SOUND AND PRACTICAL METHODS FOR FULL-SYSTEM TIMING CHANNEL CONTROL

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Danfeng Zhang

August 2015

SOUND AND PRACTICAL METHODS FOR
FULL-SYSTEM TIMING CHANNEL CONTROL

Danfeng Zhang, Ph.D.
Cornell University 2015

Building systems with rigorous security guarantees is difficult, because most programming languages lack support for reasoning about security. This situation is amplified by emerging timing attacks, which reveal secrets from computation time. Recent work shows that timing channels can quickly leak sensitive information, such as private keys of RSA and AES. Such threats greatly harm the security of many emerging applications, such as cloud computing, mobile computing, and embedded systems.

This dissertation describes novel programming languages and run-time enforcement mechanisms for full-system control of timing channels. The proposed approach has two major components: A new software-hardware security interface, and control mechanisms present at separate levels of system abstraction. These control mechanisms include:

1) A type system for an imperative language, so that well-typed programs provably leak only a bounded amount of information via timing channels,
2) SecVerilog, a hardware description language that supports mostly-static, precise reasoning about information flows in hardware designs, and
3) Predictive mitigation, a general run-time mechanism that permits tunable tradeoffs between security and performance.

Evaluation on real-world security-sensitive applications suggest that the proposed approach is sound and has reasonable performance.

BIOGRAPHICAL SKETCH

Danfeng Zhang was born in Suzhou, a beautiful city in China. He obtained his Bachelor/Master of Science degrees in Computer Science in 2006/2009 from Peking University in Beijing, China. Although his research back in China has little to do with programming languages and security, Danfeng was fascinated by the area of language-based security after entering Cornell University in 2009. Since then, he has been studying for his doctorate degree, with a focus on designing programming models with rigorous security guarantees and minimal burden on programmers.

To my family

ACKNOWLEDGEMENTS

Making this dissertation and my Ph.D. degree a reality is impossible without the help from many people. First and foremost, I want to thank my advisor Andrew Myers. He has been an incredible advisor in all dimensions, such as research, teaching, writing and presentation. Conversations with Andrew have always been educational and motivating. His high standard in research and teaching made him a role model for me.

I would like to thank the other members of my committee: Dexter Kozen and Bart Selman. I have benefited a lot from their teaching and talking to them about my research. Moreover, I am very grateful to G. Edward Suh, who has been a wonderful mentor whenever I encounter hardware-related questions.

During my six years at Cornell, many friends and fellow students have helped me in various ways. It is fortunate to start my first project with Aslan Askarov. His insights and feedbacks have helped to shape some of the ideas that this dissertation is built on. It is my great fortune to have Jed Liu, Michael George, Krishnaprasad Vikram, Owen Arden, Chinawat Isradisaikul, Tom Magrino, Yizhou Zhang, Isaac Sheff, Laure Thompson and Matthew Milano as colleagues. They have always been a reliable source for constructive feedbacks on research papers and presentations. I am also grateful to Yao Wang, Andrew Ferraiuolo and Rui Xu: I learnt so much about hardware designs and Verilog from them in the SecVerilog project.

The Ph.D. journey is mostly enjoyable, but is also sometimes painful. Last but not least, I wish to thank my family for being so supportive of my academic pursuits. To Mom and Dad, thank you for always being on my side since my childhood. To my wife, thank you for your accompany, understanding and sharing the joys and pains in these years.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Timing channels have long been a difficult and important problem for computer security. The difficulty has long been recognized since the 70's [47, 26, 66], but their importance has been reinforced by recent work that shows timing channels can quickly leak sensitive information. Attacks exploit the timing of cryptographic operations [43, 13] and of web server responses [11]. These attacks work even without cooperation of any software on the system being timed. If the system contains malicious code or hardware (e.g., [74]), timing can also be exploited as a robust covert channel. Further, timing channels can be exploited stealthily, at low risk to the attacker [51].

Controlling timing channels is nevertheless extremely challenging, because confidential information can affect timing throughout the entire computer system. At the software level, a branch or loop conditioned on secret values creates timing channels. For instance, a branch condition depends on the private key in an early implementation of RSA, resulting in exploitable timing channels [43]. Moreover, even machine instructions with different operators can take variable time [18]. At the hardware level, shared hardware resources such as the data cache also creates timing channels. For example, cache probing attacks (e.g., [68, 65, 32]) exploit the timing channel that arises because accesses to memory locations by one process affect the cache, and thereby observably affect the timing behavior of later accesses by other processes. The cache is not the only problem. Attacks have also been shown that exploit timing channels arising from other components: instruction and data caches [2], branch predictors and branch target buffers [3], and shared functional units [87].

This dissertation introduces a sound and practical approach for full-system timing channel control. The core of this approach is a new software-hardware security interface, which for the first time, enables accurate reasoning about timing channels at the software-language level. Unlike previous work on language-based timing channel control, such as code transformation [4], this interface supports more realistic programs and hardware. For example, it can be implemented on hardware with an instruction cache, branch predictors, shared functional units, and so on. Such a new security interface forms a rigorous security contract between the software (language) level and the hardware implementation, enabling provable control of timing channels throughout the entire computer system.

On the software side, this dissertation presents a type system that provides fine-grained reasoning about timing channels, assuming the hardware follows the security contract. The type system can distinguish between benign timing variations and those carrying confidential information, and can distinguish between multiple distinct security levels. This fine-grained reasoning about timing channels improves the tradeoff between security and performance: benign timing variations provably leak no confidential information, so only the timing of code fragments that carry confidential information needs to be controlled.

On the hardware side, the security contract is formalized into three security requirements, guiding secure hardware designs. To formally verify such secure designs, this dissertation presents SecVerilog, a new hardware design language that statically checks information flows within hardware, including flows via timing channels. Unlike previous approaches, SecVerilog enables flexible, fine-grained reuse and sharing of hardware across security domains, via a novel type

system with dependent types. The benefit is that almost no additional run-time overhead is added, since SecVerilog checks information flows at compile time; little chip area and energy consumption is added since hardware resources can be shared securely across security domains.

To control timing channels arising from code fragments that do leak information, this dissertation proposes predictive mitigation, a practical method for a broad class of computing systems. Unlike previous general techniques for timing channel mitigation, this framework offers tunable tradeoffs between security and performance, so that programmers may improve system performance with a programmer-specified leakage function. By incorporating mechanisms for predictive mitigation of timing channels, the aforementioned software-level type system can also permit an expressive programming model, where applications with a provably bounded amount of timing leakage are allowed.

The soundness and effectiveness of the proposed approach are demonstrated on applications previously shown to be vulnerable to timing attacks, as well as security benchmarks. A complete MIPS processor with modern processor features, such as data hazard detection and data bypassing, is formally verified in SecVerilog. The results suggest that the combination of language-based mitigation and secure hardware works well, with overheads of only 1% in chip area, critical path delay and power consumption. Moreover, performance overhead for the verified applications running on the verified processor is about 20% on average .

1.1 Timing channel control: challenges

Timing channels are perhaps the most challenging aspect of information flow security, because confidential information can affect timing in various ways: at the software level, a branch or loop conditioned on secret values creates timing channels [43]; at the hardware level, shared hardware resources such as the data cache also create timing channels [68, 65, 18, 32].

1.1.1 Direct and indirect timing dependencies

We first define *direct* and *indirect* timing dependencies to distinguish timing channels arising from software and hardware respectively. We call timing channels visible at the source-language level *direct timing dependencies*. In the following example, control flow affects timing.

```
1 if (h)
2   sleep(1);
3 else
4   sleep(10);
5 sleep(h);
```

Assume `h` holds confidential data and that `sleep` (e) suspends execution of the program for the amount of time specified by e. Since line 4 takes longer to execute than line 2, one bit of `h` is leaked through timing. Such control-flow-related timing channels are real issues in practice, as demonstrated by attacks on RSA [43, 13]. Another source of direct timing dependencies is operations whose execution time depends on parameter values, such as the `sleep` command at line 5. In general, even machine instructions can take variable time [18].

Modern hardware also creates *indirect timing dependencies* in which execution time depends on hardware state that has no source-level representation. The following code shows that the data cache is one source of indirect dependencies.

```
1 if (h1)
2   h2:=l1;
3 else
4   h2:=l2;
5 l3:=l1;
```

Suppose only `h1` and `h2` are confidential and that neither `l1` nor `l2` are cached initially. Even though both branches have the same instructions and similar memory access patterns, executing this code fragment is likely to take less time when `h1` is not zero: because `l1` is cached at line 2, line 5 runs faster, and the value of `h1` leaks through timing.

Some timing attacks [65, 32] exploit data cache timing dependencies to infer AES encryption keys, but indirect dependencies arising from other hardware components have also been exploited to construct attacks: instruction and data caches [2], branch predictors and branch target buffers [3], and shared functional units [87].

1.1.2 Limitations of previous approaches

Due to the existence of indirect timing dependencies, software-based solutions are doomed to fail. Much prior language-based work uses simple, implicit models of timing, and no previous work fully addresses indirect dependencies. For example, type systems have been proposed to prevent timing channels, but are very restrictive. Often (e.g., [86, 78, 73, 76]) timing behavior of the program is assumed to be accurately described by the number of steps taken in an opera-

tional semantics. This assumption does not hold even at the machine-language level. As a result, these prior methods fail to handle timing channels arising from hardware features, such as data cache.

Some previous work uses program transformation to remove indirect dependencies, though only those arising from the data cache. The main idea is to equalize the execution time of different branches, but a price is paid in expressiveness, since these languages either rule out loops with confidential guards (as in [4, 34, 10]), or limit the number of loop iterations (as in [56, 18]). Moreover, these methods do not handle all indirect timing dependencies; for example, the instruction cache is not handled, so verified programs remain vulnerable to other indirect timing attacks [87, 3, 2].

Recent work in the architecture community has aimed for hardware-based solutions to timing channels. Their hardware designs implicitly rely on assumptions about how software uses the secure hardware, but these assumptions have not been rigorously defined or formally verified. For example, the cache design by Wang and Lee [88] works only under the assumption that the AES lookup table is preloaded into the cache and that the load time is not observable to the adversary [44].

1.2 Sound and practical full-system timing channel control

Timing channels cannot be controlled effectively at the software level only. Hardware mechanisms can help, but do a poor job of controlling language-level leaks such as direct timing dependencies. The question, then, is how to usefully and accurately characterize the timing semantics of code at the source level. Our

insight is to combine the language-level and hardware-level mechanisms, via a new cross-domain interface.

1.2.1 Hardware abstraction and assumptions

Throughout this dissertation, we use the term *machine environment* to refer to all hardware state that is invisible at the language level but that is needed to predict timing. Timing channels relying on indirect dependencies are at best difficult to reason about at the language level—the semantics of programming languages and even of instruction set architectures (ISAs) hide information about execution time by abstracting away low-level implementation details. For instance, it is difficult to reason about timing without knowing how the data cache works.

We assume all (software-level) information is associated with a *security label*, describing the confidentiality of the information. Labels ℓ_1 and ℓ_2 are ordered, written $\ell_1 \sqsubseteq \ell_2$, if ℓ_2 describes a confidentiality requirement that is at least as strong as that of ℓ_1. It is secure for information to flow from label ℓ_1 to label ℓ_2 if $\ell_1 \sqsubseteq \ell_2$. We assume there are at least two distinct labels L (low) and H (high) such that $\mathrm{L} \sqsubseteq \mathrm{H}$ and $\mathrm{H} \not\sqsubseteq \mathrm{L}$. The label of public information is L; that of secret information is H.

Accordingly, we can logically partition the machine environment according to the security labels associated with hardware state. For example, a commodity cache with no partition is a special case where all cache lines are associated with the same label. In general, different partitions in a partitioned cache [88] can be associated with different labels. For hardware components such as a pipeline, we can time-multiplex their use according to different security labels.

1.2.2 Timing interface and software enforcements

To concisely track how the machine environment affects timing, and how information flows into the machine environment, we propose a new cross-domain timing interface, in form of two *timing labels*. In particular, we associate two timing labels with each command in the program. The first of these labels is the command's *read label* ℓ_r. The read label is an upper bound on the label of machine environment that affects the run time of the command. For example, the run time of a command with ℓ_r = L depends only on cache state with label L or below. The second of these labels is the command's *write label* ℓ_w. The write label is a lower bound on the label of machine environment that the command can modify. It ensures that the labels of machine environment reflect the confidentiality of information that has flowed into that machine environment.

For example, suppose that there is only one (low) data cache, which to be conservative means that anyone can learn from timing whether a given memory location is cached. Therefore, both the read and write label of every command must be L. The example in Section 1.1.1 is then annotated as follows, where the first label in brackets is the read label, and the second, the write label.

```
1 if (h1)[L,L]
2   h2:=l1;[L,L]
3 else
4   h2:=l2;[L,L]
5 l3:=l1;[L,L]
```

This example is insecure because execution of lines 2 and 4 is conditioned on the high variable `h1`. Therefore these lines are in a *high context*, one in which the program counter label [21] is high. If lines 2 and 4 update cache state in the usual way, the low write label permits low hardware state to be affected by `h1`.

This insecure information flow is a form of *implicit flow* [21], but one in which hardware state with no language-level representation is being updated.

Since lines 2 and 4 occur in a high context, the write label of these commands must be H for this program to be secure. Consequently, the hardware may not update low parts of the machine environment. One way to avoid modifying the low parts of a commodity cache is to deactivate it in high contexts, since the entire cache is low. A generalization of this idea is to use a partitioned cache, where different partitions are associated with different labels. In this case, cache misses in a high context then cause only the high cache partition to be updated.

With the read and write labels abstracting the timing behavior of hardware, timing channel security can be statically checked at the language level, according to the type system described in Chapter 2. Moreover, these timing labels could be inferred automatically according to the type system, reducing the burden on programmers.

1.2.3 Provably secure hardware design

The aforementioned cross-domain interface (read and write labels) communicates information flows between the software and hardware, enabling timing channels to be rigorously controlled. The interface defines a contract that both software and hardware must follow for their composition to correctly control information flows. The next question is how to design complex hardware that correctly enforces its part of the contract.

To provide provable security for hardware designs, Chapter 3 presents a

method for designing hardware that correctly, precisely, and efficiently enforces secure information flow. This method is based on a new hardware description language (HDL) called SecVerilog, which adds a security type system to Verilog so that hardware-level information flows can be checked statically. In combination with software-level information flow control, our hardware design method enables building computing systems in which all forms of information flow are tracked, including implicit flows and timing channels.

SecVerilog has several advantages over the state of the art in secure hardware design. SecVerilog checks information flows statically while providing formal security assurance and guidance to hardware designers, avoiding runtime costs of tracking and checking of information flow. The language is expressive enough to prove security of a design even when hardware resources are shared among multiple security levels that are changed at a per-cycle granularity, avoiding the duplication of hardware resources. The novel dependent type system of SecVerilog follows a modular design that decouples the program analyses required for precision from the type system, making it more amenable to future extension. Our prototype secure pipelined MIPS processor with a cache adds area and clock cycle overheads of only about 1%.

1.2.4 Quantitative control of timing channels

Strictly disallowing all timing leakage can be done as sketched thus far, but results in an impractically restrictive programming language, or computer system, because execution time is prevented from depending on confidential information in any way.

To provide a strict bound on timing channel leakage while providing practical performance, Chapter 4 introduces a general framework called *predictive mitigation*. The key idea is that given a prediction of how long a computation will take, solely based on public information, a run-time enforcement can ensure that at least that much time is consumed by simply waiting if necessary. In the case of a misprediction (i.e., when the estimate is too low), a larger prediction is generated, and the execution time is padded accordingly. Mispredictions also inflate the predictions generated by subsequent mitigated computation, so that the total timing leakage is tightly bounded.

Predictive mitigation bounds the amount of information leaked through the timing channel as a function of elapsed time. Simple mitigation schemes can ensure that no more than $\log^2(T)$ bits of information are leaked, where T is the running time (all logarithms in this dissertation are base 2). Further, an arbitrary bound on information leakage can be enforced. However, tighter bounds have a price: they can reduce system throughput and increase system latency, particularly if the system has unpredictable behavior.

Chapter 5 incorporates the predictive mitigation framework into the software language in Chapter 2 by a new command called `mitigate`. Command (`mitigate` (e, ℓ) c) executes the command c while ensuring that timing leakage is bounded. Here, the expression e computes an initial prediction for the execution time of c. The label ℓ bounds what information can be learned by observing the timing leakage c. That is, no information at level ℓ' such that $\ell' \not\sqsubseteq \ell$ can be learned from c's execution time.

1.3 Outline

This dissertation presents and explores practical methods for full-system timing channel control. The key component, a cross-domain timing interface that enables compositional enforcement, is presented in Chapter 2. Chapter 2 also includes formal restrictions that are required by the interface on hardware implementation, as well as how the timing interface enables a software-level type system that provably eliminates all timing channels, with the assumption that hardware respects the interface.

Chapter 3 introduces the SecVerilog language, which extends Verilog with expressive type annotations that enable precise reasoning about information flow. The language also comes with rigorous formal assurance: SecVerilog provably enforces timing-sensitive noninterference and thus ensures secure information flow.

Chapter 4 describes a general framework, called predictive mitigation, which provides provable tight timing-channel leakage bound for applications where a limited leakage is allowed. We start from a black-box model, where we know nothing about a system other than the output events, and then extend it to interactive systems, which receive input requests from multiple clients and deliver responses.

Chapter 5 incorporates the predictive mitigation framework (Chapter 4) into the software language introduced in Chapter 2. The result is an expressive programming model, where applications with a provably bounded amount of timing leakage are allowed.

The soundness and effectiveness of the approach proposed in this dissertation is demonstrated on real-world security-sensitive applications, in Chapter 6. The results suggest this approach controls timing channels, and has reasonable performance for these real-world applications. Chapter 7 summarizes.

The materials presented in Chapters 2, 4 and 5 are adapted from joint work with Aslan Askarov and Andrew Myers [95, 6, 94]. Chapter 3, the SecVerilog language, is adapted from joint work with Yao Wang, G. Edward Suh and Andrew Myers [96].

CHAPTER 2

LANGUAGE-BASED CONTROL AND MITIGATION OF TIMING CHANNELS

Timing channels have long been a difficult and important problem for computer security. They can be used by adversaries as side channels or as covert channels to learn private information, including cryptographic keys and passwords.

This chapter introduces a complete and effective language-based method for controlling timing channels. An important contribution of this chapter is a system of simple, static annotations that provides just enough information about the underlying language implementation to enable accurate reasoning about timing channels. These annotations form a contract (formalized in this chapter) between the software (language) level and the hardware implementation. We design a novel type system based on the annotations, and formally prove that any well-typed program has no timing channel leakage, assuming that the hardware implementation obeys the contract.

2.1 Assumptions

Recall that throughout this dissertation, we follow the terms and assumptions defined in Section 1.2.1. We use the term *machine environment* to refer to all hardware state that is invisible at the language level but that is needed to predict timing. Examples include the compiler, operating system, data cache, instruction cache, and so on. We assume all software-level information and all components of the machine environment are associated with *security labels*, describing the

corresponding confidentiality. As a special case, commodity cache with no partition has all cache lines associated with the same label. In general, different partitions in a partitioned cache [88] can be associated with different labels. For hardware components such as a pipeline, we can time-multiplex the use of them according to different security labels.

2.2 A cross-domain timing interface

The syntax and semantics of programming languages and even of instruction set architectures (ISAs) intentionally hide information about execution time by abstracting away low-level implementation details. Doing so is beneficial for simpler reasoning about non-timing-related software properties, as well as for portability. But on the other hand, hiding timing information also makes it at best difficult to reason about timing channels arising from hardware features, such as the data cache, at the language level.

To track how information flows into the machine environment, but without concretely representing the hardware state, we propose a novel cross-domain interface in this dissertation. This interface consists of two labels with each command in the program. The first of these labels is the command's *read label* ℓ_r. The read label is an upper bound on the label of hardware state that affects the run time of the command. For example, the run time of a command with ℓ_r = L depends only on hardware state with label L or below. The second of these labels is the command's *write label* ℓ_w. The write label is a lower bound on the label of hardware state that the command can modify. For example, no machine environment with a label H can be modified during the execution of a command with ℓ_r = L.

$$
\begin{aligned}
e &::= n \mid x \mid e \ \texttt{op} \ e \\
c &::= \texttt{skip}_{[\ell_r,\ell_w]} \mid (x := e)_{[\ell_r,\ell_w]} \mid c; c \mid (\texttt{while} \ e \ \texttt{do} \ c)_{[\ell_r,\ell_w]} \\
&\quad \mid (\texttt{if} \ e \ \texttt{then} \ c_1 \ \texttt{else} \ c_2)_{[\ell_r,\ell_w]} \mid (\texttt{sleep} \ e)_{[\ell_r,\ell_w]}
\end{aligned}
$$

Figure 2.1: Syntax of the language.

Next, we introduce a core language with read and write labels. Formal definition and semantics of these labels are deferred to Section 2.3.6.

2.3 A language for controlling timing channels

Figure 2.1 gives the syntax for an imperative language with timing channel control. The novel elements—read and write labels—have already been introduced. Notice that the sequential composition command itself needs no timing labels. The syntax is mostly standard: it has assignments, sequential compositions, loops and branches. Command (`sleep` e) suspends execution of the program for the amount of time specified by e.

We present our semantics in a series of modular steps. We start with a *core semantics*, a largely standard semantics for a simple while-language, which ignores timing. Next, we develop an *abstracted full semantics* that describes the timing semantics of the language more accurately while abstracting away parameters that depend on the language implementation, including the hardware and the compiler.

$$\langle \texttt{skip}_{[\ell_r,\ell_w]}, m\rangle \to \langle \texttt{stop}, m\rangle \qquad \langle (\texttt{sleep}\ e)_{[\ell_r,\ell_w]}, m\rangle \to \langle \texttt{stop}, m\rangle$$

$$\frac{\langle c_1, m\rangle \to \langle \texttt{stop}, m'\rangle}{\langle c_1; c_2, m\rangle \to \langle c_2, m'\rangle} \qquad \frac{\langle c_1, m\rangle \to \langle c_1', m'\rangle \quad c_1' \neq \texttt{stop}}{\langle c_1; c_2, m\rangle \to \langle c_1'; c_2, m'\rangle}$$

$$\frac{\langle e, m\rangle \Downarrow v}{\langle (x := e)_{[\ell_r,\ell_w]}, m\rangle \to \langle \texttt{stop}, m[x \mapsto v]\rangle}$$

$$\frac{\langle e, m\rangle \Downarrow n \quad n \neq 0 \implies i = 1 \quad n = 0 \implies i = 2}{\langle (\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2)_{[\ell_r,\ell_w]}, m\rangle \to \langle c_i, m\rangle}$$

$$\frac{\langle e, m\rangle \Downarrow n \qquad n \neq 0}{\langle (\texttt{while}\ e\ \texttt{do}\ c)_{[\ell_r,\ell_w]}, m\rangle \to \langle c; (\texttt{while}\ e\ \texttt{do}\ c)_{[\ell_r,\ell_w]}, m\rangle}$$

$$\frac{\langle e, m\rangle \Downarrow n \qquad n = 0}{\langle (\texttt{while}\ e\ \texttt{do}\ c)_{[\ell_r,\ell_w]}, m\rangle \to \langle \texttt{stop}, m\rangle}$$

Figure 2.2: Core semantics of commands.

2.3.1 Core semantics

For expressions we use a standard big-step evaluation $\langle e, m\rangle \Downarrow v$ when expression e in memory m evaluates to value v. For commands (Figure 2.2), we write $\langle c, m\rangle \to \langle c', m'\rangle$ for the transition of command c in memory m to command c' in memory m'. Note that read and write labels are not used in these rules. The rules use `stop` as a syntactic marker of the end of computation. We distinguish `stop` from the command $\texttt{skip}_{[\ell_r,\ell_w]}$ because `skip` is a real command that may consume some measurable time (e.g., reading from the instruction cache), whereas `stop` is purely syntactic and takes no time at all. Since time is not part of the core semantics, `sleep` behaves like `skip`.

2.3.2 Abstracted full language semantics

The core semantics discussed so far ignores timing; the job of the full language semantics is to supply a complete description of timing so that timing channels can be precisely identified.

Writing down a full semantics as a set of transition rules would define the complete timing behavior of the language. But this semantics would be useful only for a particular language implementation on particular hardware. Instead, we permit *any* full semantics that satisfies a certain set of properties yet to be described. What is presented here is therefore a kind of abstracted full semantics in which only the key properties are fixed. This approach makes the results more general.

These key properties fall into two categories, which we call *faithfulness requirements* and *security requirements*. The faithfulness requirements (Section 2.3.5) are mostly straightforward; the security requirements (Section 2.3.6) are more subtle.

2.3.3 Configurations

Configurations in the full semantics have the form $\langle c, m, E, G\rangle$. As in the core semantics, c and m are the current program and memory. Component E is the machine environment, and G is the global clock. In general G can be measured in any units of time, but we interpret it as machine clock cycles hereafter. We write $\langle c, m, E, G\rangle \rightarrow \langle c', m', E', G'\rangle$ for evaluation transitions.

The full semantics of expression evaluation obviously also needs to be small-

step, but we choose a presentation style that elides the details of expression evaluation for simplicity.

As before, the machine environment E represents hardware state that may affect timing but that is not needed by the core semantics. Hardware components captured by E include the data cache and instruction cache, the branch prediction buffer, the translation lookaside buffer (TLB), and other low-level components. The machine environment might also include hidden state added by the compiler for performance optimization.

For example, if one considers only the timing effects of data and instruction caches, denoted by D and I respectively, E could be a configuration of the form $E = \langle D, I \rangle$.

Note that while both the memory m and the machine environment E can affect timing, only the memory affects program control flow. This is the reason to distinguish them in the semantics. The environment E can be completely abstract as long as the properties for the full semantics are satisfied. This separation also ensures that the core semantics is completely standard.

The separation of m and E also clarifies possibilities for hardware design. For instance, it is possible for confidential data to be stored securely in a public partition of E, but not in public memory (cf. Section 2.4.1).

2.3.4 Threat model

To evaluate whether the programming language achieves its security goals, we need to describe the power of the adversary in terms of the semantics. We as-

sociate an adversary with a security level ℓ_A bounding what information the adversary can observe directly. To represent the confidentiality of memory, we assume that an environment Γ maps variable names to security levels. If a memory location (variable) has security level ℓ that flows to ℓ_A (that is, $\ell \sqsubseteq \ell_A$), the adversary is able to see the contents of that memory location. By monitoring such a memory location for changes, the adversary can also measure the times at which the location is updated.

Two memories m_1 and m_2 are *ℓ-equivalent*, denoted $m_1 \sim_\ell m_2$, when they agree on the contents of locations at level ℓ and below:

$$m_1 \sim_\ell m_2 \triangleq \forall x \,.\, \Gamma(x) \sqsubseteq \ell \,.\, m_1(x) = m_2(x)$$

Intuitively, ℓ-equivalence of two memories means that an observer at level ℓ cannot distinguish these two memories.

Projected equivalence We define *projected equivalence* on memories to require equivalence of variables with *exactly* level ℓ:

$$m_1 \simeq_\ell m_2 \triangleq \forall x \,.\, \Gamma(x) = \ell \,.\, m_1(x) = m_2(x)$$

We assume there is a corresponding projected equivalence relation on machine environments. If two machine environments E_1 and E_2 have equivalent ℓ-projections, denoted $E_1 \simeq_\ell E_2$, then ℓ-level information that is stored in these environments is indistinguishable. The precise definition of projected equivalence depends on the hardware and perhaps the language implementation. For example, for a two-level partitioned cache containing some entries at level L and some at level H, two caches have equivalent H-projections if they contain the same cache entries in the H portion, regardless of the L entries.

Using projected equivalence it is straightforward to define ℓ-equivalence on machine environments:

$$E_1 \sim_\ell E_2 \triangleq \forall \ell' \sqsubseteq \ell \,.\, E_1 \simeq_{\ell'} E_2$$

2.3.5 Faithfulness requirements for the full semantics

The faithfulness requirements for the full semantics comprise four properties: adequacy, deterministic execution, sequential composition, as well as accurate sleep duration.

Adequacy specifies that the core semantics and the full semantics describe the same executions: for any transition in the core semantics there is a matching transition in the full semantics and vice versa.

Property 1 (Adequacy of core semantics) $\forall m, c, c', E, G$.

$$(\exists E', G' \,.\, \langle c, m, E, G\rangle \to \langle c', m', E', G'\rangle) \Leftrightarrow \langle c, m\rangle \to \langle c', m'\rangle$$

We also require that the full semantics be deterministic, which means that the machine environment E completely captures the possible influences on timing.

Property 2 (Deterministic execution) $\forall m, c, E, G$.

$$\langle c, m, E, G\rangle \to \langle c_1, m_1, E_1, G_1\rangle \wedge \langle c, m, E, G\rangle \to \langle c_2, m_2, E_2, G_2\rangle \implies E_1 = E_2 \wedge G_1 = G_2$$

Since the core semantics is already deterministic, determinism of the machine environment and time components suffices.

Sequential composition must correctly accumulate time and propagate the machine environment.

Property 3 (Sequential composition)

1. $\forall c_1, c_2, m, E, G$.

$\langle c_1, m, E, G\rangle \rightarrow \langle \texttt{stop}, m', E', G'\rangle \Leftrightarrow \langle c_1; c_2, m, E, G\rangle \rightarrow \langle c_2, m', E', G'\rangle$

2. $\forall c_1, c_2, c_1', m, E, G$ *such that* $c_1' \neq \texttt{stop}$.

$\langle c_1, m, E, G\rangle \rightarrow \langle c_1', m', E', G'\rangle \Leftrightarrow \langle c_1; c_2, m, E, G\rangle \rightarrow \langle c_1'; c_2, m', E', G'\rangle$

Finally, the `sleep` command must take the correct amount of time. When its argument is negative, it is assumed to take no time.

Property 4 (Accurate sleep duration) $\forall n, m, E, G, \ell_r, \ell_w$.

$$\langle (\texttt{sleep}\ n)_{[\ell_r, \ell_w]}, m, E, G\rangle \rightarrow \langle \texttt{stop}, m, E', G'\rangle \Rightarrow G' = G + \max(n, 0)$$

Discussion The faithfulness requirements are mostly straightforward. The assumption of determinacy might sound unrealistic for concurrent execution. But if information leaks through timing because some other thread preempts this one, the problem is in the scheduler or in the other thread, not in the current thread. Deterministic time is realistic if we interpret G as the number of clock cycles the current thread has used.

Property 5 (Write label) *Given a labeled command $c_{[\ell_r,\ell_w]}$, and a level ℓ such that $\ell_w \not\sqsubseteq \ell$*

$$\forall m, E, G \,.\, \langle c_{[\ell_r,\ell_w]}, m, E, G\rangle \to \langle c', m', E', G'\rangle \implies E \simeq_\ell E'$$

Property 6 (Read label) *Given any command $c_{[\ell_r,\ell_w]}$,*

$$\begin{aligned}\forall m_1, m_2, E_1, E_2, G \,.\, (&\forall x \in \mathbf{Vars}_1(c_{[\ell_r,\ell_w]}) \,.\, m_1(x) = m_2(x))\\ &\wedge E_1 \sim_{\ell_r} E_2\\ &\wedge \langle c_{[\ell_r,\ell_w]}, m_1, E_1, G\rangle \to \langle c_1, m_1', E_1', G_1\rangle\\ &\wedge \langle c_{[\ell_r,\ell_w]}, m_2, E_2, G\rangle \to \langle c_2, m_2', E_2', G_2\rangle \implies G_1 = G_2\end{aligned}$$

Property 7 (Single-step machine-environment noninterference) *Given any labeled command $c_{[\ell_r,\ell_w]}$, and any level ℓ,*
$\forall m_1, m_2, E_1, E_2, G_1, G_2 \,.\, m_1 \sim_\ell m_2 \wedge E_1 \sim_\ell E_2$

$$\begin{aligned}&\wedge \langle c_{[\ell_r,\ell_w]}, m_1, E_1, G_1\rangle \to \langle c_1, m_1', E_1', G_1'\rangle\\ &\wedge \langle c_{[\ell_r,\ell_w]}, m_2, E_2, G_2\rangle \to \langle c_2, m_2', E_2', G_2'\rangle \implies E_1' \sim_\ell E_2'\end{aligned}$$

Figure 2.3: Security requirements.

2.3.6 Security requirements for the full semantics

For security, the full semantics also must satisfy certain properties to ensure that read and write labels accurately describe timing. These properties are specified as constraints on the full semantics that must hold after each evaluation step. In the formalization of these properties, we quantify over labeled commands with the form $c_{[\ell_r,\ell_w]}$: that is, all commands except sequential composition.

Write labels The write label ℓ_w is the lower bound on the parts of the machine environment that a single evaluation step modifies. Property 5 in Figure 2.3 formalizes the requirements on the machine environment: executing a labeled command $c_{[\ell_r,\ell_w]}$ cannot modify parts of the environment at levels to which ℓ_w does not flow.

Example Consider program $\texttt{sleep}(h)_{[\ell_r,\mathrm{H}]}$ under the two-level security lattice $\mathrm{L} \sqsubseteq \mathrm{H}$. This command is annotated with the write label H. The only level ℓ such that $\ell_w \not\sqsubseteq \ell$ is $\ell = \mathrm{L}$. In this case, Property 5 requires that an execution of $\texttt{sleep}(h)_{[\ell_r,\mathrm{H}]}$ does not modify L parts of the machine environment.

Consider program $\texttt{sleep}(h)_{[\ell_r,\mathrm{L}]}$ which has write label L. Because there is no security level ℓ such that $\mathrm{L} \not\sqsubseteq \ell$, Property 5 does not constrain the machine environment for this command.

Read labels The read label ℓ_r of a command specifies which parts of the machine environment may affect the time necessary to perform the single next evaluation step. For a compound command such as `if` or `while`, this time does not include time spent in subcommands.

Property 6 in Figure 2.3 formalizes the requirement that read labels accurately capture the influences of the machine environment. This formalization uses the $\mathbf{Vars}_1$ function, which identifies the part of memory that may affect the timing of the next evaluation step—that is, a set of variables. We need $\mathbf{Vars}_1$ because parts of the memory can also affect timing, such as e in `sleep` (e). A simple syntactic definition of $\mathbf{Vars}_1$ conservatively approximates the timing influences of memory, but a more precise definition might depend on particularities of the hardware implementation. For `skip`, this set is empty; for $x := e$ and `sleep` (e), the set consists of x and all variables in expression e; for `if` e `then` c_1 `else` c_2 and `while` e `do` c, it contains only variables in e and excludes those in subcommands, since only e is evaluated during the next step.

In the definition in Figure 2.3, equality of G_1 and G_2 means that a single step takes exactly the same time. Both configurations take the same time, because

m_1 and m_2 must agree on all variables x that are evaluated in this step. This expresses our assumption that values of variables other than those explicitly evaluated in a single step cannot influence its timing. Machine environments E_1 and E_2 are required to be ℓ_r-equivalent, to ensure that parts of the machine environment other than those at ℓ_r and below also cannot influence its timing.

Consider command `sleep` $(h)_{[\mathrm{L},\ell_w]}$ with read-label $\ell_r = \mathrm{L}$, with respect to all possible pairs of memories m_1, m_2 and machine environments E_1, E_2. Whenever $m_1(h)$ and $m_2(h)$ have different values, Property 6 places no restrictions on the timing of this command regardless of E_1, E_2. When $m_1(h) = m_2(h)$, we require that if E_1 and E_2 are L-equivalent, the resulting time must be the same. To satisfy such a property, the H parts of the machine environment cannot affect the evaluation time.

Single-step noninterference Property 5 specifies which parts of the machine environment can be modified. However, it does not say anything more about the nature of the modifications. For example, consider a three-level security lattice $\mathrm{L} \sqsubseteq \mathrm{M} \sqsubseteq \mathrm{H}$, and a command $(x := y)_{[\mathrm{M},\mathrm{M}]}$, where both the read label and write label are M. Property 5 requires that no modifications to L parts of the environment are allowed, but modifications to the M level are not restricted. This creates possibilities for insecure modifications of machine environments when H-parts of the machine environment propagate into the M-parts. To control such propagation, we introduce Property 7 in Figure 2.3. Note that here level ℓ is independent of read or write labels.

