in `msg.sender` and the amount of ether sent in `msg.value`), or the block in which the invocation transaction is mined (such as the miner's timestamp in `block.timestamp`).

In this exploratory work, we consider a restricted subset of Solidity, shown in Figure 2. Notably, the fragment we consider does not include loops. The three main types of declarations within a contract are type declarations, property declarations and methods. Type declarations consist of C-like structs and enums, and mappings (associative arrays implemented as hash tables). Although properties and methods are reminiscent of object oriented programming, it is somewhat a confusing analogy: contracts are "instantiated" by the account creating transaction; this will allocate the properties of the contract in the global storage and call the constructor (the method with the same name as the contract). Despite the C++/Java-like access modifiers, all properties of a contract are stored in the Ethereum ledger, and as such, the internal state of all contracts is completely public. Methods are compiled in EVM into a single function that runs when a transaction is sent to the contract's address. This transaction handler matches the requested method signature with the list of non-internal methods, and calls the relevant one. If no match is found, a fallback handler is called instead (in Solidity, this is the unnamed method).

2.1 Translation to F*

We perform a shallow translation of Solidity to F* as follows:

1. contracts are translated to F* modules;
2. type declarations are translated to type declarations: enums become sums of nullary data constructors, structs become records, and mappings become F* maps;
3. all contract properties are packaged together within a `state` record, where each property is a reference;
4. each method gets translated to a function, no defunctionalization is required since Solidity is first-order only;
5. we rewrite `if` statements that have a continuation depending on whether one branch ends in `return` or `throw` (moving the continuation in the other branch) or not (we then duplicate the continuation in each branch).
6. to translate assignments, we keep an environment of local, state, and ambient global variable names: local variable declarations and assignments are translated to `let` bindings; globals are replaced with library calls; state properties are replaced with `update` on the `state` type;
7. built-in method calls (e.g.`address.send()`) are replaced by library calls.

We show a minimalistic Solidity contract and its F* translation in Figure 3. The only type annotation added by the translation is a custom `Eth` effect on the contract's methods, which we describe in Section 2.2. The `Solidity` library defines the `mapping` type (a reference to a map) and the associated functions `update_map` and `lookup`. Furthermore, it defines the numeric types used in Solidity, which are unsigned 256-bit by default.

2.2 An effect for detecting vulnerable patterns

The example in Figure 3 captures two major pitfalls of Solidity programming. First, many contracts fail to realize that `send` and its variants are not guaranteed to succeed (`send` returns a `bool`). This is highly surprising for Solidity programmers because all other runtime errors (such as running out of gas or call stack overflows) trigger an exception. Such exceptions (including the ones triggered by `throw`) revert all transactions and all changes to the contract's properties. This is *not* the case of `send`: the programmer needs to undo side effects manually when it returns `false`, e.g. **if**(!addr.send(x)) **throw**.

The other problem illustrated in `MyBank` is reentrancy. Since transactions are also method calls, calling `send` is a transfer of program control. Consider the following malicious contract:

```
contract Malicious {
  uint balance;
  MyBank bank = MyBank(0xdeadbeef8badf00d...);

  function Malicious(){
     balance = msg.value;
     bank.Deposit.value(balance)();
     bank.Withdraw.value(0)(balance); // forwarding gas
  }

  function (){ // fallback function
    bank.Withdraw.value(0)(balance);
  }
}
```

It attacks the `Withdraw` method of `MyBank` by calling recursively into it at the point where it does its `send`. The `if` condition in the second `Withdraw` call is still satisfied (because the balances are updated after `send`, and there is no check that it was successful). Even though the `send` in the second call to `Withdraw` is guaranteed to fail (because unlike method calls, `send` allocates only 2300 gas for the call), it still corrupts the balance by decreasing twice, causing an unsigned integer underflow. After corrupting the balance,

```
contract MyBank {
  mapping (address ⇒ uint) balances;

  function Deposit() {
    balances[msg.sender] += msg.value;
  }

  function Withdraw(uint amount) {
    if(balances[msg.sender] ≥ amount) {
      msg.sender.send(amount);
      balances[msg.sender] −= amount;
    }
  }

  function Balance() constant returns(uint) {
    return balances[msg.sender];
  }
}
```

```
module MyBank
open Solidity

type state = { balances: mapping address uint; }
val store : state = {balances = ref empty_map}

let deposit () : Eth unit =
  update_map store.balances msg.sender
      (add (lookup store.balances msg.sender) msg.value)

let withdraw (amount:uint) : Eth unit =
  if (ge (lookup store.balances msg.sender) amount) then
    send msg.sender amount;
    update_map store.balances msg.sender
       (sub (lookup store.balances msg.sender) amount)

let balance () : Eth uint =
  lookup store.balances msg.sender
```

Figure 3. A simple bank contract in Solidity translated to F*

the malicious contract can freely withdraw any remaining funds in the bank.

Using the effect system of F*, we now show how to detect some vulnerable patterns such as unchecked `send` results in translated contracts. The base construction is a combined exception and state monad (see [9] for details) with the following signature:

```
EST (a:Type) = h0:heap // input heap
  → send_failed:bool // send failure flag
  → Tot (option (a ∗ heap) // result and new heap, or exception
           ∗ bool) // new failure flag

return (a:Type) (x:a) : EST a =
  fun h0 b0 → Some (x, h0), b0

bind (a:Type) (b:Type) (f:EST a) (g:a → EST b) : EST b =
  fun h0 b0 →
    match f h0 b0 with
    | None, b1 → None, b1 // exception in f: no output heap
    | Some (x, h1), b1 → g x h1 b1 // run g, carry failure flag
```

The monad carries a `send_failure` flag to record whether or not a `send()` or external call may have failed so far. It is possible to enforce several different styles based on this monad; for instance, one may want to enforce that a contract always throws when a send fails. As an example, we defined the following effect based on EST:

```
effect Eth (a:Type) = EST a
  (fun _ b0 → not b0) // Start in non-failsure state
  (fun h0 b0 r b1 →
     // What to do when a send failed
     b1 ⟹ (match r with | None → True // exception
             | Some (_, h1) → no_mods h0 h1)) // no writes
```

The standard library then defines the post-condition of `throw` to `fun h0 b0 r b1 → b0=b1 ∧ is_None r` and the post-condition of `send` to `fun h0 b0 r b1 → r == Some (b1, h0)`.

Simply by typechecking extracted methods in the `Eth` effect, we can detect dangerous patterns such as the send() followed by an unconditional write to the `balances` table in `MyBank`. Note that the safety condition imposed by `Eth` is not sufficient to prevent reentrency attacks, as there is no guarantee that the state modifictions before and after send preserve the functional invariant of the contract. Therefore, this analysis is useful for detecting dangerous patterns and enforcing a failure handling style, but it doesn't replace a manual F* proof that the contract is correct.

Evaluation Despite the limitations of our tool (in particular, it doesn't support many syntactic features of Solidity), we are able to translate and typecheck 46 out of the 396 contracts we collected on `https://etherscan.io`. Out of these, only a handful are valid in the `Eth` effect. This is a clear sign that a large scale analysis of published contract is likely to uncover widespread vulnerabilities; we leave such analysis to future work.

3. Decompiling EVM Bytecode to F*

In this section we present EVM*, a decompiler for EVM bytecode that we use to analyze contracts for which the Solidity source is unavailable (as is the case for the majority of live contracts in the Ethereum blockchain), as well as low-level properties of contracts. A third use case of the decompiler that we do not further explore in this paper is to use EVM* together with Solidity* to check the equivalence between a Solidity program and the bytecode output by the Solidity compiler, thus ensuring not only that the compiler did not introduce bugs, but also that any properties verified at the source level are preserved. This equivalence proof could be done, for instance, using rF* [1] a version of F* with relational refinement types.

EVM* takes as input the bytecode of a contract as stored in the blockchain and translates it into a representation in F*. The decompiler performs a stack analysis to identify jump destinations in the program and detect stack under- and overflows. The result is an equivalent F* program that, morally, operates on a machine with infinite single-assignment registers which we translate as let bindings.

The EVM is a stack-based machine with a word size of 256 bits [10]. Bytecode programs have access to a word-addressed non-volatile storage modeled as a word array, a word-addressed volatile memory modeled as an array of bytes, and an append-only non-readable event log. The instruction set includes the usual arithmetic and logic operations (e.g. ADD, XOR), stack and memory operations (e.g. PUSH, POP, MSTORE, MLOAD, SSTORE, SLOAD), control flow operations (e.g. JUMP, CALL, RETURN), instructions to inspect the environment and blockchain (e.g. BALANCE, TIMESTAMP), as well as specialized instructions unique to EVM (e.g. SHA3, CREATE, SUICIDE). As a peculiarity, the instruction JUMPDEST is used to mark valid jump destinations in the code section of a contract, but behaves as a NOP at runtime. This is convenient for identifying potential jump destinations during decompilation, as jumping to an invalid address halts execution.

The static analysis done by EVM* marks stack cells as either of 3 types: 1. Void for initialized cells, 2. Local for results of operations, and 3. Constant for immediate arguments of PUSH operations The analysis identifies jumpable addresses and blocks, contiguous sections of code starting at a jumpable address and ending in a halting or control flow instruction (we treat branches of conditionals as independent blocks). A block summary consists of the address of its entry point, its final instruction, and a representation of the initial and final stacks summarizing the block effects on the stack. An entry point may be either the 0 address, an address marked with JUMPDEST, an immediate argument of a PUSH used in a jump, or a fall-through address of a conditional.

As a result of the static analysis, EVM* emits F* code, using variables bound in let bindings instead of stack cells. Many instructions can be eliminated in this way; the analysis keeps an accurate account of the offsets of instructions in the remaining code. Because the instructions eliminated may incur gas charges, we keep track of the fuel consumption by instrumenting the code with calls to burn, a library function whose sole effect is to accumulate gas charges. Figure 4 shows the F* code decompiled from the `Balance` method of the `MyBank` contract in Fig. 3.

We wrote a reference cost model for bytcode operations that can be used to prove bounds on the gas consumption of contract methods. As an example, Fig. 5 shows a type annotation for the entry point of the `MyBank` contract decompiled to F* that proves that a method call to the `Balance` function will consume at most 390 units of gas.

```
let x_29 = pow [0x02uy] [0xA0uy] in
let x_30 = sub x_29 [0x01uy] in
let x_31 = get_caller () in
let x_32 = land x_31 x_30 in
burn 17 (* opcodes: SUB, CALLER, AND, PUSH1 00, SWAP1, DUP2 *);
mstore [0x00uy] x_32;
burn 9 (* opcodes: PUSH1 20, DUP2, DUP2 *);
mstore [0x20uy] [0x00uy];
burn 9 (* opcodes: PUSH1 40, SWAP1, SWAP2 *);
let x_33 = sha3 [0x00uy] [0x40uy] in
let x_34 = sload x_33 in
burn 9 (* opcodes: PUSH1 60, SWAP1, DUP2 *);
mstore [0x60uy] x_34;
loadLocal [0x60uy] [0x20uy] (* returned value *)
```

Figure 4. Decompiled version of the `Balance` method of the `MyBank` contract, instrumented with gas consumption.

```
val myBank: unit → ST word
  (requires (fun h → sel h mem = 0 ∧ sel h gas = 0 ∧
    nonZero (eqw
      (div (get_calldataload [0x00uy]) (pow [0x02uy] [0xE0uy]))
      [0xF8uy; 0xF8uy; 0xA9uy; 0x12uy]))) // hash of Balance method
  (ensures (fun h0 _ h1 → sel h1 gas ≤ 390))

let myBank () =
  burn 6 (* opcodes: PUSH1 60, PUSH1 40 *);
  mstore [0x40uy] [0x60uy];
  ...
  let x_28 = eqw [0xF8uy; 0xF8uy; 0xA9uy; 0x12uy] x_3 in
  burn 10 (* opcode JUMPI *);
  if nonZero x_28 then
  begin (* offset: 165 *)
    // decompiled code of Balance method
  end
```

Figure 5. A proof of a bound on the gas consumed by a call to the `Balance` method of `MyBank`.

