Fall-Through Semantics for Mitigating Timing-Based Side Channel Leaks

Aniket Mishra
Indian Institute of Technology Gandhinagar, India

Abhishek Bichhawat
Indian Institute of Technology Gandhinagar, India

Abstract

With the recent advent of exploits like Spectre and Meltdown, the mitigation of side-channel attacks has become an important concern for security researchers. In this paper, we focus on timing-based side channels introduced through conditional branching on secret information within programs. We introduce a language that allows a programmer to write conditionals branching on secrets within its syntax, but has a semantics that keeps execution time constant with respect to an adversary under an observationally equivalent memory. We differ from other approaches that use program analysis methods, opting instead to modify the operational semantics to enforce the necessary properties. We formalize the semantics for our language with timing leak mitigations in Rocq (previously, Coq) and prove that these semantics satisfy the property of timing-sensitive non-interference. Since our system describes a mitigation approach for timing leaks in a general high-level imperative language, we believe that our semantics can be used as a basis for compiler construction for other high-level imperative languages that seek to be safe from timing side channels.

2012 ACM Subject Classification Theory of computation → Operational semantics; Theory of computation → Logic and verification

Keywords and phrases Timing leaks, information flow control, runtime monitor, type system, side-channel attacks

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2025.44

Supplementary Material *Software*: `https://github.com/satiscugcat/timing-side-channels/` archived at `swh:1:dir:ea55fb0069b70b63d7fed91de597f07191be8ed5`

Funding This work is funded in part by the DST Inspire Faculty Fellowship and the SERB Startup Research Grant.

Acknowledgements We would like to thank the anonymous reviewers for their feedback on this work.

1 Introduction

Traditionally, cybersecurity has dealt with the analysis of overt channels, which allow an attacker to both send and receive information, bypassing standard security protocols and opening up exploits that manipulate a system's behaviour. A side channel, on the other hand, is one where there is no such direct communication; instead, information is leaked through side effects of a program's execution. These side effects usually depend on the specific implementation of the program (which can vary even if it meets the input-output specifications) or the properties of the machine that executes the program. For instance, the amount of power the program consumes, the cache accesses, or even the time taken by the program to execute can leak information about secrets in the program [24]. Thus, side channels can leak information without the victim ever observing any change of the program's behaviour themselves.

One of the ways by which timing side channels are introduced is via the usage of control flow constructs in a program, as different branches of execution within the program may correspond to a difference in the time taken in the execution of each of these branches. An adversary that is able to track these times can leverage this information to steal data related to the branching invariant.

Cache side channels, on the other hand, are introduced due to the time taken to access data from the memory/cache. If some value has already been cached due to some prior access, it is much faster to retrieve that value as compared to it being accessed from the memory store. This difference in the time to access data from the cache and the main memory has been exploited as a cache side channel. Speculative execution has been a prominent reason for these side channel leaks and has been heavily exploited in the Spectre attacks [15, 19, 25, 39].

Another approach to avoid timing and cache-timing leaks in programs is to write code that follows *constant-time programming* [22]. Within this approach, certain programming practices such as avoiding the usage of conditionals, employing bit manipulation, etc. are followed to mitigate these side channels. However, not only does this lead a loss of expressivity and clarity with less straightforward code, it has also been shown that due to the presence of compiler optimisations, it is difficult to write code that is truly constant-time [20]. Moreover, it also becomes difficult to verify if a piece of code is constant-time in dynamic environments like the Web where a lot of the code is only available at execution time.

In this work, we propose a novel dynamic approach to tackle timing side channel leaks by modifying the runtime semantics of how a program executes. We mitigate this problem by formalising the semantics of a programming language, which syntactically allows its programs to contain conditionals, but its evaluation semantics are such that the programs are executed in a way that does not lead to timing leaks via control flow constructs. By modifying the semantics of how a program executes, we ensure that any optimizations included by the compilers are accounted for in ensuring that the program does not leak information due to timing side channel.

We propose a fall-through semantics that accounts for the timing leaks through branching in programs on secret values without requiring any static program transformations and without conflicting with the intended semantics of the program. We describe these semantics for an imperative language to show its enforcement. We prove that our fall-through semantics prevent any information leaks due to explicit flows, implicit flows and through timing side-channels by showing that our semantics satisfies timing-sensitive non-interference. We also prove that our approach is semantics-preserving. In addition to this, we have formalised these semantics within the Rocq (Coq) theorem prover [38] and thus have machine checked proofs [28] for the relevant security properties.

2 Background

We provide a brief overview of basic concepts in runtime enforcement of information flow control (IFC) [21, 37]. In dynamic IFC, a security label is attached with every value, which is an element of a pre-determined lattice and is an upper bound on the security levels of all the values that have influenced the computation that led to the value.

Typically, all values are classified into security levels or labels and the policy is a lattice over these labels. Information is only allowed to flow up the lattice having a well-defined partial ordering ($\sqsubseteq$), join ($\sqcup$), and meet ($\sqcap$) operations. For illustration purposes, often the smallest nontrivial lattice $L \sqsubseteq H$ is used, which specifies that public (low, L) data must not be influenced by confidential (high, H) data.

Information flows can happen either explicitly, implicitly or indirectly through covert side-channels.

Explicit flows arise as a result of values being assigned to others. For instance, there is an explicit flow of information from both z and y to x in the statement $x = y + z$. To account for explicit flows, we update the label of the computed value (x in our example) with the least upper bound of the labels of the operands in the computation.

Implicit flows arise from control dependencies. For example, in the program l = *false; if (h)* l = *true;*, there is an implicit flow from h to the final value of l. To handle these flows, we can maintain a program-context label (*pc*-label), which is an upper bound on the labels of all the values that have influenced the control flow thus far. Thus, if the value in h has label H, then *pc* will be H within the if branch. To prevent leaks from secret contexts to public variables, we disallow assignments to a variable carrying a public value, in a secret context (also known as the no-sensitive-upgrade (NSU) check [7]). When assigning inside a branch, the final computed value has the label of the operands and the program context to indicate the level of secrecy. This is necessary to guarantee the standard security property of non-interference.

The above checks guarantee a variant of non-interference that does not account for leaks due to indirect side-channels like termination or timing. Timing side channels are introduced as a result of the implementation of a program, specifically through control flow. Control flow constructs like conditionals introduce timing side channels because different branches of a conditional correspond to different execution times for a program. As an example of how this may leak information, consider a simple use case of a program that checks an input against some password.

```
1    bool matchpwd(int* input, size_t n){
2        if (n != pwd_length) return false;
3        for (int i = 0; i < n; i++) {
4            if (input[i] != pwd[i]) return false;
5        }
6        return true;
7    }
```

The first branch returns immediately if the actual password length is different from the length of the input. If the length of the input is same as that of the password, it enters the loop requiring more time for processing as compared to the former case. The loop iterates depending on how many characters of the password match with the input. Based on the *time taken* for the program to return, an adversary can figure out whether the length of the input matches the password and if the different characters in the input match that of the password.

Further, information stored in the cache can affect the timing behavior of program execution as data not in cache takes more time for retrieval as compared to data already in the cache. If the data has been retrieved in a secret context, the presence of data in the cache reveals information about the value of the secret context.

To guarantee timing-sensitive non-interference, we need to show that if any run of a program starting from a specific memory store takes time T to execute, then all runs of the same program starting from an equivalent memory store (that looks the same to an adversary who cannot observe secret values) take the same time T to execute while resulting in an equivalent final store. We also need to show that the order and timing of memory reads/writes remains the same across such equivalent memory stores.