2.4 A sketch of secure hardware

To illustrate how the requirements for the full language semantics enable secure hardware design, we sketch two possible ways for a design of cache and TLB to realize Properties 5–7, and reason about their security informally. In Chapter 3, we present a formal method for verifying more complex designs (e.g., a complete MIPS processor).

For simplicity, we assume that the two-point label lattice $L \sqsubseteq H$ throughout this section.

We start with a standard single-partition data cache similar to current commodity cache designs and then explore a more sophisticated partitioned cache similar to that in prior work [88].

2.4.1 Choosing machine environments

The machine environment does not need to include all hardware state. It should be precise enough to ensure that equivalent commands take the same time in equal environments, but no more precise. Including state with no effect on timing leads to overly conservative security enforcement that hurts performance.

For example, consider a data cache, usually structured as a set of cache lines. Each cache line contains a tag, a data block and a valid bit. Let us compare two possible ways to describe this as a machine environment: a more precise modeling of all three fields—a set of triples $\langle$tag, data block, valid bit$\rangle$—versus a coarser modeling of only the tags and valid bits—a set of pairs $\langle$tag, valid bit$\rangle$.

The coarse-grained abstraction of data cache state is adequate to predict execution time, since for most cache implementations, the contents of data blocks do not affect access time at all. The fine-grained abstraction does not work as well. For example, consider the command `h := h'` occurring in a low context. That is, variables `h` and `h'` are confidential, but the fact that the assignment is happening is not. With the fine-grained abstraction, the low part of the cache cannot record the value of `h` if Property 7 is to hold, because the low-equivalent memories m_1 and m_2 appearing in its definition may differ on the value of `h'`. However, with the coarse-grained abstraction, the location `h` can be stored in low cache, because Property 7 holds without making the value of `h'` part of the machine environment.

The coarse-grained abstraction shows that high variables can reside in low cache without hurting security in at least some circumstances. This treatment of cache is quite different from the treatment of memory, because public memory cannot hold confidential data. Without the formalization of Property 7, it would be difficult to reason about the security of this treatment of cache. Yet this insight is important for performance: otherwise, code with a low timing label cannot access high variables using cache.

2.4.2 Realization on standard hardware

At least some standard CPUs can satisfy the security requirements (Properties 5–7). Intel's family of Pentium and Xeon processors has a "no-fill" mode in which accesses are served directly from memory on cache misses, with no evictions from nor filling of the data cache.

Our approach can be implemented by treating the whole cache as low, and therefore disallowing cache writes from high contexts. For each block of instructions with $\ell_w = \mathrm{H}$, the compiler inserts a no-fill start instruction before, and a no-fill exit instruction after.

It is easy to verify that Properties 5–7 hold, as follows:

Property 5 For commands with $\ell_w = \mathrm{L}$, this property is vacuously true since there is no ℓ such that $\mathrm{L} \not\sqsubseteq \ell$. Commands with $\ell_w = \mathrm{H}$ are executed in "no-fill" mode, so the result is trivial.

Property 6 Since there is only one (L) partition, $E_1 \sim_{\ell_r} E_2$ is equivalent to $E_1 = E_2$. The property can be verified for each command. For instance, consider command `sleep` $(e)_{[\ell_r,\ell_w]}$. The condition $\forall x \in \mathbf{Vars}_1(c_{[\ell_r,\ell_w]}).m_1(x) = m_2(x)$ ensures that $m_1(e) = m_2(e)$. Thus, this command is suspended for the same time. Moreover, since $E_1 = E_2$, cache access time must be the same according to Property 2. So, we have $G_1 = G_2$.

Property 7 We only need to check the L partition, which can be verified for each command. For instance, consider command `sleep` $(e)_{[\ell_r,\ell_w]}$. When $\ell_w = \mathrm{H}$, the result is true simply because the cache is not modified. Otherwise, the same addresses (variables) are accessed. Since initial cache states are equivalent, identical accesses yields equivalent cache states.

2.4.3 A more efficient realization

A more efficient hardware design might partition both the cache(s) and the TLB according to security labels. Let us assume both the cache and TLB are equally, statically partitioned into two parts: L and H. The hardware accesses different parts as directed by a timing label that is provided from the software level. Here we focus on the correctness of this hardware design; a simulation of this design is discussed in Section 6.2.

One subtle issue is consistency, since data can be stored in both the L and the H partitions. We avoid inconsistency by keeping only one copy in the cache and TLB. In any CPU pipeline stage that accesses memory when the timing label is H, both H and L partitions are searched. If there is a cache miss, data is installed in the H partition. When the timing label is L, only the L partition is searched. However, to preserve consistency, instead of fetching the data from next level or memory, the controller moves the data from the H partition if it already exists there. To satisfy Property 6, the hardware ensures this process takes the same time as a cache miss.

We can informally verify Properties 5–7 for this design as well:

Property 5 When the write label is L, this property holds trivially because there is no label such that $\mathrm{L} \not\sqsubseteq \ell$. When the write label is H, a new entry is installed only in the H partition, so $E \sim_{\mathrm{L}} E'$.

Property 6 The premise of Property 6 ensures that all variables evaluated in a single step have identical values, so any variation in execution time is due to

the machine environment. When the read label is H, $E_1 \sim_{\mathrm{H}} E_2$ ensures that the machine environments are identical; therefore, the access time is also identical. When the read label is L, the access time depends only on the existence of the entry in L-cache/TLB. Even if the data is in the H partition, the load time is the same as if there were an L-partition miss.

Property 7 This requirement requires noninterference for a single step. Contents of the H partition can affect the L part in the next step only when data is stored in the H partition and the access has a timing label L. Since data is installed into the L part regardless of the state of the H partition, this property is still satisfied.

Discussion on formal proof and multilevel security We have discussed efficient hardware for a two-level label system. Verification of multilevel security hardware is more challenging. In Chapter 3, we present SecVerilog to formally verify multilevel security hardware.

2.5 A type system for controlling timing channels

Next, we present the security type system for our language. We show that the type system eliminates all timing channels in a well-type program, with the assumption that Properties 1–7 hold.

$$\frac{pc \sqsubseteq \ell_w}{\Gamma, pc, \tau \vdash \texttt{skip}_{[\ell_r,\ell_w]} : \tau \sqcup \ell_r} \text{ T-SKIP}$$

$$\frac{\Gamma \vdash e : \ell \qquad pc \sqsubseteq \ell_w \qquad \ell \sqcup pc \sqcup \tau \sqcup \ell_r \sqsubseteq \Gamma(x)}{\Gamma, pc, \tau \vdash x := e_{[\ell_r,\ell_w]} : \Gamma(x)} \text{ T-ASGN}$$

$$\frac{\Gamma \vdash e : \ell \qquad pc \sqsubseteq \ell_w}{\Gamma, pc, \tau \vdash (\texttt{sleep}\ (e))_{[\ell_r,\ell_w]} : \tau \sqcup \ell \sqcup \ell_r} \text{ T-SLEEP}$$

$$\frac{\Gamma \vdash e : \ell \qquad pc \sqsubseteq \ell_w \quad \Gamma, \ell \sqcup pc, \ell \sqcup \tau \sqcup \ell_r \vdash c_i : \tau_i \quad i = 1, 2}{\Gamma, pc, \tau \vdash (\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2)_{[\ell_r,\ell_w]} : \tau_1 \sqcup \tau_2} \text{ T-IF}$$

$$\frac{\Gamma \vdash e : \ell \quad pc \sqsubseteq \ell_w \quad \ell \sqcup \tau \sqcup \ell_r \sqsubseteq \tau' \quad \Gamma, \ell \sqcup pc, \tau' \vdash c : \tau'}{\Gamma, pc, \tau \vdash (\texttt{while}\ e\ \texttt{do}\ c)_{[\ell_r,\ell_w]} : \tau'} \text{ T-WHILE}$$

$$\frac{\Gamma, pc, \tau \vdash c_1 : \tau_1 \qquad \Gamma, pc, \tau_1 \vdash c_2 : \tau_2}{\Gamma, pc, \tau \vdash c_1; c_2 : \tau_2} \text{ T-SEQ}$$

Figure 2.4: Typing rules: commands.

2.5.1 Security type system

Typing rules for expressions have form $\Gamma \vdash e : \ell$ where Γ is the security environment (a map from variables to security labels), e is the expression, and ℓ is the type of the expression. The rules are standard [72] and we omit them here. Typing rules for commands, in Figure 3.5, have form $\Gamma, pc, \tau \vdash c : \tau'$. Here pc is the usual program-counter label [72], τ is the timing *start-label*, and τ' is the timing *end-label*. The timing start- and end-labels bound the level of information that flows into timing before and after executing c, respectively. When timing end-labels are not relevant, we write $\Gamma, pc, \tau \vdash c$. We use $\Gamma \vdash c$ to denote $\Gamma, \bot, \bot \vdash c$.

All rules enforce the constraint $\tau \sqsubseteq \tau'$ because timing dependencies accumu-

late as the program executes. Every rule also propagates the timing end-labels of subcommands. This can be seen most clearly in the rule for sequential composition (T-SEQ): the end-label from c_1 is the start-label for c_2.

All remaining rules require $pc \sqsubseteq \ell_w$. This restriction, together with Property 5, ensures that no confidential information about control flow leaks to the low parts of the machine environment. We do not require $\tau \sqsubseteq \ell_w$ because we assume the adversary cannot directly observe the timing of updates to the machine environment. This assumption is reasonable since the ISA gives no way to check whether a given location is in cache.

Rule (T-SKIP) takes the read label ℓ_r into account in its timing end-label. The intuition is that reading from confidential parts of the machine environment should be reflected in the timing end-label.

Rule (T-ASGN) for assignments $x := e$ requires $\ell \sqcup pc \sqcup \tau \sqcup \ell_r \sqsubseteq \Gamma(x)$, where ℓ is the level of the expression. The condition $\ell \sqcup pc \sqsubseteq \Gamma(x)$ is standard. We also require $\tau \sqcup \ell_r \sqsubseteq \Gamma(x)$, to prevent information from leaking via the timing of the update, from either the current time or the machine environment. The timing end-label is set to $\Gamma(x)$, bounding all sources of timing leaks.

Notice that the write label ℓ_w is independent of the label on x. The reason is that ℓ_w is the interface for software to tell hardware which state may be modified. A low write label on an assignment to a high variable permits the variable to be stored in low cache.

Because `sleep` has no memory side effects, rule (T-SLEEP) is slightly simpler than that for assignments; the timing end-label conservatively includes all sources of timing information leaks.

Rule (T-IF) restricts the environment in which branches c_1 and c_2 are type-checked. As is standard, the program-counter label is raised to $\ell \sqcup pc$. The timing start-labels are also restricted to reflect the effect of reading from the ℓ_r-parts of the machine environment and of the branching expression. Rule (T-WHILE) imposes similar conditions on end-label τ', except that τ' can also be used as both start- and end-labels for type-checking the loop body.

We have seen that for security, the write label of a command must be higher than the label of the program counter. There is no corresponding restriction on the read label of a command. The hardware may be able to provide better performance if a higher read label is chosen. For instance, in most cache designs, reading from the cache changes its state. The cache can only be used when $\ell_r = \ell_w$, so this condition should be satisfied for best performance.

2.5.2 Machine-environment noninterference

An important property of the type system is that it guarantees machine environment noninterference. This property requires execution to preserve low-equivalence of memory and machine environments.

Theorem 1 (Memory and machine-environment noninterference)

$$\begin{aligned}\forall E_1, E_2, m_1, m_2, G, c, \ell \,.\, \Gamma \vdash c \wedge m_1 \sim_\ell m_2 \wedge E_1 \sim_\ell E_2 \\ \wedge \langle c, m_1, E_1, G\rangle \rightarrow^* \langle \texttt{stop}, m_1', E_1', G_1\rangle \\ \wedge \langle c, m_2, E_2, G\rangle \rightarrow^* \langle \texttt{stop}, m_2', E_2', G_2\rangle \\ \implies m_1' \sim_\ell m_2' \wedge E_1' \sim_\ell E_2'\end{aligned}$$

Theorem 1 guarantees the adversary obtains no information by observing public parts of the memory and machine environments. Therefore, for any well-typed program, information leakage via storage channels and machine environment is eliminated.

More importantly, the type system guarantees that a well-typed program leaks no information via timing channels. To formalize this property, we start from *observable assignment events*.

Observable assignment events As discussed in Section 2.3.4, an adversary at level ℓ_A observes memory, including timing of updates to memory, at levels up to ℓ_A. To formally define adversary observations, we refine our presentation of the language semantics with observable assignment events.

Let $\alpha \in \{(x, v, t), \epsilon\}$ range over observable events, which can be either an assignment to variable x of value v at time t, or an empty event ϵ. An event (x, v, G') is generated by assignment transitions $\langle x := e, m, E, G\rangle \rightarrow \langle \texttt{stop}, m', E', G'\rangle$, where $\langle m, e\rangle \Downarrow v$, and by all transitions whose derivation includes a subderivation of such a transition.

We write $\langle c, m, E, G\rangle \Rrightarrow (\mathbf{x}, \mathbf{v}, \mathbf{t})$ if configuration $\langle c, m, E, G\rangle$ produces a sequence of events $(\mathbf{x}, \mathbf{v}, \mathbf{t}) = (x_1, v_1, t_1) \ldots (x_n, v_n, t_n)$ and reaches a final configuration $\langle \texttt{stop}, m', E', G'\rangle$ for some m', E', G'.

ℓ_A-observable events An event (x, v, t) is observable to the adversary at level ℓ_A when $\Gamma(x) \sqsubseteq \ell_A$. Given a configuration $\langle c, m, E, G\rangle$ such that $\langle c, m, E, G\rangle \Rrightarrow (\mathbf{x}, \mathbf{v}, \mathbf{t})$, we write $\langle c, m, E, G\rangle \Rrightarrow_{\ell_A} (\mathbf{x}', \mathbf{v}', \mathbf{t}')$ for the longest subsequence of $(\mathbf{x}, \mathbf{v}, \mathbf{t})$ such that for all events (x_i, v_i, t_i) in $(\mathbf{x}', \mathbf{v}', \mathbf{t}')$ it holds that $\Gamma(x_i) \sqsubseteq \ell_A$.

For example, for program $l_1 := l_2; h_1 := l_1$, the H-adversary observes two assignments: $\langle c, m, E, G\rangle \Rrightarrow_{\mathrm{H}} (l_1, v_1, t_1), (h_1, v_2, t_2)$ for some v_1, t_1, v_2 and t_2. For the L-adversary, we have $\langle c, m, E, G\rangle \Rrightarrow_{\mathrm{L}} (l_1, v_1, t_1)$, which does not include the assignment to h_1.

Timing-sensitive noninterference Now we are ready to state the most important property of the type system. That is, all well-typed programs are timing channel free.

Theorem 2 (Timing-sensitive noninterference)

$$\begin{aligned}\forall E_1, E_2, m_1, m_2, G, c, \ell \,.\, \Gamma \vdash c \wedge m_1 \sim_\ell m_2 \wedge E_1 \sim_\ell E_2 \\ \wedge \langle c, m_1, E_1, G\rangle \Rrightarrow_\ell (\mathbf{x_1}, \mathbf{v_1}, \mathbf{t_1}) \wedge \langle c, m_2, E_2, G\rangle \Rrightarrow_\ell (\mathbf{x_2}, \mathbf{v_2}, \mathbf{t_2}) \\ \Longrightarrow \mathbf{x_1} = \mathbf{x_2} \wedge \mathbf{v_1} = \mathbf{v_2} \wedge \mathbf{t_1} = \mathbf{t_2}\end{aligned}$$

In other words, given two different secrets, an adversary who observes memory, including timing of updates to memory, at levels up to ℓ, will always observe the same observations regardless of the secrets.

Proofs For technical reasons, we defer the proofs of Theorem 1 and Theorem 2 until Chapter 5, where these results become corollaries of more general results presented in that chapter.

A note on termination The definition of memory and machine noninterference in Theorem 1 is presented in the batch-style termination-insensitive

form [5]. Such definitions are simple but ordinarily limit one's results to programs that eventually terminate. Because termination channels are a special case of timing channels, using a batch-style definition is not fundamentally limiting here.

2.6 Related work

Control of internal timing channels has been studied from different perspectives, and several papers have explored a language-based approach. Low observational determinism [93, 37] can control these channels by eliminating dangerous races.

External timing channels are harder to control. Much prior language-based work on external timing channels uses simple, implicit models of timing, and no previous work fully addresses indirect dependencies. Type systems have been proposed to prevent timing channels [86], but are very restrictive. Often (e.g., [86, 78, 73, 76]) timing behavior of the program is assumed to be accurately described by the number of steps taken in an operational semantics. This assumption does not hold even at the machine-language level, unless we fully model the hardware implementation in the operational semantics and verify the entire software and hardware stack together. Our approach adds a layer of abstraction so software and hardware can be designed and verified independently.

Some previous work uses program transformation to remove indirect dependencies, though only those arising from data cache. The main idea is to equalize the execution time of different branches, but a price is paid in expressiveness, since these languages either rule out loops with confidential guards (as

in [4, 34, 10]), or limit the number of loop iterations [56, 18]. These methods do not handle all indirect timing dependencies; for example, the instruction cache is not handled, so verified programs remain vulnerable to other indirect timing attacks [87, 3, 2].

Secure multi-execution [22, 42] provides timing-sensitive noninterference yet is probably less restrictive than the prior approaches discussed above. The security guarantee is weaker than in our approach: that the number of instructions executed, rather than the time, leaks no information for incomparable levels. Extra computational work is also required per security level, hurting performance, and no quantitative bound on leakage is obtained.

Though security cannot be enforced purely at the hardware level, hardware techniques have been proposed to mitigate timing channels. Targeting cache-based timing attacks, both static [67] and dynamic [88, 89, 50] mechanisms, based on the idea of partitioned cache, have been proposed. Such designs are ad hoc and hard to verify against other attacks. For example, Kong et al. [44] show vulnerabilities in Wang's cache design [88]. SecVerilog, the new hardware description language introduced in Chapter 3, and languages designed by Li et al. [49, 48] are statically verifiable hardware description languages for building hardware that is information-flow secure by construction. We defer a detailed comparison to Chapter 3.

CHAPTER 3

A HARDWARE DESIGN LANGUAGE FOR TIMING-SENSITIVE INFORMATION-FLOW SECURITY

Chapter 2 shows that a new timing contract can communicate information flows between the software and hardware, enabling timing channels to be rigorously controlled in the entire computer system. Left open is the question of how to design complex hardware that correctly enforces its part of the contract.

This chapter fills in the critical missing piece, offering a method for designing hardware that correctly and precisely enforces secure information flow. Our approach is based on a new hardware description language, called SecVerilog, that statically checks information flows within hardware using a security type system. This hardware design method enables computing systems in which all forms of information flow are tracked, including explicit flows, implicit flows, and flows via timing channels.

3.1 Background and approach

3.1.1 Information flow control in hardware

Recall that information flow control aims to ensure that all information flows in a system respect a security policy. For this purpose, information in the system is associated with a *security level* drawn from a lattice $\mathcal{L}$ whose partial ordering $\sqsubseteq$ specifies which information flows are allowed. As defined in Section 1.2.1, a lattice with two security levels L (low, public) and H (high, secret) can be used

to forbid information labeled as H from flowing into L ($H \not\sqsubseteq L$) while allowing the other direction ($L \sqsubseteq H$).

The goal of SecVerilog is to enforce fine-grained information flow control for hardware designs in a statically verifiable fashion. With SecVerilog, hardware designers specify hardware-level information flow policies by annotating wires and registers with security labels and specifying a security lattice. Then, the SecVerilog type system statically checks and verifies timing-sensitive information flow properties within hardware at design time. While we use a simple lattice with two security levels (L and H) in our examples, the approach applies to an arbitrary security lattice.

3.1.2 Threat model

We follow the threat model in Section 2.3.4. In particular, we assume a *software-level adversary*, who can observe *all* information at or below a certain security level that we will call low (L). We assume the adversary may either directly or indirectly (e.g., by measuring the timing of L instructions) observe machine environment state at or below the level L. Hence, both storage and timing channels [47] are considered.

Moreover, we target synchronous circuits driven by a fixed-frequency clock. We assume the software-level adversary has no physical access to the hardware, and we do not consider physical attacks such as directly tapping internal circuits. Therefore, the adversary may only observe machine environment at the granularity of a clock cycle. In other words, the instable circuit state between clock ticks is invisible to the adversary.

```
1 if (h1)[L]
2   h2:=l1;[H]
3 else
4   h2:=l2;[H]
5 l3:=l1;[L]
```

Security policy on hardware designs:

1) The high partition cannot affect the timing of instructions with label L,
2) the low partition cannot be modified when the timing label is H, and
3) the contents of the high partition cannot affect those of the low partition.

Figure 3.1: An example of full-system timing channel control. The well-typed program on the left is secure if the hardware enforces the security policy on the right.

Furthermore, we do not consider side channels that require physical proximity, such as power consumption analysis.

3.1.3 Controlling timing channels in hardware

The ability to verifiably control fine-grained information flow in hardware can enhance security in many applications. One notable example, and a focus of this chapter, is designing efficient hardware that controls timing channels.

Our goal is an efficient hardware design that enforces the complex security policy required by the full-system timing channel control mechanism proposed in Chapter 2. Recall that in this approach, the security of the whole system rests on a concise contract between the software and hardware, provably controlling timing channels if both meet their requirements. For example, the code fragment in Figure 3.1 illustrates a well-typed program, in which timing labels are shown in brackets; `h1` and `h2` are confidential, and other variables are public[1].

[1]The general contract in Chapter 2 uses two timing labels, called the *read* and *write* labels. SecVerilog is expressive enough to verify the general contract, which is implemented in our verified MIPS processor (Section 6.3). For simplicity, we assume these two labels are equal in most of examples.

```
1  reg[18:0]{L} tag0[256],tag1[256];
2  reg[18:0]{H} tag2[256],tag3[256];
3  wire[7:0]{L} index;
4  //Par(0)=Par(1)=L Par(2)=Par(3)=H
5  wire[1:0]{Par(way)} way;
6  wire[18:0]{Par(way)} tag_in;
7  wire{Par(way)} write_enable;
8
9  always @(posedge clock) begin
10   if (write_enable) begin
11     case (way)
12     0: tag0[index]=tag_in;
13     1: tag1[index]=tag_in;
14     2: tag2[index]=tag_in;
15     3: tag3[index]=tag_in;
16     endcase
17   end
18 end
```

(a) SecVerilog code for cache tags

```
1  wire{L} isLoad,isStore;
2  wire{L} hit0,hit1; // hitX: 1 iff way
3  wire{H} hit2,hit3; // X gets a cache hit
4  //LH(0)=L  LH(1)=H
5  wire{LH(timingLabel)} stall, hit, timingLabel;
6  reg[2:0]{LH(timingLabel)} dFsmState;
7
8  assign stall = ((isLoad | isStore) &
9    (~hit | (dFsmState != DFSM_IDLE)));
10 assign hit = (timingLabel == 0) ?
11   ((hit0|hit1)?1:0):((hit0|hit1|hit2|hit3)?1:0);
12 ...
13 case (dFsmState)
14   DFSM_IDLE: begin
15     // load hit
16     if (isLoad && hit) begin
17       dFsmState <= DFSM_IDLE;// nonblocking
18   ...
19 endcase
```

(b) SecVerilog code for a cache controller

Figure 3.2: SecVerilog extends Verilog with security label annotations (shaded in gray).

Here, the existence of `h1` in data cache, rather than the value of `h1`, can affect the execution time of line 1. Hence, line 1 has a timing label of L. The benefit of these timing labels is that only the timing of instructions with H timing labels needs to be controlled and mitigated at software level, as long as the security policy in Figure 3.1 is enforced on hardware.

3.1.4 Example: secure cache design

Designing hardware to meet the complex security policy in Figure 3.1 is challenging. As an illustration, we consider designing a secure cache, statically partitioned between security levels L (low) and H (high) as proposed in prior work [67, 88]. The L and H partition correspond to the L and H machine environment respectively.

Figure 3.2(a) presents a simplified fragment of SecVerilog code to update cache tags. For now, ignore the shaded annotations. This design logically partitions a 4-way set-associative cache so that ways 0 and 1 (`tag0` and `tag1`) are used as the L partition, and the other ways (`tag2` and `tag3`) are used as the H partition. The code writes a new cache tag to a way specified by `way` when `write_enable` is asserted.

This simple example shows the intricacy of correctly enforcing the aforementioned security policy in hardware. First, `tag_in` must not contain high information when `way` is 0 or 1, to prevent the H partition from affecting the state of the L partition (`tag0` and `tag1`). Second, `write_enable`, which controls whether a write occurs, cannot be influenced by high information when `way` is 0 or 1 (an instance of implicit flows [72]). Verifying these restrictions is tricky since the cache partition that `tag_in` and `write_enable` belong to can change at run time.

More challenging is to enforce secure timing: the H partition cannot affect the timing of instructions with timing label L. A simplified fragment of the SecVerilog code for the cache controller is shown in Figure 3.2(b), where `timingLabel` represents the timing label of a cache access, propagated from the software level, `hit`i ($0 \leq i \leq 3$) indicates if way i gets a cache hit, and the `stall` signal indicates when a cache access completes.

Since the `stall` signal affects the execution time of an instruction, a secure design must ensure that only the L partition can affect the execution time when `timingLabel` is 0 (encoding L). Verifying this property is difficult, since the cache controller may access H data even when `timingLabel` is 0 (e.g., to execute line 1 of the example in Figure 3.1). Perhaps counterintuitively, this access is se-

cure, because timing may be affected by the *existence* of H data in the cache but not by the value of the data. Moreover, the `hit` and `dFsmState` signals, which affect `stall` (line 8), are shared across both cache partitions. A secure design must ensure that no information leaks through these shared variables, which is difficult since their uses are spread across multiple statements (lines 13–19 only show a snippet).

3.1.5 The SecVerilog approach

SecVerilog extends Verilog with the ability to give each variable a *label* that specifies the security level of the variable. In Figure 3.2, these labels are the shaded annotations, which indicate, e.g., that variables `tag0` and `tag1` are labeled L whereas `tag2` and `tag3` are labeled H. Using these annotations, the SecVerilog type system automatically verifies information flow properties of Verilog code at compile time.

Programming languages that provide the ability to label variables have been developed before [9, 59, 75], but their labels are not expressive enough to handle practical hardware designs where resources need to be shared across security levels. In effect, the security levels might be changed at run time. We use dependent types to address this challenge.

Consider the example in Figure 3.2(a). The labels of `way`, `write_enable`, and `tag_in` depend on which cache way is being accessed. In fact, we observe that a precise dependent label can be assigned to these variables without any change to the Verilog code. The proper label is `Par(way)`, where the name `Par` denotes a type-level function that maps 0 and 1 to level L, and 2 and 3 to level H (concisely,

`Par` = $\{0 \mapsto \mathrm{L}, 1 \mapsto \mathrm{L}, 2 \mapsto \mathrm{H}, 3 \mapsto \mathrm{H}\}$). Intuitively, these dependent labels express a lightweight invariant on variables (e.g., when `way` is 0, `write_enable` must have level L).

For the example in Figure 3.2(b), `stall`, `hit` and `dFsmState` can be labeled with `LH(timingLabel)` where `LH` = $\{0 \mapsto \mathrm{L}, 1 \mapsto \mathrm{H}\}$ to ensure that they can be affected only by the low partition when `timingLabel` is 0.

Such invariants can be maintained by the type system described in Section 3.3. For instance, to ensure that the explicit flow from `tag_in` to `tag0` at line 12 in Figure 3.2(a) is secure, the type system generates a proof obligation (`way` $= 0 \Rightarrow$ `Par(way)` $\sqsubseteq L$), meaning that when `way` is 0, information flow from `tag_in` (with label `Par(way)`) to `tag0` (with label L) is permissible. This proof obligation can easily be discharged by an external solver.

The soundness of our type system (Section 3.4) guarantees that all security violations are detected at compile time. For example, consider the case when `timingLabel` is 0 in line 11 in Figure 3.2(b). If the H partition, such as variable `hit2`, were accessed in that case, an error would be reported because the type system would generate an invalid proof obligation: (`timingLabel` $= 0) \Rightarrow H \sqsubseteq$ `LH(timingLabel)`.

3.1.6 Benefits over previous approaches

Our approach enjoys several benefits compared with prior efforts with verifiable information-flow security for hardware [84, 49, 48]. First, verification is done at compile time, avoiding run-time overhead and detecting errors at an

Program	`Prog`	$::= B_1 \ldots B_n$
Thread	B	$::=$ `always` $@(\gamma)\ c$
Trigger	γ	$::=$ `posedge clock` | `negedge clock` | $\vec{v}$
Cmds	c	$::=$ $\texttt{skip}_\eta$ | `begin` $c_1; \ldots; c_n;$ `end`
		| $v =_\eta e$ | $v \Leftarrow_\eta e$ | $\texttt{if}_\eta\ (e)\ c_1$ `else` c_2
Expr	e	$::= v \mid n \mid$ `uop` $e \mid e$ `bop` e
Vars	x, y, v	$\in$ **Vars**

Figure 3.3: Syntax of SecVerilog.

early design stage. This is not possible with GLIFT [84] and Sapper [48]. Second, variables and logic can be shared across multiple security levels (e.g., `way` and `hit` are shared with various timing labels), which is not possible with Caisson [49]. Moreover, SecVerilog adds little programming effort: Verilog code can be verified almost as-is, with annotations (security labels) required only for variable declarations.

3.2 SecVerilog: Syntax and semantics

Except for added annotations, SecVerilog has essentially the same syntax and semantics as Verilog [30]. It builds on the synthesizable subset of the Verilog language. The target language of our compiler is synthesizable Verilog from which hardware can be generated using existing tools. We restrict to synthesizable code because unsynthesizable Verilog code is used only for testing purposes; it has no effect on the final hardware.

A core subset of SecVerilog is shown in Figure 3.3. We choose this subset because it includes all interesting features, and the omitted features (e.g., `case`, `assign` and the ternary conditional) can be translated into the core language.

A SecVerilog program (`Prog`) consists of a set of variable declarations and a set of thread definitions that use these variables. Variable v can represent either a register or a wire. The difference is that wires are stateless, and must be driven by other signals. We do not distinguish them in the syntax.

"Always blocks" (B) in (Sec)Verilog are similar to *threads* from the software perspective. Each `always` block translates into a hardware module that operates in parallel to other modules.

Threads are activated by triggers. A trigger γ can either be a change to the clock signal (`posedge/negedge` means the rising/falling edge of the clock signal), or a change to a variable in a variable list $\vec{v}$. For example, commands in the `always` block at line 9 in Figure 3.2(a) are activated at every rising edge of the clock signal.

Commands c are similar to those in software languages. Symbols η are unique identifiers for program points and can be ignored for now. A feature of Verilog not found in most programming languages is the distinction between blocking assignment $v =_\eta e$ and nonblocking assignment $v \Leftarrow_\eta e$. The effects of blocking assignments are visible immediately, but those of nonblocking assignments are delayed until the end of the current time unit. For example, consider the two code fragments $x = 1; y \Leftarrow x$ and $x \Leftarrow 1; y \Leftarrow x$. If the value of x is initially 0, then y becomes 1 in the first piece of code, but 0 in the second.

We provide a formal operational semantics for SecVerilog in Section 3.5.1.

Level	$\ell \in \mathcal{L}$
Family	$f \in \mathbb{Z}_n \to \mathcal{L}$
Label	$\tau ::= \ \ell \mid f(v) \mid \tau_1 \sqcup \tau_2 \mid \tau_1 \sqcap \tau_2$

Figure 3.4: Syntax of security labels.

3.3 SecVerilog: Type system

The SecVerilog type system statically controls information flow in a rigorous and verifiable way. The most novel features of the type system include: 1) mutable, dependent security labels, 2) a permissive yet sound way of controlling label channels, and 3) a modular design that decouples the program analyses required for precision from the type system. These novel features are essential for statically verifying highly efficient, practical hardware designs.

3.3.1 Type syntax

Types in SecVerilog are simply Verilog types extended with security label expressions, whose syntax is shown in Figure 3.4. The simplest form of label τ is a concrete security level ℓ drawn from the security lattice $\mathcal{L}$.

Unlike in most previous work on language-based security, SecVerilog supports *dynamic labels*: labels that can change at run time. A dynamic label $f(v)$ is constructed using a type-valued function f applied to a variable v. Type-valued functions are needed in order to decode the simple values that the hardware can convey into labels from the lattice $\mathcal{L}$.

$$\frac{}{\Gamma, pc, \mathcal{M} \vdash \texttt{skip}_\eta}\ \text{T-SKIP} \qquad \frac{\Gamma, pc, \mathcal{M} \vdash c_1 \quad \Gamma, pc, \mathcal{M} \vdash c_2}{\Gamma, pc, \mathcal{M} \vdash c_1; c_2}\ \text{T-SEQ}$$

$$\frac{\Gamma \vdash e : \tau \quad v \notin \texttt{FV}(\Gamma(v)) \quad \models P(\bullet\eta) \Rightarrow \tau \sqcup pc \sqsubseteq \Gamma(v)}{\Gamma, pc, \mathcal{M} \vdash v =_\eta e \quad \Gamma, pc, \mathcal{M} \vdash v \Leftarrow_\eta e}\ \text{T-ASSIGN}$$

$$\frac{\Gamma \vdash e : \tau \quad \begin{array}{ll} v \in \texttt{FV}(\Gamma(v)) & \models P(\bullet\eta) \Rightarrow pc \sqsubseteq \Gamma(v)\ ^{\text{if } v \notin \mathcal{M}} \\ v' \notin \Gamma & \models P(\bullet\eta), v' = \lfloor e \rfloor_a \Rightarrow \tau \sqcup pc \sqsubseteq \Gamma(v)\{v'/v\} \end{array}}{\Gamma, pc, \mathcal{M} \vdash v =_\eta e \qquad \Gamma, pc, \mathcal{M} \vdash v \Leftarrow_\eta e}\ \text{T-ASSIGN-REC}$$

$$\frac{\Gamma \vdash e : \tau \quad \begin{array}{l} \Gamma, pc \sqcup \tau, \mathcal{M} \cap \texttt{DA}(\eta) \vdash c_1 \\ \Gamma, pc \sqcup \tau, \mathcal{M} \cap \texttt{DA}(\eta) \vdash c_2 \end{array}}{\Gamma, pc, \mathcal{M} \vdash \texttt{if}_\eta\ (e)\ c_1\ \texttt{else}\ c_2}\ \text{T-IF}$$

Figure 3.5: Typing rules: commands.

Dynamic labels are needed to accurately describe information flows in complex hardware designs, where resources are used by multiple security levels. One example is the label `Par(way)` used in Figure 3.2(a). Note that all security labels in SecVerilog, including dynamic labels and label-decoding functions, only exist for compile-time type checking; they have no run-time manifestation.

Because security labels can mention terms (in particular, variables such as `way`), the type system has dependent types. Dependent security types have been explored in some prior work on security type systems that track information flow (e.g., [59, 85, 97]), where they provide valuable expressive power. However, in order to support analysis of hardware security, the type system for SecVerilog includes some unique features: first, the use of type-valued functions for label decoding, and second, even more unusual, the presence of mutable variables in types—that is, types may depend on variables whose value can change at run time.

The design philosophy of SecVerilog is to offer an expressive language with a low annotation burden, along with fast, automatic type checking. Following this philosophy, the only kind of term to which a label decoding function can be applied is a variable. This restriction ameliorates two problems: first, the undecidability of type equality involving general program expressions, and second, side effects changing the meaning of types.

Despite this restriction, dependent types in SecVerilog nevertheless turn out to be expressive enough for the intended use in hardware design. Restricting dependent types allows type checking to be fast (e.g., two seconds to verify a complete MIPS CPU in Section 6.3.1) and fully automatic. The syntax also alleviates the resulting limitations on expressiveness by allowing joins ($\sqcup$) and meets ($\sqcap$) of labels.