4. Conclusion

Our preliminary experiments in using F* to verify smart contracts show that the type and effect system of F* is flexible enough to express and prove non-trivial properties. In parallel, Luu et al. [6] used symbolic execution to detect flaws in EVM bytecode programs, and an experimental Why3 [5] formal verification backend is now available from the Solidity web IDE [4].

The examples we considered are simple enough that we did not have to write a full implementation of EVM bytecode. We plan to complete a verified reference implementation and use it to verify that the output of the Solidity compiler is functionally equivalent to the sources.

We implemented EVM* and Solidity* in OCaml. It would be interesting to implement and verify parts of these tools using F* instead. For instance, we could prove that the stack and control flow analysis done in EVM* is sound with respect to a stack machine semantics.

References

[1] G. Barthe, C. Fournet, B. Grégoire, P.-Y. Strub, N. Swamy, and S. Zanella-Béguelin. Probabilistic relational verification for cryptographic implementations. In *41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14*, pages 193–205. ACM, 2014.

[2] V. Buterin. Critical update re: Dao vulnerability. `https://blog.ethereum.org/2016/06/17/critical-update-re-dao-vulnerability`, 2016.

[3] Ethereum. Solidity documentation – Release 0.2.0. `http://solidity.readthedocs.io/`, 2016.

[4] Ethereum. Solidity-browser. `https://ethereum.github.io/browser-solidity`, 2016.

[5] J.-C. Filliâtre and A. Paskevich. Why3 — where programs meet provers. In *22nd European Symposium on Programming, ESOP '13*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, 2013.

[6] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor. Making smart contracts smarter. Cryptology ePrint Archive, Report 2016/633, 2016. `http://eprint.iacr.org/2016/633`.

[7] S. Nakamoto. Bitcoin: A peer-to-peer electronic cash system. `http://bitcoin.org/bitcoin.pdf`.

[8] N. Swamy, C. Fournet, A. Rastogi, K. Bhargavan, J. Chen, P. Strub, and G. M. Bierman. Gradual typing embedded securely in javascript. In *POPL '14*, pages 425–438. ACM, 2014.

[9] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoué, and S. Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '16*, pages 256–270. ACM, 2016.

[10] G. Wood. Ethereum: A secure decentralised generalised transaction ledger. `http://gavwood.com/paper.pdf`.

A Better Facet of Dynamic Information Flow Control

Minh Ngo
INRIA, France
nguyen-nhat-minh.ngo@inria.fr

Nataliia Bielova
INRIA, France
nataliia.bielova@inria.fr

Cormac Flanagan
UCSC, USA
cormac@ucsc.edu

Tamara Rezk
INRIA, France
tamara.rezk@inria.fr

Alejandro Russo
Chalmers University of Technology, Sweden
russo@chalmers.se

Thomas Schmitz
UCSC, USA
tschmitz@ucsc.edu

ABSTRACT

Multiple Facets (MF) is a dynamic enforcement mechanism which has proved to be a good fit for implementing information flow security for JavaScript. It relies on multi executing the program, once per each security level or view, to achieve soundness. By looking inside programs, MF encodes the views to reduce the number of needed multi-executions.

In this work, we extend Multiple Facets in three directions. First, we propose a new version of MF for arbitrary lattices, called Generalised Multiple Facets, or GMF. GMF strictly generalizes MF, which was originally proposed for a specific lattice of principals. Second, we propose a new optimization on top of GMF that further reduces the number of executions. Third, we strengthen the security guarantees provided by Multiple Facets by proposing a termination sensitive version that eliminates covert channels due to termination.

KEYWORDS

Multiple Facets; Dynamic Information Flow Control; Secure Multi-Execution; Noninterference

1 INTRODUCTION

JavaScript has become the de facto programming language of the Web. Web browsers daily execute thousands of JavaScript lines which usually have access to confidential information, for example cookies that mark that the user in a web session is authenticated. It is not surprising that JavaScript is a common target for attacks. While browsers deploy security measures in the form of access control (e.g., SOP and CSP), they are insufficient [12, 17, 30] to protect confidentiality of data.

Information flow control (IFC) is a promising technology which provides a systematic solution to handle unintentional or malicious leaks of confidential information. Recently, dynamic IFC analyses have received a lot of attention [1–3, 5, 7, 9, 10, 14, 26, 33], due, in part, to its applicability to JavaScript—where static analyses are rather an awkward fit [29].

WWW '18 Companion, April 23–27, 2018, Lyon, France

ACM ISBN 978-1-4503-5640-4/18/04.
https://doi.org/10.1145/3184558.3186348

In order to scale, a suitable IFC technique for the web not only needs to be dynamic but also needs to reduce to the minimum the modifications required to existing JavaScript code. In this light, an interesting dynamic IFC technique which fulfills both of these requirements consists in executing several copies of a program: one execution per each security level or view. In that manner, each copy of the program (view) depends only on information observable to the corresponding security level, where no leaks are therefore possible. Secure Multi Execution (SME) [14] and Multiple Facets (MF) [3] are two techniques based on this idea.

Both techniques have been proved to be a good fit for information flow security in the web since they have been successfully implemented as extensions of the Firefox browser [13, 33].

Although both SME and MF are based on multi-executions, they present important differences [7]. On one hand, SME is black-box [24], i.e., it is a mechanism that does not *look inside* programs but rather change the semantics of inputs and outputs to ensure security. For a moment, we assume a scenario where security levels are simply sets of principals (e.g., web origins) which denote those authorities with confidentiality concerns over data. In such a scenario, SME needs to spawn one execution for any possible set of principals—where the number of executions grows exponentially with respect to the number of principals! Instead, MF [3] is designed to reduce the number of multi-executions and the memory footprint of SME. It does so by inspecting programs code and multi-executing instructions and multiplexing memory only when needed. While MF is more resource-friendly than SME, SME provides stronger security guarantees when it comes to leaks via abnormal termination [7].

Our broad goal is to augment the efficiency of techniques based on MF and SME to general cases. In particular, we discovered that MF might sometimes spawn more multi-executions than SME—something that is counter-intuitive when considering the purpose of MF (see Section 2). Our first contribution consists on a novel technique to further reduce the number of multi-executions (and memory footprint) of MF. Our second contribution is to generalize MF to work for *arbitrary finite lattices* (see Section 3) rather than being restricted to the security lattice of principals as in the original proposal [3]. This becomes useful when, for instance, a program depends on 5 security levels. In such case, as stated originally, MF will need to encode them by using (at least) 3 principals ($2^3 > 5$), and thus execute the program $2^3 = 8$ times, while SME will execute it only 5 times (one per security level). Finally, we combine MF and SME into a single new dynamic IFC mechanism in order to provide security guarantees as strong as SME (i.e., termination sensitive

$$\textsc{skip}\ \frac{}{(\textbf{skip}, \mu) \Downarrow \mu} \qquad \textsc{assign}\ \frac{v = \mu(e)}{(x := e, \mu) \Downarrow \mu[x \mapsto v]}$$

$$\textsc{if}\ \frac{\mu(e) = v \qquad (P_v, \mu) \Downarrow \mu'}{(\textbf{if}\ e\ \textbf{then}\ P_{\mathrm{tt}}\ \textbf{else}\ P_{\mathrm{ff}}, \mu) \Downarrow \mu'} \qquad \textsc{seq}\ \frac{(P_1, \mu) \Downarrow \mu' \qquad (P_2, \mu') \Downarrow \mu''}{(P_1; P_2, \mu) \Downarrow \mu''}$$

$$\textsc{while}\ \frac{(\textbf{if}\ e\ \textbf{then}\ P; \textbf{while}\ e\ \textbf{do}\ P\ \textbf{else}\ \textbf{skip}, \mu) \Downarrow \mu'}{(\textbf{while}\ e\ \textbf{do}\ P, \mu) \Downarrow \mu'}$$

Figure 1: Language semantics

non-interference) while avoiding multi-executions as much as our optimized version of MF allows it. All proofs can be found in [23].

2 BACKGROUND ON SME AND MF

In this section, we discuss how on one hand, the underpinning mechanism in MF reduces the number of executions compared to SME, and on the other hand, may run more multi executions than SME because of the security lattice based on principals. Our goal here is partly pedagogical and partly to motivate and provide intuition on the optimization proposed in Section 4.

Language and Semantics To investigate the foundation of multiple facets, we use a simple, deterministic while language. Its syntax includes programs P, variables x, expressions e, and values v. We use the symbol $\oplus$ for binary expression operators. A value is either an integer value or a boolean value.

(programs) $P ::= \textbf{skip} \mid x := e \mid \textbf{if}\ e\ \textbf{then}\ P_1\ \textbf{else}\ P_2 \mid \textbf{while}\ e\ \textbf{do}\ P \mid P_1; P_2$

(expressions) $e ::= v \mid x \mid e \oplus e$

Figure 1 presents standard big-step semantics of the language. Memories μ map variables to values; we overload the notation of memory and use $\mu(e)$ as the evaluation function for expression e in memory μ, where $\mu(v) = v$ and $\mu(e_1 \oplus e_2) = \mu(e_1) \oplus \mu(e_2)$. We write $(P, \mu) \Downarrow \mu'$ to mean that the evaluation of program P on memory μ terminates with memory μ'. We use $\mu[x \mapsto v]$ for the memory μ' where $\mu'(y) = \mu(y)$ if $y \neq x$, and $\mu'(y) = v$ if $y = x$.

MF may use fewer resources than SME SME [14] multi executes programs, in a blackbox manner, as many times as security levels in a lattice. Let's define an SME memory as a function that maps each variable to an array of values, one value per security level. For the sake of simplicity, let's consider first a security lattice with only two elements H and L where $H \not\sqsubseteq L$ is the only disallowed flow. Thus, an SME memory $\hat{\mu}$ maps variables to an array of 2 (possibly different) values: one corresponding to the H view and one corresponding to the L view. Let's denote such array of values as $\langle v_1 : v_2 \rangle$, where v_1 is a private, H, view and v_2 is a public, L, view. Assume that $H(\hat{\mu})$ (resp. $L(\hat{\mu})$) is a memory in the standard semantics, obtained by projection of $\hat{\mu}$, mapping variables to single values of the high view (resp. low view). Then, the SME monitoring rule[1] for such a language can be given by the relation $\Downarrow_{SME-TINI}$ as follows:

[1] We give here the termination insensitive version of SME.

$$\text{SME-TINI}\ \frac{(P, H(\hat{\mu})) \Downarrow \mu_1 \qquad (P, L(\hat{\mu})) \Downarrow \mu_2}{(P, \hat{\mu}) \Downarrow_{SME-TINI} \mu_1 \odot \mu_2}$$

where $\odot$ combines two normal memories into a SME memory in such a way that $H(\mu_1 \odot \mu_2) = \mu_1$ and $L(\mu_1 \odot \mu_2) = \mu_2$. The SME mechanism will blindly execute the program as many times as possible views (or positions of the array) may exist.