2.1 Threat Model

An adversary may observe all public values in the memory store. More generally, an adversary at a security level ℓ in the lattice can observe all values in the memory store that are labelled $k \sqsubseteq \ell$; all such values are considered public with respect to the adversary while values that have labels $k \not\sqsubseteq \ell$ are secret to the adversary. Our threat model assumes that the adversary has knowledge of the address of every memory access and the point in time during the execution of the program where this was performed. The adversary can additionally observe the time it takes for a program to execute and whether or not a program has terminated. Finally, the adversary can also observe when any given memory location has been read from or written to during the execution of the program. We assume that the time needed to perform an operation is standard and does not depend on the number of bits set to 0/1 in the operands. Similarly, we assume that the time taken to read/write a value from/to a memory location is the same regardless of the value. However, our approach can easily be extended to account for these differently by storing fine-grained timing information for different constants, variables and their types.

3 Related Works

Prior approaches have focused on verification of absence of timing leaks in programs or generating constant-time code. Our approach, on the other hand, focuses on runtime enforcement of mitigation of timing related leaks in programs, which, to the best of our knowledge, has not been explored before. The major difference between our work and the prior works is that our focus is on building a runtime that can mitigate timing leaks as opposed to performing program transformations at the source-level. The idea is to enforce this at the architectural level to reduce additional sources of timing leaks. We discuss some of the aforementioned prior works next.

Johan Agat presented one of the first approaches to mitigating timing leaks by proposing a transformation of the program to account for time spent in each of the branches [1], specifically by inserting dummy code that would take the same amount of time to execute as the real code. The program counter security model [29] is another such work, which pursued the problem of timing side-channels by transforming C code using a static checker that detects violations due to timing channels. The paper also introduces a novel method to formally capture the behaviour of various kinds of side channels, call the "program counter transcript model". Future approaches built on top of these works or used their approach to provide more accurate and robust mitigation approaches by transforming original programs to constant-time code. While our work does not transform the program code, the runtime semantics emulates a similar behavior to prevent leaks through the timing channel.

Some approaches have been proposed to verify the absence of timing side-channel leaks in program source-codes [4, 8, 14, 31, 33, 34, 40]. Approaches have also targeted verifying programs at the intermediate code or representation level to guarantee freedom from timing leaks [2, 3, 9] while some other approaches provide the same guarantees for assembly or machine-code [14, 17]. Some testing and fuzzing-based approaches target generation of concrete inputs to test for the presence of timing vulnerabilities in code [5]. However, these approaches focus on detection of side-channel leaks rather than preventing leaks due to them.

Certain other works [13, 18, 26, 27, 32, 36, 39, 41, 42] use various programming language or software engineering-based approaches for program transformation or repair to modify a program to remove timing leaks. Borrello et al. [12] do this at the level of the LLVM IR. Many of these techniques also rely on random input testing/fuzzing, and thus do not

completely guarantee that the generated code is side channel free. All of these require the entire code to be available for analysis beforehand, and thus are unfit for dynamic runtime environments. Works such as those of Barthe et al. [10], Qin et al. [35], and Arranz Olmos et al. [6] present approaches to show the preservation of constant-time property of programs during compilation. Other works deal with providing guarantees by implementing DSLs which enforce the required properties at the language level, for example FaCT [16] for C, and SecWasm [11] for WASM. Swivel [30] and WaSCR [23] deal with the constant time execution of WASM code in its generality.

4 Language

4.1 Syntax

Next, we describe the syntax of a simple imperative language (shown in Figure 1) on which we describe our enforcement approach. Our approach focuses on modifying the runtime semantics of the language to ensure that program executions are free from any leaks due to timing side channels. Expressions and commands are represented by e and c, respectively. Binary operators are represented as $\oplus$. n represents a constant value (either a number, or a boolean) and x represents a variable, which stores a value n having the label k. The memory-store μ is a mapping from variable names (x) to labeled values, n^k. Commands include the standard no-op, assignment, sequencing, branching and looping commands while expressions include constants, variables and binary operations.

In our language, k represents a security label. The label pc represents the security level of the current program context.

$$\mu := \cdot \mid \mu, x \mapsto n^k$$
$$e := n \mid x \mid e_1 \oplus e_2$$
$$c := \mathsf{skip} \mid x := e \mid c_1; c_2 \mid \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \mid \mathsf{while}\ e\ \mathsf{do}\ c$$

Figure 1 Syntax of WHILE.

Additionally, our semantics employ a 3-tuple structure of the form $\mathsf{debranch}(c, n, k)$, where n is a boolean and k is a level. This is not exposed in the surface level syntax, and is only used in intermediate evaluation steps.

4.2 Semantics

To formally define the information released by a program starting from some initial memory containing secret values, we define a big-step semantics for the language shown in Figure 1. To handle timing side channels, we introduce a debranch struct that accounts for timing leaks arising from the control flow of a program. Program configurations for expressions and commands are given as $e \mid \mu$ and $c \mid \mu$, respectively; the expression e and command c are respectively evaluated under the initial memory store μ, which is a map from variable names to labeled-values.

The judgement $e \mid \mu \Downarrow_{pc} T, n^k$ defines expression evaluation. It means that the expression e evaluated under the memory store μ in the program security context pc results in a constant n having the label k.

$$\text{CONST}\quad \frac{}{n \mid \mu \Downarrow_{pc} \langle \texttt{CONST} \rangle,\ n^{pc}} \qquad \text{VAR}\quad \frac{\mu(x) = n^k}{x \mid \mu \Downarrow_{pc} \langle \texttt{VAR}_\texttt{x} \rangle,\ n^{pc \sqcup k}}$$

$$\text{OPER}\quad \frac{e_1 \mid \mu \Downarrow_{pc} T_1, n_1^{k_1} \qquad e_2 \mid \mu \Downarrow_{pc} T_2, n_2^{k_2} \qquad n = n_1 \oplus n_2}{e_1 \oplus e_2 \mid \mu \Downarrow_{pc} \langle T_1, T_2, \texttt{OPER}_\oplus \rangle,\ n^{k_1 \sqcup k_2}}$$

Figure 2 Semantics for expression evaluation.

Command execution judgements have the form: $c \mid \mu \Downarrow_{pc} T \mid \mu'$ – where a command c executed under the memory store μ in the program security context pc modifies the initial store and generates a new memory store μ'.

Both judgements additionally generate T, which is a list of symbols corresponding to the evaluation steps taken by the program during its execution. For example $\langle \texttt{VAR}_\texttt{x}, \texttt{ASSN}_y \rangle$ would be list generated during evaluation of the command $y := x$. The notation T_1, T_2 represents the concatenation of the lists T_1 and T_2. The , operator indicates an append operation between two T lists. We assume that there is some constant extra amount of time that is required during each evaluation step of the program (where this constant may depend on which variable is being accessed, if any variable is accessed in the evaluation step). Under this assumption, proving the identity of the lists (as defined above) across execution traces over observationally equivalent memory stores proves our desired property, that is, the timing of each individual memory access is the same across observationally equivalent memory stores.

In any context where a boolean is expected (expressions involving the && and ! operators, within a conditional, or as the second element in the debranch 3-tuple), numbers are treated as falsy. This does not affect any of our results however, and any coercion scheme from numbers to booleans could be used.

The semantics for expression evaluation are defined in Figure 2 and the command execution semantics are defined in Figure 3. We describe the semantics in detail next.

The expression evaluation rules are mostly standard except for the additional timing related information captured by the semantics. The information captured is an account of the computation steps involved in the evaluation of the expression, that is dependent on the order of evaluation of the expressions. For instance, in CONST, $\langle \texttt{CONST} \rangle$ indicates that one constant expression has been evaluated; the resulting constant n has the label of the current program context pc. Similarly, for VAR, the variable x, evaluates to the value stored in μ while its security level is the join of the previous label k and the current program context pc, and generating the list $\langle \texttt{VAR}_\texttt{x} \rangle$, recording the variable that was read from too. The OPER inductively evaluates the sub-expressions e_1 and e_2 and the final T list is an order-dependent composition (thus, the expression $e_2 \oplus e_1$ would generate a different list) of the individual timing lists, such that each computation rule that is used in the evaluation of the final expression is recorded.