3.3.2 Typing rules

Typing rules for expressions have the form $\Gamma \vdash e : \tau$ where Γ is a typing environment that maps variables to security labels, e is the expression, and τ is its label. Since these rules are mostly standard [72], we leave the details in Section 3.5.

The typing rules for commands are shown in Figure 3.5. The typing judgment has the form $\Gamma, pc, \mathcal{M} \vdash c$. Similar to the usual program-counter label [72] for software languages, pc is used to control implicit flows. More interesting is $\mathcal{M}$, which tracks a set of variables that *must* be modified in all alternative executions. The type system uses $\mathcal{M}$ to improve its precision, as we see shortly.

In the next three sections, we explore the challenges of designing the SecVerilog type system and along the way explain the rules of Figure 3.5 in more depth.

```
1 reg[7:0] {H} secret, {L} public, {L} x;
2 reg[7:0] {LH(x)} y; // LH(0)=L LH(1)=H
3 always@(posedge clock) begin
4   if (x==1) begin y ⇐ secret; end
5   else      begin public ⇐ y; end
6   end
7 end
```

Figure 3.6: An example of implicit declassification.

3.3.3 Mutable dependent security labels

Dependent types need to mention mutable variables in practical hardware designs. For example, the variable `way` in Figure 3.2(a) can be modified whenever a new read request comes to the data cache, updating which cache way to use. Mutability creates challenges for the soundness of the type system. We begin by illustrating these challenges.

Implicit declassification Whenever a variable changes, the meaning of any security label that depends on it also changes. To be secure, SecVerilog needs to prevent such changes from implicitly declassifying information. Consider the example in Figure 3.6. This code is clearly insecure since it copies `secret` into `public` when `x` changes from 1 to 0 (not shown for brevity).

At the assignment to `y` in the first branch, its level is H, but at the assignment to `public`, the level of `y` has become L. The insecurity arises from the change to the label of `y` during the execution, while its content remains the same. In other words, if `x` changes from 1 to 0, the label of `y` cannot protect its content.

We rely on a dynamic mechanism to ensure register contents are erased when the old label is not bounded by the new one. This is captured by the

$$\frac{\langle \sigma, e \rangle \Downarrow n \quad \sigma' = \texttt{switch}(v, \sigma[v \mapsto n])}{\langle \sigma, v =_\eta e \rangle \Downarrow \sigma'} \text{ S-ASGN1}$$

$$\texttt{switch}(v, \sigma)(v') = \begin{cases} 0 & \text{if } v' \neq v \wedge v \in \texttt{FV}(\Gamma(v')) \\ \sigma(v) & \text{otherwise} \end{cases}$$

Figure 3.7: Dynamic erasure of contents.

small-step rule for assignments, shown in Figure 3.7. Note that since wires in hardware are stateless, this rule only applies to registers. The rule (S-ASGN1) ensures that after an assignment that changes the label of a variable, that variable's value is zeroed out. Code to dynamically zero out registers is automatically inserted as part of the translation to Verilog. In the rule, the expression $\texttt{FV}(\tau)$ returns the set of free variables in type τ.

While this dynamic mechanism may affect the functionality of the original hardware design, we believe that it is not a major issue in practice for the following reasons:

1. Dynamic erasure happens very rarely in our design experience. Most variables with dynamic labels are wires in our prototype processor design (e.g., `way`, `tag_in` and `write_enable` in Figure 3.2(a)). So the dynamic mechanism has no effect on these variables.

2. For registers with dynamic labels, this clearing is indeed necessary for security; hardware designers need to explicitly implement it anyway. Consider `dFsmState` in Figure 3.2(b), the state of the cache controller. It is reset anyway in a secure design, when the pipeline is flushed in the case that the timing label changes from H to L.

3. Further, the compiler can notify a designer when automatic clearing is generated, and ask the designer to explicitly approve such changes.

```
1  reg{H} high;
2  reg{L} low, low';
3  reg{LH(x)} x;
4  //LH(0)=L LH(1)=H
5  ...
6  if (high) begin
7    x ⇐ 1;
8  end
9  if (x==0 && low==1) begin
10   low' ⇐ 0;
11 end
12 low ⇐ 1;
13 ...
```

(a) Insecure program with a label channel.

```
1  reg{H} hit2, hit3;
2  reg[1:0]{Par(way)} way;
3  // Par(0)=Par(1)=L
4  // Par(2)=Par(3)=H
5  ...
6  if (hit2 || hit3) begin
7    way ⇐ (hit2?2'b10:2'b11);
8  end
9  else begin
10   way ⇐ 2'b10;
11 end
12 ...
```

(b) No-sensitive-upgrade rejects secure code.

```
1  reg{H} high;
2  reg{L} low, low';
3  reg{Par(x)} x;
4  // Par(0)=Par(1)=L
5  // Par(2)=Par(3)=H
6  ...
7  if (x==0) begin
8    low ⇐ 1;
9  end
10 else begin
11   high ⇐ 1;
12 end
13 low' ⇐ low;
14 ...
```

(c) Flow-sensitive systems reject secure code.

Figure 3.8: Examples illustrating the challenges of controlling label channels.

Label channels Mutable dependent types create *label channels* in which the value of a label becomes an information channel. For instance, consider the code snippet in Figure 3.8(a). This example appears secure as the assignment to `low'` only occurs when the label of `x` is L (when `x` is 0). When `high` is 1, the label of `x` becomes H, which correctly protects the secrecy of `high`. However, this code is insecure because the *change of label* `x` also leaks information. Suppose that the variables represent flip-flops that are initialized to ($\texttt{x} = 0, \texttt{low} = 0, \texttt{low}' = 1$) on a reset. The value of `x` in the second clock cycle after a reset is determined by the value of `high` in the first cycle; 1 if `high` is 1, 0 if `high` is 0. Then, `low'` in the third clock cycle reflects the value of `x` in the second cycle, leaking information from `high` to `low'`.

Similar vulnerabilities have also been observed in the literature on flow-sensitive security types, in which security labels of a variable may change dynamically (e.g., [71, 38, 8]). However, prior solutions are all too conservative (i.e., they reject secure programs) for practical hardware designs.

The first approach is *no-sensitive-upgrade* [8], which forbids raising a low label to high in a high context. However, this restriction rules out useful secure code, such as the secure code in Figure 3.8(b), adapted from our partitioned cache design. This code selects a cache way to write to. Variables `hit2` and `hit3`, representing the existence of a hit in high cache, have label H. No-sensitive-upgrade rejects this program, since `way` might be L before the assignment.

The second approach [38, 71] raises the label of variables modified in any branch to the context label (the label of the branch condition). Returning to the example in Figure 3.8(a), the label of `x` would become H because of the if-statement at lines 6–8. This over-approximation can be too conservative as well. For example, consider the secure code in Figure 3.8(c). Here, the label of `x` specifies an invariant: whenever `x` is 0 or 1 (i.e., `Par(x)`=L), nothing is leaked by the value of `x` nor by the time at which its value changes. Hence, `low`'s transition to 1 at line 8 is secure. However, the approach in [38] raises the label of `low` to `Par(x)` after the if-else statement. This conservative label of `low` makes checking at line 13 fail, since there is a flow from H to L when `x` is 2 or 3. Even a more permissive approach rejects this secure code. When `x` is 2 or 3, the dynamic monitor described in [71] tracks a set of variables that may be modified in another branch (`low` in this case), and raises their label to the context label (H). Hence, line 13 is still rejected.

We propose a more permissive mechanism that accepts the secure programs in Figure 3.8(b) and 3.8(c). Our insight is that no-sensitive-upgrade is needed for security, but only when the modified variable *might not* be assigned in an alternative path. For example, in Figure 3.8(b), the variable `way` is modified in both branch paths. Here, the label of `way` is checked for both branch paths on

the assignments to `way` (line 7 and 10), ensuring that the label of `way` must be higher than the context label (H) at the merge point. In other words, the fact that the label of `way` becomes H leaks no information. Hence, the no-sensitive-upgrade check is unnecessary in this case. This insight is formally justified in our soundness proof in Section 3.5.

This insight motivates using a *definite-assignment analysis*, which identifies variables that must be assigned to in any possible execution. Definite assignment analysis is a common static program analysis useful for detecting uninitialized variables. Since SecVerilog, like Verilog, has no aliasing, definite-assignment analysis is simple; we omit the details.

We assume an analysis that returns $\texttt{DA}(\eta)$, variables that must be assigned to in any possible execution of the command at location η. The type system propagates this information to branches, so that for an assignment to v, the no-sensitive-upgrade check is avoided if v must be assigned to in other paths. For example, the program in Figure 3.8(b) is well-typed because the variable `way` is modified in both branches, avoiding the limitations of [8]. Moreover, the type system still enables the remaining (necessary) no-sensitive-upgrade checks. So there is no need to raise the label of a variable assigned to in an alternative path. For example, there is no need to check the assignment to `low` at line 8 of Figure 3.8(c) in a high context (the else branch), avoiding the limitations of [38, 71]. Soundness is preserved despite the extra permissiveness (see Section 3.4).

3.3.4 Constraints and hypotheses

The design goal of SecVerilog is to achieve both soundness and precision, with a low annotation burden. The key to precision is to make enough information about the run-time values of variables available to the type system. For instance, consider the assignment to `hit` at line 10 in our cache controller (Figure 3.2(b)). To rule out an insecure flow from `hit2` and `hit3` (with label H) to `hit` (with label `LH(timingLabel)`), the type system must ensure $\mathrm{H} \sqsubseteq$ `LH(timingLabel)`. In other words, in any possible evaluation of the assignment, the label H must be bounded by `LH(timingLabel)`. In fact, this must be true because the condition `timingLabel=1` holds whenever the assignment happens (note that `timingLabel` is a single bit). However, a naive type system without knowledge of run-time values of `timingLabel` has to conservatively reject the program.

We use a modular design to separate the concerns of soundness and precision of our type system. In this design, the type system, along with a race-condition analysis in Section 3.4.3, ensures soundness (i.e., *observational determinism* in Section 3.4.2). The precision of the type system is improved further, without harming soundness, by integrating two program analyses: a *predicate transformer* analysis and the definite-assignment analysis already discussed.

Specifically, the type system generates proof obligations: partial orderings that must hold on pairs of security labels, regardless of the run-time values of those labels. To statically check a partial ordering on labels, we might require the partial ordering to hold for *any* possible values of free variables:

$$\tau_1 \sqsubseteq \tau_2 \Leftrightarrow \forall \vec{n} \,.\, \tau_1\,\{\vec{n}/\vec{v}\} \sqsubseteq \tau_2\,\{\vec{n}/\vec{v}\}$$

where $\vec{v} = \mathtt{FV}(\tau_1) \cup \mathtt{FV}(\tau_2)$, and $\mathtt{FV}(\tau)$ is the free variables in τ. However, this static approximation is too conservative.

To escape this conservatism, the type system uses a more precise approximation of the possible hardware states that can arrive at each program point. We denote the facts that program analysis has derived about the hardware states as predicates indexed by command identifiers η. The predicates $P(\bullet\eta)$ and $P(\eta\bullet)$ respectively denote overapproximations of the hardware states that can exist before and after the execution of the command at location η. Using even simple program analyses to generate these predicates considerably improves the precision of information flow analysis without harming soundness. Returning to our example, supposing that the program analysis can derive the predicate $P(\bullet\eta)$ = (`timingLabel = 1`). The type system then only needs to know that the flow from H to `LH(timingLabel)` is secure when `timingLabel` is 1. This requirement can be expressed as an (easily verified) constraint:

$$\texttt{timingLabel} = 1 \Rightarrow \mathrm{H} \sqsubseteq \texttt{LH(timingLabel)}$$

3.3.5 Generating state predicates

Many techniques can be used to generate predicates describing the run-time state, with a tradeoff between precision and complexity. For example, weakest preconditions [23] could be used. However, shallow knowledge of run-time state is enough for our type system to be effective. We use a simple abstract interpretation to propagate predicates forward through each thread definition, starting from the predicate `true` and overapproximating the postcondition at each program point. The rules defining this analysis are given in Figure 3.9. The algorithm generates predicates in linear time.

Expression results are coarsely approximated by tracking only constant val-

$$\lfloor n \rfloor_a = n \qquad \lfloor v \rfloor_a = v \qquad \lfloor e \rfloor_a = \top \text{ (otherwise)}$$

$$\lfloor e_1 \ \texttt{bop} \ e_2 \rfloor_b = \begin{cases} \lfloor e_1 \rfloor_b \ \texttt{bop} \ \lfloor e_2 \rfloor_b \text{ if } \texttt{bop} \in \{\wedge, \vee\} \\ \lfloor e_1 \rfloor_a \ \texttt{bop} \ \lfloor e_2 \rfloor_a \text{ if } \texttt{bop} \in \{=, \neq\} \\ \top \text{ otherwise} \end{cases}$$

$$\lfloor \texttt{uop} \ e \rfloor_b = \begin{cases} \neg \lfloor e \rfloor_b \text{ if } \texttt{uop} \in \{\neg\}, \lfloor e \rfloor_b \neq \top \\ \top \text{ otherwise} \end{cases}$$

$$\frac{}{\{P\}\ \texttt{skip}_\eta\ \{P\}} \qquad \frac{\{P\}\ c_1\ \{Q\} \quad \{Q\}\ c_2\ \{R\}}{\{P\}\ c_1; c_2\ \{R\}}$$

$$\frac{Q = \texttt{remove}(v, P)}{\{P\}\ \ v =_\eta e\ \ \{Q \wedge (v = \lfloor e \rfloor_a)\}} \qquad \frac{}{\{P\}\ \ v \Leftarrow_\eta e\ \ \{P\}}$$

$$\frac{\{P \wedge (\lfloor e \rfloor_b)\}\ c_1\ \{Q\} \quad \{P \wedge (\neg \lfloor e \rfloor_b)\}\ c_2\ \{R\}}{\{P\}\ \texttt{if}_\eta\ (e)\ c_1\ \texttt{else}\ c_2\ \{Q \cap R\}}$$

Figure 3.9: Predicate generation in Hoare logic.

ues and variables, and replacing more complex expressions with the "unknown" value $\top$. Operators $\lfloor e \rfloor_a$ and $\lfloor e \rfloor_b$ estimate the arithmetic and boolean values of e, respectively. The result of binary operators is $\top$ if any operand is $\top$. The translation rules on commands are written as admissible weakenings of the rules of Hoare Logic [35]. They specify how to compute a postcondition from a precondition. To make reasoning practical, the rules do not derive the strongest possible postcondition—but of course it is sound to weaken postconditions. Consequently, postconditions and preconditions are represented as conjunctions. The rule for assignment weakens the strongest-postcondition rule [25] by discarding all conjuncts that mention the assigned variable, in $\texttt{remove}(v, P)$. For efficiency, the rule for `if` weakens the obvious postcondition, $Q \vee R$, by syntactically intersecting the sets of conjuncts in Q and R.

3.3.6 Discussion of typing rules

The most interesting rules in Figure 3.5 are (T-ASSIGN), (T-ASSIGN-REC) and (T-IF). Proof obligations are generated for assignments $v =_\eta e$ and $v \Leftarrow_\eta e$. These proof obligations are discharged by an external solver; our implementation uses Z3 [58]. The informal invariants the type system maintains are 1) the new label of v is more restrictive than both the context label pc and label of e, 2) the no-sensitive-upgrade check is enabled if there might not be an assignment to v in an alternative branch. Rule (T-ASSIGN-REC) checks these invariants explicitly. To check the invariant after update, the rule generates a fresh variable v' to represent the new value of v. Though pc and the security level of e may also change after the assignment, the rule checks against the old value since semantically, information flows from the old state to variable v. The no-sensitive-upgrade check is enforced with the condition $v \notin \mathcal{M}$ adding precision in the case where the variable is assigned in every branch. The single check in Rule (T-ASSIGN) is sufficient for these invariants since $\Gamma(v)$ remains the same when there is no self-dependency, and the check entails no-upgrade-check (because $pc \sqsubseteq \tau \sqcup pc$). To improve precision of the type system, predicates on states are used only as hypotheses in these proof obligations. Blocking and nonblocking assignments use the same typing rule, differing only on when the assignment takes effect.

Rule (T-IF) propagates the set of variables that must be modified in both branches ($\texttt{DA}(\eta)$) to c_1 and c_2. Taking the intersection of $\mathcal{M}$ and $\texttt{DA}(\eta)$ is needed for nested if-statements.

3.3.7 Scalability of type checking

Queries sent to Z3 are generated by typing rules (T-ASSIGN) and (T-ASSIGN-REC) in Figure 3.5. Note that these queries are essentially predicates on a (finite) lattice of security labels. In other words, only simple theories (e.g., no quantifiers, no real numbers) of the full-fledged Z3 solver are needed by the type system. These queries can be efficiently solved by Z3.

Moreover, the static analyses used by SecVerilog to enable precise type checking (definite assignment analysis and predicate generation) are both modular. Race condition analysis may vary depending on the hardware design tool, but is scalable for most tools.

For the complete MIPS CPU in Section 6.3.1, it takes a total of only two seconds to generate all 1257 constraints by the type system, and solve them with Z3, suggesting that type checking is likely to scale to larger hardware designs.

3.3.8 Well-formed typing environments

The use of dynamic labels also puts constraints on the typing environment Γ: Γ is *well-formed*, denoted $\vdash \Gamma$, when 1) no variable depends on a more restrictive variable, preventing secrets from flowing into a label, and 2) no dependencies are chained, preventing cyclic dependencies. If $\texttt{FV}(\tau)$ is the free variables in τ, this can be expressed formally as follows:

Definition 1 (Well-formedness) Γ *is well-formed iff*

$$\forall v \in \textbf{Vars} \,.\, (\forall v' \in \texttt{FV}(\Gamma(v)) \,.\, \Gamma(v') \sqsubseteq \Gamma(v)) \;\wedge\; (\forall v' \in \texttt{FV}(\Gamma(v)) \,.\, v' \neq v \Rightarrow \texttt{FV}(\Gamma(v')) = \emptyset)$$

3.4 Soundness

Central to our approach is rigorous enforcement of a strong information security property. We formalize this property in this section and show the full proofs in Section 3.5.

3.4.1 Proving hardware properties from HDL code

Our goal is to prove that the actual hardware implementation controls information flow. However, information flow is analyzed at the level of the HDL. The argument that language-level reasoning is accurate has two steps. First, the operational semantics of SecVerilog correspond directly to hardware simulation at the RTL (Register Transfer Level) of abstraction. Second, for a synchronously clocked design, these RTL simulations accurately reflect behavior of synthesized hardware; in fact, functional verification of modern hardware relies mainly on RTL simulation. Thus, HDL-level reasoning suffices to prove hardware-level security properties.

3.4.2 Observational determinism

Our formal definition of information flow security is based on *observational determinism* [70, 93], a generalization of noninterference [28] that provides a strong end-to-end security guarantee even for nondeterministic systems. Observational determinism requires that in any two executions that receive the same low (adversary-visible) input, the system's low behavior must also be indistinguish-

able regardless of both high inputs and (possibly adversarial) nondeterministic choices.

Formalizing this property in the presence of dynamic labels presents some challenges, since the security level of a variable may differ in two hardware states. We start by defining a *low-equivalence* relation $\approx_\ell$ on hardware states σ, indexed by a level $\ell \in \mathcal{L}$. Two states are low-equivalent at level ℓ if they cannot be distinguished by an adversary able to observe information only at that level or below.

We assume a typing environment Γ that maps variables to security labels. Given state σ, the security level of a variable x is: $\mathcal{T}(x, \sigma) = \ell'$, where ℓ' is the value of label $\Gamma(x)$ in σ. We formalize the low-equivalence relation as follows:

Definition 2 (Low equivalence at level ℓ) *Two states are low-equivalent at level ℓ iff any variable whose label is below ℓ in one state must have the same label and value in the other:*

$$\forall \sigma_1, \sigma_2 \,.\, \sigma_1 \approx_\ell \sigma_2 \iff \forall x \in \mathbf{Vars} \,.\, (\mathcal{T}(x, \sigma_1) \sqsubseteq \ell \Leftrightarrow \mathcal{T}(x, \sigma_2) \sqsubseteq \ell) \wedge (\mathcal{T}(x, \sigma_1) \sqsubseteq \ell \Rightarrow \sigma_1(x) = \sigma_2(x))$$

It is straightforward to check that $\approx_\ell$ is an equivalence relation. Note that we require the level of x to be bounded by ℓ in σ_2 whenever $\mathcal{T}(x, \sigma_1) \sqsubseteq \ell$. This definition corresponds to our adversary model: all variables below ℓ are observable to the adversary. For example, consider the case $\Gamma(x) = \mathtt{LH}(x), \sigma_1(x) = 0$ and $\sigma_2(x) = 1$. Since x has different labels in the two states, $\sigma_1 \not\approx_L \sigma_2$. This is necessary because the ability to make the observation itself leaks information.

An *event* is a pair (t, σ), meaning that state σ occurred at clock cycle t. Assuming synchronous logic, events are produced only when a clock tick occurs

(formalized as the semantic rule (S-CLOCK) in Section 3.5.1). A *trace* T is a countably infinite sequence of events. We write $\langle \sigma, \texttt{Prog} \rangle \hookrightarrow T$ if executing `Prog` with initial states σ produces a trace T. Since the semantics is nondeterministic, there can be multiple traces T such that $\langle \sigma, \texttt{Prog} \rangle \hookrightarrow T$. Two traces are low-equivalent when the states in traces are clockwise low-equivalent.

We formalize observational determinism as follows:

Definition 3 (Observational Determinism) *Program* `Prog` *obeys observational determinism if for any low-equivalent states σ_1 and σ_2, execution from those states always produces low-equivalent traces:*

$$\sigma_1 \approx_L \sigma_2 \wedge \langle \sigma_1, \texttt{Prog} \rangle \hookrightarrow T_1 \wedge \langle \sigma_2, \texttt{Prog} \rangle \hookrightarrow T_2 \implies T_1 \approx_L T_2$$

Note that traces include the clock-cycle counter, so this definition is timing-sensitive, controlling timing channels.

In principle, observational determinism restricts expressiveness, since the scheduling of low assignments must be deterministic even when refinements cannot leak secret information. In practice, it rules out few useful designs, since nondeterministic behavior is caused by race conditions, which for hardware design are usually bugs.

3.4.3 Soundness of SecVerilog

The type system in Section 3.3 along with a race-condition analysis ensures that well-typed SecVerilog programs satisfy observational determinism.

Race freedom Today's synchronous hardware design methods disallow race conditions in order to produce deterministic systems. Existing synthesis tools prevent races by ensuring that only one thread updates each variable once per clock cycle. Intuitively, a program is race-free if the sequence of thread executions does not affect the synchronized state. This assumption is formalized as the following property.

Definition 4 (Race Freedom) *Program c is* race free *if for any state σ,*

$$\langle \sigma, c \rangle \hookrightarrow T_1 \wedge \langle \sigma, c \rangle \hookrightarrow T_2 \implies T_1 = T_2$$

Soundness proof We use the notation $\langle c, \sigma \rangle \Downarrow \sigma'$ for a big step: fully evaluating command c in state σ results in state σ'. To simplify notation, $\mathcal{V}(\tau, \sigma)$ represents the security level resulting from evaluating type τ in σ. We sketch one lemma and two theorems in this section and defer formal proofs to Section 3.5.

The first lemma states that any variable assigned to in a high context has a high label in the final state.

Lemma 1 (Confinement) *Let $\langle \sigma, c \rangle \Downarrow \sigma'$. If c can be typed under a given program counter label pc and well-formed typing environment Γ, then for every variable v assigned in command c, we have*

$$\mathcal{V}(pc, \sigma) \sqsubseteq \mathcal{T}(v, \sigma')$$

The next theorem states that running a command atomically to finish enforces noninterference.

Theorem 3 (Single-command noninterference) *If the states σ_1, σ_2 are low-equivalent at the beginning of a clock cycle, running any well-typed command c in σ_1 and σ_2 produces low-equivalent states at the beginning of next cycle as well:*

$$(\vdash \Gamma) \wedge (\Gamma \vdash c) \wedge (\sigma_1 \approx_L \sigma_2) \wedge \langle \sigma_1, c \rangle \Downarrow \sigma_1' \wedge \langle \sigma_2, c \rangle \Downarrow \sigma_2' \implies \sigma_1' \approx_L \sigma_2'$$

Finally, any well-typed SecVerilog program obeys observational determinism and is therefore secure:

Theorem 4 (Soundness of the type system) *If a SecVerilog program is well-typed under any well-formed typing environment, the program obeys observational determinism:*

$$(\vdash \Gamma) \wedge (\Gamma \vdash \texttt{Prog}) \wedge (\sigma_1 \approx_L \sigma_2) \wedge \langle \sigma_1, \texttt{Prog} \rangle \hookrightarrow T_1 \wedge \langle \sigma_2, \texttt{Prog} \rangle \hookrightarrow T_2 \implies T_1 \approx_L T_2$$

3.5 Soundness proof

3.5.1 Semantics

We now present a formal small-step operational semantics of SecVerilog. Though expressed more concisely, this semantics is largely motivated and justified by prior semantics for Verilog [30, 24].

We separate the semantics into command-level and thread-level semantics. A command-level configuration consists of a global store σ (a map from variables to values), a command c to be executed (or `stop`), a set of active assign-

$$\textsc{S-Skip}\ \frac{}{\langle \sigma, \texttt{skip}, \texttt{AS}, \texttt{NB}\rangle \to \langle \sigma, \texttt{stop}, \texttt{AS}, \texttt{NB}\rangle}$$

$$\textsc{S-Seq1}\ \frac{\langle \sigma, c_1, \texttt{AS}, \texttt{NB}\rangle \to \langle \sigma', \texttt{stop}, \texttt{AS}', \texttt{NB}'\rangle}{\langle \sigma, c_1; c_2, \texttt{AS}, \texttt{NB}\rangle \to \langle \sigma', c_2, \texttt{AS}', \texttt{NB}'\rangle}$$

$$\textsc{S-Seq2}\ \frac{\langle \sigma, c_1, \texttt{AS}, \texttt{NB}\rangle \to \langle \sigma', c_1', \texttt{AS}', \texttt{NB}'\rangle}{\langle \sigma, c_1; c_2, \texttt{AS}, \texttt{NB}\rangle \to \langle \sigma', c_1'; c_2, \texttt{AS}', \texttt{NB}'\rangle}$$

$$\textsc{S-Assign}\ \frac{\langle e, \sigma\rangle \Downarrow n \quad \texttt{AS}' = \texttt{AS} \cup \{(v, n)\}}{\langle \sigma, v =_\eta e, \texttt{AS}, \texttt{NB}\rangle \to \langle \sigma[v \mapsto n], \texttt{stop}, \texttt{AS}', \texttt{NB}\rangle}$$

$$\textsc{S-NBAssign}\ \frac{\langle e, \sigma\rangle \Downarrow n \quad \texttt{NB}' = \texttt{NB} \cup \{(v, n)\}}{\langle \sigma, v \Leftarrow_\eta e, \texttt{AS}, \texttt{NB}\rangle \to \langle \sigma, \texttt{stop}, \texttt{AS}, \texttt{NB}'\rangle}$$

$$\textsc{S-If}\ \frac{\langle e, \sigma\rangle \Downarrow v \quad v \neq 0 \Rightarrow i = 1 \quad v = 0 \Rightarrow i = 2}{\langle \sigma, \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2, \texttt{AS}, \texttt{NB}\rangle \to \langle \sigma, c_i, \texttt{AS}, \texttt{NB}\rangle}$$

Figure 3.10: Small-step operational semantics of commands.

$$\textsc{S-Adv}\ \frac{\langle \sigma, c_i, \emptyset, \texttt{NB}\rangle \to \langle \sigma', c_i', \texttt{AS}, \texttt{NB}'\rangle \quad \vec{c'} = \vec{c}\{c_i'/c_i\} \cup B\ \lfloor_{\texttt{AS}}}{\langle t, \sigma, \vec{c}, B, \texttt{NB}\rangle \to \langle t, \sigma', \vec{c'}, B - B\ \lfloor_{\texttt{AS}}, \texttt{NB}'\rangle}$$

$$\textsc{S-Trans}\ \frac{\texttt{NB} \neq \emptyset \quad \nexists i\ .\ \langle \sigma, c_i, \emptyset, \texttt{NB}\rangle \to \langle \sigma', c_i', \texttt{AS}, \texttt{NB}'\rangle \quad m' = \texttt{apply}(m, \texttt{NB})}{\langle t, \sigma, \vec{c}, B, \texttt{NB}\rangle \to \langle t, \sigma', B\ \lfloor_{\texttt{NB}}, B - B\ \lfloor_{\texttt{NB}}, \emptyset\rangle}$$

$$\textsc{S-Clock}\ \frac{\nexists i\ .\ \langle \sigma, c_i, \emptyset, \emptyset\rangle \to \langle \sigma', c_i', \texttt{AS}, \texttt{NB}\rangle}{\langle t, \sigma, \vec{c}, B, \emptyset\rangle \overset{(t,\sigma)}{\to} \langle t + 1, \sigma, \mathcal{S}, \mathcal{C}, \emptyset\rangle}$$

Figure 3.11: Small-step operational semantics of threads.

ments AS and a set of pending assignments NB. Commands merely accumulate AS and NB; their use will be clear in the thread-level semantics.

The semantics of commands are presented in Figure 3.10. Most rules are self-explanatory. The most interesting part is the difference between (S-ASSIGN) and (S-NBASSIGN), where the latter captures the delayed effect of nonblocking assignments.

The thread-level semantics in Figure 3.11 is also complicated by the need to defer effects of nonblocking assignments until the end of the current clock cycle. A thread-level configuration consists of the current clock cycle counter (t), global store (σ), a set of active commands to be executed in parallel ($\vec{c}$), a set of inactive combinational blocks (B) and a set of delayed updates accumulated from the execution of commands (NB).

We explain these rules by imagining the run of sequential blocks S and combinational blocks C from initial state $\langle 0, \sigma, \emptyset, \emptyset, \emptyset \rangle$ for some initial store σ. The (S-CLOCK) rule applies when the system is quiescent: no thread can make any progress and there are no pending nonblocking assignments. This applies to the initial state. When a clock tick occurs (the clock counter increases), all sequential blocks are activated[2] and all combinational blocks are waiting to be activated. Rule (S-ADV) then applies as long as there are activated and unfinished commands. In this step, an arbitrary thread c_i is scheduled and executed nondeterministically, using the rules in Figure 3.10. Combinational blocks that are activated by the execution ($B \downarrow_{\mathtt{AS}}$) are moved to the active commands. All previously active threads remain the same, except that the scheduled thread advances by one step ($\vec{c}\left\{c_i'/c_i\right\}$).

[2]to overapproximate all schedules, sequential logic on falling edges are also activated.

When all active threads finish, delayed assignments accumulated in `NB` take effect by rule (S-TRANS). To do that, the store is updated by applying all changes in `NB` in the order that assignments are added to `NB` ($\texttt{apply}(\sigma, \texttt{NB})$). At this point, the semantics allows all possible orders of applying assignments in `NB` to model all possible schedulers. Meanwhile, combinational logic triggered by these delayed assignments ($B \downarrow_{\texttt{NB}}$) is activated. Notice that the semantics permits race conditions. Therefore, different positions of an assignment in `NB` may result in different store states. *Events*, pairs of (t, σ), are produced only by the rule (S-CLOCK) since we focus on synchronous logic. We write $\langle \sigma, \texttt{Prog} \rangle \hookrightarrow T$ if $\langle 0, \sigma, \emptyset, \emptyset, \emptyset \rangle \hookrightarrow T$.

3.5.2 Typing rules

The typing rules for expressions and threads are shown in Figure 3.12 and Figure 3.13 respectively. The rules for expressions are standard. To type-check a combinational `always` block (rule (T-ALWAYS-COMB)), the join of all variables' label in the sensitive list is used as the *pc* label since the execution of block c reveals the fact that some of these variables are modified. The must-be-modified set of variables $\mathcal{M}$ is set to $\emptyset$ since combinational logic only executes when a variable in the sensitive list changes. On the other hand, the rule (T-ALWAYS-SEQ) uses $\bot$ as the *pc* label since c is executed whenever a clock tick comes. We say a program is well-typed under Γ ($\Gamma \vdash \texttt{Prog}$) when all blocks are well-typed. $\mathcal{M}$ is initialized to **Vars** since sequential logic is always triggered by the clock.

$$\frac{}{\Gamma \vdash n : \bot}\ \text{T-CONST} \qquad \frac{\Gamma(v) = \tau}{\Gamma \vdash v : \tau}\ \text{T-VAR} \qquad \frac{\Gamma \vdash e : \tau_1 \qquad \Gamma \vdash e' : \tau_2}{\Gamma \vdash e\ \texttt{bop}\ e' : \tau_1 \sqcup \tau_2}\ \text{T-BOP}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{uop}\ e : \tau}\ \text{T-UOP}$$

Figure 3.12: Typing rules: expressions.

$$\frac{\Gamma \vdash v_i : \tau_i \quad \Gamma, \sqcup\tau_i, \emptyset \vdash c}{\Gamma \vdash \texttt{always}\ @(\vec{v})\ c}\ \text{T-ALWAYS-COMB}$$

$$\frac{\Gamma, \bot, \mathbf{Vars} \vdash c}{\Gamma \vdash \texttt{always}\ @(\texttt{posedge}\ (\texttt{negedge}\ )\texttt{clock})\ c}\ \text{T-ALWAYS-SEQ}$$

Figure 3.13: Typing rules: threads.

3.5.3 Proofs

To aid the proof, we first define a big-step semantics of SecVerilog, shown in Figure 3.14 and 3.15. It is easy to check that this semantics defines one particular run of all threads in SecVerilog, according to the small-step semantics. Showing two runs starting from low-equivalent memory in this big-step semantics obeys observational determinism is sufficient to prove for any possible scheduling in the small-step semantics, due to the race-freedom assumption.

To simplify notation, we write $\langle \sigma, c \rangle$ instead of $\langle \sigma, c, \texttt{AS}, \texttt{NB} \rangle$ when `AS` and `NB` are irrelevant. To aid the proof, in the big-step semantics, `AS` and `NB` are extended with a security level ℓ, the concrete level of v after the execution. One

S-SKIP

$$\frac{}{\langle \sigma, \texttt{skip}, \texttt{AS}, \texttt{NB} \rangle \Downarrow \langle \sigma, \texttt{AS}, \texttt{NB} \rangle}$$

S-SEQ

$$\frac{\langle \sigma, c_1, \texttt{AS}, \texttt{NB} \rangle \Downarrow \langle \sigma'', \texttt{AS}'', \texttt{NB}'' \rangle \quad \langle \sigma'', c_2, \texttt{AS}'', \texttt{NB}'' \rangle \Downarrow \langle \sigma', \texttt{AS}', \texttt{NB}' \rangle}{\langle \sigma, c_1; c_2, \texttt{AS}, \texttt{NB} \rangle \Downarrow \langle \sigma', \texttt{AS}', \texttt{NB}' \rangle}$$

S-ASSIGN

$$\frac{\langle e, \sigma \rangle \Downarrow n \quad \sigma' = \sigma\{v \mapsto n\} \quad \texttt{AS}' = \texttt{AS} \cup \{(v, n, \mathcal{T}(v, \sigma'))\}}{\langle \sigma, v =_\eta e, \texttt{AS}, \texttt{NB} \rangle \Downarrow \langle \sigma', \texttt{AS}', \texttt{NB} \rangle}$$

S-NBASSIGN

$$\frac{\langle e, \sigma \rangle \Downarrow n \quad \texttt{NB}' = \texttt{NB} \cup \{(v, n, \mathcal{T}(v, \sigma\{v \mapsto n\}))\}}{\langle \sigma, v \Leftarrow_\eta e, \texttt{AS}, \texttt{NB} \rangle \Downarrow \langle \sigma, \texttt{AS}, \texttt{NB}' \rangle}$$

S-IF1

$$\frac{\langle e, \sigma \rangle \Downarrow 1 \quad \langle \sigma, c_1, \texttt{AS}, \texttt{NB} \rangle \Downarrow \langle \sigma', \texttt{AS}', \texttt{NB}' \rangle}{\langle \sigma, \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2, \texttt{AS}, \texttt{NB} \rangle \Downarrow \langle \sigma', \texttt{AS}', \texttt{NB}' \rangle}$$

S-IF2

$$\frac{\langle e, \sigma \rangle \Downarrow 0 \quad \langle \sigma, c_2, \texttt{AS}, \texttt{NB} \rangle \Downarrow \langle \sigma', \texttt{AS}', \texttt{NB}' \rangle}{\langle \sigma, \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2, \texttt{AS}, \texttt{NB} \rangle \Downarrow \langle \sigma', \texttt{AS}', \texttt{NB}' \rangle}$$

Figure 3.14: Big-step operational semantics of commands.