Consider a program $h := l$ where initial views for variables l and h are given by: $\hat{\mu}(h) = \langle 1 : 0 \rangle$ and $\hat{\mu}(l) = \langle 1 : 1 \rangle$. In SME, using the SME-TINI rule, the assignment will be executed twice: once with $H(\hat{\mu}) = [h \mapsto 1, l \mapsto 1]$ for the high view and once with $L(\hat{\mu}) = [h \mapsto 0, l \mapsto 1]$ for the low view. After execution, the final SME memory will map h to $\langle 1 : 1 \rangle$. One way to reduce the number of executions is to exploit the knowledge that the high and the low view for variable l are equal, i.e., $H(\hat{\mu})(l) = L(\hat{\mu})(l)$. Since the semantics is deterministic, there is no need to execute the program twice. We can use this knowledge by specialising SME at the granularity of commands and include the following assignment rule:

$$\text{SME-}\textsc{optim}\ \frac{H(\hat{\mu})(e) = L(\hat{\mu})(e) \qquad (x := e, L(\hat{\mu})) \Downarrow \mu}{(x := e, \hat{\mu}) \Downarrow_{SME} \hat{\mu}[x \mapsto \langle \mu(x), \mu(x) \rangle]}$$

Notice that this SME optimization requires to *look inside* the shape of the program to evaluate if expression e of an assignment satisfies the hypothesis.

In general, in order to reduce the number of executions using the multi-execution technique of SME-TINI, it is sufficient to (i) identify in an SME memory which values in the array of values are equal and (ii) remember which values correspond to which views. MF uses the multi-execution technique, implements (i) and (ii) and hence, reduces the number of executions. MF encodes values in SME memories (arrays with as many positions as lattice elements) as ordered binary trees, where the order is given by the elements of the lattice. For example, for a SME memory where $\hat{\mu}(h) = \langle 1 : 0 : 0 : 0 \rangle$ for a lattice of 4 elements with top element $\top$, an equivalent MF memory encodes this array as $\langle \top ? 1 : 0 \rangle$ with the meaning that 1 is the view for $\top$ and 0 for the rest. Every execution that depends on that value, will multi execute twice instead of 4 times as in SME.

Moreover, MF further uses the view information provided by the encoding in order to multi execute less in case of branching commands. For example, for SME-TINI with SME memory $\hat{\mu}(h) = \langle 1 : 0 : 0 : 0 \rangle$ the program:

```
1: if h = 0 then
2:     h := h + 1
```

executes 4 times (where the assignment at line 2 executes 3 times).

Using the MF memory encoding $\hat{\mu}(h) = \langle \top ? 1 : 0 \rangle$, MF remembers that at line 2 there is no possible observation for the view $\top$ (because for view $\top$ the value of h is 1 so it doesn't take the then branch). Hence, the assignment $h := h + 1$ only executes once with a memory where h is 0 (the view of variable h corresponding to the 3 levels which are not $\top$).

For a program $h := l$, where $\hat{\mu}(l)$ is $\langle 1 : 1 \rangle$ in SME, MF keeps only the value 1: a single value represents the fact that all views can observe the same value. Thus the assignment $h := l$ executes once (and all future executions dependent on h will also be reduced).

Hence when encoding of an SME memory can be reduced effectively, multi executions are reduced accordingly. As shown in the following sections, preservation of MF memories encoding through execution requires: to represent arrays of values as trees called faceted values and to evaluate expressions depending on faceted values. In particular, the definition of the evaluation of expressions on faceted values depends highly on the shape of expressions and their values according to different views, and thus is contradictory to the blackbox property of a monitor.

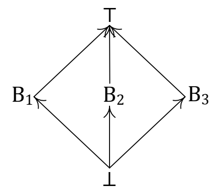

Figure 2: Lattice $\langle \mathcal{L}_{\mathbf{B}}, \sqsubseteq \rangle$

MF may run more multi executions than SME Original MF has one limitation with respect to SME: it was designed only for a security lattice of principals: for n principals, such a lattice contains 2^n security levels. The following Ad Exchange platform [35] example demonstrates that MF may be less efficient than SME in practice, when the security lattice is not based on principals.

Example 2.1. An Ad Exchange platform needs to put an advertisement on a publisher's website. For that, it implements a Real-time Bidding (RTB) system [36], where advertisers can bid for the space on the publisher's website to get their ad published. The system receives as input all the bid offers from bidders and sorts them. According to the RTB algorithm, the second best offer wins.

We present the lattice of 5 elements for this example in Fig. 2. For simplicity, we consider only 3 bidders called B_1, B_2, and B_3, an Ad Exchange ($\top$ level) which is able to see all the bids, and a public view $\perp$. Because MF is designed for a principal lattice, to encode 5 security levels, it uses 3 principals k_1, k_2, and k_3, and create a lattice of $8 = 2^3$ levels, and thus has a potential to run some parts of the program 8 times, while SME always executes the program 5 times.

We consider one test that naively checks the order of bid offers and decides the winner. The encoding of the lattice is: $\top = \{k_1, k_2, k_3\}$, $B_i = \{k_i\}$, and $\perp = \emptyset$.

1: *winner* := 0;
2: *test* := $(x_1 \leq x_2)$ and $(x_2 \leq x_3)$;
3: **if** *test* **then** *winner* := 2 **else skip**

The bid values from bidders are $x_1 = \langle k_1 \,?\, 10 : 0 \rangle$, $x_2 = \langle k_2 \,?\, 5 : 0 \rangle$, and $x_3 = \langle k_3 \,?\, 7 : 0 \rangle$. Thus, the resulting value of *test* at line 2 is

$$\langle k_1 \,?\, \langle k_2 \,?\, \langle k_3 \,?\, \mathrm{ff} : \mathrm{ff} \rangle : \langle k_3 \,?\, \mathrm{ff} : \mathrm{ff} \rangle \rangle : \langle k_2 \,?\, \langle k_3 \,?\, \mathrm{tt} : \mathrm{ff} \rangle : \langle k_3 \,?\, \mathrm{tt} : \mathrm{tt} \rangle \rangle \rangle.$$

Therefore, the original MF executes the if instruction 8 times with 3 useless executions for levels $\{k_1, k_2\}$, $\{k_2, k_3\}$, and $\{k_1, k_3\}$.

Moreover, because different views of a variable may contain the same values, MF may execute the same statement several times. For example, in the execution described above, original MF executes the then branch 3 times, while it only needs to run once since the threes executions for the then branch can be merged into one.

3 MF FOR ARBITRARY SECURITY LATTICE

We present an extension to the original Multiple Facets mechanism [3] for an arbitrary security lattice $\langle \mathcal{L}, \sqsubseteq \rangle$, which we call Generalised Multiple Facets mechanism, or *GMF*. Similarly to Multiple Facets, GMF operates over a *faceted memory* $\hat{\mu}$ that maps variables to simple values or faceted values. A *faceted value* is of the form $\langle l \,?\, V_1 : V_2 \rangle$ where $l \in \mathcal{L}$ is a security level, and V_i can be either a faceted value or a simple value. The first facet V_1 of $\langle l \,?\, V_1 : V_2 \rangle$ is called *private*, and visible to the observers at security level l or higher levels in the lattice; the second facet V_2 is called *public*, and visible to security levels that are lower or incomparable to l. We use V as a meta-variable for faceted values or simple values. Every evaluation in GMF (see Fig. 6) is marked with a set of security levels pc, for which the current computation is visible.

3.1 Expression evaluation

By $\hat{\mu}^{pc}(e)$ we denote the evaluation of expression e in faceted memory $\hat{\mu}$ with set of security levels pc. The definition of $\hat{\mu}^{pc}(e)$ is presented in Fig. 4. For example, consider the evaluation of x when the faceted value x in memory $\hat{\mu}$ is $\langle l \,?\, V_1 : V_2 \rangle$. To define which facet is useful given a pc, we consider the following cases:

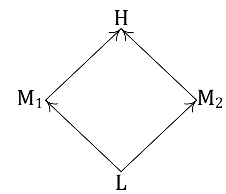

Figure 3: Lattice $\langle \mathcal{L}_{\diamond}, \sqsubseteq \rangle$

- All the levels in pc are greater than or equal to l, denoted $l \preccurlyeq pc$ (i.e. $\forall l' \in pc.\ l \sqsubseteq l'$): the evaluation can use the private facet V_1 because the public facet V_2 is anyway not useful for every level in this pc.
- All the levels in pc are lower than or incomparable to l, denoted $l \not\preccurlyeq pc$ (i.e. $\forall l' \in pc.\ l \not\sqsubseteq l'$): the evaluation can only use the public facet V_2 because V_2 is a facet visible to any view that is lower than or incomparable to l.
- Otherwise, we say that l and pc are *incomparable* and denote it by $l \,|||\, pc$ (i.e. $\exists l', l'' \in pc.\ l \sqsubseteq l' \wedge l \not\sqsubseteq l''$): we first evaluate V_1 with $pc_1 = \{l' \in pc \mid l \sqsubseteq l'\}$ – the set of all levels in pc which are greater than or equal to l. Then, we evaluate V_2 with $pc_2 = pc \setminus pc_1$ which is the set of all levels in pc which are lower than or incomparable to l. Finally, we combine the two results in a new faceted value.

To evaluate a variable x, we use a special unary operator $\ominus^{pc}(\hat{\mu}(x))$, which returns the value that is visible to all the levels in the pc. Let's consider the case of $\ominus^{pc}(\langle l \,?\, V_1 : V_2 \rangle)$. Notice that, if pc and l are incomparable, meaning that there are some levels in pc that are higher than or equal to l and other levels in pc that are lower than or incomparable to l, denoted by $l \,|||\, pc$, then the evaluation returns the faceted value $\langle l \,?\, \ominus^{pc_1}(V_1) : \ominus^{pc_2}(V_2) \rangle$. The form of the result of $\hat{\mu}^{pc}(e)$ is described in Lemma 3.1.

Lemma 3.1. *If $\hat{\mu}^{pc}(e) = \langle l \,?\, V_1 : V_2 \rangle$, then $l \,|||\, pc$.*

Example 3.2 (Expression evaluation). Consider the lattice $\langle \mathcal{L}_{\diamond}, \sqsubseteq \rangle$ from Fig. 3, and the evaluation of $x + y$ in $\hat{\mu}$, where $\hat{\mu}(x) = \langle M_1 \,?\, 10 : 0 \rangle$ and $\hat{\mu}(y) = \langle M_2 \,?\, 5 : 0 \rangle$.