The SKIP command does not alter the memory store and returns a new list with SKIP while the sequencing command (SEQ) executes c_1 under the memory store μ and c_2 executes under the modified memory store μ'. The timing information of both these commands is added along with SEQ to record that a sequencing operation was executed. For the assignment statements (ASSN), we require that the final value stored in the variable has a label equivalent to at least the program context (which is obtained through the expression evaluation). The timing list generated also records the variable that it wrote to.

$$\text{SKIP}\quad \frac{}{\mathsf{skip} \mid \mu \Downarrow_{pc} \langle \texttt{SKIP} \rangle \mid \mu} \qquad \text{ASSN}\quad \frac{e \mid \mu \Downarrow_{pc} T, n^k}{x := e \mid \mu \Downarrow_{pc} \langle T, \texttt{ASSN}_{\texttt{x}} \rangle \mid \mu, x \mapsto n^k}$$

$$\text{SEQ}\quad \frac{c_1 \mid \mu \Downarrow_{pc} T_1 \mid \mu' \qquad c_2 \mid \mu' \Downarrow_{pc} T_2 \mid \mu''}{c_1; c_2 \mid \mu \Downarrow_{pc} \langle T_1, T_2, \texttt{SEQ} \rangle \mid \mu''}$$

$$\text{IF-HIGH}\quad \frac{e \mid \mu \Downarrow_{pc} T, n^k \qquad \mathsf{debranch}(c_1, n, pc) \mid \mu \Downarrow_k T_1 \mid \mu' \qquad \mathsf{debranch}(c_2, !n, pc) \mid \mu' \Downarrow_k T_2 \mid \mu'' \qquad k \not\sqsubseteq pc}{\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \mid \mu \Downarrow_{pc} \langle T, T_1, T_2, \texttt{IF} - \texttt{HIGH} \rangle \mid \mu''}$$

$$\text{IF-LOW}\quad \frac{e \mid \mu \Downarrow_{pc} T_1, n^k \qquad c = \left\{ \begin{array}{ll} c_1 & \mathit{if}\ n_1 = \mathsf{true} \\ c_2 & \mathit{otherwise} \end{array} \right\} \qquad c \mid \mu \Downarrow_{pc} T_2 \mid \mu' \qquad k \sqsubseteq pc}{\mathsf{if}\ e_1\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \mid \mu \Downarrow_{pc} \langle T_1, T_2, \texttt{IF} - \texttt{LOW} \rangle \mid \mu'}$$

$$\text{WHILE-F}\quad \frac{e \mid \mu \Downarrow_{pc} T, \mathsf{false}^k}{\mathsf{while}\ e\ \mathsf{do}\ c \mid \mu \Downarrow_{pc} \langle T, \texttt{WHILE} - \texttt{EXIT} \rangle \mid \mu}$$

$$\text{WHILE-T}\quad \frac{e \Downarrow_{pc} T_1, \mathsf{true}^k \qquad c \mid \mu \Downarrow_{pc} T_2 \mid \mu' \qquad \mathsf{while}\ e\ \mathsf{do}\ c \mid \mu' \Downarrow_{pc} T_3 \mid \mu'' \qquad k \sqsubseteq pc}{\mathsf{while}\ e\ \mathsf{do}\ c \mid \mu \Downarrow_{pc} \langle T_1, T_2, T_3, \texttt{WHILE} - \texttt{CONT} \rangle \mid \mu''}$$

Figure 3 Semantics for command evaluation.

For looping statements (WHILE-T), we enforce that the program context label is at least as secret as the label of the invariant, i.e., $k \sqsubseteq pc$ to ensure that the program does not loop on secret values; if the invariant evaluates to false (WHILE-F), the execution terminates. This is a standard assumption for constant-time execution as the maximum number of iterations a looping statement makes over all possible runs cannot be accurately determined. One could establish upper bounds, but any such upper bounds will naturally end up being extremely conservative (going up to possibly, $2^31 - 1$), and thus extremely inefficient performance-wise. Works such as those of Borrello et al. [12], that do try to provide such upper bounds do so via heuristics, and do not provide formal guarantees about the upper bound. The same restriction does not apply to conditionals, since either branch of a conditional runs at most one time (mutually exclusive to one another). The timing information accounts for the commands inside the loop.

The most interesting command evaluation rule is that of the branching statement. When branching on a predicate having a label less secret than the current program context (IF-LOW), the execution proceeds normally as in the unmodified semantics. However, when branching on predicates having a secret label (IF-HIGH), the execution follows the *debranching semantics* under the current pc as shown in Figure 4, which we explain next.

4.3 Debranching Semantics

The debranching command is given by $\mathsf{debranch}(c, n, \ell)$ where c is the command from the secret then branch ($n = \mathit{true}$) or the else branch ($n = \mathit{false}$) and ℓ is the previous security level at which the branching occurred.

DEB-SKIP

$$\frac{}{\mathsf{debranch}(\mathsf{skip}, n, \ell) \mid \mu \Downarrow_{pc} \langle \mathtt{DEB-SKIP} \rangle \mid \mu}$$

DEB-ASSN-TRUE

$$\frac{e \mid \mu \Downarrow_{pc} T_1, n^k \qquad x \mid \mu \Downarrow_{pc} T_2, n_1^{k_1}}{\mathsf{debranch}(x := e, (true), \ell) \mid \mu \Downarrow_{pc} \langle T_1, T_2, \mathtt{DEB-ASSN_x} \rangle \mid \mu, x \mapsto n^k}$$

DEB-ASSN-FALSE

$$\frac{e \mid \mu \Downarrow_{pc} T_1, n^k \qquad x \mid \mu \Downarrow_{pc} T_2, n_1^{k_1}}{\mathsf{debranch}(x := e, (false), \ell) \mid \mu \Downarrow_{pc} \langle T_1, T_2, \mathtt{DEB-ASSN_x} \rangle \mid \mu, x \mapsto n_1^k}$$

DEB-SEQ

$$\frac{\mathsf{debranch}(c_1, n, \ell) \mid \mu \Downarrow_{pc} T_1 \mid \mu' \qquad \mathsf{debranch}(c_2, n, \ell) \mid \mu' \Downarrow_{pc} T_2 \mid \mu''}{\mathsf{debranch}(c_1; c_2, n, \ell) \mid \mu \Downarrow_{pc} \langle T_1, T_2, \mathtt{DEB-SEQ} \rangle \mid \mu''}$$

DEB-IF-HIGH

$$\frac{\begin{array}{c} e_1 \mid \mu \Downarrow_{\ell} T_3, n_1^{k_\ell} \qquad e_1 \mid \mu \Downarrow_{pc} T_4, n_1^{k_{pc}} \qquad k_\ell \not\sqsubseteq \ell \qquad n_1 \& \& n = n' \qquad !n_1 \& \& n = n'' \\ \mathsf{debranch}(c_1, n', \ell) \mid \mu \Downarrow_{k_{pc}} T_1 \mid \mu' \qquad \mathsf{debranch}(c_2, n'', \ell) \mid \mu' \Downarrow_{k_{pc}} T_2 \mid \mu'' \end{array}}{\mathsf{debranch}(\text{ if } e_1 \text{ then } c_1 \text{ else } c_2, n, \ell) \mid \mu \Downarrow_{pc} \langle T_1, T_2, T_3, T_4, \mathtt{DEB-IF-HIGH} \rangle \mid \mu''}$$

DEB-IF-LOW

$$\frac{c = \left\{ \begin{array}{ll} \mathsf{debranch}(c_1, n, \ell) & \textit{if } n_1 = \mathsf{true} \\ \mathsf{debranch}(c_2, n, \ell) & \textit{otherwise} \end{array} \right\} \qquad \begin{array}{c} e_1 \mid \mu \Downarrow_{\ell} T_1, n_1^k \\ c \mid \mu \Downarrow_{pc} T_2 \mid \mu' \end{array} \qquad k \sqsubseteq \ell}{\mathsf{debranch}(\mathsf{if}\ e_1\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, n, \ell) \mid \mu \Downarrow_{pc} \langle T_1, T_2, \mathtt{DEB-IF-LOW} \rangle \mid \mu'}$$