S-ADV

$$\frac{\vec{c} \neq \emptyset \quad \begin{array}{c} \langle \sigma, c_1, \emptyset, \texttt{NB} \rangle \Downarrow \langle \sigma', \texttt{AS}, \texttt{NB}' \rangle \\ \vec{c'} = (\vec{c} - \{c_1\}) \cup B \downarrow_{\texttt{AS}} \end{array}}{\langle t, \sigma, \vec{c}, B, \texttt{NB} \rangle \rightarrow \langle t, \sigma', \vec{c'}, B - B \downarrow_{\texttt{AS}}, \texttt{NB}' \rangle}$$

S-TRANS

$$\frac{\texttt{NB} \neq \emptyset \quad \vec{c} = \emptyset \quad \sigma' = \texttt{apply}(\sigma, \texttt{NB})}{\langle t, \sigma, \vec{c}, B, \texttt{NB} \rangle \rightarrow \langle t, \sigma', B \downarrow_{\texttt{NB}}, B - B \downarrow_{\texttt{NB}}, \emptyset \rangle}$$

S-CLOCK

$$\frac{\vec{c} = \emptyset}{\langle t, \sigma, \vec{c}, B, \emptyset \rangle \overset{(t,\sigma)}{\rightarrow} \langle t + 1, \sigma, \mathcal{S}, C, \emptyset \rangle}$$

Figure 3.15: Big-step operational semantics of threads.

exception is the rule (S-NBAssign), where ℓ is a hypothetical security level, as if the delayed assignment occurs immediately as a blocking assignment. Notice that this change does not affect the semantics: the purpose is solely to facilitate the proof.

The projection up to level L ($\restriction_L$) of $\mathtt{AS}$ is the longest subsequence of $\mathtt{AS}$ such that $\forall (x, v, \ell) \in \mathtt{AS} \restriction_L$ we have $\ell \sqsubseteq L$. We define $\mathtt{NB} \restriction_L$ in a similar way. Similar to the definition of low-equivalence on memory, $\mathtt{AS}_1 \approx_L \mathtt{AS}_2 \Leftrightarrow \mathtt{AS}_1 \restriction_L = \mathtt{AS}_2 \restriction_L$ and $\mathtt{NB}_1 \approx_L \mathtt{NB}_2 \Leftrightarrow \mathtt{NB}_2 \restriction_L = \mathtt{NB}_2 \restriction_L$.

We first prove several useful lemmas, and then show the type system enforces noninterference.

Lemma 2 *Let $\langle \sigma, v =_\eta e \rangle \Downarrow \sigma'$ (or $\langle \sigma, v \Leftarrow_\eta e \rangle \Downarrow \sigma'$). We have*

$$\forall u \in \mathtt{FV}(\Gamma(v)) \,.\, (u \neq v \Rightarrow \sigma(u) = \sigma'(u))$$

Proof. Trivial for $\Leftarrow$ since $\sigma' = \sigma$. For blocking assignments (=), since $u \in \mathtt{FV}(\Gamma(v))$, we have $\mathtt{FV}(\Gamma(u)) = \emptyset$ due to the well-formedness of Γ. Hence, by the semantics of assignment and `switch`, u will not be zeroed. So $\sigma(u) = \sigma'(u)$. □

Lemma 3 (Assignment) *Let $\langle \sigma, v =_\eta e, \emptyset, \mathtt{NB} \rangle \Downarrow \langle \sigma', (v, n, \ell), \mathtt{NB} \rangle$. If $\vdash \Gamma \wedge \Gamma, pc, \mathcal{M} \vdash v =_\eta e$ and $\Gamma \vdash e : \tau$, we have the following properties:*

1. *if $v \notin \mathtt{FV}(\Gamma(v))$, then $\mathcal{V}(pc, \sigma) \sqcup \mathcal{V}(\tau, \sigma) \sqsubseteq \ell$ and $\mathcal{V}(pc, \sigma) \sqsubseteq \mathcal{T}(v, \sigma)$*
2. *if $v \in \mathtt{FV}(\Gamma(v))$, then $\mathcal{V}(pc, \sigma) \sqcup \mathcal{V}(\tau, \sigma) \sqsubseteq \ell$ and $\mathcal{V}(pc, \sigma) \sqsubseteq \mathcal{T}(v, \sigma)$ if $v \notin \mathcal{M}$*
3. *$\forall v \notin \mathtt{FV}(\Gamma(u)) \wedge u \neq v \,.\, \sigma(u) = \sigma'(u) \wedge \mathcal{T}(u, \sigma) = \mathcal{T}(u, \sigma')$*

Proof. By induction on the typing rules.

1. By the typing rule (T-ASSIGN), we have $P(\bullet\eta) \to pc \sqcup \tau \sqsubseteq \Gamma(v)$. By the correctness of P and Lemma 5, $\mathcal{V}(pc, \sigma) \sqcup \mathcal{V}(\tau, \sigma) = \mathcal{V}(pc \sqcup \tau, \sigma) \sqsubseteq \mathcal{T}(v, \sigma)$. By the property of $\sqcup$, $\mathcal{V}(pc, \sigma) \sqsubseteq \mathcal{T}(v, \sigma)$.

   Now consider any $u \in \mathtt{FV}(\Gamma(v))$, we have $u \neq v$ by assumption. By Lemma 2, $\sigma(u) = \sigma'(u)$. This is true for all $u \in \mathtt{FV}(\Gamma(v))$, hence $\mathcal{T}(v, \sigma) = \mathcal{T}(v, \sigma')$. Therefore, $\mathcal{V}(pc, \sigma) \sqcup \mathcal{V}(\tau, \sigma) \sqsubseteq \mathcal{T}(v, \sigma') = \ell$, and $\mathcal{V}(pc, \sigma) \sqcup \mathcal{V}(\tau, \sigma) \sqsubseteq \mathcal{T}(v, \sigma)$.

2. When $v \notin \mathcal{M}$, we have $P(\bullet\eta) \to pc \sqsubseteq \Gamma(v)$ by the typing rule (T-ASSIGN-REC). By the correctness of P and Lemma 5, $\mathcal{V}(pc, \sigma) \sqsubseteq \mathcal{T}(v, \sigma)$.

   By the typing rule (T-ASSIGN-REC), we have $P(\bullet\eta), v' = [\![e]\!] \to \tau \sqsubseteq \Gamma(v)\{v'/v\}$. By the correctness of $\lfloor e \rfloor_a$, $\lfloor e \rfloor_a = n$ where $\langle e, \sigma \rangle \Downarrow n$. We extend σ to σ_e s.t. $\forall u \in \mathbf{Vars} \,.\, \sigma_e(u) = \sigma(u) \wedge \sigma_e(v') = n$. Easy to check σ_e satisfies the precondition. By Lemma 5, $\mathcal{T}(pc, \sigma_e) \sqcup \mathcal{T}(\tau, \sigma_e) = \mathcal{T}(pc \sqcup \tau, \sigma_e) \sqsubseteq \sigma_e(\Gamma(v)\{v'/v\})$. Since σ_e agrees with σ for all variables except v', which is not in pc and τ, $\mathcal{T}(pc, \sigma) \sqcup \mathcal{T}(\tau, \sigma) \sqsubseteq \sigma_e(\Gamma(v)\{v'/v\})$.

   Now consider any $u \in \mathtt{FV}(v)$ such that $u \neq v$. By Lemma 2, $\sigma'(u) = \sigma(u) = \sigma_e(u)$. By the semantics, $\sigma'(v) = n = \sigma_e(v')$. Hence, $\mathcal{T}(v, \sigma') = \sigma_e(\Gamma(v)\{v'/v\})$. Therefore, $\mathcal{V}(pc, \sigma) \sqcup \mathcal{V}(\tau, \sigma) \sqsubseteq \mathcal{T}(v, \sigma') = \ell$.

3. $\sigma(u) = \sigma'(u)$ is clear from the semantics of assignment. For any $w \in \mathtt{FV}(\Gamma(u))$, we prove $\sigma(w) = \sigma'(w)$ by contradiction.

   Otherwise, we have $w = v$ or $v \in \mathtt{FV}(\Gamma(w))$ from the semantics of assignment. In the first case, $v \in \mathtt{FV}(\Gamma(u))$ which contradicts our assumption that $v \notin \mathtt{FV}(\Gamma(u))$.

   In the second case, $v \in \mathtt{FV}(\Gamma(w))$. The case $w = v$ is already considered. When $w \neq v$, by the well-formedness of Γ, we have $\mathtt{FV}(\Gamma(w)) = \emptyset$ since $w \in \mathtt{FV}(\Gamma(u))$. This contradicts the fact that $v \in \mathtt{FV}(\Gamma(w))$).

Since $\sigma(w) = \sigma'(w)$ for all $w \in \mathtt{FV}(\Gamma(u))$, $\mathcal{T}(u, \sigma) = \mathcal{T}(u, \sigma')$.

□

Lemma 4 (NBAssignment) *Let* $\langle \sigma, v \Leftarrow_\eta e, \mathtt{AS}, \emptyset \rangle \Downarrow \langle \sigma', \mathtt{AS}, (v, n, \ell) \rangle$. *If* $\vdash \Gamma \wedge \Gamma, pc, \mathcal{M} \vdash v \Leftarrow_\eta e$ *and* $\Gamma \vdash e : \tau$, *we have the following properties:*

1. *if* $v \notin \mathtt{FV}(\Gamma(v))$, *then* $\mathcal{V}(pc, \sigma) \sqcup \mathcal{V}(\tau, \sigma) \sqsubseteq \ell$ *and* $\mathcal{V}(pc, \sigma) \sqsubseteq \mathcal{T}(v, \sigma)$
2. *if* $v \in \mathtt{FV}(\Gamma(v))$, *then* $\mathcal{V}(pc, \sigma) \sqcup \mathcal{V}(\tau, \sigma) \sqsubseteq \ell$ *and* $\mathcal{V}(pc, \sigma) \sqsubseteq \mathcal{T}(v, \sigma)$ *if* $v \notin \mathcal{M}$
3. $\forall v \notin \mathtt{FV}(\Gamma(u)) \wedge u \neq v \,.\, \sigma(u) = \sigma'(u) \wedge \mathcal{T}(u, \sigma) = \mathcal{T}(u, \sigma')$

Proof. Similar to the proof for blocking assignments since they have the same typing rules. The last claim holds trivially since $\sigma = \sigma'$. □

Lemma 5 *The lifted partial order on labels* ($\sqsubseteq$) *is conservative:*

$$\forall \sigma, \tau_1, \tau_2 \,.\, P(\sigma) \wedge P \Rightarrow \tau_1 \sqsubseteq \tau_2 \implies \mathcal{V}(\tau_1, \sigma) \sqsubseteq \mathcal{V}(\tau_2, \sigma)$$

Proof. Clear from the definition of $\sqsubseteq$. □

Lemma 6 *Low expressions may only contain low variables:*

$$\Gamma \vdash e : \tau \wedge \mathcal{V}(\tau, \sigma) \sqsubseteq L \implies \textit{for all } v \textit{ in } e\ \mathcal{T}(v, \sigma) \sqsubseteq L$$

Proof. By induction on the structure of expressions.

- n: trivial.

- v: by the typing rule, $\Gamma(v) = \tau$. Hence, $\mathcal{T}(v, \sigma) = \mathcal{V}(\Gamma(v), \sigma) = \mathcal{V}(\tau, \sigma) \sqsubseteq L$.
- e_1 bop e_2: by the typing rule, $\Gamma \vdash \tau_1 \wedge \Gamma \vdash \tau_2 \wedge \tau = \tau_1 \sqcup \tau_2$. Hence, $\mathcal{V}(\tau_1 \sqcup \tau_2, \sigma) = \mathcal{V}(\tau_1, \sigma) \sqcup \mathcal{V}(\tau_2, \sigma) \sqsubseteq L$, which gives us $\mathcal{V}(\tau_1, \sigma) \sqsubseteq L$ and $\mathcal{V}(\tau_2, \sigma) \sqsubseteq L$. By the induction hypothesis, for all v in e_1 or e_2 (that is, in e), $\mathcal{T}(v, \sigma) \sqsubseteq L$.
- uop e: by the induction hypothesis.

□

Lemma 7 *Low expressions must have the same concrete label in low-equivalent memories:*

$$\sigma_1 \approx_L \sigma_2 \wedge \Gamma \vdash e : \tau \wedge \mathcal{V}(\tau, \sigma_1) \sqsubseteq L \implies \mathcal{V}(\tau, \sigma_1) = \mathcal{V}(\tau, \sigma_2)$$

Proof. By induction on the structure of τ.

- ℓ: $\mathcal{V}(\ell, \sigma_2) = \ell = \mathcal{V}(\ell, \sigma_1)$.
- $f\ i$: by the well-formedness of Γ, $\Gamma(i) \sqsubseteq f\ i$. By Lemma 5, $\mathcal{T}(i, \sigma_1) = \mathcal{V}(\Gamma(i), \sigma) \sqsubseteq \mathcal{V}(f\ i, \sigma_1)$. By the transitivity of $\sqsubseteq$, $\mathcal{T}(i, \sigma_1) \sqsubseteq L$. Since $\sigma_1 \approx_L \sigma_2$, $\mathcal{T}(i, \sigma_2) \sqsubseteq L$ as well and $\mathcal{V}(i, \sigma_1) = \mathcal{V}(i, \sigma_2)$. Hence, $\mathcal{V}(f\ i, \sigma_1) = \mathcal{V}(f\ i, \sigma_2)$.
- $\tau_1 \sqcup \tau_2$: $\mathcal{V}(\tau_1, \sigma_1) \sqcup \mathcal{V}(\tau_2, \sigma_1) = \mathcal{V}(\tau_1 \sqcup \tau_2, \sigma_1) \sqsubseteq L$. Hence, $\mathcal{V}(\tau_1, \sigma_1) \sqsubseteq L \wedge \mathcal{V}(\tau_2, \sigma_1) \sqsubseteq L$. By the induction hypothesis, $\mathcal{V}(\tau_1, \sigma_1) = \mathcal{V}(\tau_1, \sigma_2)$ and $\mathcal{V}(\tau_2, \sigma_1) = \mathcal{V}(\tau_2, \sigma_2)$. Therefore, $\mathcal{V}(\tau_1 \sqcup \tau_2, \sigma_1) = \mathcal{V}(\tau_1 \sqcup \tau_2, \sigma_2)$.
- $\tau_1 \sqcap \tau_2$: since no typing rule generates meet labels, there must be a variable v such that $\Gamma(v) = \tau_1 \sqcap \tau_2$. By the well-formedness of Γ, $\forall u \in \mathtt{FV}(\tau_1 \sqcap \tau_2)\ .\ u \sqsubseteq \tau_1 \sqcap \tau_2$. By Lemma 5, $\mathcal{V}(u, \sigma_1) \sqsubseteq \mathcal{V}(\tau_1 \sqcap \tau_2, \sigma_1) \sqsubseteq L$. By the induction hypothesis, we have $\mathcal{V}(u, \sigma_1) = \mathcal{V}(u, \sigma_2)$. Since this is true for all variables in $\tau_1 \sqcap \tau_2$, $\mathcal{V}(\tau_1 \sqcap \tau_2, \sigma_1) = \mathcal{V}(\tau_1 \sqcap \tau_2, \sigma_2)$.

□

Lemma 8 *Low expressions must evaluate to the same value in low-equivalent memories:*

$$\sigma_1 \approx_L \sigma_2 \wedge \Gamma \vdash e : \tau \wedge \mathcal{V}(\tau, \sigma_1) \sqsubseteq L \wedge \langle e, \sigma_1 \rangle \Downarrow n_1 \wedge \langle e, \sigma_2 \rangle \Downarrow n_2 \implies \mathcal{V}(\tau, \sigma_2) \sqsubseteq L \wedge n_1 = n_2$$

Proof. We have $\mathcal{V}(\tau, \sigma_2) = \mathcal{V}(\tau, \sigma_1) \sqsubseteq L$ by Lemma 7.

By Lemma 6, for all v in e, we have $\mathcal{T}(v, \sigma_1) \sqsubseteq L$ and $\mathcal{T}(v, \sigma_2) \sqsubseteq L$. Since $\sigma_1 \approx_L \sigma_2$, $\sigma_1(v) = \sigma_2(v)$. This is true for all v in e, hence $n_1 = n_2$. □

Lemma 9 (PC Subsumption) *If a command can be typed under a given program counter label pc, it can also be typed under a lower level pc′:*

$$\Gamma, pc, \mathcal{M} \vdash c \wedge pc' \sqsubseteq pc \implies \Gamma, pc', \mathcal{M} \vdash c$$

Proof. By induction on the typing rules.

- Case T-Seq, T-If: by the induction hypothesis.
- Case T-Assign: from the typing rule, we have $P(\bullet\eta) \Rightarrow \tau \sqcup pc \sqsubseteq \Gamma(x)$ where $\Gamma \vdash e : \tau$. Since $\tau \sqcup pc' \sqsubseteq \tau \sqcup pc$, $P(\bullet\eta) \Rightarrow \tau \sqcup pc' \sqsubseteq \Gamma(x)$. Hence, $\Gamma, pc', \mathcal{M} \vdash x =_\eta e$.
- Case T-Assign-Rec: similar to T-Assign.

□

Lemma 10 (Aliveness Subsumption) *If a command can be typed under a given alive set $\mathcal{M}$, it can also be typed under a larger set $\mathcal{M}'$:*

$$\Gamma, pc, \mathcal{M} \vdash c \wedge \mathcal{M} \subseteq \mathcal{M}' \Rightarrow \Gamma, pc, \mathcal{M}' \vdash c$$

Proof. By induction on the typing rules.

- Case T-Seq, T-If: by the induction hypothesis.
- Case T-Assign, T-Stop: trivial since $\mathcal{M}$ is not used in the promise.
- Case T-Assign-Rec: since $\mathcal{M} \subseteq \mathcal{M}'$, for all $v \notin \mathcal{M}'$, we have $v \notin \mathcal{M}$. Since $\Gamma, pc, \mathcal{M} \vdash v =_\eta e$, we have $P(\bullet\eta) \Rightarrow \tau \sqcup pc \sqsubseteq \Gamma(v)$. Hence, $\Gamma, pc, \mathcal{M}' \vdash v =_\eta e$.

□

Proof of Lemma 1 [Confinement] If $\langle \sigma, c \rangle \Downarrow \sigma'$ and c can be typed under a given program counter label pc and well-formed typing environment Γ, then for every v assigned to in c, we have

$$\mathcal{V}(pc, \sigma) \sqsubseteq \mathcal{T}(v, \sigma') \wedge \mathcal{V}(pc, \sigma) \sqsubseteq \mathcal{V}(pc, \sigma')$$

Proof. By induction on the structure of c.

- $c_1; c_2$: by the typing rule, we have $\Gamma, pc, \mathcal{M} \vdash c_i$ where $i \in \{1, 2\}$. From semantics of sequential statement, we have

$$\frac{\langle \sigma, c_1 \rangle \Downarrow \sigma'' \quad \langle \sigma'', c_2 \rangle \Downarrow \sigma'}{\langle \sigma, c_1; c_2 \rangle \Downarrow \sigma'}$$

By the induction hypothesis, for every u assigned to in c_1, we have $\mathcal{V}(pc, \sigma) \sqsubseteq \mathcal{T}(u, \sigma'')$ and $\mathcal{V}(pc, \sigma) \sqsubseteq \mathcal{V}(pc, \sigma'')$. Also, for every w assigned to in c_2, we have $\mathcal{V}(pc, \sigma'') \sqsubseteq \mathcal{T}(w, \sigma')$ and $\mathcal{V}(pc, \sigma'') \sqsubseteq \mathcal{V}(pc, \sigma')$.

Therefore, $\mathcal{V}(pc, \sigma) \sqsubseteq \mathcal{V}(pc, \sigma')$ is straightforward by the transitivity of $\sqsubseteq$.

Hence, for every v assigned to in $c_1; c_2$. If v is assigned to in c_2, we have $\mathcal{V}(pc, \sigma) \sqsubseteq \mathcal{V}(pc, \sigma'') \sqsubseteq \mathcal{T}(v, \sigma')$. If v is assigned to in c_1 but not in c_2, we have $\mathcal{V}(pc, \sigma) \sqsubseteq \mathcal{T}(v, \sigma'')$. It must hold that $\mathcal{T}(v, \sigma'') = \mathcal{T}(v, \sigma)$ since otherwise, there must be some $v' \in \mathtt{FV}(\Gamma(v))$ be assigned to in c_2. However, by the semantics of assignment, either $v = v'$ or v is assigned to in c_2. Contradiction.

- $v =_\eta e$: By Lemma 3, we have $\mathcal{V}(pc, \sigma) \sqcup \mathcal{V}(\tau, \sigma) \sqsubseteq \mathcal{T}(v, \sigma')$ no matter $v \in \mathtt{FV}(\Gamma(v))$ or not. Therefore, $\mathcal{V}(pc, \sigma) \sqsubseteq \mathcal{T}(v, \sigma')$.

  By the semantics of assignment, $v \in \mathtt{FV}(\Gamma(u))$ may also be assigned to. By the well-formedness of Γ, we have $\Gamma(v) \sqsubseteq \Gamma(u)$. By Lemma 5, $\mathcal{T}(v, \sigma') \sqsubseteq \mathcal{T}(u, \sigma')$. Hence, $\mathcal{V}(pc, \sigma) \sqsubseteq \mathcal{T}(u, \sigma')$ as well.

  Now consider $u \in \mathtt{FV}(pc)$. If u is not assigned to, we have $\mathcal{V}(u, \sigma') = \mathcal{V}(u, \sigma)$. Otherwise, $\mathcal{V}(pc, \sigma) \sqsubseteq \mathcal{T}(u, \sigma')$ by the argument above. Due to the well-formedness of Γ, $\Gamma(u) \sqsubseteq pc$. By Lemma 5, $\mathcal{T}(u, \sigma) \sqsubseteq \mathcal{V}(pc, \sigma)$. Hence, $\mathcal{V}(u, \sigma) \sqsubseteq \mathcal{V}(u, \sigma')$. Putting together, for every $u \in \mathtt{FV}(pc)$, $\mathcal{V}(u, \sigma) \sqsubseteq \mathcal{V}(u, \sigma')$. Therefore, $\mathcal{V}(pc, \sigma) \sqsubseteq \mathcal{V}(pc, \sigma')$.

- `if` e `then` c_1 `else` c_2: by the typing rule (T-IF), we have $\Gamma \vdash e : \tau$ and $\Gamma, pc \sqcup \tau, \mathcal{M}' \vdash c_i$ where $i \in \{1, 2\}$ and $\mathcal{M}' = \mathcal{M} \cap \mathtt{DA}(\eta)$. By Lemma 9 and Lemma 10, $\Gamma, pc, \mathcal{M} \vdash c_i$.

  Consider the case when the if branch is taken. From semantics of if state-

ment, we have

$$\frac{\langle \sigma, e\rangle \Downarrow 1 \quad \langle \sigma, c_1\rangle \Downarrow \sigma'}{\langle \sigma, \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2\rangle \Downarrow \sigma'}$$

By the induction hypothesis, we have $\mathcal{V}(pc, \sigma) \sqsubseteq \mathcal{T}(v, \sigma')$ for every v assigned to in c_1 , which is the same set of variables assigned to in $\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2$.

$\mathcal{V}(pc, \sigma) \sqsubseteq \mathcal{V}(pc, \sigma')$ can be derived directly from the induction hypothesis.

The case when the else branch is taken is similar.

□

Lemma 11 (Definite Assignments) *If* $\langle \sigma, c, \texttt{AS}, \texttt{NB}\rangle \Downarrow \langle \sigma', \texttt{AS}', \texttt{NB}'\rangle$ *and* c *can be typed under some* $\mathcal{M}$ *returned by a definite-assignments analysis,* pc *and well-formed* Γ*, then we have*

$$\forall (v, n, \ell) \in (\texttt{AS}' - \texttt{AS}) \cup (\texttt{NB}' - \texttt{NB})\ .\ v \notin \mathcal{M} \Rightarrow \mathcal{V}(pc, \sigma) \sqsubseteq \mathcal{T}(v, \sigma)$$

Proof. By induction on the structure of c.

- $c_1; c_2$: by the typing rule, both c_1 and c_2 can be typed under $\mathcal{M}$. By the induction hypothesis if (v, n, ℓ) is generated in c_1. Otherwise, by Lemma 1, $\mathcal{V}(pc, \sigma) \sqsubseteq \mathcal{V}(pc, \sigma'')$. Therefore, $\mathcal{V}(pc, \sigma) \sqsubseteq \ell$ by the induction hypothesis.
- $v =_\eta e$: by Lemma 3.
- $v \Leftarrow_\eta e$: by Lemma 4.

- `if` e `then` c_1 `else` c_2: by the typing rule (T-IF), we have $\Gamma \vdash e : \tau$ and $\Gamma, pc \sqcup \tau, \mathcal{M}' \vdash c_i$ where $i \in \{1, 2\}$ and $\mathcal{M}' = \mathcal{M} \cap \mathtt{DA}(\eta)$. By Lemma 9 and Lemma 10, $\Gamma, pc, \mathcal{M} \vdash c_i$. Consider evaluation rules (S-IF1) and (S-IF2), $\mathcal{V}(pc, \sigma) \sqsubseteq \ell$ by the induction hypothesis.

□

Proof of Theorem 3 [Single-command Noninterference]

$$\vdash \Gamma \wedge \Gamma \vdash c \wedge \sigma_1 \approx_L \sigma_2 \wedge \langle \sigma_1, c \rangle \Downarrow \sigma_1' \wedge \langle \sigma_2, c \rangle \Downarrow \sigma_2' \implies \sigma_1' \approx_L \sigma_2'$$

Proof. Induction on the structure of c.

- $c_1; c_2$: by the typing rule, we have $\Gamma, pc, \mathcal{M} \vdash c_i$ where $i \in \{0, 1\}$. From evaluation rule, we have

$$\frac{\langle \sigma, c_1, \mathtt{AS}, \mathtt{NB} \rangle \Downarrow \langle \sigma'', \mathtt{AS}'', \mathtt{NB}'' \rangle \quad \langle \sigma'', c_2, \mathtt{AS}'', \mathtt{NB}'' \rangle \Downarrow \langle \sigma', \mathtt{AS}', \mathtt{NB}' \rangle}{\langle \sigma, c_1; c_2, \mathtt{AS}, \mathtt{NB} \rangle \Downarrow \langle \sigma', \mathtt{AS}', \mathtt{NB}' \rangle}$$

  From the induction hypothesis, $\sigma_1'' \approx_L \sigma_2'' \wedge \mathtt{AS}_1'' \approx_L \mathtt{AS}_2'' \wedge \mathtt{NB}_1'' \approx_L \mathtt{NB}_2''$. Using the induction hypothesis again on c_2, we get $\sigma_1' \approx_L \sigma_2' \wedge \mathtt{AS}_1' \approx_L \mathtt{AS}_2' \wedge \mathtt{NB}_1' \approx_L \mathtt{NB}_2'$.

- $v =_\eta e$: From the evaluation rules, we have for $i \in \{1, 2\}$

$$\frac{\langle \sigma_i, e \rangle \Downarrow n_i \quad \sigma_i' = \mathtt{switch}(v, \sigma_i[v \mapsto n_i])}{\langle \sigma_i, v =_\eta e, \mathtt{AS}_i, \mathtt{NB}_i \rangle \Downarrow \langle \sigma_i', \mathtt{AS}_i', \mathtt{NB}_i' \rangle}$$

  Let $\vdash e : \tau$ and (v, n_1, ℓ_1) is generated for $\mathtt{AS}_1$, (v, n_2, ℓ_2) is generated for $\mathtt{AS}_2$. First consider the case when $\ell_1 \sqsubseteq L$.

We have $\mathcal{V}(\tau, \sigma_1) \leq \ell_1 \sqsubseteq L$ by Lemma 3. By Lemma 28, $n_1 = n_2$. So $\sigma'_1(v) = \sigma'_2(v)$.

Next, we prove $\ell_2 \sqsubseteq L$. Consider any $u \in \texttt{FV}(v) \wedge u \neq v$. By the well-formedness of Γ, $\Gamma(u) \sqsubseteq \Gamma(v)$. By Lemma 5, $\mathcal{T}(u, \sigma'_1) \sqsubseteq \mathcal{T}(v, \sigma'_1) \sqsubseteq L$. By Lemma 2, we also have $\sigma'_1(u) = \sigma_1(u)$. Hence, $\mathcal{T}(u, \sigma_1) = \mathcal{T}(u, \sigma'_1) = \ell \sqsubseteq L$. By the assumption, $\sigma_1 \approx_L \sigma_2$, so $\mathcal{T}(u, \sigma_1) = \mathcal{T}(u, \sigma_2)$. By Lemma 2 again, we have $\mathcal{T}(u, \sigma'_2) = \mathcal{T}(u, \sigma_2) = \mathcal{T}(u, \sigma_1) = \mathcal{T}(u, \sigma'_2)$. This is true for all $u \in \texttt{FV}(v) \wedge u \neq v$. Since we already shown $\sigma'_1(v) = \sigma'_2(v)$, $\ell_2 = \mathcal{T}(v, \sigma'_2) = \mathcal{T}(v, \sigma'_1) = \ell_1 \sqsubseteq L$. Hence, we have $\texttt{AS}'_1 \approx_L \texttt{AS}'_2$.

For variables $v \notin \texttt{FV}(\Gamma(u)) \wedge u \neq v$. There are two possibilities: $\mathcal{T}(u, \sigma_1) = \mathcal{T}(u, \sigma_2) \sqsubseteq L$ or $\not\sqsubseteq L$ by Lemma 7. By Lemma 3, $\mathcal{T}(u, \sigma_1) = \mathcal{T}(u, \sigma'_1)$ and $\mathcal{T}(u, \sigma_2) = \mathcal{T}(u, \sigma'_2)$. Hence, $\mathcal{T}(u, \sigma'_1) = \mathcal{T}(u, \sigma'_2) \sqsubseteq L$ or $\not\sqsubseteq L$ by Lemma 7. In the first case, we have $\sigma'_1(u) = \sigma_1(u) = \sigma_2(u) = \sigma_2(u)'$ by Lemma 3. In the second case, $\approx_L$ has no restriction on the values of u.

For variables $v \in \texttt{FV}(\Gamma(u)) \wedge u \neq v$, $\sigma'_1(u) = \sigma'_2(u) = 0$ by the semantics. Now consider any $w \in \texttt{FV}(\Gamma(u))$. We observe that $\texttt{FV}(\Gamma(w)) = \emptyset$ by the well-formedness of Γ. Hence, $v \notin \texttt{FV}(\Gamma(w)) \vee w = v$. By the argument above, for any w, $\mathcal{T}(w, \sigma'_1) \sqsubseteq L \Leftrightarrow \mathcal{T}(w, \sigma'_2) \sqsubseteq L$ and $\mathcal{T}(w, \sigma'_1) \sqsubseteq L \Rightarrow \sigma'_1(w) = \sigma'_2(w)$. If $\forall w \in \texttt{FV}(\Gamma(u)) \,.\, \mathcal{T}(w, \sigma'_1) \sqsubseteq L$, we have $\sigma'_1(w) = \sigma'_2(w)$. So $\mathcal{T}(u, \sigma'_1) \sqsubseteq L \Leftrightarrow \mathcal{T}(u, \sigma'_2) \sqsubseteq L$ because $\mathcal{T}(u, \sigma'_1) = \mathcal{T}(u, \sigma'_2)$. Otherwise, there is some w such that $\mathcal{T}(w, \sigma'_1) \not\sqsubseteq L$ and $\mathcal{T}(w, \sigma'_2) \not\sqsubseteq L$. By the well-formedness of Γ, $\Gamma(w) \sqsubseteq \Gamma(u)$. By Lemma 5, $\mathcal{T}(u, \sigma'_1) \not\sqsubseteq L$ and $\mathcal{T}(u, \sigma'_2) \not\sqsubseteq L$ as well. Still $\mathcal{T}(u, \sigma'_1) \sqsubseteq L \Leftrightarrow \mathcal{T}(u, \sigma'_2) \sqsubseteq L$ holds.

Next, consider the case when $\ell_v \not\sqsubseteq L$. It must be true that $\mathcal{T}(v, \sigma'_2) \not\sqsubseteq L$ since otherwise, we can derive $\ell_v \sqsubseteq L$ as above.

For variables $v \in \texttt{FV}(\Gamma(u)) \wedge u \neq v$, $\Gamma(v) \sqsubseteq \Gamma(u)$ due to the well-formedness of

Γ. Hence, $\mathcal{T}(u, \sigma_1') \not\sqsubseteq L$ and $\mathcal{T}(u, \sigma_2') \not\sqsubseteq L$.

For variables $v \notin \mathtt{FV}(\Gamma(u)) \wedge u \neq v$. There are two possibilities: $\mathcal{T}(u, \sigma_1) = \mathcal{T}(u, \sigma_2) \sqsubseteq L$ or $\not\sqsubseteq L$ by Lemma 7. By Lemma 3, $\mathcal{T}(u, \sigma_1) = \mathcal{T}(u, \sigma_1')$ and $\mathcal{T}(u, \sigma_2) = \mathcal{T}(u, \sigma_2')$. Hence, $\mathcal{T}(u, \sigma_1') = \mathcal{T}(u, \sigma_2') \sqsubseteq L$ or $\not\sqsubseteq L$ by Lemma 7. In the first case, we have $\sigma_1'(u) = \sigma_1(u) = \sigma_2(u) = \sigma_2(u)'$ by Lemma 3. In the second case, $\approx_L$ has no restriction on the values of u.

$\mathtt{NB}_1' \approx_L \mathtt{NB}_2'$ is trivial since $\mathtt{NB}_1 = \mathtt{NB}_1$ and $\mathtt{NB}_2 = \mathtt{NB}_2'$

- $v \Leftarrow_\eta e$: $\sigma_1' \approx_L \sigma_2'$ is trivial since $\sigma_1 = \sigma_1'$ and $\sigma_2 = \sigma_2'$.

  Let $\vdash e : \tau$ and (v, n_1, ℓ_1) is generated for $\mathtt{NB}_1$, (v, n_2, ℓ_2) is generated for $\mathtt{NB}_2$. First consider the case when $\ell_1 \sqsubseteq L$.

  We have $\mathcal{V}(\tau, \sigma_1) \sqsubseteq \ell_1 \sqsubseteq L$ by Lemma 4. By Lemma 28, $n_1 = n_2$. So $\sigma_1'(v) = \sigma_2'(v)$. Similar to the proof for blocking assignments, we can prove $\ell_2 \sqsubseteq L$. Hence, $\mathtt{NB}_2 \approx_L \mathtt{NB}_2'$.

  Next, consider the case when $\ell_1 \not\sqsubseteq L$. It must be true that $\ell_2 \not\sqsubseteq L$ since otherwise, we can derive $\ell_1 \sqsubseteq L$ as above. Hence, $\mathtt{NB}_1' \approx_L \mathtt{NB}_2'$.

  $\mathtt{AS}_1' \approx_L \mathtt{AS}_2'$ is trivial since $\mathtt{AS}_1 = \mathtt{AS}_1$ and $\mathtt{AS}_2 = \mathtt{AS}_2'$

- $\mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2$: let $\Gamma \vdash e : \tau$. Since $\sigma_1 \approx_L \sigma_2$, there are two possibilities due to Lemma 7: $\mathcal{V}(\tau, \sigma_1) = \mathcal{V}(\tau, \sigma_2) \sqsubseteq L$ or $\not\sqsubseteq L$. First case is easy by the induction hypothesis.