Suppose that $pc = \{M_1, H\}$. Since all the levels in pc are higher than or equal to M_1, the evaluation of x returns $\hat{\mu}^{pc}(x) = 10$. Since pc and M_2 are incomparable, the evaluation of y returns $\hat{\mu}^{pc}(y) = \langle M_2 \,?\, 5 : 0 \rangle$. Next, the evaluation of $10 +^{pc} \langle M_2 \,?\, 5 : 0 \rangle$ is split into two: one uses a facet visible to M_2 (and hence H), and another one

$$
\begin{aligned}
\hat{\mu}^{pc}(v) &= v\\
\hat{\mu}^{pc}(x) &= \ominus^{pc}(\hat{\mu}(x))\\
\hat{\mu}^{pc}(e_1 \oplus e_2) &= \hat{\mu}^{pc}(e_1) \oplus^{pc} \hat{\mu}^{pc}(e_2)\\
\ominus^{pc}(v) &= v\\
\ominus^{pc}(\langle l\,?\,V_1 : V_2\rangle) &= \begin{cases} \ominus^{pc}(V_1) & \text{if } l \preccurlyeq pc\\ \ominus^{pc}(V_2) & \text{if } l \not\preccurlyeq pc\\ \langle l\,?\,\ominus^{pc_1}(V_1) : \ominus^{pc_2}(V_2)\rangle & \text{otherwise}\end{cases}\\
v_1 \oplus^{pc} v_2 &= v_1 \oplus v_2\\
v \oplus^{pc} \langle l\,?\,V_1 : V_2\rangle &= \begin{cases} v \oplus^{pc} V_1 & \text{if } l \preccurlyeq pc\\ v \oplus^{pc} V_2 & \text{if } l \not\preccurlyeq pc\\ \langle l\,?\,(v \oplus^{pc_1} V_1) : (v \oplus^{pc_2} V_2)\rangle & \text{otherwise}\end{cases}\\
\langle l\,?\,V_1 : V_2\rangle \oplus^{pc} V &= \begin{cases} V_1 \oplus^{pc} V & \text{if } l \preccurlyeq pc\\ V_2 \oplus^{pc} V & \text{if } l \not\preccurlyeq pc\\ \langle l\,?\,(V_1 \oplus^{pc_1} V) : (V_2 \oplus^{pc_2} V)\rangle & \text{otherwise}\end{cases}
\end{aligned}
$$

where $pc_1 = \{l' \in pc \mid l \sqsubseteq l'\}$ and $pc_2 = pc \setminus pc_1$.

Figure 4: Expression evaluation

$$
\mu \uparrow_{\Gamma}^{def} (x) = \begin{cases} \mu(x) & \text{if } \Gamma(x) = \text{glb}(\mathcal{L}),\\ \langle \Gamma(x)\,?\,\mu(x) : def(x)\rangle & \text{otherwise.}\end{cases}
$$
$$
\hat{\mu}|_{\Gamma}(x) = l(\hat{\mu})(x) \text{ where } l = \Gamma(x)
$$

Figure 5: Functions for faceted and normal memories.

uses a public facet that will be visible to M_1.

$$
\begin{aligned}
\hat{\mu}^{pc}(x+y) &= \hat{\mu}^{pc}(x) +^{pc} \hat{\mu}^{pc}(y)\\
&= \ominus^{pc}(\langle M_1\,?\,10 : 0\rangle) +^{pc} \ominus^{pc}(\langle M_2\,?\,5 : 0\rangle)\\
&= 10 +^{pc} \langle M_2\,?\,5 : 0\rangle = \langle M_2\,?\,10 +^{\{H\}} 5 : 10 +^{\{M_1\}} 0\rangle\\
&= \langle M_2\,?\,15 : 10\rangle
\end{aligned}
$$

3.2 Semantics

We abuse the notation and use l as a *projection function* on simple values, faceted values and faceted memories. For any V, $l(V)$ returns the value in V which is visible to users at level l. For any $\hat{\mu}$, $l(\hat{\mu})$ returns the memory in $\hat{\mu}$ which is visible to users at level l.

$$
l(v) = v \qquad l(\langle l_1\,?\,V_1 : V_2\rangle) = \begin{cases} l(V_1) & \text{if } l_1 \sqsubseteq l,\\ l(V_2) & \text{otherwise.}\end{cases}
$$
$$
l(\hat{\mu})(x) = l(\hat{\mu}(x))
$$

The projection function l is used in the definition of $\hat{\mu}|_{\Gamma}$ function that converts a faceted memory to a simple memory (see Fig. 5).

The semantics of GMF is defined in Fig. 6 as a big-step evaluation relation $\Gamma \vdash (P,\mu) \Downarrow_{GMF} \mu'$, where program P is executed in a memory μ and a security environment Γ that maps variables to security levels in a given security lattice $\langle \mathcal{L}, \sqsubseteq\rangle$.

The main rule GMF first constructs a faceted memory from the standard memory using the transformation $\mu \uparrow_{\Gamma}^{def}$ from Fig. 5, where glb($\mathcal{L}$) is the greatest lower bound of $\mathcal{L}$. The resulting faceted memory keeps original value of each variable x in a private facet, and adds default values (defined by *def* function) in a public facet. In a special case when the level of x is the smallest level in a lattice, we keep only a simple value $\mu(x)$ that is visible to all security levels. We then evaluate the program with the constructed faceted memory and $pc = \mathcal{L}$. The resulting faceted memory is transformed back to a normal memory by using the projection function $\hat{\mu}|_{\Gamma}$.

$$
\text{GMF}\ \boxed{\dfrac{(P, \mu \uparrow_{\Gamma}^{def}) \downarrow_G^{\mathcal{L}} \hat{\mu}'}{\Gamma \vdash (P,\mu) \Downarrow_{GMF} \hat{\mu}'|_{\Gamma}}}
$$

$$
\text{GSkip}\ \dfrac{}{(\textbf{skip}, \hat{\mu}) \downarrow_G^{pc} \hat{\mu}} \qquad \text{GAssign}\ \dfrac{}{(x := e, \hat{\mu}) \downarrow_G^{pc} \hat{\mu}[x \mapsto \hat{\mu}^{pc}(e)]}
$$

$$
\text{GSeq}\ \dfrac{(P_1, \hat{\mu}) \downarrow_G^{pc} \hat{\mu}' \qquad (P_2, \hat{\mu}') \downarrow_G^{pc} \hat{\mu}''}{(P_1; P_2, \hat{\mu}) \downarrow_G^{pc} \hat{\mu}''}
$$

$$
\text{GIf-C}\ \dfrac{\hat{\mu}^{pc}(e) = v \qquad (P_v, \hat{\mu}) \downarrow_G^{pc} \hat{\mu}'}{(\textbf{if } e \textbf{ then } P_{\mathrm{tt}} \textbf{ else } P_{\mathrm{ff}}, \hat{\mu}) \downarrow_G^{pc} \hat{\mu}'}
$$

$$
\text{GIf-S}\ \dfrac{\begin{array}{c}\hat{\mu}^{pc}(e) = \langle l\,?\,V_1 : V_2\rangle \qquad pc_1 = \{l' \in pc \mid l \sqsubseteq l'\}\\ pc_2 = pc \setminus pc_1 \qquad \hat{\mu}_1 = \hat{\mu} \uplus (y \mapsto V_1) \qquad \hat{\mu}_2 = \hat{\mu} \uplus (y \mapsto V_2)\\ P' = \textbf{if } y \textbf{ then } P_1 \textbf{ else } P_2 \qquad (P', \hat{\mu}_1) \downarrow_G^{pc_1} \hat{\mu}_1' \qquad (P', \hat{\mu}_2) \downarrow_G^{pc_2} \hat{\mu}_2'\end{array}}{(\textbf{if } e \textbf{ then } P_1 \textbf{ else } P_2, \hat{\mu}) \downarrow_G^{pc} (\hat{\mu}_1' \setminus\!\!\setminus y) \otimes^l (\hat{\mu}_2' \setminus\!\!\setminus y)}
$$

$$
\text{GWhile}\ \dfrac{(\textbf{if } e \textbf{ then } P; \textbf{while } e \textbf{ do } P \textbf{ else skip}, \hat{\mu}) \downarrow_G^{pc} \hat{\mu}'}{(\textbf{while } e \textbf{ do } P, \hat{\mu}) \downarrow_G^{pc} \hat{\mu}'}
$$

where $\hat{\mu}_1 \otimes^l \hat{\mu}_2(x) = [\![\langle l\,?\,\hat{\mu}_1(x) : \hat{\mu}_2(x)\rangle]\!]$

Figure 6: Multiple facets for arbitrary security lattice

The semantics rules for skip, sequence and while loop are straightforward. The GAssign rule uses a faceted evaluation $\hat{\mu}^{pc}(e)$ defined in Section 3.1.

Before describing the semantics of if instruction, we first define several auxiliary functions. Let $dom(\hat{\mu})$ be the domain of $\hat{\mu}$ and y be a fresh variable, i.e. $y \notin \text{dom}(\hat{\mu})$). By $\hat{\mu} \uplus (y \mapsto V)$ we denote a new memory $\hat{\mu}'$, such that $\text{dom}(\hat{\mu}') = \text{dom}(\hat{\mu}) \cup \{y\}$, $\hat{\mu}'(y) = V$ and for all $x \in \text{dom}(\hat{\mu})$, $\hat{\mu}'(x) = \hat{\mu}(x)$. By $\hat{\mu} \setminus\!\!\setminus y$, we remove y from the domain of $\hat{\mu}$, that is, $\hat{\mu} \setminus\!\!\setminus y$ constructs a new memory $\hat{\mu}'$, where $\text{dom}(\hat{\mu}') = \text{dom}(\hat{\mu}) \setminus \{y\}$ and for all $x \neq y$, $\hat{\mu}(x) = \hat{\mu}'(x)$.

Consider the evaluation of the if instruction **if** e **then** P_1 **else** P_2 with $\hat{\mu}$ and pc. If e is evaluated to a constant value (tt or ff), then only P_{tt} or P_{ff} is evaluated (see rule GIf-C).

When e is evaluated to a faceted value $\langle l\,?\,V_1 : V_2\rangle$, we construct a new program **if** y **then** P_1 **else** P_2, where y is a fresh variable. From Lemma 3.1, we have that $l \parallel\!| pc$, and hence $pc_1 = \{l' \in pc \mid l \sqsubseteq l'\}$ and $pc_2 = pc \setminus pc_1$ are non-empty. In this case, we run the new program **if** y **then** P_1 **else** P_2 twice: once with the "higher view" than l, i.e., with $pc_1 = \{l' \in pc \mid l \sqsubseteq l'\}$ and y set to a private facet V_1, and another time with "lower or incomparable view" than l, i.e. with $pc_2 = pc \setminus pc_1$ and y set to a public facet V_2. We then combine the resulting memories using the $\otimes^l$ operator. The combination of faceted memories is based on the fact that when pc is split into pc_1 and pc_2 in the GIf-S rule, all levels in pc_1 is larger than or equal to l, and all levels in pc_2 is smaller than or incomparable to l.

$$[\![v]\!] = v$$

$$[\![\langle l \mathbin{?} V_1 : V_2\rangle]\!] = \begin{cases} [\![V]\!] & \text{if } V_1 = V_2,\\ [\![\langle l \mathbin{?} V_{11} : V_{22}\rangle]\!] & \text{elseif } l_1 \sqsubseteq l,\ l \sqsubseteq l_2,\ V_1 = \langle l_1 \mathbin{?} V_{11} : V_{12}\rangle,\\ & \qquad V_2 = \langle l_2 \mathbin{?} V_{21} : V_{22}\rangle,\\ [\![\langle l \mathbin{?} V_{11} : V_2\rangle]\!] & \text{elseif } l_1 \sqsubseteq l,\ V_1 = \langle l_1 \mathbin{?} V_{11} : V_{12}\rangle,\\ [\![\langle l \mathbin{?} V_1 : V_{22}\rangle]\!] & \text{elseif } l \sqsubseteq l_2,\ V_2 = \langle l_2 \mathbin{?} V_{21} : V_{22}\rangle,\\ \langle l \mathbin{?} [\![V_1]\!] : [\![V_2]\!]\rangle & \text{otherwise.} \end{cases}$$

Figure 7: Optimisation of a faceted value.