DEB-WHILE-F

$$\frac{e \mid \mu \Downarrow_{\ell} T, \mathsf{false}^k}{\mathsf{debranch}(\mathsf{while}\ e\ \mathsf{do}\ c, n, \ell) \mid \mu \Downarrow_{pc} \langle T, \mathtt{DEB-WHILE-EXIT} \rangle \mid \mu}$$

DEB-WHILE-T

$$\frac{\begin{array}{c} e \Downarrow_{\ell} T_1, \mathsf{true}^k \\ \mathsf{debranch}(c, n, \ell) \Downarrow_{pc} T_2 \mid \mu' \qquad \mathsf{debranch}(\mathsf{while}\ e\ \mathsf{do}\ c, n, \ell) \mid \mu' \Downarrow_{pc} T_3 \mid \mu'' \end{array} \qquad k \sqsubseteq \ell}{\mathsf{debranch}(\mathsf{while}\ e\ \mathsf{do}\ c, n, \ell) \mid \mu \Downarrow_{pc} \langle T_1, T_2, T_3, \mathtt{DEB-WHILE-CONT} \rangle \mid \mu''}$$

Figure 4 Debranching semantics.

We distinguish the debranching commands' timing information from the normal commands that account for the fact that a secret branch has been encountered. The DEB-SKIP executes as SKIP without any modification to the store.

The DEB-ASSN-TRUE and DEB-ASSN-FALSE rules define the semantics of assignment operations in secret contexts. Depending on whether the assignment operation is in the "true" branch or the "false" branch, either the evaluated value of e is written in the memory store or the original value of x is written back to the memory store. Note that the label of the value in x may change due to the additional program context label being joined with the label of the value in x. The timing includes the time taken to evaluate x, e and the memory store write, which is captured by $\mathtt{DEB-ASSN_x}$. Note that here too we record the variable which we write to.

In DEB-ASSN-FALSE we upgrade the label of the value in the store even though we do not change the value itself. Without this upgrade, the value may leak information because in one branch of the program, the value (and label) would get updated while in the other branch it would not. Thus, the visibility of the value in the store would be secret dependent. In our semantics, the value becomes hidden regardless of the branch the program takes.

The DEB-SEQ rule inductively evaluates c_1 and c_2 under the debranching semantics, the result of which is the final store μ''. The timing accounts for the time taken in executing c_1 and c_2 and an indication that these were sequenced commands ($\texttt{DEB} - \texttt{SEQ}$).

The rule that branches on more secret values in a secret context requires both c_1 and c_2 to run in the more secret context. In DEB-IF-HIGH, the context switches are made because of the no-sensitive-upgrade check [7]. The results of the actions performed on the store are dependent on the prior value of n and the value obtained through evaluation of e_1. The DEB-WHILE-T rule requires that the context does not increase any further and the evaluation happens in the current context. This prevents any non-deterministic behavior due to the change in the predicate values.

4.4 Example

With the help of a simple example, we explain how the debranching semantics addresses the issue of timing-based side-channel leaks.

```
1    while (i < l)
2    do
3        if (input[i] != pwd[i])
4        then r := 1
5        else skip;
6        i := i + 1;
7    return r
```

In the example above, consider a two-point lattice $L \sqsubseteq H$ and two runs of the program where

$$\mu_1 = [\texttt{l} \mapsto 3^L; \texttt{i} \mapsto 0^L; \texttt{input} \mapsto [1,2,3]^L; \texttt{pwd} \mapsto [1,2,3]^H]$$

$$\mu_2 = [\texttt{l} \mapsto 3^L; \texttt{i} \mapsto 0^L; \texttt{input} \mapsto [3,2,3]^L; \texttt{pwd} \mapsto [1,2,3]^H]$$

and the initial $pc = L$.

In the first run with μ_1, the WHILE-T computes the value $n = true$ with time $\texttt{VAR}_\texttt{i}, \texttt{VAR}_\texttt{l}, \texttt{OPER}_<$ and executes the sequencing command which then executes the branch. The branch predicate evaluates to $\texttt{false}^\texttt{H}$ with time $\texttt{VAR}_{\texttt{input}[0]}, \texttt{VAR}_{\texttt{pwd}[0]}, \texttt{OPER}_{!=}$ as the first elements are the same and hence the debranch instruction is called with $\mathsf{debranch}(r := 1, false, L)$ under $pc = H$. The DEB-ASSN-FALSE is called, which evaluates r and constant 1 with times $\texttt{DEB} - \texttt{VAR}_\texttt{r}$ and $\texttt{DEB} - \texttt{CONST}$ respectively. The time to update the store with the original value of $r = 0$ is captured with $\texttt{DEB} - \texttt{ASSN}_\texttt{r}$. The next command $\mathsf{debranch}(skip, true, L)$ under $pc = H$ executes without any change to the memory and records $\texttt{DEB} - \texttt{SKIP}$. The second part of the sequencing command is the assignment, which computes the value $i + 1$ generating times $\texttt{CONST}$, $\texttt{VAR}_\texttt{i}$ and $\texttt{OPER}_+$ followed by $\texttt{ASSN}_\texttt{i}$ to update i. Similarly, the next two runs with μ_1 execute without effectively changing the value of μ_1 and returning the value of $r = 0$ meaning the comparison was successful generating the same sequence of timing information.

In the second run with μ_2, the WHILE-T computes the value $n = true$ with time $\texttt{VAR}_\texttt{i}, \texttt{VAR}_\texttt{l}, \texttt{OPER}_<$ and executes the sequencing command which then executes the branch. The branch predicate evaluates to $\texttt{true}^\texttt{H}$ with time $\texttt{VAR}_{\texttt{input}[0]}, \texttt{VAR}_{\texttt{pwd}[0]}, \texttt{OPER}_{!=}$ as the first elements are not the same and hence the debranch instruction is called with $\mathsf{debranch}(r := 1, true, L)$

under $pc = H$. The DEB-ASSN-TRUE is called, which evaluates r and constant 1 with times $\texttt{DEB} - \texttt{VAR}_\texttt{r}$ and $\texttt{DEB} - \texttt{CONST}$ respectively. The time to update the store with the value of $r = 1$ is captured with $\texttt{DEB} - \texttt{ASSN}_\texttt{r}$. The next command $\mathsf{debranch}(skip, false, L)$ under $pc = H$ executes without any change to the memory and records $\texttt{DEB} - \texttt{SKIP}$. The second part of the sequencing command is the assignment, which computes the value $i+1$ generating times $\texttt{CONST}$, $\texttt{VAR}_\texttt{i}$ and $\texttt{OPER}_+$ followed by $\texttt{ASSN}_\texttt{i}$ to update i. The next two runs with μ_2' execute without effectively changing the value of μ_2' and returning the value of $r = 1$ meaning the comparison failed. The timing for the remaining two runs is the same as the first run.

From the two runs, we observe that the final values of r^H are indistinguishable to an L-adversary and the timing information in both the runs is syntactically equal. In the normal run of the program, the assignment of $r = 1$ is executed only in one run, which takes a different execution time as compared to the skip operation. Furthermore, within our semantics, all the variables are read from and written to at the same time regardless of secret values. In a normal run, this is not the case, as r is written to in one run, but not the other. This demonstrates the ability of our semantics to mitigate cache side channels too.

5 Formalising Timing Leaks

The end-to-end security property usually established to ensure confidentiality is non-interference. Non-interference means that two runs of the same program starting from any two memory stores that are *observationally equivalent* for any adversary end with two memory stores that are also observationally equivalent for that adversary. For our observation model, where the adversary also records the time taken to complete execution, the timing information for all runs of the program should be observationally equivalent to the adversary.