  When $\not\sqsubseteq L$, the interesting case is different branches are taken, say σ_1 evaluates to σ_1' and generates $\mathtt{AS}_1'$ and $\mathtt{NB}_1'$; σ_2 evaluates to σ_2' and generates $\mathtt{AS}_2'$ and $\mathtt{NB}_2'$.

  By the typing rules, we have $\Gamma, pc \sqcup \tau, \mathcal{M}' \vdash c_i$, where $i \in \{1, 2\}$. We denote the triple of variables, values and levels in $(\mathtt{AS}_1' - \mathtt{AS}_1) \cup (\mathtt{NB}_1' - \mathtt{NB}_1)$ as $\mathtt{assign}(c_1)$ and that for second evaluation $\mathtt{assign}(c_2)$ respectively. By

Lemma 1, we have $\forall(v_1, n_1, \ell_1) \in \texttt{assign}(c_1)\ .\ \mathcal{V}(pc \sqcup \tau, \sigma_1) \sqsubseteq \ell_1$ and $\forall(v_2, n_2, \ell_2) \in \texttt{assign}(c_2)\ .\ \mathcal{V}(pc \sqcup \tau, \sigma_2) \sqsubseteq \ell_2$. Since $\mathcal{V}(\tau, \sigma_1) \not\sqsubseteq L$ and $\mathcal{V}(\tau, \sigma_2) \not\sqsubseteq L$, $\ell_1 \not\sqsubseteq L$ and $\ell_2 \not\sqsubseteq L$. Hence we have $\texttt{AS}'_1 \approx_L \texttt{AS}'_2$ and $\texttt{NB}'_1 \approx_L \texttt{NB}'_2$.

Hence, $\forall(v, n, \ell) \in \texttt{assign}(c_1) \cap \texttt{assign}(c_2)$, we have $\mathcal{T}(v, \sigma'_1) \not\sqsubseteq L$ and $\mathcal{T}(v, \sigma'_2) \not\sqsubseteq L$. That is, σ'_1 and σ'_2 are low-equivalent on these variables.

$\forall(v, n, \ell) \in \texttt{assign}(c_1) - \texttt{assign}(c_2)$, it must be true that $v \notin \texttt{DA}(\eta)$. From the typing rules, $\Gamma, pc \sqcup \tau, \mathcal{M}' \vdash c_i$, where $\mathcal{M}' = \mathcal{M} \cap \texttt{DA}(\eta) \subseteq \texttt{DA}(\eta)$. Since $v \notin \texttt{DA}(\eta)$, $v \notin \mathcal{M}'$ as well. By Lemma 11, $\mathcal{V}(\tau, \sigma_1) \sqsubseteq \mathcal{V}(pc \sqcup \tau, \sigma_1) \sqsubseteq \mathcal{T}(v, \sigma_1)$. Since $\mathcal{V}(\tau, \sigma_1) \not\sqsubseteq L$, $\mathcal{T}(v, \sigma_1) \not\sqsubseteq L$ as well. Since $\sigma_1 \approx_L \sigma_2$, we have $\mathcal{T}(v, \sigma_2) \not\sqsubseteq L$ by Lemma 7. Since v is not assigned in c_2, $\sigma'_2(v) = \sigma_2(v)$. Hence, $\mathcal{T}(v, \sigma'_2) \not\sqsubseteq L$. Therefore, σ'_1 and σ'_2 agrees on these variables as well. Similarly, we can prove for all v in $v \in \texttt{assign}(c_2) - \texttt{assign}(c_1)$.

$\forall v \notin \texttt{assign}(c_1) \cup \texttt{assign}(c_2)$, v is not assigned in both c_1 and c_2. Hence, $\sigma'_1(v) = \sigma_1(v) \wedge \sigma'_2(v) = \sigma_2(v)$. Therefore, σ'_1 and σ'_2 must agree on v since $\sigma_1 \approx_L \sigma_2$.

□

Next, we prove several lemmas that are useful for the results on the thread-level semantics.

Lemma 12 (High PC)

$$\forall pc, \sigma, c\ .\mathcal{V}(pc, \sigma) \not\sqsubseteq L \wedge (\vdash \Gamma \wedge \Gamma, pc, \emptyset \vdash c) \wedge \langle \sigma, c, \texttt{AS}, \texttt{NB} \rangle \Downarrow \langle \sigma', \texttt{AS}', \texttt{NB}' \rangle \Rightarrow$$
$$\sigma \approx_L \sigma' \wedge \texttt{AS} \approx_L \texttt{AS}' \wedge \texttt{NB} \approx_L \texttt{NB}'$$

Proof. For variables not assigned to in c, σ and σ' are trivially low-equivalent for them. Moreover, $\texttt{AS}, \texttt{AS}', \texttt{NB}, \texttt{NB}'$ contain no events for these variables.

Now consider any v that is assigned to in c. By Lemma 1, $\mathcal{V}(pc, \sigma) \sqsubseteq \mathcal{T}(v, \sigma')$. Since $\mathcal{V}(pc, \sigma) \not\sqsubseteq L$, $\mathcal{T}(v, \sigma') \not\sqsubseteq L$ as well. For the label of v in σ, because the typing rules for commands only narrows the set $\mathcal{M}$, which is $\emptyset$, $\mathcal{M}$ must be $\emptyset$ when any assignment in c is typed. Hence, by Lemma 3, $\mathcal{V}(pc, \sigma) \sqsubseteq \mathcal{T}(v, \sigma)$. Given $\mathcal{V}(pc, \sigma) \not\sqsubseteq L$, $\mathcal{T}(v, \sigma) \not\sqsubseteq L$ for any v assigned to in c. Hence, σ and σ' are low-equivalent for variables assigned in c as well. Moreover, by Lemma 3 and 4, $\mathcal{V}(pc, \sigma) \sqsubseteq \mathcal{V}(pc, \sigma) \sqcup \mathcal{V}(\tau, \sigma) \sqsubseteq \ell$, where ℓ is label of the generated assignment events in $\texttt{AS}' - \texttt{AS}$ and $\texttt{NB}' - \texttt{NB}$. Since $\mathcal{T}(pc, \sigma) \not\sqsubseteq L$, $\ell \not\sqsubseteq L$ as well.

Therefore, $\sigma \approx_L \sigma' \wedge \texttt{AS} \approx_L \texttt{AS}' \wedge \texttt{NB} \approx_L \texttt{NB}'$. □

Lemma 13 (Stable event labels) *If an assignment event (v, n, ℓ) is produced in some clock cycle with cycle counter t, then $\mathcal{T}(v, \sigma) = \ell$ for all configurations $\langle t', \sigma, \vec{c}, B, \texttt{NB}\rangle$ where $t' = t$ after the event is produced.*

Proof. By the semantics, we know there exists some store σ_0, when the event is generated, such that $\mathcal{T}(v, \sigma_0) = \ell$. If $\mathcal{T}(v, \sigma_0) \neq \mathcal{T}(v, \sigma)$, it must be true that at least a variable in $\texttt{FV}(\Gamma(v))$ is modified in between. However, by the rule for dynamic erasure of contents (S-ASGN1), modifying any variable that v's type depends on resets the value of v. This contradicts the race-free assumption. Hence, $\mathcal{T}(v, \sigma) = \mathcal{T}(v, \sigma_0) = \ell$. □

Lemma 14 (Delayed assignment) *If $\texttt{NB}_1 \approx_L \texttt{NB}_2$ and $\sigma_1 \approx_L \sigma_2$, then we have*

$$\texttt{apply}(\sigma_1, \texttt{NB}_1) \approx_L \texttt{apply}(\sigma_2, \texttt{NB}_2)$$

Proof. By induction on the length of $\texttt{NB}_1$ and $\texttt{NB}_2$. Denote the first events $\texttt{NB}_1$ and $\texttt{NB}_2$ as (v_1, n_1, ℓ_1) and (v_2, n_2, ℓ_2) respectively.

When $\ell_1 \sqsubseteq L$ and $\ell_2 \sqsubseteq L$, it must be true that $v_1 = v_2 \wedge n_1 = n_2$ by the definition of low-equivalence on assignment events. Hence, $\texttt{apply}(\sigma_1, v_1 = n_1) \approx_L \texttt{apply}(\sigma_2, v_2 = n_2)$. The result for the remaining events in $\texttt{NB}_1$ and $\texttt{NB}_2$ is true by induction hypothesis.

When at least one event has a label not bounded by L. Without losing generality, assume $\ell_1 \not\sqsubseteq L$. By Lemma 13, $\mathcal{T}(v_1, \sigma_1) = \ell_1 \not\sqsubseteq L$. We next show $\mathcal{T}(v_1, \sigma_1\{v_1 \mapsto n_1\}) = \ell_1 \not\sqsubseteq L$ as well.

Suppose the event (v_1, n_1, ℓ_1) is generated when the following rule is applied:

$$\frac{\langle e, \sigma_0\rangle \Downarrow n_1 \quad \texttt{NB}' = \texttt{NB} \cup \{(v_1, n_1, \mathcal{T}(v_1, \sigma_0\{v_1 \mapsto n_1\}))\}}{\langle \sigma_0, v_1 \Leftarrow_\eta e, \texttt{AS}, \texttt{NB}\rangle \Downarrow \langle \sigma_0, \texttt{AS}, \texttt{NB}'\rangle}$$

By the dynamic erasure of contents rule (S-ASGN1), $\forall x \in \texttt{FV}(v_1)\ .\ \sigma_0(x) = \sigma_1(x)$. Hence, $\mathcal{T}(v_1, \sigma_1\{v_1 \mapsto n_1\}) = \mathcal{T}(v_1, \sigma_0\{v_1 \mapsto n_1\}) = \ell_1 \not\sqsubseteq L$.

Therefore, $\texttt{apply}(\sigma_1, (v_1, n_1, \ell_1)) \approx_L \sigma_1 \approx_L \sigma_2$. The result for the remaining events in $\texttt{NB}_1$ and $\texttt{NB}_2$ is true by the induction hypothesis.

□

Before proving our final soundness theorem, we first define a low projection of command sequence. In the thread-level semantics, a sequence of commands to be executed, $\vec{c}$ as the third element in the configuration, can either be part of a sequential logic, or part of a combinational logic triggered by some event (v, n, ℓ) according to the semantics. We write $\vec{c} \upharpoonright L$ as the longest subsequence

of $\vec{c}$ such that any command in $\vec{c} \upharpoonright L$ is not triggered by some event (v, n, ℓ) where $\ell \not\sqsubseteq L$. Two command sequences are low-equivalent ($\approx_L$) when they have identical L-projections. Similarly, two sets of threads are low-equivalent when their command sets are low-equivalent.

We define a low-equivalent relation thread-level configurations in the following way:

$$\langle t_1, \sigma_1, \vec{c_1}, B_1, \mathtt{NB}_1 \rangle \approx_L \langle t_2, \sigma_2, \vec{c_2}, B_2, \mathtt{NB}_2 \rangle \Leftrightarrow$$
$$t_1 = t_2 \wedge \sigma_1 \approx_L \sigma_2 \wedge \vec{c_1} \approx_L \vec{c_2} \wedge B_1 \approx_L B_2 \wedge \mathtt{NB}_1 \approx_L \mathtt{NB}_2$$

Proof of Theorem 4 [Soundness]

$$\vdash \Gamma \wedge \Gamma \vdash \mathtt{Prog} \Rightarrow \sigma_1 \approx_L \sigma_2 \wedge \langle \sigma_1, \mathtt{Prog} \rangle \hookrightarrow T_1 \wedge \langle \sigma_2, \mathtt{Prog} \rangle \hookrightarrow T_2 \implies T_1 \approx_L T_2$$

Proof. We prove a stronger result, a thread-level noninterference result for the small-step semantics of threads. The desired result is a direct implication of this stronger result. We proceed by induction on the number of steps in the thread-level semantics.

Base case: given σ_1, σ_2 and `Prog`, the machine starts from the states $\langle 0, \sigma_1, \mathcal{S}, C, \emptyset \rangle$ and $\langle 0, \sigma_2, \mathcal{S}, C, \emptyset \rangle$, where $\mathcal{S}$ and C are the sequential and combinational threads of `Prog`. So the initial configurations are low-equivalent.

For the induction step, we induct on the number of steps in the thread-level semantics. To simplify notation, we refer to the configurations before (after) the semantic steps starting from σ_1 as $\mathtt{conf}_1$ ($\mathtt{conf}'_1$), and that from σ_2 as $\mathtt{conf}_2$ ($\mathtt{conf}'_2$).

- (S-ADV): since $\vec{c_1} \approx_L \vec{c_2}$, either the same command is executed in $\texttt{conf}_1$ and $\texttt{conf}_2$, or at least one command executed is from combinational logic, and it is triggered by some event (v, n, ℓ) where $\ell \not\sqsubseteq L$ by definition.

  In the first case, by Theorem 3, $\sigma_1' \approx_L \sigma_2'$, $\texttt{AS}_1' \approx_L \texttt{AS}_2'$ and $\texttt{NB}_1' \approx_L \texttt{NB}_2'$. Since $\texttt{AS}_1' \approx_L \texttt{AS}_2'$, $B \downarrow_{\texttt{AS}}$ may only differ for the ones triggered by (v, n, ℓ) where $\ell \not\sqsubseteq L$. Hence, $\vec{c_1'} \approx_L \vec{c_2'}$, where $\vec{c_1'}$ and $\vec{c_2'}$ are the third dimension of $\texttt{conf}_1'$ and $\texttt{conf}_2'$. Therefore, $\texttt{conf}_1' \approx_L \texttt{conf}_2'$.

  In the second case, we assume it is the command executed under σ_1, say c_1, that is triggered by some assignment (v, n, ℓ) where $\ell \not\sqsubseteq L$. By Lemma 13, $\mathcal{T}(v, \sigma) = \ell \not\sqsubseteq L$. By the typing rule (T-ALWAYS-COMB) and Lemma 9, c_1 can be typed with pc label $\Gamma(v)$ and $\mathcal{M} = \emptyset$. Hence, by Lemma 26, we have $\sigma_1' \approx_L \sigma_1 \wedge \texttt{NB}_1' \approx_L \texttt{NB}_1$ and $\emptyset \approx_L \texttt{AS}_1$. Hence, commands in $B \downarrow_{\texttt{AS}_1}$ are all triggered by (v, n, ℓ) where $\ell \not\sqsubseteq L$. So, $B_1 - B_1 \downarrow_{\texttt{AS}_1} \approx_L B_1$.

  Therefore, $\texttt{conf}_1' \approx_L \texttt{conf}_1 \approx_L \texttt{conf}_2$.

- (S-TRANS): by Lemma 14, $\sigma_1' \approx_L \sigma_2'$. Since $\texttt{NB}_1 \approx_L \texttt{NB}_2$, $B \downarrow_{\texttt{NB}}$ may only differ for the ones triggered by (v, n, ℓ) where $\ell \not\sqsubseteq L$. Hence, $B_1 \downarrow_{\texttt{NB}_1} \approx_L B_2 \downarrow_{\texttt{NB}_2}$. Therefore, $\texttt{conf}_1'$ and $\texttt{conf}_2'$.

- (S-CLOCK): trivial.

□

3.6 Related work

Verifiable secure hardware Dynamic information flow tracking is applied at the logic-gate level in GLIFT [84, 82, 62, 63, 83]. Dynamic checks in the initial GLIFT design [84] add high overheads in area, power, and performance. Subsequent work [62, 63, 83] checks designs before fabrication, but enumerates all possible states through gate-level simulation, an approach unlikely to scale to large designs without rigid time-and-space multiplexing. SecVerilog allows more flexible resource sharing and identifies security issues early in the design process.

Sapper [48] also adds logic for tracking information flows, incurring run-time overhead. Sapper cannot capture the dependencies between types and values needed for complex security policies. For example, it would not be possible to use the label `LH(timingLabel)` for variable `stall`, as shown in Figure 3.2(b), to capture the policy that the label of `stall` must be L when `timingLabel` is 0.

Caisson [49] supports static analysis but with purely static security levels that prevent fine-grained sharing of hardware resources across security levels. E.g., `write_enable`, `tag_in` and `stall` in Figure 3.2 would require duplication (per security level) since their labels cannot be determined at compile time. Duplicated resources must be controlled by extra encoders and decoders, adding run-time overhead (Section 6.3).

Dynamic security labels Some prior type systems for information flow also support limited forms of dynamic labels [59, 97, 85, 39, 81, 31, 52]. The type-valued functions needed to express the communication of security levels at the

hardware level are absent in most of these, and none permit dynamic labels to depend on mutable variables, a feature key to allowing SecVerilog to verify practical hardware designs. The modular design of the SecVerilog type system makes it more amenable to future extension. Fine [79] and F* [80] can verify stateful information flow policies, modeling state changes with affine types. Affine types suffice for functional programming, but HDLs need SecVerilog's new feature of dependence on mutable variables.

Flow-sensitive information flow control Flow-sensitive information flow control [71, 38, 8], where security labels may change during execution, encounters label channels similar to those observed in our type system. Our type system controls these channels more permissively (Section 3.3.3) because it captures the dependency between types and values.

Dependent type systems Dependent types have been widely studied and have been applied to practical programming languages (e.g., [92, 91, 59, 17, 7]). Information flow adds new challenges, such as precise, sound handling of label channels. RHTT [61] supports rich information flow policies with dependent types, but has much more complex specifications and verification is not automatic.

CHAPTER 4

PREDICTIVE MITIGATION OF TIMING CHANNELS

Strictly disallowing all timing leakage can be done as sketched thus far, but results in an impractically restrictive programming language, or computer system, because execution time is not permitted to depend on confidential information in any way.

To enable general computation with a strict bound on timing channel leakage while providing practical performance, this chapter introduces a general framework called *predictive mitigation*. We start from a simple black-box system model, and then generalize it to interactive systems.

4.1 Simple mitigation schemes

4.1.1 Black-Box system model

We begin with a simple model of a computing system that produces externally observable events whose timing may introduce a timing channel. Because the mitigation scheme works regardless of the internal details of computation, the

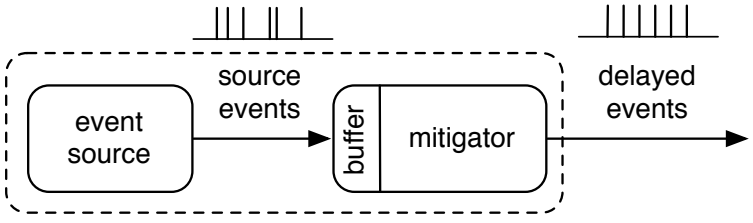


Figure 4.1: System overview.

computing system is treated simply as a black-box source of events. As depicted in Figure 4.1, the event source generates events that are delayed by the mitigation mechanism so that their times of delivery convey less information. Delaying events while preserving their order is the only behavior of the mitigator that we consider.

We ignore for now the attributes of events other than time. These attributes include the actual content of an event and also the choice of communication medium (e.g., different networks, or even visual displays or sound) over which it can be conveyed. Both content and choice of medium can be viewed as storage channels [47], which we assume are controlled by other means. Therefore we assume that the only information requiring control is encoded in the times at which events arrive from the source. This separate treatment of timing and storage channels is justified in Section 4.2.5.

We assume the attacker observes delayed events and knows the design of the mitigator though not its internal state. The goal of the attacker is to communicate information from inside the event source to the outside. Therefore the attacker consists of two parts: an *insider* that controls the timing of source events, and an external *observer* that attempts to learn sensitive information from this timing channel. The *content* of the events may also be observable to the attacker, which motivates our choice to not have the mitigator generate dummy events. The observer may combine information from both the content and timing of messages. In real world this corresponds to attacker-controlled software that communicates seemingly benign messages on a storage channel, but transmits sensitive information using timing.

As shown in the figure, it is useful to allow the mitigation system to buffer

events in a queue so that the event source can run ahead, generating more events without waiting for previous event to be delivered. We consider adding input events to the system model in Section 4.2.6.

4.1.2 Leakage measures

We can bound the amount of information that leaks through the adversary's observations through a combinatorial analysis of the number of possible distinct observations the adversary can make. An observation consists of a sequence of times at which events are released by the mitigator. For a total number of n possible distinct observations, the information leakage is at most $\log(n)$ bits.

Two other ways to measure information leakage have recently been popular. The information-theoretic measure of *mutual information* has a long history of use; it is advocated, for example, by Denning [21], and has been used for the estimation of covert channel capacity, including timing channel capacity, in much prior work (e.g., [54, 55, 57]). Recently, *min-entropy leakage* has become a popular measure, motivated by the observation that two systems with the same leakage according to mutual information may have very different security properties [77]. Fortunately, the combinatorial analysis used here is sufficiently conservative that it bounds both the mutual information and the min-entropy measures of leakage.

The information-theoretic (Shannon) entropy of a finite distribution X over its n possible values is written as $\mathcal{H}(X)$. It achieves its maximal value of $\log(n)$ bits when all n possible values have equal probability. Suppose that O is the distribution over n possible timing observations by the adversary, and S is the

distribution over possible secrets that the adversary wants to learn. The mutual information between O and S, written $\mathcal{I}(O;S)$, is equal to $\mathcal{H}(O) - \mathcal{H}(O|S)$, where $\mathcal{H}(O|S)$ is the conditional entropy of O on S—how much entropy remains in O once S is fixed. In our context, the conditional entropy describes how effectively the adversary encodes the secrets S into the observations O. But since conditional entropy is always positive, the mutual information between O and S is at most $\mathcal{H}(O)$, or $\log(n)$.

Smith argues [77] that the min-entropy of a distribution is a better basis for assessing the vulnerability introduced by quantitative leakage because it describes the chance that an adversary is able to guess the value of the secret in one try. The min-entropy of a distribution is defined as $\mathcal{H}_\infty(O) = -\log V(O)$ where $V(O)$ is the worst-case *vulnerability* of O to being guessed: the maximum over the probabilities of all values in O. Let us write $P(o|s)$ for the probability of observation o given secrets s. Köpf and Smith [46] show that the min-entropy channel capacity from S to O is equal to $\log \sum_{o \in O} \max_{s \in S} P(o|s)$. This capacity is maximized when $P(o|s) = 1$ at all o, in which case it is equal to $\log(n)$. Therefore $\log(n)$ is a conservative bound on this measure of leakage as well.

4.1.3 Quantizing time

A very simple mitigation scheme that has been explored in prior work [29, 27, 13] permits events to leave the mitigator only at scheduled times that are multiples of a particular time quantum q. We refer to the times when events are permitted as *slots*, which in this case occur at times $q, 2q$, etc. Without loss of generality, let us use $q = 1$ to analyze this scheme.

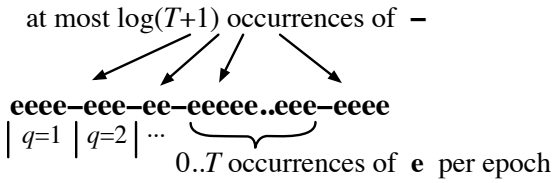
Suppose we allow the system to run for time T, and during that time there is an event ready to be delivered in every slot except that at some point the event source may stop producing events (effectively, it terminates). The total number of events delivered must be an integer between 0 and T. Because all the slots filled with events precede all the empty slots, the external observer can make at most $T + 1$ possible distinct observations. According to information theory, the maximum amount of information that can be transmitted by one of $T + 1$ possible observations is achieved when the possible observations are uniformly distributed. This value, in bits, is the log base 2 of the number of possible observations, or $\log(T + 1)$. For $q \neq 1$, it is $\log(\frac{T+1}{q})$.

4.1.4 A basic mitigation scheme: fast doubling

An asymptotically logarithmic bound on leakage sounds appealing, but such leakage bounded is achieved only for an event source that fills every slot with an event. In the general case, maximum leakage from the simple quantizing approach is one bit per quantum, leading to an unpleasant tradeoff between security and performance.

However, a sublinear (in fact, polylogarithmic) bound is achievable even if the event source misses some slots. Perhaps the simplest way to achieve this is to double the quantum q every time a slot is missed, which we call *fast doubling*. Doubling the quantum ensures that in time T there can be at most $\log(T + 1)$ misses. Effectively, the event source is penalized for irregular behavior. For the penalty will be effective, the multiplicative factor need not be 2; the number of misses will grow logarithmically for any multiplicative factor greater than 1.

We can represent all behaviors of the resulting system as strings constructed from the symbols **e** (for an event that fills a slot) and − (for a missed slot). A given string generated by the regular expression $(\mathbf{e}|-)^*$ precisely determines the times at which events emerge from the mitigator, so the distinct strings correspond exactly to the possible external timing observations. Therefore, the maximum of the expected number of bits of information transmitted by time T is the log base 2 of the number of strings that can be observed within time T. These strings contain at most $\log(T+1)$ occurrences of −. Between and around these occurrences are consecutive sequences of between 0 and T filled slots (**e**'s), as suggested by this figure:



Each sequence of **e**'s falls into a different *epoch* with its own characteristic quantum. There are at most $\log(T+1)+1$ epochs, so the number of possible strings observable within time T is at most $(T+1)^{\log(T+1)+1}$. The maximum information content of the timing channel is the log of this number, or $\log(T+1)\cdot(\log(T+1)+1) = \log^2(T+1)+\log(T+1)$. This is bounded above by $(1+\epsilon)\log^2 T$ where ϵ can be made arbitrarily small for sufficiently large T. With the more careful combinatorial analysis given next, we can show leakage is bounded by $O(\log T(\log T - \log\log T))$. In either case, timing leakage is $O(\log^2(T))$, which is a slowly growing function of time.

A more precise bound on leakage of the basic scheme This derivation is based on the fact that each possible string can be determined by the placement

of the misses, that is, the locations of "–" in the string. For m misses in time T, there are at most $\binom{T}{m}$ different strings. So

$$\begin{aligned}\text{All possible strings} &\leq \sum_{m=0}^{\log T} \binom{T}{m} \leq (\log T + 1)\binom{T}{\log T} \\ &\leq (\log T + 1)\frac{T^{\log T}}{(\log T)!}\end{aligned}$$

Thus, the leakage can be no more than $\log(\log T + 1) + \log^2 T - \log((\log T)!)$, and by Stirling's approximation,

$$\log((\log T)!) = \log T \log\log T - \log T + o(\log T)$$

So the whole leakage term is $O(\log T(\log T - \log\log T)))$.

4.1.5 Slow-doubling mitigation

Doubling on every miss performs poorly if the event source is quiescent for long periods. The quantum-doubling scheme can be refined further to accommodate quiescent periods, by doubling the quantum only when a missed slot follows a filled slot (that is, a – after an **e**). With this mitigator, no performance penalty is suffered by an event source that is initially quiescent, but then generates all its output in a rapid series of events.

In this case we have epochs consisting of sequences like "––––" and "**eeee**". There can be at most $2\log(T + 1)$ epochs, and there can be at most T strings per epoch, so the information content of the channel is no more than $2\log(T)\log(T + 1) \leq (2 + \epsilon)\log^2 T$. Thus, slow doubling gives much more flexibility without changing asymptotic information leakage.

In the next section we see that both the fast and slow doubling schemes are

instances of a more general framework for epoch-based timing mitigation, enabling further important refinements such as adaptively reducing the quantum.

4.2 General epoch-based mitigation

The common feature of the mitigation schemes introduced thus far is that the mitigator divides time into epochs. During each epoch the mitigator operates according to a fixed schedule that predicts the future behavior of the event source. As long as the schedule predicts behavior accurately, the event source leaks no timing information except for the length of the epoch. However, a misprediction by the mitigator causes it to construct a new schedule; because this choice is in general observable by the adversary, some information leaks.

We can describe the mitigation schemes seen so far in these terms. For example, the slow doubling scheme has "**e**" epochs in which the mitigator predicts there will be an event ready for slots spaced at the current quantum q. It also has "$-$" epochs in which the mitigator predicts there will be no event ready for slots spaced at the quantum q. On a mispredicted slot (a miss) during an "**e**" epoch, the mitigator switches to a "$-$" epoch with a doubled quantum.

Let us now explore this framework more formally, to enable generating and analyzing a variety of mitigation schemes that meet specified bounds on timing channel transmission.

4.2.1 Mitigation

The mitigator is oblivious to the content of the events and does not alter their content. From the mitigator's point of view, source events and delayed events are considered abstractly as timestamps at which the events are received and delivered respectively.

Let source events be denoted by a monotonic sequence $s_1 \dots s_n$, where $0 \leq s_1$ and each s_i specifies when the i-th event is received by the mitigator. We denote the mitigator by M. Given a sequence of source events $s_1 \dots s_n$, let $\mathsf{M}(s_1 \dots s_n)$ be the sequence of possibly delayed timestamps $d_1 \dots d_m$ produced by the mitigator. The sequence is again monotonically increasing; also, we have $m \leq n$ and $s_i \leq d_i$. The last inequality means the mitigator cannot produce events before they are received from the source.

A mitigation scheme is *online* if the delayed sequence does not depend on timing or contents of future source messages. In this dissertation we are only interested in online mitigation schemes.

Timing leakage Because timing of the events may depend on the sensitive data at the source, any variation in observed event timing creates an information channel. The larger the number of different observable variations, the more information can be transmitted over this channel. When events are mitigated, the number of possible sequences of events that a mitigator M can deliver by time T is

$$\mathbf{M}(T) = |\{d_1 \dots d_m = \mathsf{M}(s_1 \dots s_n) \mid d_m \leq T\}|$$

The amount of information that can be leaked by such mitigator, when the running time is bounded by T, is a logarithm of $\mathbf{M}(T)$.

Definition 5 (Leakage of the mitigator) *Given a mitigator* M*, the leakage of the mitigator* M *is* $\log \mathbf{M}(T)$*.*

This definition implicitly assumes that the mitigator can control the timing of events with perfect precision, but also credits the adversarial observer with perfect measurement abilities. More realistically, we can assume that the mitigator can control timing to at least the measurement precision of the observer, in which case the above formula still bounds leakage.

Bounding leakage We specify the security requirements for timing leakage as a bound, expressed as a function on running time T.

Definition 6 (Bounding mitigator leakage) *Given a mitigator* M*, and a leakage bound* $B(T)$*, we say that the leakage of* M *is bounded by* $B(T)$ *if for all* T*, we have* $\log \mathbf{M}(T) \leq B(T)$*.*

4.2.2 Epoch-based mitigation

In this dissertation we focus on a specific class of mitigators that we dub *epoch-based* mitigators. An *epoch* represents a period of time during which the behavior of the mitigator meets the epoch *schedule*.

An epoch schedule is a sequence of epoch *predictions*, one for each slot. Epoch predictions can be either *positive* or *negative*. A positive prediction, de-

noted by $[t]^+$, means the mitigator expects to be able to deliver an event at time t. A negative prediction, denoted by $[t]^-$, says that no source events are expected to be available for delivery at time t. We may simply write t when the sign of the prediction is not important for the context. A prediction is an element of $\mathbb{R} \times \{+, -\}$, because times t are real-valued. An epoch schedule S is therefore a function from slot indices (from the natural numbers $\mathbb{N}$) to predictions.

Definition 7 (Epoch schedule) *An epoch schedule is a function* $S : \mathbb{N} \to \mathbb{R} \times \{+, -\}$, *where* $S(n)$ *is a* prediction *for the n-th slot in the epoch.*

We say that a positive prediction $S(n) = [t]^+$ *holds* or *is valid* if at time t, the mitigator can deliver an event; in this case, this is also the n-th event in the epoch. A negative prediction $S(n) = [t]^-$ holds when no source events are available at time t.

Conversely, *failing* a positive prediction $[t]^+$ means that there are no events (available or buffered) to be delivered at time t. Failing a negative prediction $[t]^-$ means that there are buffered source events that have not yet been delivered by time t.

When a mitigator prediction $S(n)$ fails at the n-th slot, we observe an *epoch transition*. In addition to prediction failure, an epoch transition may be caused by *mitigator adjustments*. For example, the mitigator might adjust for a faster rate of source events, or might improve performance by flushing or partially flushing the buffer queue. We can now formally define an epoch:

Definition 8 (Epoch) *An epoch is a triple* (τ, τ', S) *where timestamps* τ *and* τ' *correspond to the beginning and the end of the epoch, and* S *is the epoch schedule.*

When the number of the epoch is important, we write S_N for the schedule in epoch N.

Example Revisiting the basic doubling scheme from Section 4.1.4, we see that the prediction for every N-th epoch that starts at time t is given by the function $S_N(i) = [t + i \cdot 2^N]^+$.

For the slow doubling scheme of Section 4.1.5, every odd prediction is positive—it expects the events to be delivered at regular intervals, and every even prediction is negative—no events are expected from the source. These predictions can be expressed as follows:

$$S_N(i) = \begin{cases} [t + i \cdot 2^k]^+ & \text{if } N = 2k - 1 \\ [t + i \cdot 2^k]^- & \text{if } N = 2k \end{cases}$$

On the form of schedules Most of the examples of schedules in this dissertation are *constant-quantum* functions, where prediction times depend linearly on the epoch sequence number of the events. However, when timing pattern of the source events is well-understood, a finer prediction, described by an arbitrary function, could yield better performance. From the standpoint of security, the form of the schedule is irrelevant as long as the mitigator satisfies the leakage bound discussed in Section 4.2.4.

4.2.3 Leakage of epoch-based mitigators

Epoch-based design allows us to reduce the analysis of epoch-based mitigation to the analysis of individual epochs and of the transitions between them.

Variations within an epoch Because prediction times during an epoch are deterministic, the only source of timing variation within an epoch is the number of valid predictions. The latter is the key element in bounding the number of possible event sequences within an epoch. The number of valid predictions is bounded by the duration of the epoch, which itself is bounded by the current running time $T + 1$. Therefore the current running time $T + 1$ is a bound on the number of variations within each epoch.

Transition variations Epoch transitions may depend on source events too. Therefore, one needs to take into account the number of possible schedules for the next epoch. We denote by Λ_N the number of possible schedules when transitioning from epoch N to epoch $N + 1$.

The exact number of transition variations depends on the particular mitigation scheme. In the two schemes described thus far, the transition into a new epoch occurs only when a miss occurs, and only one new schedule is possible; hence, for all epochs N, we have $\Lambda_N = 1$.

An example mitigator for which Λ_N is greater than 1 is an *adaptive scheme* that uses the average rate of the previously received source events to choose the new schedule. In this case, Λ_N can be bounded by the current running time $T + 1$.

Section 4.3.1 describes the convergence experiment where the number of possible predictions for a given epoch is chosen from a fixed table and is exactly two.

Bound on the number of total variations Consider an epoch-based mitigator at time T that has reached at most N epochs. Assume that within each epoch

the number of variations is at most $T + 1$, and the number of possible transition variations into epoch i is Λ_i, where i ranges from 1 to N. We include Λ_N to accommodate the transition from epoch N to epoch $N+1$ at time T. We can bound the number of possible variations of such a mitigator by a function $\mathbf{M}(T, N)$:

$$\mathbf{M}(T, N) = (T + 1)^N \cdot \Lambda_1 \cdot \Lambda_2 \cdot \ldots \cdot \Lambda_N$$

The leakage of this mitigator is bounded by the logarithm

$$\log \mathbf{M}(T, N) = N \log(T + 1) + \sum_{j=1}^{N} \log \Lambda_j \tag{4.1}$$

We refer to the term $\log(T + 1)$ as *epoch leakage* and to the terms $\log \Lambda_i$ as *transition leakage*.

Basic schemes revisited Revisiting the simple mitigators from Section 4.1.1, we see that because $\Lambda_N = 1$, the leakage of such mitigators is bounded by $N \log(T + 1)$.