Notice that the form of a faceted value constructed by combining values can be reduced. For example, a faceted value of the form $\langle H \mathbin{?} \langle M_1 \mathbin{?} V_{11} : V_{12}\rangle : V_2\rangle$ can be reduced to $\langle H \mathbin{?} V_{11} : V_2\rangle$ because $M_1 \sqsubseteq H$ and the projection of the original value at any level is either V_{11} or V_2. We use the optimisation on the constructed faceted values from Fig. 7.

Therefore, in the GIf-S rule after the evaluation of P' in two contexts, we combine the resulting faceted memories $\hat{\mu}'_1 \backslash y$ and $\hat{\mu}'_2 \backslash y$ and apply an optimisation operator $[\![]\!]$ for each newly constructed faceted value. The correctness of $[\![]\!]$ used to optimize faceted values is proven in Lemma 3.3.

Lemma 3.3. *For all l, all V, it follows that $l(V) = l([\![V]\!])$.*

Example 3.4 (Evaluation of if instruction). Consider the security lattice $\langle \mathcal{L}_\diamond, \sqsubseteq\rangle$ from Fig. 3 and the evaluation of the following program P with $pc = \mathcal{L}_\diamond$ and $\hat{\mu}$, where $\hat{\mu}(x) = \langle M_1 \mathbin{?} \langle H \mathbin{?} \mathtt{tt} : \mathtt{ff}\rangle : \mathtt{tt}\rangle$.

1: **if** x **then** $z := 10$ **else** $z := 5$

The evaluation follows the GIf-S rule since $M_1 ||| \mathcal{L}_\diamond$. We construct $P' =$ **if** y_1 **then** P_1 **else** P_2 and first evaluate P' with $pc_1 = \{l' \in pc \mid M_1 \sqsubseteq l'\} = \{M_1, H\}$ and $\hat{\mu}_1 = \hat{\mu} \uplus (y \mapsto \langle H \mathbin{?} \mathtt{tt} : \mathtt{ff}\rangle)$, and then evaluate P' with $pc_2 = pc \setminus pc_1 = \{M_2, L\}$, $\hat{\mu}_2 = \hat{\mu} \uplus (y \mapsto \mathtt{tt})$.

Since $pc_1 = \{H, M_1\}$ and $\hat{\mu}^{pc}(y) = \langle H \mathbin{?} \mathtt{tt} : \mathtt{ff}\rangle$, the evaluation of P' with pc_1 and $\hat{\mu}_1$ is split again to two evaluations: one with $P'' =$ **if** t **then** P_1 **else** P_2, $pc_{11} = \{H\}$, and $\hat{\mu}_{11} = \hat{\mu}_1 \uplus (t \mapsto \mathtt{tt})$; and the other one with P'', $pc_{12} = \{M_1\}$, and $\hat{\mu}_{12} = \hat{\mu}_1 \uplus (t \mapsto \mathtt{ff})$.

The evaluation of P'' with pc_{11} and with pc_{12} follow the GIf-C rule and we get two faceted memories $\hat{\mu}'_{11}$ and $\hat{\mu}'_{12}$, where $\hat{\mu}'_{11}(z) = 10$ and $\hat{\mu}'_{12}(z) = 5$. Then, $\hat{\mu}'_{11} \backslash\!\backslash t$ and $\hat{\mu}'_{12} \backslash\!\backslash t$ are combined and we get $\hat{\mu}'_1$, where $\hat{\mu}'_1(z) = \langle H \mathbin{?} 10 : 5\rangle$.

The evaluation of P_2 with pc_2 follows the GIf-C rule and the result is $\hat{\mu}'_2$, where $\hat{\mu}'_2(z) = 10$. At this point, $\hat{\mu}'_1 \backslash\!\backslash y_1$ and $\hat{\mu}'_2 \backslash\!\backslash y_1$ are combined and the result is $\hat{\mu}'$, where $\hat{\mu}'(z) = \langle M_1 \mathbin{?} \langle H \mathbin{?} 10 : 5\rangle : 10\rangle$.

Example 3.5 (Evaluation with the GMF rule). Consider the lattice $\langle \mathcal{L}_\diamond, \sqsubseteq\rangle$ from Fig. 3 and program P from Example 3.4 with one more instruction $x := x_1 > x_2$. Suppose that $\Gamma(x_1) = M_1$, $\Gamma(x_2) = H$, $\Gamma(z) = H$, $\mu(x_1) = 10$, $\mu(x_2) = 5$, the default values for x_1 and x_2 are respectively 100 and 20 [2]. Let $\hat{\mu} = \mu \uparrow^{def}_{\Gamma}$. It follows that $\hat{\mu}(x_1) = \langle M_1 \mathbin{?} 10 : 100\rangle$ and $\hat{\mu}(x_2) = \langle H \mathbin{?} 5 : 20\rangle$.

1: $x := x_1 > x_2$
2: **if** x **then** $z := 10$ **else** $z := 5$

[2]The values and default values for x_1 and x_2 are chosen so that the value of x after the evaluation of the assignment instruction is $\langle M_1 \mathbin{?} \langle H \mathbin{?} \mathtt{tt} : \mathtt{ff}\rangle : \mathtt{tt}\rangle$.

Following GMF rule, the program is evaluated with $pc = \mathcal{L}_\diamond = \{H, M_1, M_2, L\}$. For the assignment instruction, the value of x is updated to $\hat{\mu}^{pc}(x_1 > x_2) = \langle M_1 \mathbin{?} \langle H \mathbin{?} \mathtt{tt} : \mathtt{ff}\rangle : \mathtt{tt}\rangle$. The rest of the evaluation is described in Example 3.4, and the resultant faceted memory is $\hat{\mu}'$, where $\hat{\mu}'(z) = \langle M_1 \mathbin{?} \langle H \mathbin{?} 10 : 5\rangle : 10\rangle$.

The memory after the application of rule GMF is $\mu' = \hat{\mu}'|_\Gamma$. Since $\Gamma(z) = H$, the value of z is $\mu'(z) = H(\langle M_1 \mathbin{?} \langle H \mathbin{?} 10 : 5\rangle : 10\rangle) = 10$.

3.3 Equivalence to SME-TINI and Security Guarantee

SME-TINI. The semantics of SME-TINI, termination-insensitive version of SME, for an arbitrary security lattice is presented below, where $\vec{\mu}$ is a vector that maps levels to normal memories; $\mu \uplus^l \Gamma$ constructs a memory where values of variables at levels that are not visible to l are replaced by default values; $\odot_\Gamma(\vec{\mu})(x) \triangleq \vec{\mu}[\Gamma(x)](x)$ constructs a memory by combining all memories in $\vec{\mu}$; and def is a function mapping variables to default values.

$$\text{SME-TINI}\ \frac{\forall l \in \mathcal{L} : (P, \mu \uplus^l \Gamma) \Downarrow \vec{\mu}[l]}{\Gamma \vdash (P, \mu) \Downarrow_{SME-TINI} \odot_\Gamma(\vec{\mu})}$$

$$\mu \uplus^l \Gamma \triangleq \begin{cases} def(x) & \text{if } \Gamma(x) \not\sqsubseteq l,\\ \mu(x) & \text{if } \Gamma(x) \sqsubseteq l. \end{cases}$$

We now prove that SME-TINI enforces *termination-insensitive noninterference (TINI)*. Two memories μ and μ' are *equivalent at l w.r.t.* Γ (denoted by $\mu =^\Gamma_l \mu'$) iff for all x, $\Gamma(x) \sqsubseteq l \implies \mu(x) = \mu'(x)$. When Γ is clear from the context, $\mu =^\Gamma_l \mu'$ is written as $\mu =_l \mu'$.

Definition 3.6 (TINI). An enforcement mechanism A is *termination insensitive non-interferent (TINI)* if for all security environments Γ, programs P, and memories μ_1, and μ_2, we have

$$\mu_1 =_l \mu_2 \ \wedge\ \Gamma \vdash (P, \mu_1) \Downarrow_A \mu'_1 \ \wedge \Gamma \vdash (P, \mu_2) \Downarrow_A \mu'_2 \implies \mu'_1 =_l \mu'_2.$$

Theorem 3.7. *SME-TINI is TINI.*

Equivalence to SME-TINI. To prove the equivalence between GMF and SME-TINI, we formally define the semantic equivalence of two mechanisms.

Definition 3.8. Two enforcement mechanisms A and B are equivalent if for any Γ, P and μ, we have that $\Gamma \vdash (P, \mu) \Downarrow_A \mu'$ iff $\Gamma \vdash (P, \mu) \Downarrow_B \mu'$.

We next establish the relation between the execution with GMF semantics and the execution with the standard semantics.

Lemma 3.9. *$(P, \hat{\mu}) \Downarrow^{pc}_G \hat{\mu}'$ iff $(P, l(\hat{\mu})) \Downarrow l(\hat{\mu}')$ for all $l \in pc$.*

Thanks to Lemma 3.9, we now prove the equivalence of GMF and SME-TINI.

Theorem 3.10. *GMF and SME-TINI are equivalent.*

As a consequence, we have that GMF is TINI.

Remark 3.1. *MF [3] is constructed for a set of principals. When the set $\mathbf{P}$ of principals is fixed, we can use GMF to encode MF: we construct the lattice $\langle 2^{\mathbf{P}}, \subseteq\rangle$, where each element is a set of principals; we prove that GMF for $\langle 2^{\mathbf{P}}, \subseteq\rangle$ and MF for $\mathbf{P}$ are equivalent [23].*

4 OPTIMIZING GMF

In Section 3, we presented the semantics of Generalised Multiple Facets (GMF) for arbitrary lattice and have proven it to be equivalent to SME-TINI. However, GMF from Fig. 6 can be further optimised and avoid repeating evaluations of the same commands. The following example demonstrates the sub-optimality of GMF.

Example 4.1 (GMF is not optimal). We consider the below program from Example 2.1. The lattice is $\langle \mathcal{L}_{\mathrm{B}}, \sqsubseteq \rangle$ from Fig. 2.

1: $winner := 0$;
2: $test := (x_1 \leq x_2)$ and $(x_2 \leq x_3)$;
3: **if** $test$ **then** $winner := 2$ **else skip**

Suppose that the bid offers of B_1, B_2, and B_3 are respectively 10, 5, and 7, and the default values for B_i are 0. W.r.t. this setting, the initial faceted memory is $\hat{\mu}$, where $\hat{\mu}(x_1) = \langle B_1 \,?\, 10 : 0 \rangle$, $\hat{\mu}(x_2) = \langle B_2 \,?\, 5 : 0 \rangle$, and $\hat{\mu}(x_3) = \langle B_3 \,?\, 7 : 0 \rangle$. We consider the execution of the program with GMF.

After line 2, $test = \langle B_1 \,?\, \langle B_2 \,?\, \mathrm{ff} : \mathrm{ff} \rangle : \langle B_2 \,?\, \mathrm{ff} : \langle B_3 \,?\, \mathrm{tt} : \mathrm{tt} \rangle \rangle \rangle$. Following the semantics of GMF, the assignment instruction $winner :=$ 2 is evaluated twice with $pc_{B3} = \{B_3\}$, and $pc_{\perp} = \{\perp\}$; the **skip** instruction is evaluated three times with $pc_{\top} = \{\top\}$, $pc_{B1} = \{B_1\}$, and $pc_{B2} = \{B_2\}$.