Observational equivalence of values and stores is formalized using the relation $\sim_\ell$, where ℓ is the level of the attacker, and defined in Definition 1 and 2.

Definition 1 states that for an adversary at level ℓ, two values are equivalent if they are either public with respect to the adversary and have the same base value or both secret with respect to the adversary. Definition 2 states that two memory stores are equivalent with respect to the attacker if all values in the two stores are equivalent with respect to the attacker. The formalization of each of these equivalences in Rocq are shown immediately following the formal definitions. In the Rocq code, `Level` indicates the label, while `rel` is the partial-order relation between labels. `MemStore` is the memory store mapping variables to labeled-values.

▶ **Definition 1** (Value Observational Equivalence). *Two values $v_1 = n_1^{l_1}$ and $v_2 = n_2^{l_2}$ are observationally equivalent at level ℓ, written $v_1 \sim_\ell v_2$, iff either:*

1. *$n_1 = n_2$, $l_1 = l_2 \sqsubseteq \ell$ (or)*
2. *$l_1 \not\sqsubseteq \ell$ and $l_2 \not\sqsubseteq \ell$*

```
Inductive ValueObservationalEquivalent {rel: Level -> Level -> Type} {
    latticeProof: JoinSemilattice rel}: Primitive -> Level -> Level ->
    Primitive -> Level -> Type :=
| LowProof {n1 n2: Primitive} {l1 l2 l: Level} (nEq: n1 = n2) (lEq: l1 =
    l2) : ValueObservationalEquivalent n1 l1 l n2 l2
| HighProof (n1 n2: Primitive) {l1 l2 l: Level} (l1High: rel l1 l ->
    False) (l2High: rel l2 l -> False): ValueObservationalEquivalent n1
    l1 l n2 l2.
```

▶ **Definition 2** (Memory Store Observational Equivalence). *Two memory stores μ_1 and μ_2 are observationally equivalent at level ℓ, written as $\mu_1 \sim_\ell \mu_2$ iff $\forall x. \mu_1(x) \sim_\ell \mu_2(x)$*

```
Definition MemStoreObservationalEquivalent {rel: Level -> Level -> Type}
    {latticeProof: JoinSemilattice rel} (mu1: MemStore) (l: Level) (mu2:
    MemStore): Type :=
forall (x: Var), @ValueObservationalEquivalent rel latticeProof (fst (mu1
     x)) (snd (mu1 x)) l  (fst (mu2 x)) (snd (mu2 x)).
```

Using the observational equivalence, we define the timing equivalence for a program c with respect to an attacker at level ℓ in Definition 3.

▶ **Definition 3** (Timing Sensitive Non-interference). *A program c is said to not leak information via timing side channels, for any arbitrary level ℓ, if for any two memory stores μ_1, μ_2, $c \mid \mu_1 \Downarrow_\ell T_1 \mid \mu_1'$ and $c \mid \mu_2 \Downarrow_\ell T_2 \mid \mu_2'$, then $\mu_1 \sim_\ell \mu_2 \implies (T_1 = T_2 \ \wedge \ \mu_1' \sim_\ell \mu_2')$*

6 Soundness and Correctness

6.1 Soundness

To prove timing sensitive non-interference for our semantics, we establish a few helper lemmas to prove the final theorem. All of our formalization is mechanically proven in Rocq [28]. The theorem statements in Rocq are shown in the listings following the formal definitions.

We start by establishing the equivalence of expression evaluation under equivalent memory stores (Lemma 6). The lemma states that evaluating an expression under two observationally equivalent memory stores results in values $n_1^{k_1}$ and $n_2^{k_2}$ that are also observationally equivalent at level ℓ. We show the timing equality for expression evaluation separately in Lemma 9.

▶ **Lemma 4** (Lower Bound on Label from Expression Evaluation).

$$\forall e.\forall \mu.\forall \ell.e \mid \mu \Downarrow_\ell T, n^k \implies \ell \sqsubseteq k$$

▶ **Corollary 5** (Lowest Value for Expression Labels).

$$\forall e.\forall \mu.\forall \ell.e \mid \mu \Downarrow_\ell T, n^k \wedge k \sqsubseteq \ell \implies k = \ell$$

▶ **Lemma 6** (Expression Equivalence under Memory Store Equivalence). *If $\forall e.\forall \ell.\forall \mu_1.\forall \mu_2.$ $e \mid \mu_1 \Downarrow_\ell T_1, n_1^{k_1}$ $e \mid \mu_2 \Downarrow_\ell T_2, n_2^{k_2}$ then $n_1^{k_1} \sim_\ell n_2^{k_2}$*

Proof. By induction on the expression e. ◀

```
Lemma MemStoreEquivalenceImplExpressionEquivalence:
  forall {binop_eval: BinOp -> Primitive -> Primitive -> Primitive}
    {rel: Level -> Level -> Type}
    {lattice: JoinSemilattice rel}
    {e: Expression} {mu1 mu2: MemStore} {l k1 k2: Level}
    {n1 n2: Primitive} {T1 T2: TimingList}
    (p1: @ExpressionBigStep binop_eval rel lattice e mu1 l T1 n1 k1)
    (p2: @ExpressionBigStep binop_eval rel lattice e mu2 l T2 n2 k2)
    (memEq: @MemStoreObservationalEquivalent rel latticeProof mu1 l mu2),
  @ValueObservationalEquivalent rel latticeProof n1 k1 l n2 k2.
```

Theorem 7 shows that our debranching semantics preserve memory store equivalence, i.e., if we start from equivalent memory stores and execute a command c under the debranching semantics, we end up with equivalent memory stores.

▶ **Theorem 7** (Debranching preserves Memory Store Equivalence). *For any program c, levels ℓ, pc_1, pc_2 such that $l \sqsubset pc_1$ and $l \sqsubset pc_2$, and boolean values n_1, n_2 $\forall \mu_1.\forall \mu_2.\mu_1 \sim_\ell \mu_2 \implies$ $\mu_1' \sim_\ell \mu_2'$ where $\mathsf{debranch}(c, n_1, \ell) \mid \mu_1 \Downarrow_{pc_1} T_1 \mid \mu_1'$ and $\mathsf{debranch}(c, n_2, \ell) \mid \mu_2 \Downarrow_{pc_2} T_2 \mid \mu_2'$*

Proof. By induction on structure of c. ◀

```
Theorem DebranchPreservesMemEq
        {binop_eval: BinOp -> Primitive -> Primitive -> Primitive}
        {rel: Level -> Level -> Type}
        {lattice: JoinSemilattice rel} :
  forall {c: Command} {n1 n2: bool} {mu1 mu2 mu1' mu2': MemStore}
    {l pc1 pc2: Level} {T1 T2: TimingList}
    (p1: @DebranchBigStep binop_eval rel lattice
                          (Debranch c n1 l) mu1 pc1 T1 mu1')
    (p2: @DebranchBigStep binop_eval rel lattice
                          (Debranch c n2 l) mu2 pc2 T2 mu2')
    (memEq: @MemStoreObservationalEquivalent rel lattice mu1 l mu2)
    (l_rel_pc1: rel l pc1) (l_not_pc1: l <> pc1)
    (l_rel_pc2: rel l pc2) (l_not_pc2: l <> pc2),
  @MemStoreObservationalEquivalent rel latticeProof mu1' l mu2'.
```

Finally, we show that command evaluation preserves memory store equivalence, i.e., starting from attacker-equivalent stores, running a program c under our semantics results in attacker-equivalent final stores.