4.2.4 Bounding leakage

Using Definition 6 and Equation 4.1 we may derive a leakage bound for epoch-based mitigation.

$$N \log(T + 1) + \sum_{j=1}^{N} \log \Lambda_j \leq B(T) \tag{4.2}$$

Furthermore, if we consider mitigators where the transition variations are fixed—that is, there is $\lambda_{\max} \geq \log \Lambda_j$ for all j—then the leakage bound criterion for such mitigators can be expressed as a bound on the number of epochs.

$$N \leq \frac{B(T)}{\log(T + 1) + \lambda_{\max}} \tag{4.3}$$

Define the *deferral point* D_N for epoch N to be the solution to the equation $N = B(T)/(\log(T+1) + \lambda_{\max})$. The importance of D_N is that until D_N there must be at most N epochs; that is, the start of the $N+1$-th epoch has to be deferred until D_N. Because the $N+1$-th epoch starts with the misprediction at the N-th epoch, this leads us to the only security constraint for the choice of schedule S_N. Namely, for all events i, we should have $\forall\, N \geq 1\,.\, S_N(i) \geq D_N$, and consequently,

$$\forall N \geq 1\,.\, S_N(1) \geq D_N \tag{4.4}$$

Example Consider the basic scheme from Section 4.1.4, which has prediction function $S_N(i) = [t + i \cdot 2^{N-1}]^+$. For this scheme, we have $\lambda_{\max} = 0$. Consider bound $\log^2(T+1)$, which leads to deferral points D_N for N-th epoch $D_N = 2^N - 1$. The leakage bound requires $S_N(1) \geq D_N$. Since in the basic scheme, the N-th epoch starts at 0 for $N = 1$, and at least at time $\sum_{i=0}^{N-2} 2^i$ for all $N \geq 2$, therefore the leakage bound follows from $S_N(1) = 0 + 2^0 = 1 \geq 2^1 - 1 = D_N$ for $N = 1$, and $S_N(1) \geq \sum_{i=0}^{N-2} 2^i + 2^{N-1} = 2^{N-1} - 1 + 2^{N-1} = 2^N - 1 = D_N$ for all $N \geq 2$.

Figure 4.2 shows the deferral points for the basic scheme from Section 4.1.1. Here the bound is $B(T) = \log^2 T$, $\lambda_{\max} = 0$, and the deferral points correspond to the intersections of the curves $N \log T$ with the bound curve.

Adaptive mitigators In the absence of a misprediction for a sufficiently long time, the difference $B(T) - N\log(T+1) - \sum_{j=1}^{N} \log \Lambda_j$ may allow an extra epoch transition. An epoch transition is *adaptive* when it is initiated by the mitigator rather than by a misprediction. Equations 4.2 and 4.3 are useful as design criteria for adaptive transitions. In particular, an adaptive transition is secure when

$$B(T) - N\log(T+1) - \sum_{j=1}^{N} \log \Lambda_j \geq \log(T+1) + \log \Lambda_{N+1}$$

Here Λ_{N+1} is the number of possible new predictions for epoch number $N+1$.

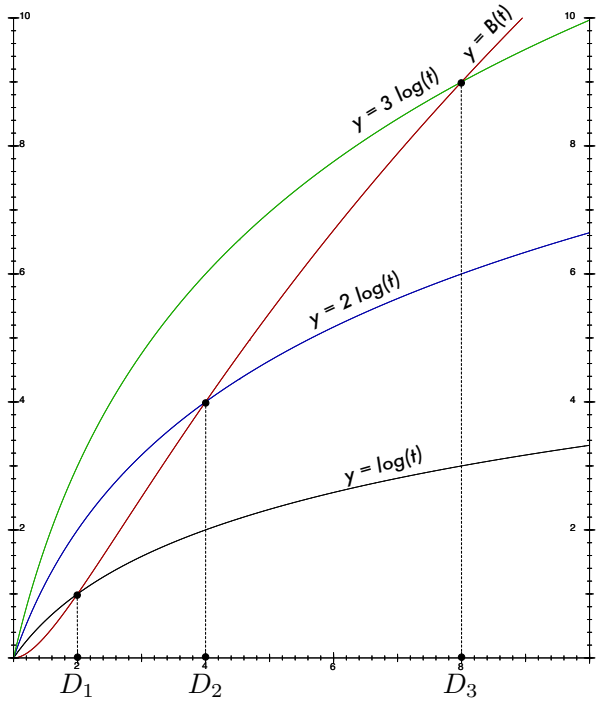


Figure 4.2: Target bound, capacity approximation for individual epochs, and deferral points.

One use for adaptive transitions is to help reduce the size of the event buffer. Past deferral points, the mitigator can choose to release more than one event from the buffer. The number of choices for how many events can be flushed from the buffer then contribute to Λ_N for the mitigation scheme at that deferral point. Prudent mitigator design probably avoids completely emptying the buffer, since an empty buffer may risk an unpredicted miss.

A second example of using adaptive transitions to improve performance is given in Section 4.3.1.

On the choice of bound functions Because the epoch-based mitigation scheme is parametric on the choice of the bound function $B(T)$, we briefly discuss possible choices for practical bounds.

Recall that we assume the number of processed events in an epoch may leak information. Under this assumption, the most draconian bound possible is $\log T$. Enforcing such a bound effectively restricts output to a single epoch for the entire run of the program. In case of a misprediction, all subsequent events would have to be delayed until the end of the program.

Our simple and adaptive mitigators use the polylogarithmic bound $\log^2 T$, which appears to make a reasonable trade-off between performance and security. However, as this section illustrates, even with this more relaxed bound, the distance between deferral bounds increases exponentially.

A third choice corresponds to a larger, more permissive, class of bounds such as $kT + \log^n T$, for $n \geq 2$ and small (or zero) k. We have not explored such bounds in this dissertation, though it is possible that a linear, albeit slowly growing, bound may be useful in bringing deferral points closer in practice.

4.2.5 Mixing storage and timing

A variety of information flow control techniques have been developed for controlling leakage through storage channels. We can now show that these techniques combine well with timing mitigation.

We use the information theoretic measure of mutual information, to measure leakage. Given random variables A and B, their mutual information $\mathcal{I}(A;B)$ is the information that A conveys about B, and vice versa. It is defined as $\mathcal{I}(A;B) = \mathcal{H}(A) + \mathcal{H}(B) - \mathcal{H}(A,B)$, where the function $\mathcal{H}$ gives the entropy of a distribution. Note that the entropy of a variable with n possible values is maximized when all n outcomes are equally probable, in which case it is $\log n$ bits.

Assume X is a random variable that corresponds to secret input, Y is a random variable that corresponds to the storage channel, and Z is a random variable that corresponds to the timing channel. The amount of information that the attacker gains by observing both storage and timing channel is the mutual information between the secret and the joint distribution of Y and Z: that is, $\mathcal{I}(X; Y, Z)$. Similarly, the amount of information that the attacker gains by observing just the storage channel is $\mathcal{I}(X; Y)$.

The following easy theorem states that the amount of information leaked by the combination of timing and storage channels is bounded by the information leaked by the storage channel, plus the maximum information content of the timing channel.

Theorem 5 (Separation of storage channel)

$$\mathcal{I}(X; Y, Z) \leq \mathcal{H}(Z) + \mathcal{I}(X; Y)$$

Proof. We prove the theorem by using the definition of $\mathcal{I}(X; Y)$ to show that the expression $\mathcal{H}(Z) + \mathcal{I}(X; Y) - \mathcal{I}(X; Y, Z)$ is nonnegative.

$$\begin{aligned}
&\mathcal{H}(Z) + \mathcal{I}(X; Y) - \mathcal{I}(X; Y, Z) \\
&= \mathcal{H}(Z) + \mathcal{H}(X) + \mathcal{H}(Y) - \mathcal{H}(X, Y) \\
&\qquad - \mathcal{H}(X) - \mathcal{H}(Y, Z) + \mathcal{H}(X, Y, Z) \\
&= \mathcal{H}(Z) + \mathcal{H}(Y) - \mathcal{H}(X, Y) - \mathcal{H}(Y, Z) + \mathcal{H}(X, Y, Z) \\
&\geq \mathcal{H}(Z) + \mathcal{H}(Y) - \mathcal{H}(X, Y) - \mathcal{H}(Y) - \mathcal{H}(Z) + \mathcal{H}(X, Y, Z) \\
&= \mathcal{H}(X, Y, Z) - \mathcal{H}(X, Y) \geq 0
\end{aligned}$$

□

A symmetric theorem can be stated for the timing channel, but seems less useful because of the difficulty of estimating $\mathcal{I}(X;Z)$. A direct corollary to this theorem is that if the system enforces noninterference [28, 54] on the storage channel, the total secret information leaked from the system is bounded by the entropy of the timing channel.

4.2.6 Input

Event sources often communicate with the external world by accepting input, and block waiting for input when no input is available. Let us assume that the timing of input does not contain sensitive information, or at least that it is the responsibility of the input provider to control the input timing channel. The time spent by a computing system waiting for input clearly does not communicate anything about its internal state provider. Therefore, the system comprising the event source and mitigator should not be penalized for time spent blocked waiting for input. For the purposes of mitigation, the clock controlling the scheduling of slots can be stopped while the event course is blocked waiting for input. This refinement is particularly helpful when the event source is a service whose service time does not fluctuate much.

4.2.7 Leakage with beliefs about execution time

Finally, we consider a particular case of *server applications* that handle client requests. In this special case, a tighter, albeit probabilistic, bound on leakage can be established than is possible with the general framework presented thus far.

For many real applications that handle sequential client requests, such as RSA encryption and simple web services (see Section 4.3.4), execution times fall within a narrow range, regardless of the values of secrets. We show that under the assumption that the distribution of execution times is approximately as expected, expected mitigated leakage can be given a tighter bound than $\log^2 T$.

Suppose that with probability at least p, the execution time for a single request is at most T_{big}. That is, the adversarial insider controls execution time but cannot make the probability of exceeding T_{big} greater than $1 - p$. For some computations, such as blinded cryptographic operations on sufficiently isolated computers, p can be gained by sampling with randomly generated inputs. Given T_{big}, a corresponding number of epochs N_{big} can be calculated, giving the number of transitions that must occur before executions of length T_{big} are possible. For instance, in the basic doubling scheme, $N_{\text{big}} = \lceil \log(T_{\text{big}}) \rceil$. Under these assumptions, expected leakage $L(N_{\text{big}}, T)$ is derived using conditional entropy:

$$L(N_{\text{big}}, T) = p \cdot \log \mathbf{M}(T, N_{\text{big}}) + (1 - p) \cdot \mathbf{M}(T, N)$$

where, as before, $\mathbf{M}(T, N_{\text{big}})$ is the bound on the number of possible variations of a mitigator when N is at most N_{big}.

Example For the basic doubling scheme, given T_{big}, we know that $N_{\text{big}} \leq \lceil \log(T_{\text{big}} + 1) \rceil$. Using the formula for $\mathbf{M}(T, N)$, we can derive

$$\begin{aligned}
L(N_{\text{big}}, T) &= p \cdot \log \mathbf{M}(T, N_{\text{big}}) + (1 - p) \cdot \mathbf{M}(T, N) \\
&= p \cdot (N_{\text{big}} \cdot \log(T + 1) - \frac{N_{\text{big}}(N_{\text{big}} - 1)}{2}) + (1 - p) \cdot (N \cdot \log(T + 1) - \frac{N(N - 1)}{2}) \\
&\leq p \cdot (N_{\text{big}} \cdot \log(T + 1) - \frac{N_{\text{big}}(N_{\text{big}} - 1)}{2}) \\
&\quad + (1 - p) \cdot (\log^2(T + 1) - \frac{\log(T + 1)(\log(T + 1) - 1)}{2}) \\
&= p \cdot (N_{\text{big}} \cdot \log(T + 1) - \frac{N_{\text{big}}(N_{\text{big}} - 1)}{2}) + \frac{1 - p}{2} \cdot (\log^2(T + 1) + \log(T + 1))
\end{aligned}$$

This leakage bound is tighter than $(\log^2 T)$, although they have the same asymptotic complexity of $O(\log^2 T)$.

4.3 Adaptive mitigation results

Some simple experiments with predictive mitigation help us understand how the mitigator can converge on the right separation between slots.

4.3.1 Convergence

Buffering source events helps prevent slowing down the event source and absorbs temporary variations in event rate. However, it is undesirable for the buffer to grow too large, because it increases latency. If the buffer fills, the event source must be paused to allow the buffer to drain. We would like to avoid

significantly pausing well-behaved applications, because pauses could disrupt their functionality.

In this part, we focus on a simplified event source, and propose one way to add adaptive transitions to the basic mitigation mechanism of Section 4.1.4. This particular design typically allows the quantum converge to the event rate, while still keeping the information leakage lower than the desired bound. Empirical results demonstrate the convergence of the mitigator in face of many different event rates. Although currently our solution is restricted to certain input event patterns, the experiment suggests that adaptive, epoch-based mitigation may be practical for different applications.

Suppose we use the simple mitigation mechanism with constant-quantum positive predictions for every epoch. Consider an event source that generates events at a constant rate, say 1 event per every 8 seconds; call the interval between events the *event interval*. When the quantum of the mitigation system is higher than the event interval—say, 10 seconds—the mitigator begins accumulating events in its buffer queue. Eventually, an increase in the buffer size may increase the latency and reduce the throughput of the mitigator. On the other hand, if the quantum is smaller than the event rate—say, 6 seconds—then the buffer quickly drains, causing unwanted epoch transitions.

Therefore, designing an adaptive mitigation scheme that can converge roughly to the event interval of the source system is important for reducing performance overhead for practical applications.

4.3.2 Assumptions

We now show that adaptive mitigation can work for relatively well-behaved event source. To capture the behavior of an event source that generates events at some average rate but with local variation around that rate, we work with an event source that generates one event at a random point during each fixed interval. It is easy to see the optimal prediction for an event source of this type is the one whose constant quantum matches the average interval between events.

The basic intuition behind the construction of the adaptive mitigation mechanism is that the size of the buffer indicates how the quantum should be adjusted. A quantum that is too large causes the buffer to grow large; a quantum that is too small causes the buffer to empty. Both of these conditions can be taken into account by the mitigator.

4.3.3 An adaptive mitigation heuristic

Following the idea of adaptive mitigation from Section 4.2.4, we heuristically extend the basic mitigator of Section 4.1.1 to adjust future schedules based the buffer size. There is no reason to believe that the particular mechanism is optimal; we describe this mechanism as a way of illustrating what is possible with adaptive mitigation.

In this mitigator, an adaptive epoch transition happens when both of the following conditions hold:

1. the size of the buffer queue is increasing, and
2. the mitigator would meet the leakage bound even if it transitioned into a new epoch.

Note that condition (1) here is specific to the design of the current mitigator, while condition (2) is a necessary condition for all adaptive transitions, as described in Section 4.2.4.

The adaptive mitigation heuristic works as follows. It doubles the quantum on each miss transition, which lets the quantum quickly approach the event interval. Next, the mitigator adjusts the quantum closer to the event interval by raising or lowering the quantum deterministically at every adaptive transition. The current quantum ideally fluctuates around the desired quantum and finally converges to it. We constrain the mitigator to have a deterministic reduction rate, enabling a deterministic (and small) bound on possible schedule functions.

This scheme uses reduction rates that regulate how quantum size is adapted. We denote reduction rates by r_j, where j ranges from 1 to 9, such that $r_1 = 0.95$, $r_2 = 0.9$, $\ldots$, $r_9 = 0.55$. Note that the number of reduction rates and the corresponding values for this experiment have been derived empirically, based on the experimental results, reported in Section 4.3.4.

The mitigator has an internal state, which is a pair (q, j). Here q is current quantum and j is the current reduction rate. Call the condition that guards when an adaptive transition may be done an *adaptive condition*; the next state (q', j') is computed at a transition point and is derived as follows:

$$(q', j') = \begin{cases} (q/2r_j, \text{next } j) & \text{if adaptive condition holds} \\ (2q \cdot r_j, \text{next } j) & \text{if miss occurs} \end{cases}$$

where function next specifies the choice of the next reduction rate

$$\text{next } j = \begin{cases} j+1 & \text{when } j < 9 \\ 5 & \text{when } j = 9 \end{cases}$$

Using this new state, the schedule for the next epoch can be computed as $S_N(i) = [\tau + i \cdot q']^+$.

According to our discussion in Section 4.2.4, since the multiplier rates are deterministic, and there are only two possible transitions (speed up or slow down), we have $\lambda_{max} = 1$. If we set the total information leakage $B(T)$ to be $\log^2(T+1)$, the number of transitions must be no more than $(\log^2(T+1))/(\log\ (T+1)+1)$ as derived from the leakage bound criterion in Section 4.2.4. Adaptive transitions are not allowed if this constraint is not met.

4.3.4 Empirical results

Figure 4.3 illustrates how the quantum converges to the event interval through the adaptive mitigation mechanism, with event interval of 18 seconds. In this figure, the quantum is indicated by the dashed line and the buffer size is represented by the solid line. Initially the mitigator doubles the quantum quickly to 32, and then lowers the quantum because the queue size has grown, around the 350-second point. Then, the queue slowly drains because the quantum is smaller than the event interval. When the queue empties around 2000 seconds, the quantum is raised again. After several adjustments, the quantum finally converges to 18 at around 5000 seconds and stays constant thereafter. Once converged, the queue size remains small (around 2–3), ensuring low latency.

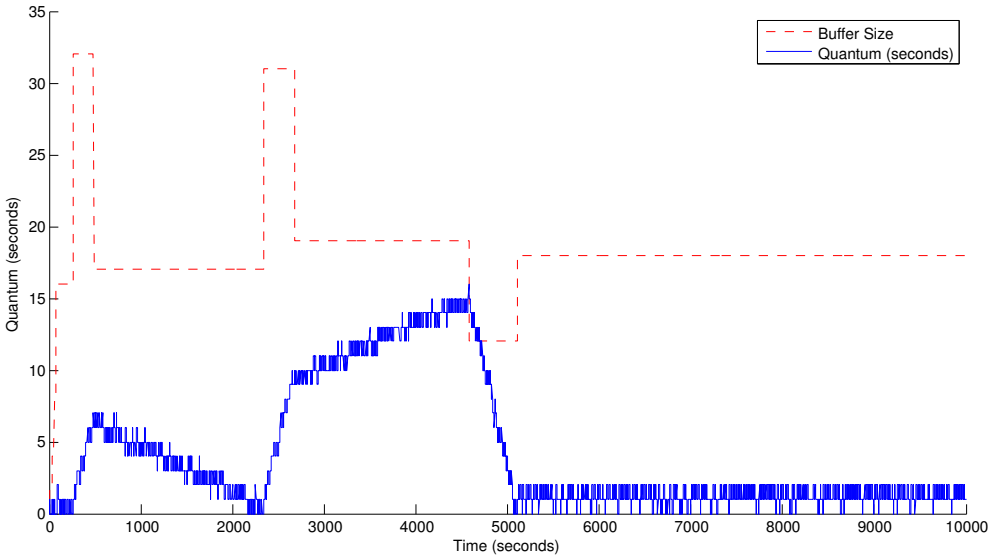


Figure 4.3: Adaptive mitigation with average interval of 18 seconds.

While perfect convergence is not required for this scheme to be useful, we tested the convergence with event intervals ranging from 1 second to 100 seconds, with the results shown in Figure 4.4. Each dot represents the final quantum arrived at by the mitigation system after different total run times. Three curves are shown, one for the final quantum after 10000 seconds, one for after 100000 seconds, and one for after 1000000 seconds. The plot shows that the adaptive mitigation heuristic converges closely to the event interval in most cases. However, there are certain cases where convergence never occurs, such as at an event interval of 42; here, the mitigation system loops among five values close to 42. The current set of reduction rates were chosen in a largely ad hoc fashion; we leave finding an optimal set for a broad class of event sources to future work.

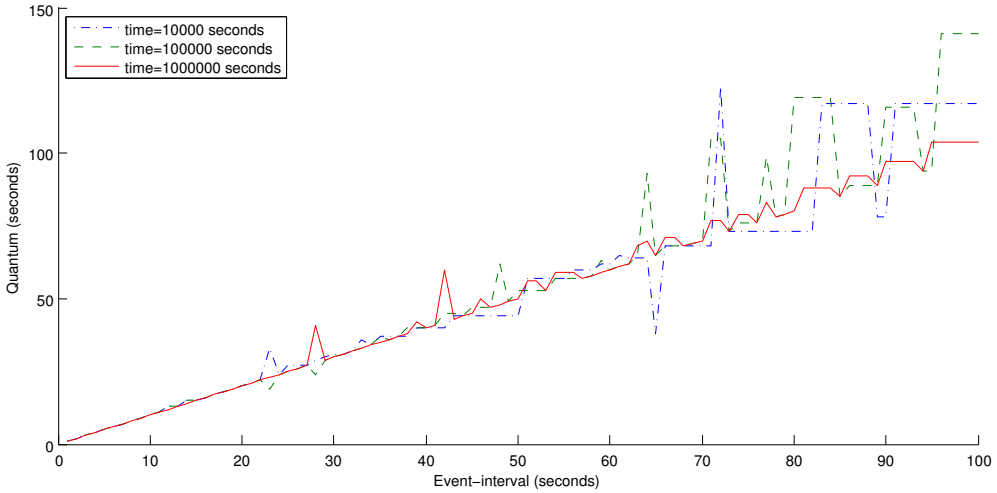


Figure 4.4: Convergence with different event intervals.

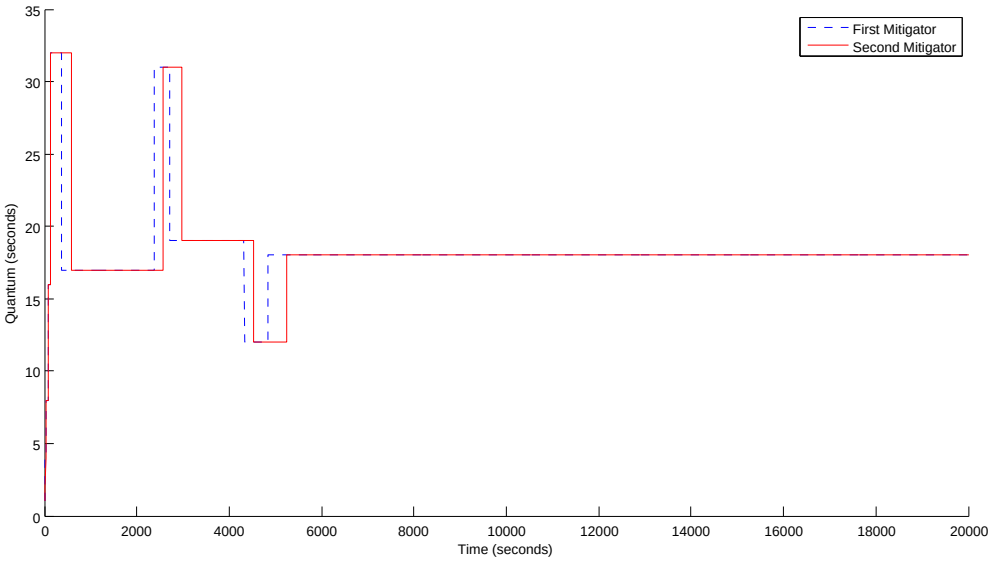


Figure 4.5: Convergence of composition of mitigators with average interval of 18 seconds.

4.3.5 Composing mitigators

Figure 4.5 illustrates convergence of composition of two adaptive mitigators. Here the first mitigator processes events received from a source system with an 18 sec. event interval. The second mitigator processes events that it receives from the first one. The lines on the graph illustrate the change of quantum values in each mitigator. Based on the similar experiments for other event intervals, we observe that composed mitigators converge in most cases. We leave identifying necessary and sufficient conditions for convergence to future work.

4.4 Application-level experiments

We evaluated the effectiveness of the basic timing mitigation mechanism on two published timing channel attacks: RSA timing channels [13] and remote web server timing channels [11].

Both experiments show that the basic mitigation mechanism of Section 4.1.4 can successfully defend against these timing channel attacks, although with a latency penalty.

4.4.1 RSA

To demonstrate the effectiveness of timing mitigation, we applied it to OpenSSL 0.9.7, a widely used open-source SSL library that was shown to be vulnerable to RSA timing channel attacks [13]. The results show that timing mitigation eliminates the time difference targeted by RSA timing channel attack, making this attack infeasible.

Experiment setup

The experiment was performed on OpenSSL 0.9.7. This version was used because it is the same version shown to be vulnerable by Brumley et al, and by default it does not use blinding to prevent timing channels. Measurements were made on a 3.16GHz Intel Core2 Due CPU, with 4G of RAM, using GCC 4.4.1. The attacker continuously asks the target to decrypt a message and records all decryption times, starting a new decryption request whenever the last one is done. The Intel CPU cycle count obtained using the `rdtsc` instruction provided a precise, accurate clock.

Attack strategy

We used the timing channel attack strategy proposed in [13] for this experiment, attacking RSA keys with 1024 bits. Instead of trying to get the secret key directly, this attack targets the smaller factor of N used in RSA key generation. More specifically, the attacker attacks q, where $N = pq$ with $q < p$. Once q (512 bits for a 1024-bit key pair) is released, the attacker can easily derive the secret key by computing $d = e^{-1} \mod (p-1)(q-1)$.

The attack works by learning a bit of q at a time, from most significant to least. In each request, the attack generates two guesses (512-bit numbers) and records the decryption time for each guess. To set up, the attacker guesses the first 2–3 bits of q by trying all possible combinations (feeding rest of the bits as 0), and plots all decryption times in a graph where the x-axis is all guesses. The first peak in the graph corresponds to q. Once the attacker has recovered the top $i-1$ bits of q, two new guesses g_1 and g_2 are generated, where

1. g_1 has the same top $i-1$ bits as q and the remaining are zero.
2. g_2 differs from g_1 only at the i^{th} bit, by setting it to 1

The attacker then computes $u_{g_1} = g_1 R^{-1} \mod N$ and $u_{g_2} = g_2 R^{-1} \mod N$ (where R is some power of 2 used in Montgomery Reduction), and measures the time to decrypt both u_{g_1} and u_{g_2}. Denote by Δ the difference between these two decryption times.

The goal of this RSA timing channel attack is to find a 0–1 gap when a certain bit of q is 0 or 1. More specifically, when the i^{th} bit of q is 0, the decryption time difference Δ will be large, otherwise small. So the attacker wins by analyzing the significance of 0–1 gap to get all bits of q. Actually, after recovering the most significant half of the bits of q, attacker can use Coppersmith's algorithm [19] to recover the rest of the bits. So we only show the 0–1 gap for the first 256 bits of q in this experiment.

Parameter choices

To overcome the effects of a multi-user environment, multiple decryptions for same guesses are necessary to cancel out the timing differences. Experimentally, we found the median time of 7 samples gives a reliable decryption time with very small variation, so this is the sample size used hereafter.

Measuring the decryption time for $n+1$ guesses ranging from $g, g+1, \ldots, g+n$ can make the 0–1 gap more significant, and thus brings more confidence in the attacker's guess, though at a computational cost to the attacker as well [13]. We chose 600 as the value for n, because it was enough to gain a significant 0–1 gap in most cases.

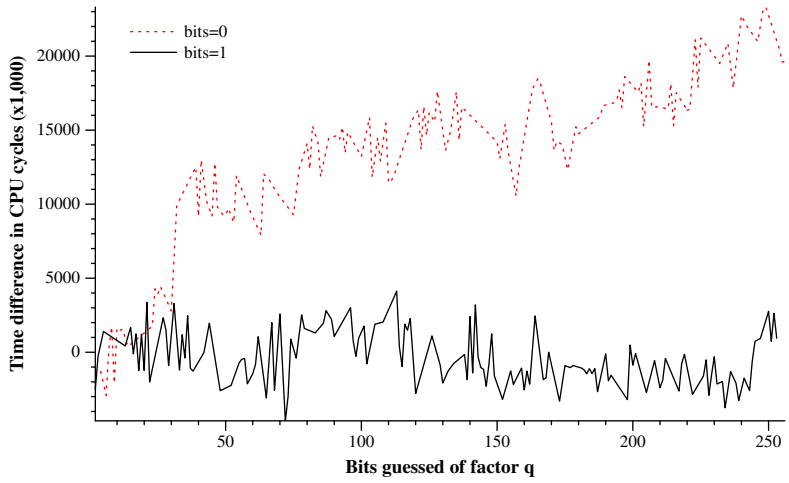
Timing mitigation of RSA Attack

Since the 0–1 gap does not depend on any specific key [13], we used a randomly generated 1024-bit key for our experiment. Figure 4.6(a) shows the result without any timing mitigation mechanism. The dotted line is the zero–one gap when the corresponding bit of q is 1, and the solid one is when the bit is 0. It is easy to see that an attacker can infer certain bits of q by observing this zero–one gap, especially when guessing a bit whose position is larger than 30. For bit indices less than 30, it is possible to increase the 0–1 gap by calculating a larger neighborhood set, with more cost to the attacker.

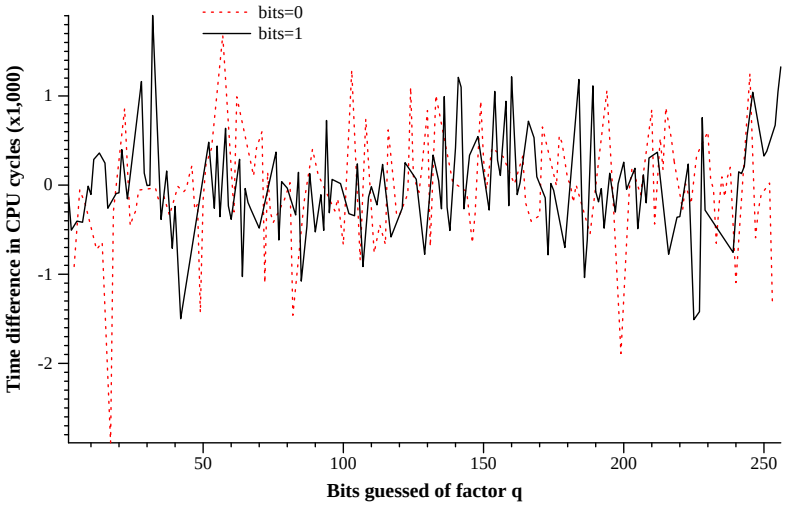
On the other hand, Figure 4.6(b) shows a RSA decryption process with the simple timing mitigation mechanism we proposed in Section 4.1.1. The timing channel attack on RSA is defeated because the two curves are indistinguishable regardless of which bit is being guessed: our timing mitigation scheme eliminates the 0–1 gap. The mitigation mechanism makes the time difference drop by four orders of magnitude, because the only source of time difference is the request time, which does not depend on the currently guessed bit.

Expected leakage

If we are willing to make assumptions about the distribution of encryption time, we can apply the method for estimating expected leakage that is discussed in Section 4.2.7. Using 1000 randomly generated inputs to estimate T_{big}, we find that 99% of them are handled within $1 * 10^8$ clock cycles, which is approximately $\frac{1\times10^8\times10^3}{3.16\times10^9} = 31.65$ ms (this is a 3.6GHz CPU). With an initial quantum of 1 ms, it is easy to see that $N_{\text{big}} = \lceil \log(31.65) \rceil = 5$. The leakage bound in this case is shown in Figure 4.7, topping out for practical purposes around 100 bits.



(a) Without mitigation



(b) With mitigation

Figure 4.6: Simple mitigation of the RSA timing attack.

4.4.2 Timing attacks on web servers

Web applications have been shown vulnerable to timing channel attacks, either by direct timing or cross-site timing. For instance, many web applications try to keep secret whether a given username is valid, by returning the same error message regardless of validity. They do this because learned usernames can be abused for spam, invasive advertising, and phishing. However, timing can

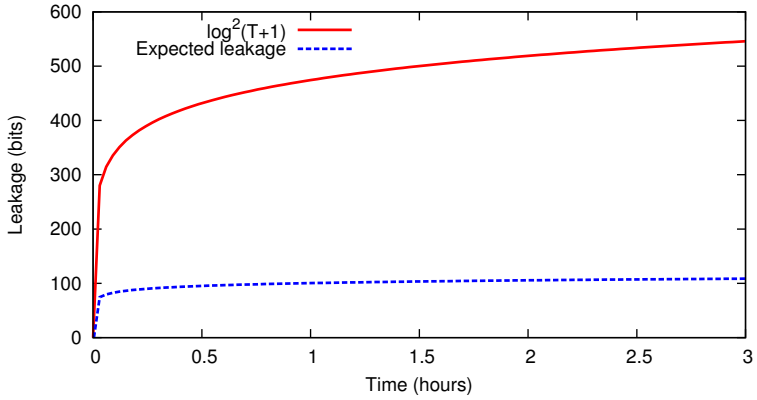


Figure 4.7: Expected leakage for RSA timing channel attack.

expose username validity, because sites usually execute different code paths for valid and invalid user names [11].

We implemented a simple web server to expose this timing channel and applied our mitigation scheme to eliminate this channel. The result shows that our mitigation mechanism is also useful in the face of web applications, although with a latency cost.

Experimental setup

We build a small HTTP web service on Tomcat 5.5.28. It takes a username/password pair as a request and checks its validity. We randomly generate 10,000 username/password pairs, and store the username with a SHA-1 password hash of passwords into Berkeley DB (Java Edition, 4.0.92) [64]. This experiment is done between two computers connected by a campus network.

The login service proceeds as follows: first, it checks the database for validity of the given username. If the username is invalid, this server just returns

an error message. Otherwise, the server computes the SHA-1 hash of the given password and checks if it matches the one stored in the database. If the password does not match, the server returns the same error message as for an invalid username, to conceal username validity. This captures the essence of a login service. However, despite its simplicity, this service also exhibits a possible timing channel, because the computation of the SHA-1 hash depends on username validity.

To reduce network timing noise, we measure the query time 20 times for each username, and choose the smallest one as our sample. For each experiment, we randomly choose 400 valid usernames from a valid username list, as well as 400 randomly generated invalid usernames to determine the timing difference between them. As in the RSA experiment, we use a sequential attacker model, where the attacker issues a query immediately after the response. To make the difference more precise, we alternately issue valid and invalid queries.

For the basic mitigation mechanism, instead of modifying the Tomcat source code, we wrap the `doGet` function in our login service servlet with code implementing the basic mitigation scheme, to control leakage of the time needed to look up and check the password. Because it is not implemented as part of Tomcat, this implementation cannot mitigate timing information communicated by web service setup time. However, the experiment still shows that timing mitigation can defend against this timing channel attack.

Results

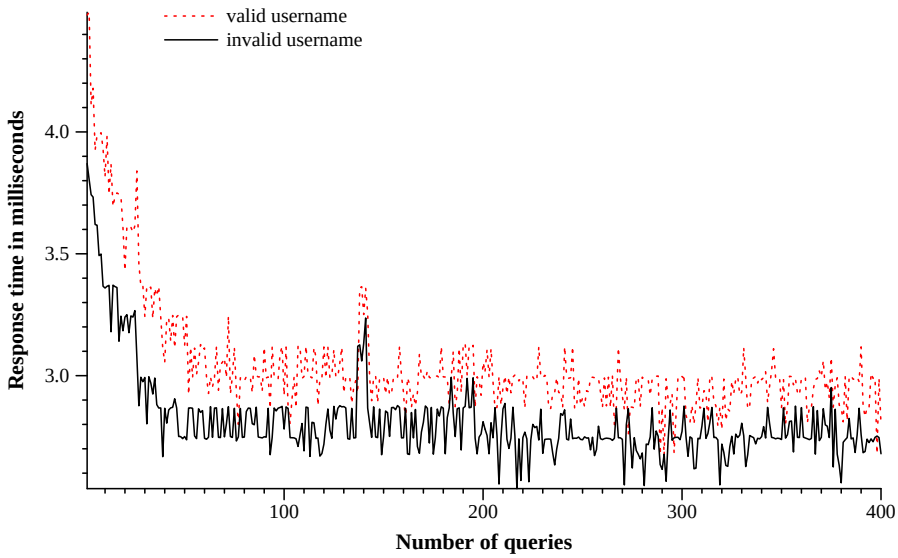
Figure 4.8(a) shows how query time differs for valid and invalid usernames. Queries for valid usernames take significantly longer, so a timing channel attack is feasible. Web server setup adds about 1.5ms latency to queries in the beginning of a run, but the query time stabilizes after around 50 queries. An attacker can determine the validity of an arbitrary username with high confidence.