The main idea of our optimisation lays in reducing the number of sub-evaluations and hence the number of faceted memory combinations. For Example 4.1, we propose a mechanism that merges the evaluations corresponding to pc_{B3} and $pc_{\perp}$ into one evaluation with $pc_1 = \{B_3, \perp\}$. This simplification is possible since $test$ denotes the same value (i.e., tt) under pc_{B3} and $pc_{\perp}$. Similarly, our simplification merges the evaluations corresponding to $pc_{\top}$, pc_{B1}, and pc_{B2}, where $test$ denotes ff, into one evaluation with $pc_2 = \{\top, B_1, B_2\}$, and thus evaluates each branch of the if command only once.

In this section, we propose semantics of *optimized GMF* (OGMF) that reduces the number of sub-evaluations, and hence is more resource-friendly than GMF.

4.1 Semantics

The ideas behind the OGMF rule, and the rules for skip, assignment, sequence, and while instructions are similar to the corresponding ones of GMF. The functions $\mu \uparrow^{def}_{\Gamma} (x)$ and $\hat{\mu}|_{\Gamma}(x)$ are defined in Fig. 5. We now explain the semantic rules for the conditional instruction.

Consider evaluation of the program **if** e **then** P_1 **else** P_2 with pc and memory $\hat{\mu}$, and $\hat{\mu}^{pc}(e) = V$. In order to evaluate each branch of the conditional only once, we split the pc in two subsets: in the first subset pc_1 the visible value of V is true, and in the remaining subset pc_2, V is false. We now have three distinct cases.

If $pc_1 = pc$, meaning that for all levels in pc, the visible value of V is true, then P_1 is evaluated (rule OIf-T). If $pc_2 = pc$, then for all levels in pc, the visible value of V is false, and only P_2 is evaluated (rule OIf-F). Finally, when pc is split in non-empty pc_1 and pc_2, then both P_1 and P_2 are evaluated, and their results ($\hat{\mu}'_1$ and $\hat{\mu}'_2$) are combined by $\hat{\mu}'_1 \oplus^{pc_1,pc_2} \hat{\mu}'_2$ (rule OIf-S) to a new faceted memory. The intuition behind this combination is that the projection of $\hat{\mu}'_1 \oplus^{pc_1,pc_2} \hat{\mu}'_2$ at $l \in pc_1$ is taken from the evaluation of P_1 and its projection at $l \in pc_2$ is taken from the evaluation of P_2.

$$\text{OGMF}\ \boxed{\dfrac{(P, \mu \uparrow^{def}_{\Gamma}) \downarrow^{\mathcal{L}}_{O} \hat{\mu}'}{\Gamma \vdash (P, \mu) \Downarrow_{OGMF} \hat{\mu}'|_{\Gamma}}}$$

$$\text{OAssign}\ \frac{}{(x := e, \hat{\mu}) \downarrow^{pc}_{O} \hat{\mu}[x \mapsto \hat{\mu}^{pc}(e)]} \qquad \text{OSkip}\ \frac{}{(\textbf{skip}, \hat{\mu}) \downarrow^{pc}_{O} \hat{\mu}}$$

$$\text{OSeq}\ \frac{(P_1, \hat{\mu}) \downarrow^{pc}_{O} \hat{\mu}' \qquad (P_2, \hat{\mu}') \downarrow^{pc}_{O} \hat{\mu}''}{(P_1; P_2, \hat{\mu}) \downarrow^{pc}_{O} \hat{\mu}''}$$

$$\text{OIf-T}\ \frac{\begin{array}{c}\hat{\mu}^{pc}(e) = V \\ pc_1 = \{l \in pc | l(V) = \mathrm{tt}\} \qquad pc_1 = pc \qquad (P_1, \hat{\mu}) \downarrow^{pc}_{O} \hat{\mu}'\end{array}}{(\textbf{if } e \textbf{ then } P_1 \textbf{ else } P_2, \hat{\mu}) \downarrow^{pc}_{O} \hat{\mu}'}$$

$$\text{OIf-F}\ \frac{\begin{array}{c}\hat{\mu}^{pc}(e) = V \qquad pc_1 = \{l \in pc | l(V) = \mathrm{tt}\} \\ pc_2 = pc \setminus pc_1 \qquad pc_2 = pc \qquad (P_2, \hat{\mu}) \downarrow^{pc}_{O} \hat{\mu}'\end{array}}{(\textbf{if } e \textbf{ then } P_1 \textbf{ else } P_2, \hat{\mu}) \downarrow^{pc}_{O} \hat{\mu}'}$$

$$\text{OIf-S}\ \frac{\begin{array}{c}\hat{\mu}^{pc}(e) = V \qquad pc_1 = \{l \in pc | l(V) = \mathrm{tt}\} \qquad pc_2 = pc \setminus pc_1 \\ pc_1 \neq \emptyset \qquad pc_2 \neq \emptyset \qquad (P_1, \hat{\mu}) \downarrow^{pc_1}_{O} \hat{\mu}'_1 \qquad (P_2, \hat{\mu}) \downarrow^{pc_2}_{O} \hat{\mu}'_2\end{array}}{(\textbf{if } e \textbf{ then } P_1 \textbf{ else } P_2, \hat{\mu}) \downarrow^{pc}_{O} \hat{\mu}'_1 \oplus^{pc_1,pc_2} \hat{\mu}'_2}$$

$$\text{OWhile}\ \frac{P' = \textbf{if } e \textbf{ then } P; \textbf{while } e \textbf{ do } P \textbf{ else skip} \qquad (P', \hat{\mu}) \downarrow^{pc}_{O} \hat{\mu}'}{(\textbf{while } e \textbf{ do } P, \hat{\mu}) \downarrow^{pc}_{O} \hat{\mu}'}$$

$$(\hat{\mu}'_1 \oplus^{pc_1,pc_2} \hat{\mu}'_2)(x) = \begin{cases} [\![\hat{\mu}'_1(x)]\!] & \text{if } \hat{\mu}'_1(x) = \hat{\mu}'_2(x), \\ [\![\mathbb{F}(\hat{\mu}'_1(x), \hat{\mu}'_2(x), pc_1, pc_2), pc_1 \cup pc_2]\!] & \text{otherwise.} \end{cases}$$

Figure 8: Optimized multiple facets for arbitrary lattice

In the definition of combination of memories for OGMF (bottom of Fig. 8), we distinguish two cases. If for some variable x, its value in both faceted memories is the same, ($\hat{\mu}'_1(x) = \hat{\mu}'_2(x)$), then we do not need to construct a new faceted value. Instead, we optimize the current value using the optimisation operator from Fig. 7.

If the values of x in $\hat{\mu}'_1(x)$ and $\hat{\mu}'_2(x)$ are different, then we construct a new faceted value $V = \mathbb{F}(V_1, V_2, pc_1, pc_2)$ and apply further optimisation on the resulting value V using a new optimisation operator that takes into account a faceted value and the current pc: $[\![V, pc]\!]$ optimizes the form of V and is described in Fig. 9. We show an example of such optimisation in Example 4.4.

To combine two faceted memories, we first construct a new faceted value by using $\mathbb{F}(V_1, V_2, pc_1, pc_2)$:

$$\mathbb{F}(V_1, V_2, pc_1, pc_2) = \langle\!\langle List(pc_1 \cup pc_2), V_1, V_2, pc_1, pc_2 \rangle\!\rangle$$

where $List(S)$ is a list of security levels from a set S, such that if l appears before l' in $List(S)$ then $l \not\sqsubseteq l'$. If the relation $\sqsubseteq$ in a given security lattice is not a total order, we can transform it into a total order $\sqsubseteq_T$ provided that $\sqsubseteq$ is a finite partial order. We can then view $List(S)$ as a list such that for any l and l' in this list, if l appears before l', then $l' \sqsubseteq_T l$.

The definition of $\mathbb{F}(V_1, V_2, pc_1, pc_2)$ uses the following operator that creates a faceted value based on an ordered list of security levels L, two faceted values, pc_1 and pc_2:

$$\langle\!\langle L, V_1, V_2, pc_1, pc_2 \rangle\!\rangle = \begin{cases} l(V_1) & \text{if } L = l, l \in pc_1, \\ l(V_2) & \text{if } L = l, l \in pc_2, \\ \langle l\, ?\, l(V_1) : \langle\!\langle T, V_1, V_2, pc_1, pc_2 \rangle\!\rangle \rangle & \\ \quad \text{if } L = l.T, T \neq [], l \in pc_1, & \\ \langle l\, ?\, l(V_2) : \langle\!\langle T, V_1, V_2, pc_1, pc_2 \rangle\!\rangle \rangle & \\ \quad \text{if } L = l.T, T \neq [], l \in pc_2. & \end{cases}$$

Notice that the form of the faceted value created by $\mathbb{F}(V_1, V_2, pc_1, pc_2)$ may be suboptimal.

Example 4.2 (Faceted value construction). Suppose that $V_1 = 2$, $V_2 = 0$, $pc_1 = \{B_3, \bot\}$, $pc_2 = \{\top, B_1, B_2\}$, $List(pc_1 \cup pc_2)$ is $\top.B_1.B_2.B_3.\bot$, and the lattice $\langle \mathcal{L}_B, \sqsubseteq \rangle$ is from Fig. 2.

Following the definition of combination of faceted memories, we have $\mathbb{F}(2, 0, pc_1, pc_2) = \langle \top\, ?\, 0 : \langle B_1\, ?\, 0 : \langle B_2\, ?\, 0 : \langle B_3\, ?\, 2 : 2 \rangle\rangle\rangle\rangle$. This value can be further reduced to $\langle B_1\, ?\, 0 : \langle B_2\, ?\, 0 : 2 \rangle\rangle$.

We therefore define an optimisation function $[\![V, pc]\!]$ that further optimises the result V of a $\mathbb{F}()$ function. The optimisation uses the observation that faceted value returned by $\mathbb{F}()$ has the form of $\langle l\, ?\, v : V' \rangle$, where V' is either a simple value or a faceted value [3].

The function $[\![V, pc]\!]$ is defined in Fig. 9. If V is of the form $\langle l\, ?\, v : v' \rangle$, then the optimisation is straightforward. We now consider the case when V is of the form $\langle l\, ?\, v : \langle l'\, ?\, v' : V' \rangle\rangle$. For demonstration, consider the lattice $\langle \mathcal{L}_B, \sqsubseteq \rangle$ from Fig. 2.

If the faceted value V is of the form $\langle \top\, ?\, v : \langle B_1\, ?\, v : V' \rangle\rangle$ (formally, $l' \sqsubseteq l$ and $v = v'$), then it can be reduced to $[\![\langle B_1\, ?\, v : V' \rangle, pc']\!]$ (formally, $[\![\langle l'\, ?\, v' : V' \rangle, pc']\!]$), where $pc' = pc \setminus \{\top\}$.