▶ **Theorem 8** (Evaluation preserves Memory Store Equivalence). *For any program c, levels pc if $\forall \mu_1.\forall \mu_2.\ c \mid \mu_1 \Downarrow_{pc} T_1 \mid \mu_1', c \mid \mu_2 \Downarrow_{pc} T_2 \mid \mu_2'$, and $\mu_1 \sim_\ell \mu_2$ then $\mu_1' \sim_\ell \mu_2'$*

Proof. By induction on c and applying Theorem 7. ◀

```
Theorem CommandPreservesMemEq
    {binop_eval: BinOp -> Primitive -> Primitive -> Primitive}
    {rel: Level -> Level -> Type}
    {latticeProof: JoinSemilattice rel} :
  forall  {c: Command} {mu1 mu2 mu1' mu2': MemStore}
    {pc: Level} {T1 T2: TimingList}
    (p1: @CommandBigStep binop_eval rel latticeProof c mu1 pc T1 mu1')
    (p2: @CommandBigStep binop_eval rel latticeProof c mu2 pc T2 mu2')
    (memEq: @MemStoreObservationalEquivalent rel latticeProof mu1 pc mu2)
    ,
  @MemStoreObservationalEquivalent rel latticeProof mu1' pc mu2'.
```

Further, we prove that with our semantics, the time-taken to evaluate an expression under two different stores, μ_1 and μ_2 is the same (Lemma 9).

▶ **Lemma 9** (Timing Security for Expressions). *If $\forall e.\forall \ell.\forall \mu_1.\forall \mu_2.e \mid \mu_1 \Downarrow_\ell T_1$ and $e \mid \mu_2 \Downarrow_\ell T_2$, then $T_1 = T_2$*

```
Lemma ExpressionTimSec
    {binop_eval: BinOp -> Primitive -> Primitive -> Primitive}
    {rel: Level -> Level -> Type} {lattice: JoinSemilattice rel}:
  forall {e: Expression} {pc1 pc2 k1 k2: Level} {mu1 mu2: MemStore}
    {n1 n2: Primitive} {T1 T2: TimingList},
    @ExpressionBigStep binop_eval rel lattice e mu1 pc1 T1 n1 k1 ->
    @ExpressionBigStep binop_eval rel lattice e mu2 pc2 T2 n2 k2 ->
  (T1 = T2).
```

Using the above result, we prove that the time taken for executing a command under the debranching semantics, starting from any two attacker-equivalent memory stores, is the same.

▶ **Theorem 10** (Debranch Timing Semantics). *For any program c, levels ℓ, pc_1, pc_2 such that $l \sqsubset pc_1$, $l \sqsubset pc_2$, and boolean values n_1, n_2,*

$$\forall \mu_1.\forall \mu_2.\mu_1 \sim_\ell \mu_2 \implies T_1 = T_2,$$

where $\mathsf{debranch}(c, n_1, \ell) \mid \mu_1 \Downarrow_{pc_1} T_1 \mid \mu_1'$ *and* $\mathsf{debranch}(c, n_2, \ell) \mid \mu_2 \Downarrow_{pc_2} T_2 \mid \mu_2'$

$$\text{N-CONST}\ \frac{}{n \mid \eta \Downarrow n} \qquad \text{N-VAR}\ \frac{\eta(x) = n}{x \mid \eta \Downarrow n} \qquad \text{N-OPER}\ \frac{e_1 \mid \eta \Downarrow n_1 \qquad e_2 \mid \eta \Downarrow n_2 \qquad n = n_1 \oplus n_2}{e_1 \oplus e_2 \mid \eta \Downarrow n}$$

Figure 5 Normal semantics for expression evaluation.

```
Lemma DebranchTimSec
    {binop_eval: BinOp -> Primitive -> Primitive -> Primitive}
    {rel: Level -> Level -> Type} {lattice: JoinSemilattice rel}:
  forall  {c: Command} {n1 n2: bool} {mu1 mu2 mu1' mu2': MemStore}
    {l pc1 pc2: Level} {T1 T2: TimingList}
    (p1: @DebranchBigStep binop_eval rel lattice (Debranch c n1 l)
                          mu1 pc1 T1 mu1')
    (p2: @DebranchBigStep binop_eval rel lattice (Debranch c n2 l)
                          mu2 pc2 T2 mu2')
    (memEq: @MemStoreObservationalEquivalent rel lattice mu1 l mu2)
    (l_rel_pc1: rel l pc1) (l_not_pc1: l <> pc1)
    (l_rel_pc2: rel l pc2) (l_not_pc2: l <> pc2),
  T1 = T2.
```

Using the above lemmas and theorems, we show that our semantics are timing-sensitive non-interferent according to the Definition 3.

▶ **Theorem 11** (Timing Sensitive Non-interference). *If*

$$\forall c.\forall \ell.\forall \mu_1.\forall \mu_2.\mu_1 \sim_\ell \mu_2 \text{ and } c \mid \mu_1 \Downarrow_\ell T_1 \mid \mu_1' \text{ and } c \mid \mu_2 \Downarrow_\ell T_2 \mid \mu_2'$$

then $T_1 = T_2$

```
Theorem CommandTimSec
    {binop_eval: BinOp -> Primitive -> Primitive -> Primitive}
    {rel: Level -> Level -> Type} {lattice: JoinSemilattice rel} :
  forall {c: Command} {mu1 mu2 mu1' mu2': MemStore}
         {pc: Level} {T1 T2: TimingList}
    (p1: @CommandBigStep binop_eval rel lattice c mu1 pc T1 mu1')
    (p2: @CommandBigStep binop_eval rel lattice c mu2 pc T2 mu2')
    (memEq: @MemStoreObservationalEquivalent rel lattice mu1 pc mu2),
  T1 = T2.
```

6.2 Correctness

We also prove the correctness of our semantics, i.e., we prove that our debranching semantics does not conflict with the intended semantics of the program. To do this, first we formalise what the intended semantics looks like. The semantics are shown in Figure 5 and Figure 6.

For expressions, we use judgement of the form: $e \mid \eta \Downarrow n$ and for commands, the judgement is of the form: $c \mid \eta \Downarrow \eta'$ where e is an expression, c is a command and η is a map from variables to primitive values, and n is a primitive value.

We start by defining store projection, which is a view of the store with values having no labels.

▶ **Definition 12** (Store Projection). *Given a memory store μ its projection $\mu_\downarrow$ is defined as the function such that* $\forall x.\mu(x) = n^k \iff \mu_\downarrow(x) = n$

Given these semantics, we prove the correctness of expression evaluation, the debranching commands and command evaluation.

$$\text{N-SKIP}\ \frac{}{\mathsf{skip} \mid \eta \Downarrow \eta} \qquad \text{N-ASSN}\ \frac{e \mid \eta \Downarrow n}{x := e \mid \eta \Downarrow \eta, x \mapsto n} \qquad \text{N-SEQ}\ \frac{c_1 \mid \eta \Downarrow \eta' \qquad c_2 \mid \eta' \Downarrow \eta''}{c_1; c_2 \mid \eta \Downarrow \eta''}$$

$$\text{N-IF}\ \frac{e \mid \eta \Downarrow n \qquad c = \left\{ \begin{array}{ll} c_1 & \textit{if } n_1 = \mathsf{true} \\ c_2 & \textit{otherwise} \end{array} \right\} \qquad c \mid \eta \Downarrow \eta'}{\mathsf{if}\ e_1\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \mid \eta \Downarrow \eta'} \qquad \text{N-WHILE-F}\ \frac{e \mid \eta \Downarrow \mathsf{false}}{\mathsf{while}\ e\ \mathsf{do}\ c \mid \eta \Downarrow \eta}$$

$$\text{N-WHILE-T}\ \frac{e \Downarrow \mathsf{true} \qquad c \mid \eta \Downarrow \eta' \qquad \mathsf{while}\ e\ \mathsf{do}\ c \mid \eta' \Downarrow \eta''}{\mathsf{while}\ e\ \mathsf{do}\ c \mid \eta \Downarrow \eta''}$$

Figure 6 Normal semantics for command evaluation.

▶ **Theorem 13** (Expression Soundness). *Given an expression e we have if $\forall \mu. \forall pc.\quad e \mid \mu \Downarrow_{pc} T, n_1^k$, then $e \mid \mu_\downarrow \Downarrow n_1$*

The expression correctness lemma states that if an expression evaluates to a value in our instrumented semantics, it also evaluates to the same value in the normal semantics.