Figure 4.8(b) shows the response time with the basic mitigation mechanism. Since server replies only at the end of the current quantum, the time difference is independent of the validity of username. Close inspection of the results reveals that there is an initial 1.5ms timing difference that is not mitigated by our implementation. This timing difference is caused by the setup of the web service, rather than by the login service we mitigated, and underscores the importance of mitigating timing end-to-end rather than on individual system components.

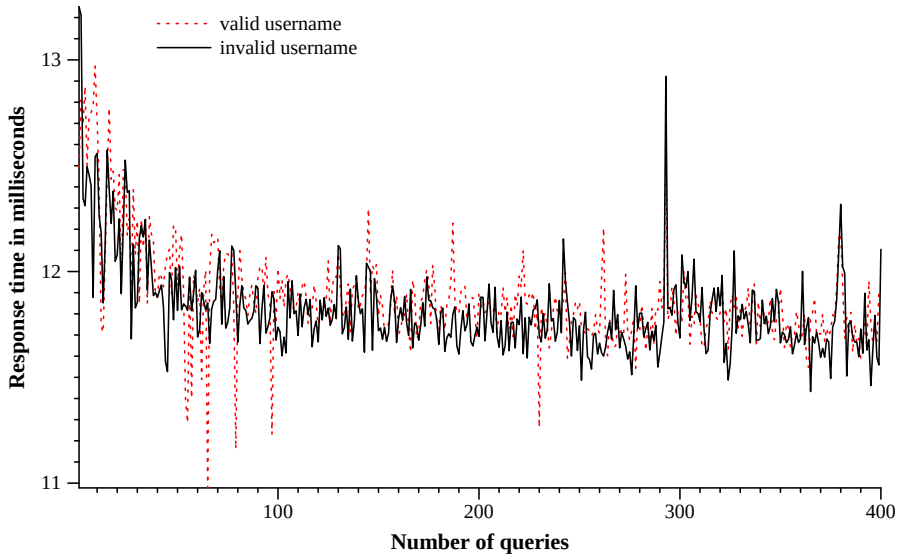
Another observation not shown in the time difference graph of RSA experiment is that our simple timing mitigation mechanism also adds a latency penalty to the web service, since the service time is unified to the closest power of 2 of the largest service time. This latency can be seen in Figure 4.8(b), where mitigation is seen to add about 9 ms latency.

Expected leakage

We applied the expected-leakage approach of Section 4.2.7 to the web service. Using 1000 random requests, we determined that 99% of them are below 8 ms. Replacing $T_{\text{big}} = \lceil \log 8 \rceil$ and $p = 0.99$ with these two numbers, the expected leakage for this application as shown in Figure 4.9, with $q_0 = 1$ ms. Clearly, the mitigated version leaks information slowly in practice.



(a) Without mitigation



(b) With mitigation

Figure 4.8: Simple mitigation of the web server timing attack.

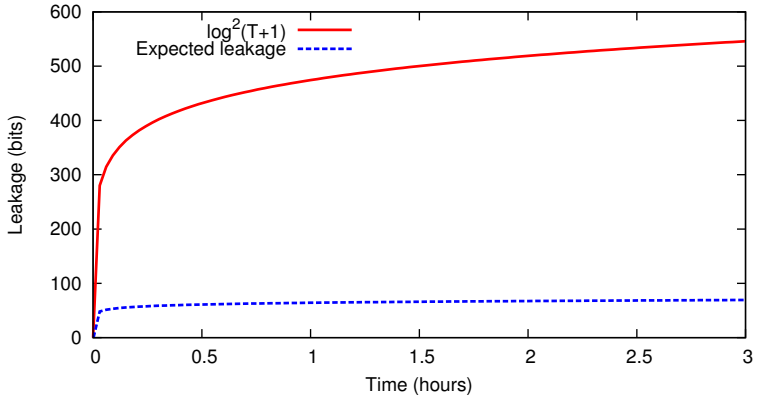


Figure 4.9: Expected leakage for web server timing attack.

4.5 Generalizing the black-box model for interactive systems

Predictive mitigation introduced so far assumes very little about the event source, which means that it can be applied to a wide range of systems. However, its very generality can make the leakage bounds conservative, and performance of the system is then hurt because the mitigator excessively delays the release of events. By refining the system model, we can make more accurate predictions and also bound timing leakage more accurately. The result is a better tradeoff between security and performance.

Timing channels in network-based services are particularly of interest for timing channel mitigation. These services are interactive systems that accept input requests from a variety of clients and send back responses. Figure 4.10 illustrates how we extend predictive mitigation for such a system.

Here, the abstract event source in the black-box model is replaced by a more concrete interactive system that accepts input messages on multiple input channels and delivers output messages to corresponding output channels. Output

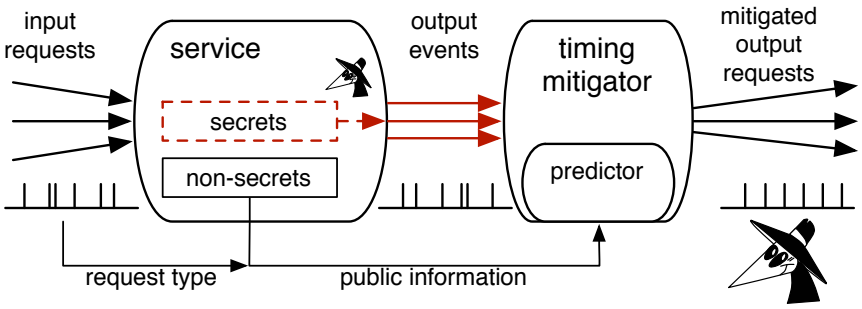


Figure 4.10: Predictive mitigation of an interactive system.

messages are passed through the timing mitigator, as before, and released by the timing mitigator in accordance with the prediction for that message. If a message arrives early, the mitigator delays it until the predicted time. If it does not arrive in time—a misprediction has happened—the mitigation starts a new epoch and makes a new, more conservative prediction.

This scheme significantly generalizes the black-box scheme. First, the time to produce each event is predicted separately, rather than requiring the mitigator to predict the entire schedule in advance—which is rather difficult for an interactive system. Second, the prediction may be computed using any public information in the system. This public information may be anything deemed public (the "non-secrets" in the diagram), possibly including some information about input requests. For example, the mitigator may use the time at which a given input request arrives to predict the time at which the corresponding output will be available for release. The model also permits the content of input requests to be partly public. Each request has an application-defined *request type* capturing what information about the request is public. If no information in the request is public, all requests have the same request type.

To see why this generalizes the original black-box scheme, consider what

happens if the prior history of mitigator predictions is the *only* information considered public when predicting the time of output events. In this case, all predictions within an epoch can be generated at the start of the epoch, yielding a completely determined schedule for the epoch. By contrast, our generalized predictive mitigation can make use of information that was not known at the start of the epoch, such as input time. Therefore, predictions can be made dynamically within an epoch.

4.6 Predictions for interactive systems

The system model described in Section 4.5 permits a great deal of flexibility in constructing predictions. We now begin to explore the possibilities.

Throughout the rest of this chapter we assume that the mitigator has an internal state, denoted by $\mathcal{S}$. In the simplest schemes, the state only records the number of epochs N, that is, $\mathcal{S} = N$. But more complex internal state is possible, as discussed in Section 4.7.2.

4.6.1 Inputs, outputs, and idling

For simplicity, we assume that inputs to and outputs from the interactive system correspond one-to-one: each input has one output and vice versa. If inputs can cause multiple output events, this can be modeled by introducing a schedule for delivering the multiple outputs as a batch.

Many services generate output events only as a response to some external

input. In the absence of inputs, such systems are idle and produce no output. If the predictor cannot take this into account when generating predictions, the failure to generate output produces gratuitous mispredictions. With generalized predictive mitigation, these mispredictions can be avoided.

For example, consider applying the original black-box scheme to a service that reliably generates results in 10ms. If the service is idle for an hour, the series of ensuing mispredictions will inflate the interval between predicted outputs to more than an hour, slowing the underlying service by more than five orders of magnitude. Clearly this is not acceptable.

Consider inputs arriving at times $inp_1, inp_2, \ldots inp_n, \ldots$, where each inp_i is the time of input i. We assume that the mitigator has some public state $\mathcal{S}$, and that this state always includes the index of the current mitigation epoch, denoted by N. Let the prediction for events for state $\mathcal{S}$ be described by a function $p(\mathcal{S})$, where p gives a bound on how long it is expected to take to compute an answer to a request in state $\mathcal{S}$.

Whenever the structure of the mitigator state is understood, we use more concrete notation. For example, in the simple mitigator we have $\mathcal{S} = N$, so we we write $p(N)$ for $p(\mathcal{S})$. Simple fast doubling has the prediction function $p(N) = 2^{N-1}$. For more complex predictors, p might depend on other (public) parameters as well. If $S_N(0)$ is the time of the start of the N-th epoch, subsequent event i in epoch N is predicted to occur at time $S_N(i)$:

$$S_N(i) = \max(inp_i, S_N(i-1)) + p(N)$$

The two terms in the expression above correspond to the predicted start of the computation for event i and the predicted amount of time it takes to com-

pute the output, respectively. To predict the start of computation for event i, we take the later of two times: the time when input i is available, and the time when event $i-1$ is delivered.

4.6.2 Multiple input and output channels

Now let us consider mitigation on multiple channels, where requests on different channels may be handled in parallel.

There are at least two reasonable concurrency models. The first model assumes that every request type has an associated process and that processes handling requests of one type do not respond to requests of other types. The second model assumes a shared pool of worker processes that can handle requests of any type as they become available.

In either model, the mitigator is permitted to use some information about which channel an input request arrives on and about the content of the request. This information about the channel and the request is considered abstractly to be the request type of the request. There is a finite set of request types numbered $1, \ldots, R$. Requests coming at time inp with request type r are represented as a pair (inp, r). A *request history* is a sequence of requests $(inp_1, r_1) \ldots (inp_i, r_i) \ldots$, where inp_i is the time of request i, and r_i is the type of the request: $1 \leq r_i \leq R$.

The mitigator makes predictions separately for each request type; however, with multiple request types, an epoch is a period of time during which predictions are met for *all* request types. A misprediction for one request type causes an epoch transition for the mitigator, and may change predictions for every re-

quest type. We denote the prediction for computation when mitigator is in state $\mathcal{S}$ on request type r by a function $p(\mathcal{S}, r)$. When the state consists only of the number of epochs ($\mathcal{S} = N$), we simply write $p(N, r)$.

Individual processes per request type

In the case where each request type has its own individual process, the prediction for output event i is

$$S_N(i) = \max(inp_i, S_N(j)) + p(N, r_i)$$

where j is the index of the previous request of type r_i; that is, $j = \max\{j' \mid j' < i \wedge r_i = r_{j'}\}$. Hence $S_N(j)$ is the prediction of the previous request of type r_i. We define $S_N(j)$ to be zero when there are no previous requests of the same type.

Example Consider a simple system with two request types A and B (for clarity we index request types with letters), and consider a mitigator with these prediction functions $p(N, r)$ for $N = 1$:

N	$p(N, A)$	$p(N, B)$
1	10	100

Assume the following input history: $(2, A)$, $(4, B)$, $(6, A)$, and $(30, B)$. That is, two inputs of type A arrive at times 2 and 6, and two of type B arrive at times 4 and 30.

The inputs $(2, A)$ and $(4, B)$ are the first requests of the corresponding types. The predictions for these requests are

$$S_1(1) = \max(2, 0) + 10 = 12$$
$$S_1(2) = \max(4, 0) + 100 = 104$$

For the next request of type A, the prediction is

$$S_1(3) = \max(6, 12) + 10 = 22$$

This prediction takes into account the amount of time it would take for the process for request type A to finish processing the last input and then to delay the message for $p(1, A)$. Similarly, the predicted output time for the fourth request $(30, B)$ is

$$S_1(4) = \max(30, 104) + 100 = 204$$

Shared worker pool

For a shared pool of worker processes, predictions must be derived with more careful computation. Suppose the system has at least n worker processes that handle input requests. To compute a prediction for input request i that arrives at time inp_i with type r_i, the mitigator needs to know two terms: when the handling of that request will start, and an estimate of how long it takes to complete the request. We assume that the completion estimate is given by $p(N, r)$ and focus instead on the first term. The main challenge is to predict when a worker will be available to process a request. For this we introduce a notion of *worker predictions*. Intuitively, worker predictions are a data structure internal to the mitigator that allows it to predict when different requests will be picked up by worker processes.

Concretely, worker predictions are n sets $W_1, \ldots, W_n$ in which every W_m contains pairs of the form (i, q). When $(i, q) \in W_m$, it means request i is predicted to be delivered at time q by worker m. Therefore, a given index i appears in at most one of the sets W_m. The function $\mathsf{avail}(W)$ predicts when a worker described by set W will be available, by choosing the time when the worker should deliver its last message.

$$\mathsf{avail}(W) \triangleq \begin{cases} \max\{q \mid (i, q) \in W\} & \text{if } W \neq \emptyset \\ 0 & \text{otherwise} \end{cases}$$

We describe next the algorithm for computing worker predictions.

Initialization In the initial state of worker predictions, all sets W_m (for $1 \leq m \leq n$) are empty.

Prediction Given an event i with input time inp_i and request type r_i, the prediction $S_N(i)$ is computed as follows:

1. The earliest available worker j is predicted to handle request i. Therefore, we find j such that $\mathsf{avail}(W_j) = \min_{1 \leq m \leq n}\{\mathsf{avail}(W_m)\}$

2. Since worker j is assumed to handle request i, we make the following prediction q for the i-th output:

$$q = \max(inp_i, \mathsf{avail}(W_j)) + p(N, r_i)$$

The prediction for S_N is $S_N(i) = q$.

3. Finally, worker predictions are updated with prediction (i, q):

$$W_j := W_j \cup \{(i, q)\}$$

Misprediction When a misprediction occurs, the mitigator resets the state of worker predictions. Consider a misprediction at time τ_N, which defines the start time of epoch N. We reset the state of worker predictions as follows:

1. For every worker m, we find the earliest undelivered request i'; that is, request received before the misprediction but not delivered by the mitigator at τ_N :

$$i' = \min\{i \mid (i, q) \in W_m \wedge inp_i < \tau_N \leq q\}$$

2. If such i' cannot be found, that is, the set in the previous equation is empty, we set W_m to $\emptyset$. Otherwise, we let $q' = \tau_N + p(N, r_{i'})$ and set $W_m = \{(i', q')\}$.

3. Note that the above step resets the state of each W_m in the worker predictions. Using these reinitialized states, we can compute predictions for the unhandled requests, i.e., all requests j with predicted time q such that $q \geq \tau_N$ according to the steps 1) and 2) described in *Prediction*.

Example Reusing the settings from the example in Section 4.6.2, we have four inputs: $(2, A)$, $(4, B)$, $(6, A)$ and $(30, B)$, and prediction function $p(1, A) = 10$ and $p(1, B) = 100$. Suppose we have two shared workers.

As described above, the worker predictions are both initialized to be empty: $W_1 = \emptyset$ and $W_2 = \emptyset$. For the first input, both workers are available; that is, $\mathsf{avail}(W_1) = \mathsf{avail}(W_2) = 0$ since W_1 and W_2 are all empty sets now. We break the tie by selecting the worker with smaller index, worker 1, and then we set the prediction for input $(2, A)$ as

$$S_1(1) = \max(2, 0) + 10 = 12$$

Finally, the worker prediction of worker 1 is updated to $\{(1, 12)\}$.

For the second input, $\mathsf{avail}(W_1) = 12$ and $\mathsf{avail}(W_2) = 0$. Worker 2 is the earliest available worker. Similarly to the first input, the prediction for the second output is $S_1(2) = \max(4, 0) + 100 = 104$. The worker prediction of worker 2 is updated to $\{(2, 104)\}$.

Computation of the predicted worker becomes more interesting for the third input $(6, A)$. We have

$$\mathsf{avail}(W_1) = \max\{q \mid (i, q) \in \{(1, 12)\}\} = 12$$
$$\mathsf{avail}(W_2) = \max\{q \mid (i, q) \in \{(2, 104)\}\} = 104$$

The mitigator picks the worker with earliest availability, worker 1. The third output is predicted at: $S_1(3) = \max(6, 12)+10 = 22$, and the prediction for worker 1 is updated to $\{(1, 12), (3, 22)\}$.

For the last input $(30, B)$, the mitigator first computes the available times for both workers:

$$\mathsf{avail}(W_1) = \max\{q \mid (i, q) \in \{(1, 12), (3, 22)\}\} = 22$$
$$\mathsf{avail}(W_2) = \max\{q \mid (i, q) \in \{(2, 104)\}\} = 104$$

Based on these values, the mitigator picks worker 1 as the predicted worker for the fourth input. The prediction for corresponding output is $S_1(4) = \max(30, 22) + 100 = 130$, and the prediction of worker 1 becomes $\{(1, 12)$, $(3, 22)$,$(4, 130)\}$.

4.7 Leakage analysis

As in Section 4.2, we use a combinatorial analysis to bound how much information leaks via predictive mitigation in interactive systems. One difference is that we take into account the interactive nature of our model and derive bounds based on the number of input requests and the elapsed time. To conservatively estimate leakage we bound the number of possible timing variations that an adversary can observe, as a function of the running time T and the length of the input history M.

We show that a leakage bound of $O(\log T \times \log M)$ can be attained, with a constant factor that depends on the choice of *penalty policy*. When there is a worst-case execution time for every request, a tighter leakage bound of $O(\log M)$ can be derived.

4.7.1 Bounding the number of variations

To bound the number of possible timing variations, we need to know three values: (1) the number of timing variations within each epoch, (2) the number of variations introduced by the schedule selector, and (3) the number of epochs.

Let us consider the number of variations within each epoch. Because messages within a single epoch are delivered according to predictions, the only source of variations within an individual epoch is *whether* there is a misprediction, and if so, *when* the misprediction occurs. This can be specified by the *length* of the epoch. When the mitigator has received at most M messages, the length of any single epoch can be at most $M + 1$.

When the mitigator transitions from epoch n to epoch $n + 1$, it chooses the schedule for the next epoch. Since the predictor can rely on public information, the "schedule" is actually an algorithm parameterized by public inputs. However, this algorithm may be chosen based on non-public inputs, in which case the choice of schedule may convey additional information to the adversary. Following Section 4.2, we denote by Λ_N the number of possible schedules when transitioning between epochs n and $n+1$. Its value depends on the details of the schedule selector. For simple mitigation schemes, where the choice of the next schedule does not depend on secrets, we have $\Lambda_N = 1$. For *adaptive* mitigation (Section 4.2.4), where the choice of schedule depends on internal state such as the size of the mitigator's message buffer, Λ_N may be greater than one.

Consider a mitigator that at time T has received at most M requests and reached at most N epochs. The number of possible timing variations of such a mitigator is at most

$$(M + 1)^N \cdot \Lambda_1 \ldots \Lambda_N$$

Measured in bits, the corresponding bound on leakage is the logarithm of the number of variations:

$$N \cdot \log(M + 1) + \sum_{i=1}^{N} \log \Lambda_i$$

Note that for the simple doubling scheme, because $\Lambda_i = 1$, we also have $\sum_{i=1}^{N} \log \Lambda_i = 0$.

We can enforce an arbitrary *enforcing bound* on leakage. Denote by $B(T, M)$ the amount of information permitted to be leaked by the mitigator. Enforcing bound $B(T, M)$ is satisfied if the mitigator ensures this inequality holds:

$$N \cdot \log(M + 1) + \sum_{i=1}^{N} \log \Lambda_i \leq B(T, M)$$

This equation requires a relationship between the number of epochs, the elapsed time, and the number of received messages. The exact nature of this relationship is determined by *penalty policies*.

4.7.2 Penalty policies

Recall that the function $p(\mathcal{S}, r)$ predicts a bound on computation time for request type r in state $\mathcal{S}$. The intuition is that the more mispredictions have happened in the past (as recorded in $\mathcal{S}$), the larger is the value of $p(\mathcal{S}, r)$. The computation is *penalized* by delivering its response later.

Designing a penalty policy function opens up a space of possibilities. The question is how mispredictions on different request types are interconnected—for example, whether a particular request type should be penalized for mispredictions on other request types, and if so, then how much.

On one side of the spectrum, we can use a *global penalty policy* that penalizes *all* request types when a misprediction occurs. If all request types are penalized, it becomes harder to trigger mispredictions on any of them in future. Therefore, this policy provides a tight bound on N. Intuitively, an adversary gains no additional power to leak information by switching between request types. However, performance of all request types is hurt by mispredictions on any request type.

On the other end of the spectrum is a *local penalty policy* in which request types are not penalized by mispredictions on other types. This improves performance but offers weaker bounds on leakage. To see this, assume that the number of mispredictions a single request type can make is N. Since penalties

are not shared between request types, with R types, as many as $R \times N$ mispredictions can occur. Timing leakage might be high if R is large; intuitively, the adversary can attack each request type independently.

Aiming for more control of the tradeoff between security and performance, we explore penalty policies that fill in the space between the global and local penalty policies. The key insight is that the request types with few mispredictions contribute little to total leakage, so they should share little penalty. This insight brings an l-level *grace period* policy. In a l-level grace period policy, request type r is only penalized by other types when the number mispredictions on r is greater than l.

For more complex penalty policies, leakage analysis becomes more challenging. In Section 4.7.4, we present an efficient and precise way of bounding N for some penalty policies.

4.7.3 Generalized penalty policies

We refine the state $\mathcal{S}$ to record the number of mispredictions for each request type. If m_r denotes the number of mispredictions on request type r, the mitigator state contains a vector of misprediction counts $\vec{m} = m_1, \ldots, m_R$. Initially all m_r are zero. When the request type r has a misprediction, m_r is increased by one. In the following, we assume $\mathcal{S} = \vec{m}$, and write the penalty function as $p(\vec{m}, r)$.

Recall that during an epoch, predictions for all types are met. Given a vector of mispredictions $\vec{m}$, the number of epochs N is simply $N = 1 + \sum_{i=1}^{R} m_i$. Thus, the problem of bounding N is the same as bounding the sum $\sum_{i=1}^{R} m_i$.

For convenience, let us focus on a family of penalty functions p that are a composition of three functions:

$$p(\vec{m}, r) = q(r) \times (\phi \circ \text{idx})(\vec{m}, r)$$

Here function $\phi(n)$ is a *baseline penalty* function, which given a penalty index n returns the prediction for n. The penalty index represents how severely this request type is penalized. It is computed by function $\text{idx}(\vec{m}, r)$, which returns the value of the index in the current state $\vec{m}$ for request type r. Finally, $q(r)$ returns an initial penalty for request type r, and allows us to model different initial estimates of how long it takes to respond to the request of type r. For instance, if one knows that request type r_1 needs at least one second, and request type r_2 needs at least 100 seconds, then one can set $q(r_1) = 1, q(r_2) = 100$.

Examples For penalty policies based on fast doubling, we set $\phi(n) = 2^n$, and $q(r) = q_0$ for all r with some initial quantum q_0. For the global penalty policy, idx can be set to $\text{idx}(\vec{m}, r) = \sum_{i=1}^{R} m_i$. For the local penalty policy, idx is chosen as $\text{idx}(\vec{m}, r) = m_r$. For an l-level grace period policy, we define idx to depend on the parameter l:

$$\text{idx}(\vec{m}, r) = \begin{cases} m_r & \text{if } m_r \leq l \\ \sum_{i=1}^{R} m_i & \text{otherwise} \end{cases}$$

4.7.4 Generalized leakage analysis

As discussed earlier, different penalty functions yield different bounds on N. While it is possible to analyze such bounds for specific penalty policies, in general it is hard to bound leakage for more complex penalty policies.

This section describes a precise method for deriving such bounds for several classes of penalty policies. We transform the problem of finding a bound on the number of epochs N into an optimization problem with R constraints, where R is the number of request types. These constraints can be nonlinear in general, but all considered classes of penalty functions can be solved in constant time.

We focus on penalty functions where $p(\vec{m}, r)$ is monotonic. Because monotonicity is natural for a "penalty", this requirement does not really constrain the generality of the analysis.

State validity We write $\vec{0}$ for the initial state $\vec{0}$ in which no mispredictions have happened. At the core of our analysis are two notions: state reachability and state validity. Informally, a state $\vec{m}$ is reachable at time T if there is a sequence of mispredictions that, starting from $\vec{0}$, lead to $\vec{m}$ by time T. To bound the number of possible epochs N at time T, it is sufficient to explore the set of all reachable states, looking for $\vec{m}$ in which $1 + \sum_{i=1}^{R} m_i$ (and therefore N) is maximized.

Enumerating all reachable states may be infeasible. In particular, an exact enumeration requires detailed assumptions about the thread model presented in Section 4.6.2. Instead, we overapproximate the set of reachable states for efficient searching of the resulting larger space.

For this, we define the notion of state validity at time T. State validity at time T is similar to reachability at time T, except that we focus only on the predicted time to respond to a request, ignoring the time needed to execute earlier requests.

We first introduce the notion of a valid successor:

Definition 9 (Valid successor) *A state $\vec{m}'$ is a* valid successor *of type j ($1 \leq j \leq R$) for state $\vec{m}$ when $m'_j = m_j + 1$ and $m'_i = m_i$ for $i \neq j$.*

For example, with three different request types ($R = 3$), the state $(0, 0, 1)$ is a valid successor of type 3 for state $\vec{0}$.

We can then define state validity:

Definition 10 (State validity for time T) *For penalty function $p(\vec{m}, r)$, a state $\vec{m}$ is a* valid state for time T *if there exists a sequence of request types $j_1, \ldots j_{n-1}, j_n$, such that, if $m_0 = \vec{0}$, it holds that for all i, $1 \leq i \leq n$ we have*

- *$\vec{m}_i$ is valid successor of type j_i for state $\vec{m}_{i-1}$.*
- *$p(\vec{m}_{i-1}, r_{j_i}) \leq T$*
- *$\vec{m}_n = \vec{m}$*

The second condition approximates whether the state $\vec{m}_{i-1}$ can make one more transition: if execution time is predicted to exceed T, no more transitions are possible.

Note that we put no requirements on the predictions for $\vec{m}_n$. The reason is that as long as there is a state $\vec{m}_{n-1}$ such that $\vec{m}_{n-1}$ can make one more misprediction to $\vec{m}_n$, then it is possible to reach state $\vec{m}_n$. Validity does not depend on whether the state can make one more transition In particular, this allows us to include all states such that after misprediction from $\vec{m}_{n-1}$ we cannot have any more mispredictions. These states are the candidates for maximizing N.

Example Consider the simple case of one request type and time 6 with prediction function $p(\vec{m}, r) = 2^{m_r}$.

State $\vec{m} = (3)$ is a *valid* state for time 6. Consider the request type sequence $1, 1, 1$. We have $\vec{m}_0 = \vec{0}$. Since $\vec{m}_1$ is a valid successor of type 1 for state $\vec{m}_0$, we have $\vec{m}_1 = (1)$. Similarly, we have $\vec{m}_2 = (2)$ and $\vec{m}_3 = (3)$. It is easy to check that $p(\vec{m}_0) = 1 \leq 6$, $p(\vec{m}_1) = 2 \leq 6$ and $p(\vec{m}_2) = 4 \leq 6$. Since $\vec{m}_3 = \vec{m}$, $\vec{m}$ is valid by definition.

However, state $\vec{m}' = (4)$ is not *valid*. Otherwise, since there is only one request type in this example, j_n must be 1. Therefore, $\vec{m}_{n-1}$ must be (3) because $\vec{m}_n$ is a valid successor of type 1 for $\vec{m}_{n-1}$. However, $p(\vec{m}_{n-1}) = 8 > 6$. This contracts the definition of validity.

Transforming to an optimization problem

In this part, we show how to get the maximal $\sum_{i=1}^{R} m_i$ among all valid states when prediction function $p(\vec{m}, r)$ is monotonic. First, we show a useful lemma.

Lemma 15 *Assume $p(\vec{m}, r)$ is monotonic. If $\vec{m}$ is a valid successor of some type j for $\vec{m}'$ such that $p(\vec{m}', j) \leq T$, then*

$$\vec{m} = (m_1, \ldots, m_R) \text{ is valid for } T \iff \vec{m}'' = (m_1, \ldots, m_{j-1}, 0, m_{j+1}, \ldots, m_R) \text{ is valid for } T$$

Proof. $\Longleftarrow$: since $\vec{m}''$ is valid, there is sequence of request types where all intermediate states satisfy the constraints. Further, we can construct a sequence of request types from $\vec{m}''$ to $\vec{m}$ by appending j to the previous sequence until $\vec{m}_i = \vec{m}$. Since $p(\vec{m}', j) \leq T$ and p is monotonic, all new states corresponding to this sequence still satisfy the constraints.

$\Longrightarrow$: by definition, there is a sequence of request types $r_1, \ldots, r_n$ such that all intermediate states satisfy constraints. Moreover, there must be a point i in this sequence such that $\forall l < i, r_l \neq j$ and $r_i = j$. Thus, the j-th element of $\vec{m}_{i-1}$ is 0.

Then, a new sequence of request types $p_1, \ldots, p_m$ exists such that $p_l = r_l, 0 \leq l \leq i-1$. For $l \geq i$, if $r_l = j$, skip this type. Otherwise, add the same type to sequence $\vec{p}$. By this construction, two properties of states occurring with $\vec{p}$ are that the j-th element is always 0, and that there is a corresponding state with sequence $\vec{r}$ such that they only differ in the j-th element. We denote the final states with request type sequence $\vec{r}$, $\vec{p}$ as $\vec{m}^r$ and $\vec{m}^p$ respectively. Since state $\vec{m}^r$ satisfies $p(\vec{m}^r, r_l) \leq T$, by monotonicity, corresponding state $\vec{m}^p$ also satisfies this condition. Since $m^r_n = \vec{m}''$, $\vec{m}''$ is valid at T. □

Lemma 15 allows us to describe valid states by R constraints. To see this, first observe that because $\vec{m}$ is valid for T, there are some j_1 and $\vec{m}'$ such that $\vec{m}$ is a valid successor of $\vec{m}'$ of type j_1. By Definition 9, $p(\vec{m}', j_1) \leq T$. This is our first constraint on the space of valid states.

By Lemma 15, the validity of $\vec{m}$ for T implies the validity of $(m_1, ..., m_{j_1-1}, 0, \ldots, m_R)$ for T. Repeating the previous step, there is some $j_2 \neq j_1$ and $\vec{m}''$ where $(m_1, ..., m_{j_1-1}, 0, \ldots, m_R)$ is a valid successor of $\vec{m}''$ of type j_2; this gives us the second constraint, $p(\vec{m}'', j_2) \leq T$. Proceeding as above, we obtain R constraints such that $\vec{m}$ is valid iff all constraints are satisfied.

Based on the properties of p, our analysis proceeds as follows. We present two different classes of p in the order of difficulty of analyzing them, starting from the easiest.

Symmetric predictions We first look at prediction policies in which all request types are penalized *symmetrically*:

1. for all i, j, such that $1 \leq i, j \leq R$ it holds that $p(m_1, \ldots, m_i, \ldots, m_j, \ldots m_R, i) = p(m_1, \ldots, m_j, \ldots, m_i, \ldots m_R, j)$.
2. for all i, j, k, such that $1 \leq i, j, k \leq R$, where $i \neq k$, and $j \neq k$ it holds that $p(m_1, \ldots, m_i, \ldots m_j, \ldots m_R, k) = p(m_1, \ldots, m_j, \ldots m_i, \ldots m_R, k)$.

These properties allow us to reorder the request types in R constraints that we have obtained earlier. For example, the first of the obtained constraints can be rewritten as $p((m_{j_1} - 1, \ldots, m_R), 1) \leq T$. Moreover, this allows us to rename the variables in the constraints without loss of generality:

$$\begin{cases} p((m_1 - 1, m_2, \ldots, m_R), 1) & \leq T \\ p((0, m_2 - 1, \ldots, m_R), 2) & \leq T \\ \ldots & \\ p((0, 0, \ldots, m_R - 1), R) & \leq T \end{cases}$$

Thus, bounding N is equivalent to finding the maximum sum $\sum_{i=1}^{R} m_i$ satisfying all the conditions.

Examples It is easy to verify that starting with same initial quantum, global, local, and l-level grace period policies penalize all request types symmetrically. We proceed with the analysis of these policies below.

1. Consider the global penalty function with fast doubling and the starting quantum $q_0 = 1$. The j-th constraint in the above system has the form

$$2^{(\sum_{i=j}^{R} m_i - 1)} \leq T$$

Here, $N = 1 + \sum_{i=1}^{R} m_i \leq \log T + 2$. This is very close to the bound $\log(T+1)+1$ given in Section 4.2.3.[1]

Using the leakage bound derived in Section 4.7.4, we obtain that for global penalty policy, when the mitigator runs for at most time T the leakage is bounded by function $B(T, M)$ where

$$B(T, M) = (\log T + 2) \cdot \log(M + 1)$$

2. Now consider the local penalty policy with the same penalty scheme and initial quantum. We have R constraints of the form:

$$2^{m_i - 1} \leq T, 1 \leq i \leq R$$

It is easy to derive $N \leq R \cdot (\log T + 1) + 1$.

Using this bound for N, we obtain that at running time time T, leakage is bounded by function $B(T, M, R)$ such that

$$B(T, M, R) = (R \cdot (\log T + 1) + 1) \cdot \log(M + 1)$$

3. We revisit the l-level grace period policy last. In this case, the j-th constraint can be split into two cases:

$$\begin{cases} m_j - 1 \leq \log T & \text{when } m_j - 1 \leq l \\ \sum_{i=j}^{R} m_i - 1 \leq \log T & \text{when } m_j - 1 > l \end{cases}$$

In general, l is ordinarily smaller than $\log T$, so N is maximized when $m_i = l + 1, 1 \leq i \leq R - 1$ and $m_R = \lfloor \log T \rfloor + 1$. Thus, $N \leq (R-1) \cdot (l+1) + \log T + 2$.

[1] Though Section 4.2.3 does not consider request types, the penalty policies considered there are effectively global penalty policies.

Using this bound for N we obtain that at running time T, leakage is bounded by function $B(T, M, R, l)$ such that

$$B(T, M, R, l) = \log(M + 1) \cdot ((R - 1) \cdot (l + 1) + \log T + 2)$$

Partially symmetric predictions Request types starting with different initial quanta, such as the setup in Section 4.7.5, make the prediction function asymmetric. We proceed as follows.

Let $q_{min} = \min_{i=1}^{R} q(r)$, and replace $p(r)$ with q_{min} for all prediction functions. The upper bound N of these replaced functions overapproximates that of asymmetric functions, since any valid state using the latter functions must be valid for the former ones. Therefore, we can obtain R constraints based on these replaced symmetric functions, as for symmetric predictions.

Non-symmetric predictions For other types of penalty functions, we can still try to partition request types into subsets such that in each subset, request types are penalized symmetrically. We then generate constraints for validity of these well-formed subsets.

More formally, we say a vector of mispredictions $\vec{m}'$ is a *subvector* of $\vec{m}$ if and only if $m'_i = 0 \vee m'_i = m_i, 1 \leq i \leq R$. A set of vectors $\vec{m}^1, \ldots, \vec{m}^k$ is a *partition* of $\vec{m}$ if all vectors are subvectors of $\vec{m}$ and for all m_i, there is one and only one $\vec{m}^j$ such that $m^j_i = m_i$.

The following lemma shows that the condition that $\vec{m}$ is valid is stronger than the validity of all subvectors. Thus, the constraints on vectors in a partition overapproximates those on the validity of $\vec{m}$.

Lemma 16 *When $p(\vec{m}, r)$ is monotonic, $\vec{m}$ is valid at time $T \implies$ any subvector of $\vec{m}$ is valid at time T.*

Proof. By definition, there is a sequence of request types $j_1, \ldots, j_n$ such that all conditions in Definition 10 are satisfied. For any subvector of $\vec{m}$, say $\vec{m}'$, we can take a projection of the sequence so that only the request types nonzero in the subvector are kept.