If the faceted value V is of the form $\langle B_1\, ?\, v : \langle B_2\, ?\, v : V' \rangle\rangle$, ($l$ and l' are incomparable and $v = v'$), and moreover for all the levels in the pc, for which either B_1 or B_2 is visible, it is guaranteed that they observe the same value v (see the definition of $cond(V, pc)$ below), then we distinguish the following two cases.

$$cond(V, pc) \triangleq V = \langle l\, ?\, v : \langle l'\, ?\, v' : V' \rangle\rangle \land$$
$$\forall l_1 \in pc : \text{glb}(l, l') \sqsubseteq l_1 \implies l_1(V) = v.$$

- If all levels in pc are greater than or equal to $\text{glb}(l, l')$ (i.e. $\text{glb}(l, l') \preccurlyeq pc$), then V is reduced to v. For example, if $pc = \{B_1, B_2, B_3\}$, $\text{glb}(B_1, B_2) = \bot$, then $\text{glb}(B_1, B_2) \preccurlyeq pc$, and thanks to the $cond(V, pc)$ we know that $B_1(V) = B_2(V) = B_3(V) = v$, then we can reduce such faceted value to simply v because value V' is not useful for such pc.
- If only some levels in pc are greater than or equal to $\text{glb}(l, l')$ (i.e. $\text{glb}(l, l') \,|||\, pc$), then V is reduced to $\langle \text{glb}(l, l')\, ?\, v : V'' \rangle$ and this value is reduced further recursively. Consider that we add one more security level L to the lattice $\langle \mathcal{L}_B, \sqsubseteq \rangle$ such that $L \sqsubseteq \bot$. If $pc = \{B_1, B_2, L\}$, $\text{glb}(B_1, B_2) = \bot$, then $\text{glb}(B_1, B_2) \,|||\, pc$ because $\bot \not\sqsubseteq L$. We then construct a set of security levels S from pc, which are higher or equal than $\text{glb}(l, l')$, and therefore the view on V from all these levels is v (because $cond(V, pc)$ holds). In our example, $S = \{B_1, B_2\}$, and we construct a new faceted value $V'' = \langle\!\langle \{L\}, V' \rangle\!\rangle = L(V')$. We then define a new $pc' = (pc \setminus S) \cup \{\text{glb}(l, l')\} = \{L, \bot\}$, and we need to keep $\text{glb}(l, l')$ in pc' because we must ensure that all the levels present in the new faceted value are also present in pc. Therefore, the reduced faceted value for our example is $[\![\langle \bot\, ?\, v : L(V') \rangle, \{\bot, L\}]\!]$.

Finally, if none of the above conditions hold then we recursively reduce the facet $\langle l'\, ?\, v' : V' \rangle$.

The correctness of $\hat{\mu}_1 \oplus^{pc_1, pc_2} \hat{\mu}_2$ in the OIf-S rule is proven in Lemma 4.3.

LEMMA 4.3. *For all levels l, variables x, sets of security levels pc_1 and pc_2, and memories $\hat{\mu}_1$ and $\hat{\mu}_2$,*

- *if $l \in pc_1$, then $l(\hat{\mu}_1 \oplus^{pc_1, pc_2} \hat{\mu}_2)(x) = l(\hat{\mu}_1)(x)$,*
- *if $l \in pc_2$, then $l(\hat{\mu}_1 \oplus^{pc_1, pc_2} \hat{\mu}_2)(x) = l(\hat{\mu}_2)(x)$.*

Example 4.4 (Optimisation of faceted value). Consider a faceted value $\langle \top\, ?\, 0 : \langle B_1\, ?\, 0 : \langle B_2\, ?\, 0 : \langle B_3\, ?\, 2 : 2 \rangle\rangle\rangle\rangle$ and $pc = \{\top, B_1, B_2, B_3, \bot\}$ from Example 4.2. We show how this value is optimised with our optimisation function $[\![,]\!]$:

$$\begin{aligned} & [\![\langle \top\, ?\, 0 : \langle B_1\, ?\, 0 : \langle B_2\, ?\, 0 : \langle B_3\, ?\, 2 : 2 \rangle\rangle\rangle\rangle, \{\top, B_1, B_2, B_3, \bot\}]\!] = \\ & = [\![\langle B_1\, ?\, 0 : \langle B_2\, ?\, 0 : \langle B_3\, ?\, 2 : 2 \rangle\rangle\rangle, \{B_1, B_2, B_3, \bot\}]\!] = \\ & = \langle B_1\, ?\, 0 : [\![\langle B_2\, ?\, 0 : \langle B_3\, ?\, 2 : 2 \rangle\rangle, \{B_2, B_3, \bot\}]\!] \rangle = \\ & = \langle B_1\, ?\, 0 : \langle B_2\, ?\, 0 : [\![\langle B_3\, ?\, 2 : 2 \rangle, \{B_3, \bot\}]\!] \rangle\rangle = \langle B_1\, ?\, 0 : \langle B_2\, ?\, 0 : 2 \rangle\rangle \end{aligned}$$

Example 4.5 (OGMF is more resource-friendly than GMF). Consider the program from Example 4.1. To show optimisation of OGMF, we evaluate it with $pc = \{\top, B_1, B_2, B_3, \bot\}$ and $\hat{\mu}$, where $\hat{\mu}(x_1) = \langle B_1\, ?\, 10 : 0 \rangle$, $\hat{\mu}(x_2) = \langle B_2\, ?\, 5 : 0 \rangle$, and $\hat{\mu}(x_3) = \langle B_3\, ?\, 7 : 0 \rangle$. After the execution of the instruction at line 2, the faceted memory is $\hat{\mu}' = \hat{\mu}[winner \mapsto 0, test \mapsto V]$, where $V = \langle B_1\, ?\, \langle B_2\, ?\, \mathtt{ff} : \mathtt{ff} \rangle : \langle B2\, ?\, \mathtt{ff} : \langle B_3\, ?\, \mathtt{tt} : \mathtt{tt} \rangle\rangle\rangle$. We consider the execution of the if instruction.

For levels $pc_1 = \{B_3, \bot\}$, the evaluation of $test$ is $\mathtt{tt} : B_3(\hat{\mu}^{pc}(test)) = \bot(\hat{\mu}^{pc}(test)) = \mathtt{tt}$. Moreover, $pc_1 \neq pc$, therefore, the rule OIf-S applies. The evaluation of the program is split to two: the first evaluation is with $P_1 = winner := 2$ and $pc_1 = \{B_3, \bot\}$; and the second evaluation is with $P_2 = \mathbf{skip}$ and $pc_2 = \{\top, B_1, B_2\}$. Each branch of the conditional will be evaluated only once.

The evaluation of P_1 with pc_1 terminates with $\hat{\mu}_1''(winner) = 2$. The evaluation of P_2 with pc_2 terminates with $\hat{\mu}_2''(winner) = 0$. These two faceted memories are combined to $\hat{\mu}''$, where $\hat{\mu}''(winner) = \langle B_1\, ?\, 0 : \langle B_2\, ?\, 0 : 2 \rangle\rangle$. The construction of this faceted memory is presented in Examples 4.2 and 4.4.

In the example above, OGMF has only two sub-evaluations, while GMF has five, moreover OGMF combines faceted memories once, while GMF combines them four times. Therefore, OGMF is more resource-friendly than GMF.

4.2 Equivalence to SME-TINI and Security Guarantee

We first establish the relation between the standard semantics and the semantics of OGMF.

LEMMA 4.6. $(P, \hat{\mu}) \downarrow_O^{pc} \hat{\mu}'$ *if and only if* $(P, l(\hat{\mu})) \Downarrow l(\hat{\mu}')$ *for all* $l \in pc$.

We now can prove the semantic equivalence result for OGMF and SME-TINI.

THEOREM 4.7. *OGMF and SME-TINI are equivalent.*

As a consequence, OGMF and GMF are equivalent even though OGMF is optimized. In addition, OGMF is TINI.

[3] The function $\mathbb{F}()$ cannot return a simple value since it is called on non-empty pc_1 and pc_2.

$$\llbracket \langle l\,?\,v : v' \rangle, pc \rrbracket = \begin{cases} v & \text{if } v = v' \\ \langle l\,?\,v : v' \rangle & \text{otherwise.} \end{cases}$$

$$\llbracket \langle l\,?\,v : \langle l'\,?\,v' : V' \rangle \rangle, pc \rrbracket = \begin{cases} \llbracket \langle l'\,?\,v : V' \rangle, pc' \rrbracket & \text{if } l' \sqsubseteq l,\ v = v', \text{ where } pc' = pc \setminus \{l\}, \\ v & \text{if } l \parallel l',\ v = v',\ \mathit{cond}(V, pc) \text{ and } \mathrm{glb}(l, l') \preccurlyeq pc, \\ \llbracket \langle \mathrm{glb}(l, l')\,?\,v : V'' \rangle, pc' \rrbracket & \text{if } l \parallel l',\ v = v',\ \mathit{cond}(V, pc) \text{ and } \mathrm{glb}(l, l') \mathrel{|||} pc, \text{ where } pc' = (pc \setminus S) \cup \{\mathrm{glb}(l, l')\}, \\ & \qquad S = \{l_1 \in pc \mid \mathrm{glb}(l, l') \sqsubseteq l_1\}, \text{ and } V'' = \langle\!\langle \mathit{List}(pc \setminus S), V' \rangle\!\rangle, \\ \langle l\,?\,v : \llbracket \langle l'\,?\,v' : V' \rangle, pc' \rrbracket \rangle & \text{otherwise, where } pc' = pc \setminus \{l\}. \end{cases}$$

$$\langle\!\langle L, V \rangle\!\rangle = \begin{cases} l(V) & \text{if } L = l, \\ \langle l\,?\,l(V) : \langle\!\langle T, V \rangle\!\rangle \rangle & \text{if } L = l.T,\ T \neq []. \end{cases}$$

Figure 9: Definition of $\llbracket V, pc \rrbracket$, and optimisation of a faceted value V with respect to the set of security levels pc.

5 A TERMINATION SENSITIVE VERSION OF MULTIPLE FACETS

A *termination sensitive* model assumes that an attacker can observe termination of evaluations. In [19], the model is explained further: an attacker at level l can observe the termination of evaluations at level l and lower. In the case of GMF and OGMF, an evaluation marked with pc is *an evaluation at l* if $l \in pc$. Notice that an evaluation is at more than one level whenever pc is not a singleton.

As illustrated by Example 5.1, GMF and OGMF do not prevent the influence of private data at higher levels to the termination of the evaluations at lower levels. In other words, GMF and OGMF do not prevent leakage on termination channel [19].

Example 5.1. Suppose that $\mathcal{L} = \{L, H\}$, where $L \sqsubseteq H$. We look at the evaluation of **if** x **then** (**while** tt **do skip**) **else skip** with $pc = \mathcal{L}$ and $\hat{\mu}(x) = \langle H\,?\,\mathrm{tt} : \mathrm{ff} \rangle$. When GMF or OGMF is used, the evaluation is split into two: one is with $pc_1 = \{H\}$, the other one is with $pc_2 = \{L\}$. The evaluation with pc_2 converges, while the evaluation with pc_1 diverges since its executing program is **while** tt **do skip**. Therefore, the evaluation of the whole program with $pc = \{L, H\}$ also diverges and hence, to an attacker at L, the evaluation at L diverges. However, if the program is evaluated with $\hat{\mu}'(x) = \langle H\,?\,\mathrm{ff} : \mathrm{ff} \rangle$, to the attacker at L, the evaluation at L converges. Based on observations on those two evaluations, an attacker at L can gain insight about the high facet of x. In other words, GMF and OGMF do not prevent the influence of data at H to the termination of the evaluation at L.

Therefore, we propose *Termination Sensitive Multiple Facets* (TSMF), a version of MF that takes into account the termination sensitive model. TSMF is a generalization of a version of MF presented in [8, Appendix A]. The basic idea of TSMF is that when an if instruction is evaluated, TSMF performs a bounded evaluation of the instruction by using OGMF. If the OGMF evaluation does not terminate within the given time bound, then the instruction is evaluated instead using SME semantics with a low-prio scheduler [14]. The security guarantees offered by TSMF are the same as SME with the same low-prio scheduler [19]. The semantics of TSMF and the proofs about its security guarantees can be found in [23].