▶ **Lemma 14** (Debranch False Ident). *Given a command c, and levels pc, ℓ we have $\forall \mu. \mu_\downarrow = \mu'_\downarrow$ where $\mathit{debranch}(c, \mathit{false}, \ell) \mid \mu \Downarrow_{pc} T \mid \mu'$*

▶ **Lemma 15** (Debranch Soundness). *Given a command c, and levels pc, ℓ we have $\forall \mu. \mu'_\downarrow = \eta$ where $\mathit{debranch}(c, \mathit{true}, \ell) \mid \mu \Downarrow_{pc} T \mid \mu'$ and $c \mid \mu_\downarrow \Downarrow_{pc} \eta$*

The final correctness theorem states that if a command evaluates under the instrumented semantics to some memory store μ', the command execution under the normal semantics produces a projected memory store $\eta = \mu'_\downarrow$.

▶ **Theorem 16** (Command Soundness). *Given a command c, and level pc, we have $\forall \mu. \mu'_\downarrow = \eta$ if $c \mid \mu \Downarrow_{pc} T \mid \mu'$ then $c \mid \mu_\downarrow \Downarrow_{pc} \eta$*

Further, we also show a (partial) completeness theorem that states that if a program, without any while loops, is executed under the normal semantics and generates a store η, under the instrumented semantics, the program generates a store μ such that $\mu \downarrow = \eta$. A stronger theorem would require proving the same with all while loops having secret predicates removed, but it is difficult to model secret predicates in the normal semantics.

As it is difficult to reason about the completeness of our semantics with respect to the normal semantics because of the loops, we limit our analysis to programs without while loops in them, as defined using the inductive predicate `NoWhile`.

```
Inductive NoWhile : Command -> Type :=
| NoWhileSkip : NoWhile SkipCommand
| NoWhileAssn: forall {x e}, NoWhile (AssnCommand x e)
| NoWhileSeq: forall {c1 c2}, NoWhile c1 -> NoWhile c2 ->
                NoWhile (SeqCommand c1 c2)
| NoWhileIf: forall {e c1 c2}, NoWhile c1 -> NoWhile c2 ->
                NoWhile (IfCommand e c1 c2).
```

The completeness theorem requires the completeness of expression evaluation and debranching, which are shown next.

▶ **Lemma 17** (Expression Evaluation Completeness). *Given an expression e, we have if* $\forall \mu. \forall pc. \quad e \mid \mu \Downarrow n_1$, *then* $\exists\ T\ k.e \mid \mu_\downarrow \Downarrow_{pc} T, n_1^k$

▶ **Lemma 18** (Debranching Completeness). *Given a command c, levels ℓ, pc and a store μ if* `NoWhile`(c) *and* $c \mid \mu_\downarrow \Downarrow_{pc} \eta$, *then* $\exists\ T\ \mu'.$*debranch*$(c, \mathit{true}, \ell) \mid \mu \Downarrow_{pc} T \mid \mu'$ *and* $\mu'_\downarrow = \eta$

Theorem 19 shows the final (partial) completeness theorem.

▶ **Theorem 19** (Relative Completeness). *Given a command c, level pc and a store μ if* `NoWhile`(c) *and* $c \mid \mu_\downarrow \Downarrow_{pc} \eta$, *then* $\exists\ T\ \mu'.c \mid \mu \Downarrow_{pc} T \mid \mu'$ *and* $\mu'_\downarrow = \eta$

```
Theorem CommandSystemCompleteness
    {binop_eval: BinOp -> Primitive -> Primitive -> Primitive}
    {rel: Level -> Level -> Type}
    {latticeProof: JoinSemilattice rel}:
  forall {c: Command} {mu: MemStore} {nu: NormalStore} {pc: Level},
    NoWhile c ->
    @NormalBigStep binop_eval c (StoreProjection mu) nu ->
    prod (EX (fun T => EX (fun mu' =>
            @CommandBigStep binop_eval rel latticeProof c mu pc T mu')))
            (StoreProjection mu' = nu).
```

7 Discussion

In our threat model, we assume the attacker has knowledge of every memory access. We model this by providing extra annotations in all the rules involving reading or writing from a variable, thus incorporating the address into the timing information. By proving that the timing information generated is independent of secrets, we thus prove that the memory accesses the program performs are independent of secrets. Thus, we believe our approach can be extended to handle cache side channels as well. However, it would involve a considerate formalisation, which we consider to be outside the scope of our paper.

Further, our focus here is on establishing a theoretical approach to solving this problem via a runtime monitor, which is generic enough to work at low-level. We leave the implementation of the semantics and performance comparisons on a real architecture to future work.

8 Conclusion and Future Work

We present a runtime approach to mitigating timing side-channel leaks in program during execution by ensuring that programs execute in constant time. By showing the independence of memory access patterns from secret values, we also provide strong guarantees against cache side channels. We formalize the semantics for our approach and prove that it satisfies timing-sensitive non-interference in Rocq theorem prover. We also show that our semantics are correct with respect to the normal execution of the program and do not produce wrong results.

In the future, we would like to extend this approach to a full-fledged programming language like LLVM IR to ensure the feasibility of the approach in the presence of constructs like return, continue, etc. that break control flow of the program. This would also allow us to evaluate the performance overheads incurred by the approach as compared to the normal execution and study the trade-offs. Another avenue of research is into the treatment of looping on high values, whether we could relax the restrictions on them and thus cover a wider assortment of programs with our semantics.

References

1 Johan Agat. Transforming out timing leaks. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '00, pages 40–53, New York, NY, USA, 2000. Association for Computing Machinery. doi:10.1145/325694.325702.

2 José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, Arthur Blot, Benjamin Grégoire, Vincent Laporte, Tiago Oliveira, Hugo Pacheco, Benedikt Schmidt, and Pierre-Yves Strub. Jasmin: High-Assurance and High-Speed Cryptography. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1807–1823, Dallas Texas USA, October 2017. ACM. doi:10.1145/3133956.3134078.

3 José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying constant-time implementations. In *Proceedings of the 25th USENIX Conference on Security Symposium*, SEC'16, pages 53–70, USA, 2016. USENIX Association. event-place: Austin, TX, USA. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/almeida.

4 Basavesh Ammanaghatta Shivakumar, Gilles Barthe, Benjamin Grégoire, Vincent Laporte, and Swarn Priya. Enforcing fine-grained constant-time policies. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, pages 83–96, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3548606.3560689.

5 Timos Antonopoulos, Paul Gazzillo, Michael Hicks, Eric Koskinen, Tachio Terauchi, and Shiyi Wei. Decomposition instead of self-composition for proving the absence of timing channels. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 362–375, Barcelona Spain, June 2017. ACM. doi:10.1145/3062341.3062378.

6 Santiago Arranz Olmos, Gilles Barthe, Lionel Blatter, Benjamin Grégoire, and Vincent Laporte. Preservation of speculative constant-time by compilation. *Proc. ACM Program. Lang.*, 9(POPL), January 2025. doi:10.1145/3704880.

7 Thomas H. Austin and Cormac Flanagan. Efficient purely-dynamic information flow analysis. In *Proc. ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security*, pages 113–124, 2009. doi:10.1145/1554339.1554353.

8 J. Bacelar Almeida, Manuel Barbosa, Jorge S. Pinto, and Bárbara Vieira. Formal verification of side-channel countermeasures using self-composition. *Science of Computer Programming*, 78(7):796–812, July 2013. doi:10.1016/j.scico.2011.10.008.

9 Gilles Barthe, Sunjay Cauligi, Benjamin Grégoire, Adrien Koutsos, Kevin Liao, Tiago Oliveira, Swarn Priya, Tamara Rezk, and Peter Schwabe. High-Assurance Cryptography in the Spectre Era. In *S&P 2021 - IEEE Symposium of Security and Privacy*, Virtual, France, May 2021. doi:10.1109/SP40001.2021.00046.