6 RELATED WORK

SME. Devriese and Piessens introduce the idea of Secure Multi-Execution [14]. Since then, many researchers have developed different aspects of this approach. Close to our work, Kashyap et al. [19] discuss how schedulers might affect security guarantees (i.e., TSNI and TINI) based on the chosen scheduler and the lattice ordering. They show several schedulers and classify them according to the strength of security guarantees and according to fairness properties. This work complement theirs by providing a similar analysis but for an interplay of MF and SME semantics. SME [14] has many implementations: as a library in Haskell [18], as an experimental web browser based on Firefox [13], as a static program transformation for both Python and JavaScript [4], and as an adaptation to reactive systems [6]. In the work above, SME preserves the semantics of secure programs up to interleaving of events. To remedy that, Zanarini et al. [37] carefully leverage SME to design a precise monitor which exactly preserves semantics of secure programs *up to termination*. Several other works [10, 26, 33] expand SME and introduce declassification. In this work, we focus on semantics guarantees up to interleaving of events—as in the SME original formulation.

MF. Austin and Flanagan introduce MF semantics [3]—a technique often referred as an optimization for SME. However, as shown by Bielova and Rezk [7], they do not provide the same security guarantees (i.e., TINI vs. TSNI) and differ in their treatment of default values. This work provides yet another look into a comparison between both techniques to show their differences, while introducing novel value-based optimizations to MF. Another work by the same authors [9] compare and contrast five dynamic techniques, including MF and SME, to mainly reason about the preservation of semantics of secure programs, a property known as *transparency*. In this work, we show that GMF and OGMF enjoy the same transparency guarantees as SME-TINI (Theorems 3.10 and 4.7).

Tools. Most information flow control tools provide TINI, e.g., Jif [21], FlowCaml [25], Laminar [27], Paragon [11], and JSFlow [16]. Similarly, termination leaks are often ignored in security tools coming from the operating system research community, e.g., Asbestos [15], HiStar [38], and Flume [20]. A few exceptions to this trend are the security libraries LIO [31] and MAC [28], which provide TSNI for concurrent programs.

Decentralized label models. The decentralized label model (DLM), allows one to express the interests of mutually-distrusting principals without a central authority [22]. The set of labels forms a pre-order where the order relationship does not require to know all the points in the relationship to determine the result of comparing two labels—bearing in mind that there might be an infinite number of labels due to the dynamic creation of principals at runtime. In a similar spirit, DC-labels [32, 34] provides a decentralized label format which allows one to express rich policies dictated by mutually-distrusting principals as propositional logic formulas (without negation). In this work, we require to know all the points in the chosen lattice in order to optimize MF as shown by OGMF. Extending our techniques to DLM or DC-labels is an interesting direction for future work.

7 CONCLUSION AND PERSPECTIVES

This work contributes to develop techniques to secure programs using dynamic information flow—a promising approach to secure existing JavaScript code. We specially focus on proposing a technique that achieves a smaller number of executions than MF (and hence smaller memory footprint) without diminishing security guarantees. We further extend our MF-based technique to work with arbitrary finite lattices (GMF) based on the observation that off-the-shelf lattices with principals are not always the most convenient ones to use. Knowing all the points in the lattice allows for further optimizations: spawning multi-executions could be done on a value-based basis (OGMF) rather than on security levels—as in original MF. Finally, we propose a hybrid approach which present an interesting balance between the number of executions and security guarantees: it behaves as OGMF as long as it can and switches to SME when termination leaks could occur (TSMF). In other words, TSMF prioritizes resource usage as long as there are no risks for termination leaks. We expect that these insights will help inform future development of multi-execution-based techniques. In fact, an intriguing question is what it would take for our optimizations (or future ones) to work on potentially infinite lattices like the DLM or DC-labels—an interesting direction for future work.

ACKNOWLEDGMENTS

We would like to thank anonymous reviewers for feedback that helped to improve this paper. This research has been partially supported by the ANR projects AJACS ANR-14-CE28-0008, CISC ANR-17-CE25-0014-01, the National Science Foundation under grants CCF-1337278 and CCF-1421016, and Swedish research agencies Vetenskapsrådet and SSF Cyber Security projects WebSec: Securing Web-driven Systems and Octopi: Secure Programming for the Internet of Things.

REFERENCES

[1] Thomas H. Austin and Cormac Flanagan. 2009. Efficient Purely-dynamic Information Flow Analysis. In *Proc. of PLAS 2009 (PLAS '09)*. 113–124.

[2] Thomas H. Austin and Cormac Flanagan. 2010. Permissive Dynamic Information Flow Analysis. In *Proc. of PLAS 2010 (PLAS '10)*. 1–12.

[3] Thomas H. Austin and Cormac Flanagan. 2012. Multiple Facets for Dynamic Information Flow. In *Proc. of POPL 2012 (POPL '12)*. 165–178.

[4] Gilles Barthe, Juan Manuel Crespo, Dominique Devriese, Frank Piessens, and Exequiel Rivas. 2012. Secure multi-execution through static program transformation. In *Formal Techniques for Distributed Systems*. Springer, 186–202.

[5] Abhishek Bichhawat, Vineet Rajani, Deepak Garg, and Christian Hammer. 2014. Generalizing Permissive-Upgrade in Dynamic Information Flow Analysis. In *Proc. of PLAS 2014 (PLAS'14)*. 15–24.

[6] N. Bielova, D. Devriese, F. Massacci, and F. Piessens. 2011. Reactive noninterference for a Browser model. In *Proc. of NSS 2011*. 97–104.

[7] Nataliia Bielova and Tamara Rezk. 2016. Spot the Difference: Secure Multi-execution and Multiple Facets. In *Proc. of ESORICS 2016*. 501–519.

[8] Nataliia Bielova and Tamara Rezk. 2016. *Spot the Difference: Secure Multi-Execution and Multiple Facets.* Technical Report. https://goo.gl/b7yoQ9.

[9] Nataliia Bielova and Tamara Rezk. 2016. A Taxonomy of Information Flow Monitors. In *Proc. of POST 2016*. 46–67.

[10] Iulia Boloşteanu and Deepak Garg. 2016. Asymmetric Secure Multi-execution with Declassification. In *Proc. of POST 2016*. 24–45.

[11] Niklas Broberg, Bart van Delft, and David Sands. 2013. Paragon for Practical Programming with Information-Flow Control.. In *Proc. of APLAS 2013 (LNCS)*, Vol. 8301. Springer, 217–232.

[12] Stefano Calzavara, Alvise Rabitti, and Michele Bugliesi. 2016. Content Security Problems?: Evaluating the Effectiveness of Content Security Policy in the Wild. In *Proc. of CCS 2016 (CCS '16)*. 1365–1375.

[13] Willem De Groef, Dominique Devriese, Nick Nikiforakis, and Frank Piessens. 2012. FlowFox: a web browser with flexible and precise information flow control. In *Proc. of CCS 2012*. ACM, 748–759.

[14] Dominique Devriese and Frank Piessens. 2010. Noninterference Through Secure Multi-execution. In *Proc. of IEEE SP 2010 (SP '10)*. 109–124.

[15] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazières, Frans Kaashoek, and Robert Morris. 2005. Labels and event processes in the Asbestos operating system. In *Proc. of SOSP 2005 (SOSP)*. ACM.

[16] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. 2014. JSFlow: Tracking information flow in JavaScript and its APIs. In *Proc. of SAC 2014*. ACM.

[17] Collin Jackson and Adam Barth. 2008. Beware of Finer-Grained Origins. In *Web 2.0 Security and Privacy (W2SP'08)*.

[18] Mauro Jaskelioff and Alejandro Russo. 2011. Secure multi-execution in haskell. In *International Andrei Ershov Memorial Conference on Perspectives of System Informatics*. Springer, 170–178.

[19] Vineeth Kashyap, Ben Wiedermann, and Ben Hardekopf. 2011. Timing- and Termination-Sensitive Secure Information Flow: Exploring a New Approach. In *Proc. of IEEE SP 2011 (SP '11)*. 413–428.

[20] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. 2007. Information Flow Control for Standard OS Abstractions. In *Proc. of SOSP 2007 (SOSP)*.

[21] Andrew C Myers. 1999. JFlow: Practical mostly-static information flow control. In *Proc. of POPL 1999*. ACM, 228–241.

[22] Andrew C Myers and Barbara Liskov. 2000. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 9, 4 (2000), 410–442.

[23] Minh Ngo, Nataliia Bielova, Cormac Flanagan, Tamara Rezk, Alejandro Russo, and Thomas Schmitz. 2017. A Better Facet of Dynamic Information Flow Control. (2017). https://goo.gl/Y2SEnw.

[24] Minh Ngo, Fabio Massacci, Dimiter Milushev, and Frank Piessens. 2015. Runtime Enforcement of Security Policies on Black Box Reactive Programs. In *Proc. of POPL 2015*.

[25] F. Pottier and V. Simonet. 2002. Information Flow Inference for ML. In *ACM Symp. on Principles of Programming Languages*. 319–330.

[26] Willard Rafnsson and Andrei Sabelfeld. 2013. Secure Multi-execution: Fine-Grained, Declassification-Aware, and Transparent. In *Proc. of CSF 2013*. 33–48.

[27] Indrajit Roy, Donald E. Porter, Michael D. Bond, Kathryn S. McKinley, and Emmett Witchel. 2009. Laminar: Practical Fine-grained Decentralized Information Flow Control. In *Proc. of PLDI 2009 (PLDI)*. ACM.

[28] Alejandro Russo. 2015. Functional Pearl: Two Can Keep a Secret, if One of Them Uses Haskell. In *Proc. of ICFP 2015 (ICFP)*. ACM.

[29] José Fragoso Santos, Thomas Jensen, Tamara Rezk, and Alan Schmitt. 2015. Hybrid Typing of Secure Information Flow in a JavaScript-Like Language. In *Proc. of TGC 2015*. 63–78.

[30] Dolière Francis Some, Nataliia Bielova, and Tamara Rezk. 2017. On the Content Security Policy Violations Due to the Same-Origin Policy. In *Proc. of WWW 2017 (WWW '17)*. 877–886.

[31] Deian Stefan, Alejandro Russo, Pablo Buiras, Amit Levy, John C Mitchell, and David Mazieres. 2012. Addressing covert termination and timing channels in concurrent information flow systems. In *Proc. of ICFP 2012*, Vol. 47. ACM, 201–214.

[32] D. Stefan, A. Russo, D. Mazières, and J. C. Mitchell. 2011. Disjunction Category Labels. In *Proc. of NordSec 2011*. Springer-Verlag.

[33] Mathy Vanhoef, Willem De Groef, Dominique Devriese, Frank Piessens, and Tamara Rezk. 2014. Stateful Declassification Policies for Event-Driven Programs. In *Proc. of CSF 2014 (CSF '14)*. 293–307.

[34] Lucas Waye, Pablo Buiras, Dan King, Stephen Chong, and Alejandro Russo. 2015. It's My Privilege: Controlling Downgrading in DC-Labels. In *International Workshop on Security and Trust Management*.

[35] Wikipedia. 2017. Ad exchange. (2017). https://en.wikipedia.org/wiki/Ad_exchange. Checked on Nov 08, 2017.

[36] Wikipedia. 2017. Real-time bidding. (2017). https://en.wikipedia.org/wiki/Real-time_bidding. Checked on Nov 08, 2017.

[37] Dante Zanarini, Mauro Jaskelioff, and Alejandro Russo. 2013. Precise enforcement of confidentiality for reactive systems. In *Proc. of CSF 2013*. IEEE, 18–32.

[38] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. 2006. Making information flow explicit in HiStar. In *USENIX Symp. on Operating Systems Design and Implementation*. USENIX.