10 Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. Secure compilation of side-channel countermeasures: The case of cryptographic "constant-time". In *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, pages 328–343, 2018. doi:10.1109/CSF.2018.00031.

11 Iulia Basys, Maximilian Algehed, Alexander Sjösten, and Andrei Sabelfeld. Secwasm: Information flow control for webassembly. In Gagandeep Singh and Caterina Urban, editors, *Static Analysis*, pages 74–103, Cham, 2022. Springer Nature Switzerland. doi:10.1007/978-3-031-22308-2_5.

12 Pietro Borrello, Daniele Cono D'Elia, Leonardo Querzoni, and Cristiano Giuffrida. Constantine: Automatic side-channel resistance using efficient control and data flow linearization. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, pages 715–733, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3460120.3484583.

13 Robert Brotzman, Shen Liu, Danfeng Zhang, Gang Tan, and Mahmut Kandemir. Casym: Cache aware symbolic execution for side channel detection and mitigation. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 505–521, 2019. doi:10.1109/SP.2019.00022.

14 Luwei Cai, Fu Song, and Taolue Chen. Towards efficient verification of constant-time cryptographic implementations. *Proc. ACM Softw. Eng.*, 1(FSE), July 2024. doi:10.1145/3643772.

15 Sunjay Cauligi, Craig Disselkoen, Klaus v. Gleissenthall, Dean Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. Constant-time foundations for the new spectre era. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, pages 913–926, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3385412.3385970.

16 Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S. Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. Fact: a dsl for timing-sensitive computation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 174–189, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3314221.3314605.

17 Jia Chen, Yu Feng, and Isil Dillig. Precise Detection of Side-Channel Vulnerabilities using Quantitative Cartesian Hoare Logic. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 875–890, Dallas Texas USA, October 2017. ACM. doi:10.1145/3133956.3134058.

18 Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. Cacheaudit: A tool for the static analysis of cache side channels. *ACM Trans. Inf. Syst. Secur.*, 18(1), June 2015. doi:10.1145/2756550.

19 Xaver Fabian, Marco Patrignani, Marco Guarnieri, and Michael Backes. Do you even lift? strengthening compiler security guarantees against spectre attacks. *Proc. ACM Program. Lang.*, 9(POPL), January 2025. doi:10.1145/3704867.

20 Lukas Gerlach, Robert Pietsch, and Michael Schwarz. Do compilers break constant-time guarantees? In *Financial Cryptography and Data Security (FC)*, 2025.

21 Joseph A. Goguen and José Meseguer. Security policies and security models. In *Proc. 1982 IEEE Symposium on Security and Privacy*, pages 11–20, 1982. doi:10.1109/SP.1982.10014.

22 Guidelines for mitigating timing side channels against cryptographic implementations. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/secure-coding/mitigate-timing-side-channel-crypto-implementation.html, 2022.

23 Liyan Huang, Junzhou He, Chao Wang, and Weihang Wang. Wascr: A webassembly instruction-timing side channel repairer. In *Proceedings of the ACM on Web Conference 2025*, WWW '25, pages 4562–4571, New York, NY, USA, 2025. Association for Computing Machinery. doi:10.1145/3696410.3714693.

24 Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '96, pages 104–113, Berlin, Heidelberg, 1996. Springer-Verlag. doi:10.1007/3-540-68697-5_9.

25 Matthew Kolosick, Basavesh Ammanaghatta Shivakumar, Sunjay Cauligi, Marco Patrignani, Marco Vassena, Ranjit Jhala, and Deian Stefan. Robust constant-time cryptography. *Proc. ACM Program. Lang.*, 9(PLDI), June 2025. doi:10.1145/3729310.

26 Rui Lima, João F. Ferreira, and Alexandra Mendes. Automatic repair of java code with timing side-channel vulnerabilities. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*, pages 1–8, 2021. doi:10.1109/ASEW52652.2021.00014.

27 Cong Ma, Dinghao Wu, Gang Tan, Mahmut Taylan Kandemir, and Danfeng Zhang. Quantifying and mitigating cache side channel leakage with differential set. *Proc. ACM Program. Lang.*, 7(OOPSLA2), October 2023. doi:10.1145/3622850.

28 Machine checked proofs. https://anonymous.4open.science/r/timing-side-channels-47BE, 2025.

29 David Molnar, Matt Piotrowski, David Schultz, and David Wagner. The program counter security model: automatic detection and removal of control-flow side channel attacks. In

Proceedings of the 8th International Conference on Information Security and Cryptology, ICISC'05, pages 156–168, Berlin, Heidelberg, 2005. Springer-Verlag. doi:10.1007/11734727_14.

30 Shravan Narayan, Craig Disselkoen, Daniel Moghimi, Sunjay Cauligi, Evan Johnson, Zhao Gang, Anjo Vahldiek-Oberwagner, Ravi Sahita, Hovav Shacham, Dean Tullsen, and Deian Stefan. Swivel: Hardening WebAssembly against spectre. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1433–1450. USENIX Association, August 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/narayan.

31 Van Chan Ngo, Mario Dehesa-Azuara, Matthew Fredrikson, and Jan Hoffmann. Verifying and Synthesizing Constant-Resource Implementations with Types. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 710–728, San Jose, CA, May 2017. IEEE. doi:10.1109/SP.2017.53.

32 Brandon Paulsen, Chungha Sung, Peter A. H. Peterson, and Chao Wang. Debreach: mitigating compression side channels via static analysis and transformation. In *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ASE '19, pages 899–911. IEEE Press, 2020. doi:10.1109/ASE.2019.00088.

33 Swarn Priya. *Formally computer-verified protections against timing-based side-channel attacks*. Theses, Université Côte d'Azur, November 2023. URL: https://hal.science/tel-04331805.

34 Jonathan Protzenko, Jean-Karim Zinzindohoué, Aseem Rastogi, Tahina Ramananandro, Peng Wang, Santiago Zanella-Béguelin, Antoine Delignat-Lavaud, Cătălin Hriţcu, Karthikeyan Bhargavan, Cédric Fournet, and Nikhil Swamy. Verified low-level programming embedded in F*. *Proceedings of the ACM on Programming Languages*, 1(ICFP):1–29, August 2017. doi:10.1145/3110261.

35 Qi Qin, JulianAndres JiYang, Fu Song, Taolue Chen, and Xinyu Xing. Dejitleak: eliminating jit-induced timing side-channel leaks. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, pages 872–884, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3540250.3549150.

36 Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: closing digital side-channels through obfuscated execution. In *Proceedings of the 24th USENIX Conference on Security Symposium*, SEC'15, pages 431–446, USA, 2015. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/rane.

37 Dorothy Elizabeth Robling Denning. *Cryptography and Data Security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1982.

38 Rocq theorem prover. https://github.com/rocq-prover/rocq, 2025.

39 Haifeng Ruan, Yannic Noller, Saeid Tizpaz-Niari, Sudipta Chattopadhyay, and Abhik Roychoudhury. Timing side-channel mitigation via automated program repair. *ACM Trans. Softw. Eng. Methodol.*, 33(8), November 2024. doi:10.1145/3678169.

40 Basavesh Ammanaghatta Shivakumar, Gilles Barthe, Benjamin Gregoire, Vincent Laporte, Tiago Oliveira, Swarn Priya, Peter Schwabe, and Lucas Tabary-Maujean. Typing High-Speed Cryptography against Spectre v1 . In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 1094–1111, Los Alamitos, CA, USA, May 2023. IEEE Computer Society. doi:10.1109/SP46215.2023.10179418.

41 Luigi Soares, Michael Canesche, and Fernando Magno Quintão Pereira. Side-channel elimination via partial control-flow linearization. *ACM Trans. Program. Lang. Syst.*, 45(2), June 2023. doi:10.1145/3594736.

42 Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. Eliminating timing side-channel leaks using program repair. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 15–26, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3213846.3213851.