By monotonicity, it is easy to check that all conditions hold in the definition. Moreover, $\vec{m}_n = \vec{m}'$. So $\vec{m}'$ is valid by definition. □

Since there are R non-zero mispredictions among all vectors in the partition, this estimation still gives R constraints.

4.7.5 Security vs. performance

As discussed informally earlier, the global penalty policy enforces the best leakage bound but has bad performance; the local penalty policy has the best performance but more leakage. We explore this tradeoff between security and performance through simulations.

Simulation setup We simulate a set of interactive system services characterized by various distributions over execution time. Initial penalty is set to be the mean of the execution time distribution. The fast doubling scheme is used, so the prediction function is $p(\vec{m}, r) = q(r) \times 2^{\text{idx}(\vec{m}, r)}$, where $q(r)$ is the mean time of simulated type r. The form of $\text{idx}(\vec{m}, r)$ is defined by penalty policies.

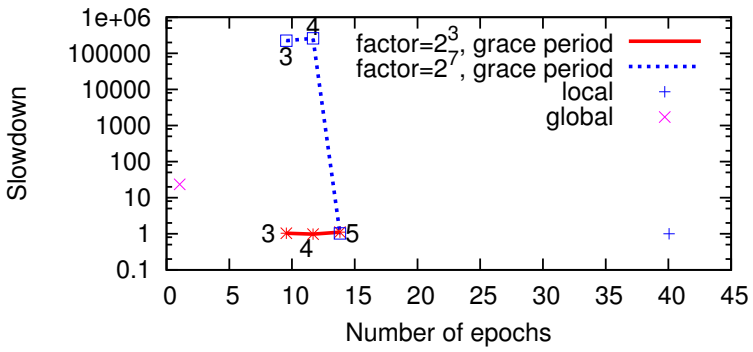


Figure 4.11: Performance vs. security.

To see the performance for requests with different variances in execution time, we simulate both *regular types* and *irregular types*. For regular types, the simulated execution time follows Poisson distribution with different means, since page view requests to a web page can be modeled as a Poisson process; for irregular types, execution time follows a perturbed normal distribution which avoids negative execution time.

Result The results in Figure 4.11 demonstrate the impact of execution-time variation on performance. The x-axis in Figure 4.11 shows the bound on number of epochs N and the y-axis shows the slowdown for all simulated request types. All values shown are normalized so that the local policy has a slowdown of 1 and so that for the number of epochs, the global policy has value 1. The standard deviation is equal to the mean multiplied by a factor ranging from 2^3 to 2^7, generating around 3 to 7 mispredictions. The number on each line denotes the grace-period level.

The results confirm the intuition that the global penalty policy has the best security but bad performance, and the local policy has the best performance. However, the l-level grace period policies have considerably fewer epochs N,

yet performance similar to that of the local policy when l is no less than m_{r_i} for most types.

When the variance of execution time increases, small grace-period level ($l = 3, 4$) can bring slowdown that is orders of magnitude higher than in the global case. The reason is that each irregular request type can trigger l mispredictions. Once misprediction of a request type is larger than l, idx($\vec{m}, r$) returns a large number. However using a larger grace-period level ($l = 5$) could restore performance at the cost of more leakage.

Penalty policies with other forms are possible to provide more options between the trade-off of security and performances. We leave a more comprehensive analysis of more penalty policies as future work.

4.7.6 Leakage with a worst-case execution time

In the analysis above, no assumption is made about execution time for each request type. The adversary can delay responses for an arbitrarily long time to covertly convey more information.

However, for some specific platforms, such as real-time systems and web applications with a timeout setting, we can assume a worst-case execution time T_w. Given this constraint, we can derive a tighter leakage bound.

The analysis works similarly to that in Section 4.7.3, but instead of using the conservative constraint $p(\vec{m}_{i-1}, r_{j_i}) \leq T$ as in Definition 10, worst-case execution time provides a tighter estimate:

$$p(\vec{m}_{i-1}, r_{j_i}) \leq T_w$$

Compared with bounding running time T, this condition more precisely approximates whether the state $\vec{m}_{i-1}$ can make one more misprediction to $\vec{m}_i$. The reason is that whenever $p(\vec{m}_{i-1}, r_{j_i}) > T_w$, the state $\vec{m}_{i-1}$ cannot have another misprediction because execution is bounded by T_w. Therefore, we can reuse the bound on the number of epochs in Section 4.7.3 by replacing T with T_w.

For example, total leakage with the assumption of worst-case execution time T_w for the global penalty policy is bounded by

$$B(T, M) = (\log T_w + 2) \cdot \log(M + 1)$$

This logarithmic bound is asymptotically the same as that achieved by the less general bucketing scheme proposed by Köpf et al. [45] for cryptographic timing channels.

For the l-grace-period penalty policy we can perform a similar analysis to derive a bound on leakage:

$$B(T, M, R, l) = \log(M + 1) \cdot ((R - 1) \cdot (l + 1) + \log T_w + 2)$$

4.8 Composing mitigators

If timing mitigation is used, we can expect large systems to be built by composing mitigated subsystems. Here, we analyze leakage of composed mitigators.

We analyze composed mitigators by considering the leakage of two gadgets: two mitigators connected either in parallel or sequentially (Figures 4.12 and 4.13). More complex systems with mitigated subsystems can be analyzed by decomposing them into these gadgets.

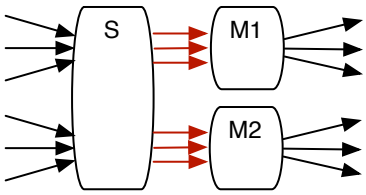


Figure 4.12: Parallel composition of mitigators.

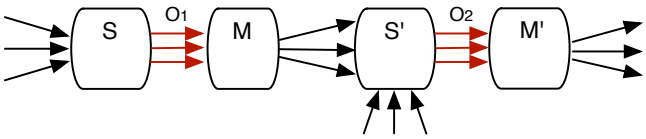


Figure 4.13: Sequential composition of mitigators.

Parallel composition Figure 4.12 is an example of parallel composition of mitigators, in which requests received by the system are handled by two independent mitigators. The bound on the leakage of the parallel composition is no greater than the sum of the bounds of the independent mitigators. To see this, denote by P the total number of variations of the parallel composition, and denote by V_1 and V_2 the number of timing variations of the first and second mitigators, respectively. We know $P \leq V_1 \cdot V_2$; consequently, the total leakage of parallel composition $\log P$ is bounded by $\log V_1 + \log V_2$. The same argument generalizes to n mitigators in parallel.

Sequential composition Suppose we have a security-critical component, such as an encryption function, and leakage from this component is controlled by a mitigator that guarantees a tight bound, say at most 10 bits of the encryption key. We can show that once mitigated, leakage of the encryption key can never exceed 10 bits, no matter how output of that component is used in the system. This is true for both Shannon-entropy and min-entropy definitions of leakage.

Consider sequential composition of two systems as depicted in Figure 4.13. Suppose that the secrets in the first system are S, and that the outputs of the first and the second mitigators are O_1 and O_2 respectively. We consider how much the output of each of the mitigators leaks about S.

We can view the outputs O_1 and O_2 as discrete random variables. Since the second service and its mitigator do not share secret S, the conditional distribution of O_2 depends only on O_1 and is conditionally independent of S (in other words, random variables S, O_1, O_2 form a Markov chain). Denoting the probability mass function of a discrete random variable X as $P(X)$, the joint distribution of these three random variables has probability mass function $P(s, o_1, o_2) = P(s)P(o_1|s)P(o_2|o_1)$. The marginal distribution $P(o_2, s)$ is $\sum_{o_1 \in O_1} P(s, o_1, o_2)$, and for any o_1, we have $\sum_{o_2 \in O_2} P(o_2|o_1) = 1$.

As discussed in Section 4.1.2, the leakage of the first mitigator using mutual information is $\mathcal{I}(S; O_1)$ and the leakage of the second is $\mathcal{I}(S; O_2)$. Then we can show that the second mitigator leaks no more information about S_1 than the first does. We formalize this in the following lemma.

Lemma 17 $\mathcal{I}(S; O_1) \geq \mathcal{I}(S; O_2)$

Proof. The proof follows from the standard *data-processing inequality* [20] and the symmetry of mutual information:

$$\begin{aligned} \mathcal{I}(S; O_2) + \mathcal{I}(S; O_1|O_2) &= \mathcal{I}(S; O_1, O_2) \\ &= \mathcal{I}(S; O_1) + \mathcal{I}(S; O_2|O_1) \end{aligned}$$

Note that S and O_2 are conditionally independent given O_1, since the second mitigator produces outputs based on only the output of the first mitigator M,

public inputs, and secrets other than S. Thus $\mathcal{I}(S;O_2|O_1) = 0$. Replacing this term with zero in the above equation, we get

$$\mathcal{I}(S;O_2) + \mathcal{I}(S;O_1|O_2) = \mathcal{I}(S;O_1)$$

Also, we know that $\mathcal{I}(S;O_1|O_2) \geq 0$, so we have

$$\mathcal{I}(S;O_1) \geq \mathcal{I}(S;O_2)$$

□

A similar result holds for min-entropy leakage.

Lemma 18 $V(S|O_1) \geq V(S|O_2)$

Proof. As discussed in Section 4.1.2, min-entropy channel capacity is defined as the maximal value of $\log \frac{V(S|O)}{V(S)}$ among all distributions on S. So it suffices to show $V(S|O_1) \geq V(S|O_2)$ for any distribution on S.

$$\begin{aligned}
V(S|O_2) &= \sum_{o_2 \in O_2} \max_{s \in S} P(s)P(o_2|s) \\
&= \sum_{o_2 \in O_2} \max_{s \in S} \sum_{o_1 \in O_1} P(s, o_1, o_2) \\
&= \sum_{o_2 \in O_2} \max_{s \in S} \sum_{o_1 \in O_1} P(s)P(o_1|s)P(o_2|o_1) \\
&\leq \sum_{o_2 \in O_2} \sum_{o_1 \in O_1} P(o_2|o_1) \max_{s \in S} P(s)P(o_1|s) \\
&= \sum_{o_1 \in O_1} \max_{s \in S}(P(s)P(o_1|s)) \sum_{o_2 \in O_2} P(o_2|o_1) \\
&= \sum_{o_1 \in O_1} \max_{s \in S} P(s)P(o_1|s) \\
&= V(S|O_1)
\end{aligned}$$

□

Discussion Parallel and sequential composition results enable deriving conservative bounds for networks of composed subsystems. The bounds derived may be quite conservative in the case where parallel mitigated systems have no secrets of their own to leak. If the graph of subsystems contains cycles, it cannot be decomposed into these two gadgets. We leave a more comprehensive analysis of mitigator composition to future work.

4.9 Experiments

To evaluate the performance and information leakage of generalized timing mitigation, we implemented mitigators for different applications. The widely used Apache Tomcat web container was modified to mitigate a local hosted application. We also developed a mitigating web proxy to estimate the overhead of mitigating real-world applications—a non-trivial homepage that results in 49 different requests and a HTTPS webmail service that requires stronger security.

We explored how to tune this general mechanism for different security and performance requirements. The results show that predictive mitigation does slow down applications to some extent; we suggest the slowdown is acceptable for some applications.

4.9.1 Mitigator design and its limitations

We define the system boundary in the following way. Inputs enter the system at the point when Tomcat dispatches requests to the servlet or JSP code. Results returned from this code are considered outputs. Thus, all timing leakage arising during the processing of the servlet and the JSP files is mitigated.

This implementation of mitigation has limitations. Because of shared hardware and operating-system resources such as filesystem caches, memory caches, buses, and the network, the time required to deliver an application response may convey information about sensitive application data. Our current implementation strategy, chosen for ease of implementation, prevents fully addressing these timing channels where they affect timing outside the system boundary as defined.

To completely mitigate timing channels, mitigation should be integrated at the operating system and hardware levels. For example, the TCP/IP stack might be extended to support delaying packets until a mitigator-specified time. With such an extension, all timing channels, including low-level interactions via hardware caches and bus contention, would be fully mitigated. Although we leave the design of such a mechanism to future work, we see no reason why a more complete mitigation mechanism would significantly change the performance and security results reported here.

4.9.2 Mitigator implementation

We implemented the mitigator as a Java library containing 201 lines of Java source code, excluding comments and the configuration file. This library provides two functions:

```
Mitigator startMitigation (String requestType);
void      endMitigation   (Mitigator miti);
```

The function `startMitigation` should be invoked when an input is available to the system, passing an application-specific request type identifier. The function `endMitigation` is used by the application when an output is ready, passing the mitigator for the related input. Calling `endMitigation` blocks the current thread until the time predicted by the mitigator.

Instead of optimizing for specific applications, we heuristically choose the following parameters for all experiments: 1. *Initial penalty*: the initial penalty for all request types is 50 ms, a delay short enough to be unnoticeable to the user. 2. *Penalty policy*: we use the 5-level grace period policy since it provides good tradeoff between security and performance as shown in 4.7.5. 3. *Penalty function*: most requests are returned within 250 ms, and the distribution is quite even. We evenly divide the first 5 epochs to make predictions more precise: 50 ms, 100 ms, 150 ms, 200 ms, 250 ms, doubling progressively thereafter. 4. *Worst-case execution time T_w*: We assume worst-case execution time for requests T_w to be 300 seconds. This is consistent with Firefox browser version 3.6.12, which uses this value as a default timeout parameter.

4.9.3 Leakage revisited

Applying the experiment settings into the formula from Section 4.7.6 with R request types, the following leakage bound obtains:

$$\begin{aligned}
&((R-1)\cdot(l+1)+(\log T_w+2))\cdot\log(M+1)\\
=&((R-1)\cdot 6+(\log 300000+2))\cdot\log(M+1)\\
\leq&(6\cdot R+15)\cdot\log(M+1)
\end{aligned}$$

where M is the number of inputs using the simple doubling scheme.

Intuitively, introducing more request types helps make the prediction more precise for each request, because processing time varies for different kinds of requests. However, the leakage bound is proportional to the number of request types. So it is important to find the right tradeoff between latency and security.

4.9.4 Latency and throughput

To enable the mitigation of unmodified web applications, we modified the open source Java Servlet and JavaServer Pages container Tomcat 6.0.29 using the mitigation library.

Experiment setup Mitigating Tomcat requires only three lines of Java code: one line generating a request type id from the HTTP request, one line to start the mitigation, and another line to end mitigation after the servlet is finished. We deployed a JSP wiki application, JSPWiki[2], in the mitigating Tomcat server

[2]http://www.jspwiki.org

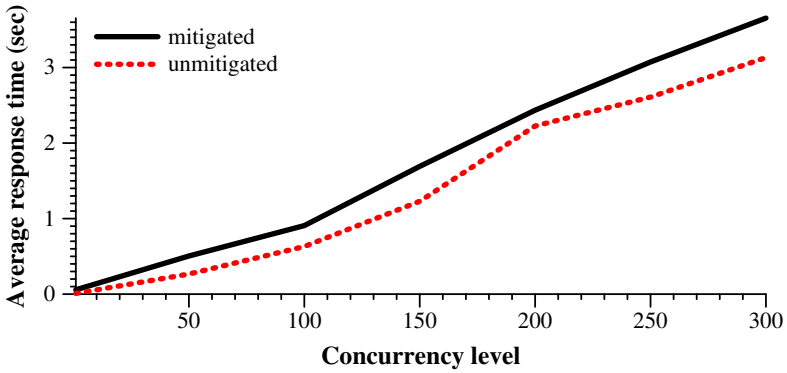


Figure 4.14: Wiki latency with and without mitigation.

to evaluate how mitigation affects both latency and throughput. Measurements were made using the Apache HTTP server benchmarking tool *ab*.[3] Since we focus on the latency and throughput overhead of requesting the main page of the wiki application, the URI is used as the request type identifier.

Results We measured the latency and throughput of the main page of JSPWiki for both the mitigated and unmitigated versions. We used a range of different concurrency settings in *ab*, controlling the number of multiple requests to perform at a time. The size of the Tomcat thread pool is 200 threads in the current implementation. For each setting, we measured the throughput for 5 minutes. The results are shown in Figure 4.14 and Figure 4.15.

When the concurrency level is 1—the sequential case—the unmitigated Wiki application has a latency around 11ms. Since the initial penalty is selected to be 50ms in our experiments, the average mitigated latency rises to about 57ms: about 400% overhead. This is simply an artifact of the choice of initial penalty.

As we increase the number of concurrent requests, the unmitigated appli-

[3]http://httpd.apache.org/docs/2.0/programs/ab.html

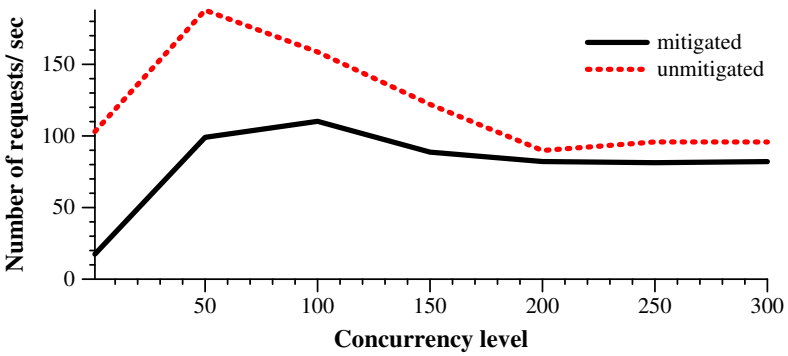


Figure 4.15: Wiki throughput with and without mitigation.

cation exhibits more latency, because concurrent requests compete for limited resources. On the other hand, the mitigation system is predicting this delayed time, and we can see that these predictions introduce less overhead: at most 90% after the concurrency level of 50; an even smaller overhead is found for higher concurrency levels.

The throughput with concurrency level 1 is much reduced from the unmitigated case: only about 1/5 of the original throughput. However, when the concurrency level reaches 50, throughput increases significantly in both cases, and the mitigated version has 52.73% of the throughput of the unmitigated version. For higher levels of concurrency, the throughput of the two versions is mostly similar.

4.9.5 Real-world applications with proxy

We evaluated the latency overhead of predictive mitigation on existing real-world web servers. To avoid the need to deploy predictive mitigation directly on production web servers, we introduce a mitigating proxy between the client browser and the target host. We modified an open source Java HTTP/HTTPS

proxy, *LittleProxy*[4], to use the mitigation library, adding about 70 LOC. We used it to evaluate latency with two remote web servers: a HTTP web page and an HTTPS webmail service.

With mitigation again done entirely at user level, timing channels that arise outside the mitigation boundary cannot be mitigated. The mitigation boundary is defined as follows: the mitigating proxy treats requests from client browser as inputs, and forwards these requests to the host. The response from the host is regarded as an output in the black-box model.

The proxy mitigates both the response time of the server and the round-trip time between the proxy and server. Only the first part corresponds to real variation that would occur with a mitigating web server. To estimate this part of latency overhead, we put the proxy in a local network with the real host. Because we found measured little variation in this configuration, the results here should estimate latency for real-world applications reasonably accurately.

HTTP web page

Unlike the previous stress test that requests only one URL, we evaluated latency overhead using a non-trivial HTTP web page, a university home page that causes 49 different requests to the server. Multiple requests bring up the opportunity of tuning the tradeoff between security and performance. Various ways to choose request types were explored:

1. TYPE/HOST: all URLs residing on the same host are treated as one request type, that is, they are predicted the same way.

[4]http://www.littleshoot.org/littleproxy/index.html

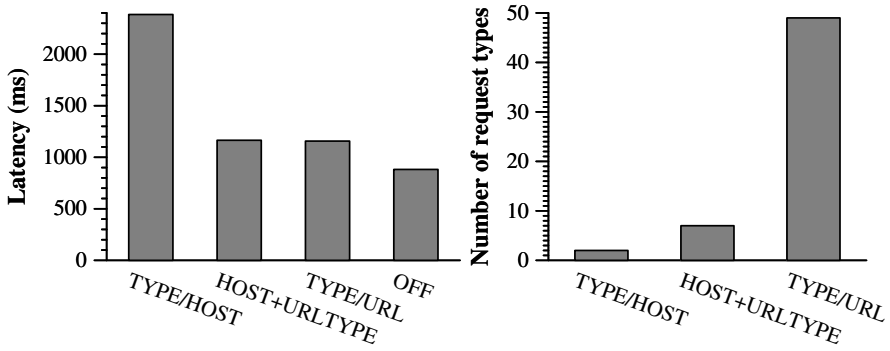


Figure 4.16: Latency for an HTTP web page.

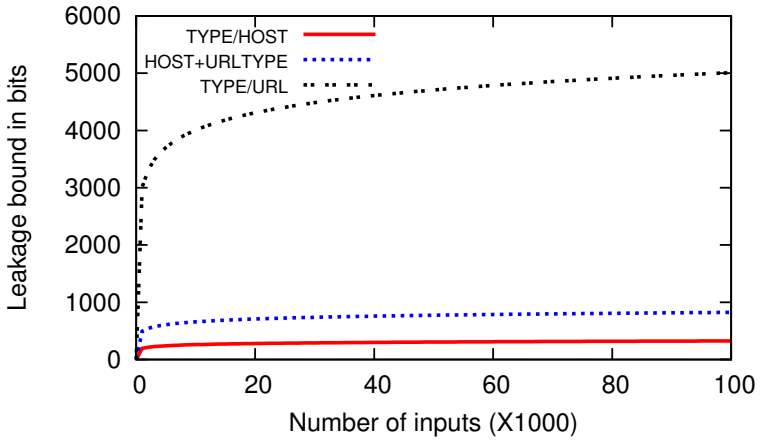


Figure 4.17: Leakage bound for an HTTP web page.

2. HOST+URLTYPE: requests on the same host are predicted differently based on the URL type of the request. We distinguish URL types based on the file types, such as JPEG files, CSS files and so on. Each of them corresponds to a different request type.

3. TYPE/URL: individual URLs are predicted differently.

Figure 4.16 shows the latency of loading the whole page and the number of request types with these options. The results show that latency in the most restrictive TYPE/HOST case almost triples that of the unmitigated case.

HOST+URLTYPE and TYPE/URL options have similar latency results, with about 30% latency overhead.

From the security point of view, the TYPE/HOST option only results in two request types: one host is in the organization, and the other one is `google-analytics.com`, used for the search component in the main page. HOST+URLTYPE introduces 6 more request types, while using the TYPE/URL option, there are as many as 49 request types. The information leakage bounds for different options are shown in Figure 4.17.

The HOST+URLTYPE choice provides a reasonable tradeoff between security and performance: it has roughly a 30% latency overhead, yet information leakage is below 850 bits for 100,000 requests.

HTTPS webmail service

We also evaluate the latency with a webmail service based on Windows Exchange Server. After the user passes Kerberos-based authentication (Auth), he is redirected to the login page (Login) and may then see the list of emails (List) or read a message (Email).

Request type selection This application accesses sensitive data, so we evaluate performance with the most restrictive scheme: one request type per host. There are actually two hosts: one host is used to serve only AuthPage.

Results We measured the latency overhead of four representative pages for this service. Each page generates from 6 to 45 different requests. The results in

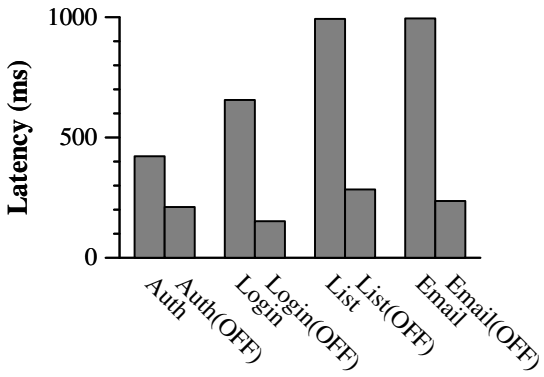


Figure 4.18: Latency overhead for HTTPS webmail service.

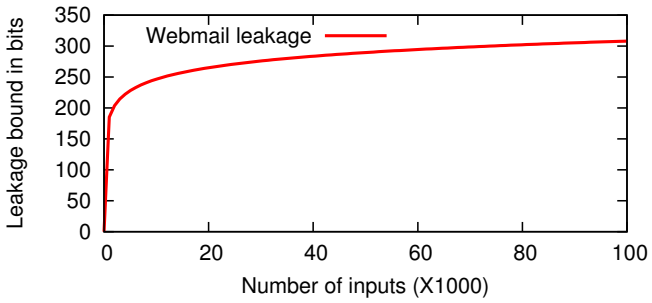


Figure 4.19: Leakage bound for HTTPS webmail service.

Figure 4.18 show that the latency overhead ranges from 2 times to 4 times for these four pages; in the worst case, latency is still less than 1 second. Also, this overhead can be reduced with different request type selection options.

Figure 4.19 shows the information leakage bound of this mitigated application. The leakage is limited to about 300 bits after 100,000 requests and grows very slowly thereafter.

4.10 Related work

Cryptographic side-channels One major motivation for controlling timing channels is the protection of cryptographic keys against side-channels arising from timing cryptographic operations. A variety of attacks that exploit timing side-channels have been demonstrated [13, 43]. Cryptographic blinding [15, 43] is a standard technique for mitigating such channels.

Köpf et al. [45, 46] introduced the mechanism of *bucketing* to mitigate timing side channels in cryptographic operations, achieving asymptotically logarithmic bounds on information leakage but with stronger assumptions than in this work. Their security analyses rely on the timing behavior of the system agreeing with a previously measured distribution of times; therefore they implicitly assume that the adversary does not control timing, and that there is a worst-case execution time. The bucketing approach does not achieve logarithmic bounds for general computation.

Quantitative information flow We advocate a quantitative approach to controlling information flow through timing channels. Like much other work on quantitative information flow [16, 53, 45, 46] we draw on information theory to obtain bounds on leakage. Millen [54] first observed that noninterference implies zero channel capacity between high and low. DiPierro et. al [69] quantify timing leaks in a language-based setting. Epoch-based mitigation is similar in spirit to Mode Security [12] which reduces covert channels to changes in modes. Unlike Mode Security, we also account for leakage within epochs, via a combinatorial analysis.

Mitigation of timing attacks Giles and Hajek present a comprehensive study of timing channels [27] in which packet arrival is represented by continuous or discrete waveforms. Similarly to us, they employ periodic quantization. However, because of the constant periods, the reduction of the timing channel bandwidth is only linear. Another difference lies in the semantics of buffer bounds: while they assume that a jammer has to release a packet from the queue when a buffer is full, our mitigators block the input source.

One prior approach to timing channel mitigation is adding noise to timing measurements. There are two ways to do this. First, we can add random delays to the time taken by various operations, which reduces the bandwidth of the timing channel, as in [36, 27]. Adding random delays sacrifices performance, and it does not asymptotically eliminate timing channels, since the noise can be eliminated to whatever degree is desired by averaging over a sequence of identical requests. Methods for creating covert timing robust against added noise have been demonstrated [51].

A second approach to mitigation, also used in [36], is that programs that read clocks are given results with random noise. This method only applies to internal timing channels that are based on reading clocks directly.

Wray [90] views every covert channel that originates from comparing two clocks as a timing channel. In this light, we focus on the channels that arise from comparing timing of the events to external reference clock that is not modulated by the attacker. Our results of Section 4.2.5 can be interpreted as mixing external timing channels and all other covert channels.

The NRL Pump [40] and its follow-ups, like Network Pump [41], are also

network service handling that handle requests. The Pump work addresses timing channels arising from message acknowledgments (which correspond to but are less general than outputs in this work). Acknowledgment timing is stochastically modulated using a moving average of past activity, and leakage in one window does not affect later windows. Therefore the NRL/Network Pumps can enforce only a linear leakage bound.

CHAPTER 5

LANGUAGE-BASED QUANTITATIVE CONTROL OF TIMING CHANNELS

The programming language in Chapter 2 disallows execution time depending on confidential information in any way. The limitation is that such a restrictive language can be impractical for applications where a limited amount of information leakage is allowed.

This chapter integrates the general predictive mitigation framework (Chapter 4) into the programming language in Chapter 2. The result is a permissive programming model which allows tight timing leakage bounds for applications.

5.1 A language with quantitative timing channel leakage

Figure 5.1 gives the full syntax for a simple imperative language for quantitative timing channel control. Compared with the syntax in Chapter 2 (Figure 2.1), the new part is the shaded `mitigate` command. As a technical convenience, each `mitigate` in the source has a unique identifier η. These identifiers are mainly used in Section 5.2; they are omitted where they are not essential.

$$
\begin{aligned}
e &::= n \mid x \mid e \ \texttt{op} \ e \\
c &::= \texttt{skip}_{[\ell_r,\ell_w]} \mid (x := e)_{[\ell_r,\ell_w]} \mid c; c \mid (\texttt{while} \ e \ \texttt{do} \ c)_{[\ell_r,\ell_w]} \\
&\quad \mid (\texttt{if} \ e \ \texttt{then} \ c_1 \ \texttt{else} \ c_2)_{[\ell_r,\ell_w]} \\
&\quad \mid (\texttt{mitigate}_{\eta} \ (e, \ell) \ c)_{[\ell_r,\ell_w]} \mid (\texttt{sleep} \ e)_{[\ell_r,\ell_w]}
\end{aligned}
$$

Figure 5.1: Syntax of the full language with the `mitigate` command.

$$\langle (\texttt{mitigate}\ (e, \ell)\ c)_{[\ell_r, \ell_w]}, m\rangle \rightarrow \langle c, m\rangle$$

Figure 5.2: Core semantics of the `mitigate` command.

$$\frac{\Gamma \vdash e : \ell \qquad pc \sqsubseteq \ell_w \qquad \Gamma, pc, \tau \sqcup \ell \sqcup \ell_r \vdash c : \tau' \quad \tau' \sqsubseteq \ell'}{\Gamma, pc, \tau \vdash (\texttt{mitigate}\ (e, \ell')\ c)_{[\ell_r, \ell_w]} : \ell \sqcup \tau \sqcup \ell_r}\ \text{T-MTG}$$

Figure 5.3: Typing rules: the `mitigate` command.

For `mitigate` we give an identity semantics for now: `mitigate` $(e, \ell)\ c$ simply evaluates to c.

Unlike other rules presented in Figure 3.5, the typing rule for `mitigate` command (T-MTG) does not propagate the timing end-labels of subcommands. Intuitively, this command "declassifies" information, justified by dynamic mechanisms that tighten information leakage from the mitigated commands. Like other rules, we require $pc \sqsubseteq \ell_w$. This restriction, together with Property 5 in Section 2.3.6, ensures that no confidential information about control flow leaks to the low parts of the machine environment.

In the added rule (T-MTG), the end-label τ' from command c is bounded by mitigation label ℓ', but τ' does not propagate to the end-label of the `mitigate`. Instead, the end-label of the `mitigate` command only accounts for the timing of evaluating expression e. This is because the predictive mitigation mechanism used at run time controls how c's timing leaks information.

5.2 Quantitative properties of the type system

The type system in Chapter 2 identifies potential timing channels in a program. We now introduce a quantitative measure of leakage for multilevel systems, and show that the type system for the extended language with `mitigate` command quantitatively bounds leakage through both timing and storage channels. The main result of this section is that information leakage can be bounded in the terms of the variation in the execution time of `mitigate` commands alone.

5.2.1 Adversary observations

As discussed earlier in Section 2.3.4, an adversary at level ℓ_A observes memory, including timing of updates to memory, at levels up to ℓ_A. The adversary does not directly observe the time of termination of the program, but this is easily simulated by adding a final low assignment to the program. To formally define adversary observations, we refine our presentation of the language semantics with *observable assignment events*.

Observable assignment events Let $\alpha \in \{(x, v, t), \epsilon\}$ range over observable events, which can be either an assignment to variable x of value v at time t, or an empty event ϵ. An event (x, v, G') is generated by assignment transitions $\langle x := e, m, E, G\rangle \rightarrow \langle \texttt{stop}, m', E', G'\rangle$, where $\langle m, e\rangle \Downarrow v$, and by all transitions whose derivation includes a subderivation of such a transition.

We write $\langle c, m, E, G\rangle \Rrightarrow (\mathbf{x}, \mathbf{v}, \mathbf{t})$ if configuration $\langle c, m, E, G\rangle$ produces a sequence of events $(\mathbf{x}, \mathbf{v}, \mathbf{t}) = (x_1, v_1, t_1) \ldots (x_n, v_n, t_n)$ and reaches a final configuration $\langle \texttt{stop}, m', E', G'\rangle$ for some m', E', G'.

ℓ_A-observable events An event (x, v, t) is observable to the adversary at level ℓ_A when $\Gamma(x) \sqsubseteq \ell_A$. Given a configuration $\langle c, m, E, G\rangle$ such that $\langle c, m, E, G\rangle \Rrightarrow (\mathbf{x}, \mathbf{v}, \mathbf{t})$, we write $\langle c, m, E, G\rangle \Rrightarrow_{\ell_A} (\mathbf{x}', \mathbf{v}', \mathbf{t}')$ for the longest subsequence of $(\mathbf{x}, \mathbf{v}, \mathbf{t})$ such that for all events (x_i, v_i, t_i) in $(\mathbf{x}', \mathbf{v}', \mathbf{t}')$ it holds that $\Gamma(x_i) \sqsubseteq \ell_A$.

For example, for program $l_1 := l_2; h_1 := l_1$, the H-adversary observes two assignments: $\langle c, m, E, G\rangle \Rrightarrow_{\mathrm{H}} (l_1, v_1, t_1), (h_1, v_2, t_2)$ for some v_1, t_1, v_2 and t_2. For the L-adversary, we have $\langle c, m, E, G\rangle \Rrightarrow_{\mathrm{L}} (l_1, v_1, t_1)$, which does not include the assignment to h_1.

5.2.2 Measuring leakage in a multilevel environment

Using ℓ_A-observable events, we can define a novel information-theoretic measure of leakage: leakage from a set of security levels $\mathcal{L}$ to an adversary level ℓ_A. We start with an observation on our adversary model and the corresponding auxiliary definition.

Because an adversary observes all levels up to ℓ_A, we can exclude these security levels from the ones that give new information. Let $\mathcal{L}_{\ell_A}$ be the subset of $\mathcal{L}$ that excludes all levels observable to ℓ_A, that is $\mathcal{L}_{\ell_A} \triangleq \{\ell' \mid \ell' \in \mathcal{L} \wedge \ell' \not\sqsubseteq \ell_A\}$. For example, for a three-level lattice $\mathrm{L} \sqsubseteq \mathrm{M} \sqsubseteq \mathrm{H}$, with $\ell_A = \mathrm{M}$, if $\mathcal{L} = \{\mathrm{M}, \mathrm{H}\}$ then $\mathcal{L}_{\ell_A} = \{\mathrm{H}\}$.

Figure 5.4(a) illustrates a general form of this definition. The adversary level ℓ_A is represented by the white point; the levels observable to the adversary correspond to the small rectangular area under the point ℓ_A. The set of security levels $\mathcal{L}$ is represented by the dashed rectangle (though in general this set does

(a) Leakage from $\mathcal{L}$ to ℓ_A (b) Variations with $\mathcal{L}^{\sqsubseteq}$

Figure 5.4: Quantitative leakage.

not have to be contiguous). The gray area corresponds to the security levels that are in $\mathcal{L}_{\ell_A}$.

Leakage from $\mathcal{L}$ to ℓ_A We measure the quantitative leakage as the logarithm (base 2) of the number of distinguishable observations of the adversary—the possible $(\mathbf{x}, \mathbf{v}, \mathbf{t})$ sequences—from indistinguishable memory and machine environments. As shown in Section 4.1.2, this measure bounds those of Shannon entropy and min-entropy, used in the literature [21, 54, 77].

Definition 11 (Quantitative leakage from $\mathcal{L}$ to ℓ_A) *Given any ℓ_A, m, and E, the leakage of program c from levels $\mathcal{L}$ to level ℓ_A, denoted by $\mathsf{Q}(\mathcal{L}, \ell_A, c, m, E)$ is defined as follows*

$$\mathsf{Q}(\mathcal{L}, \ell_A, c, m, E) \triangleq \log(|\, \{(\mathbf{x}, \mathbf{v}, \mathbf{t}) \mid \exists m', E' \,.\, (\forall \ell' \,.\, \ell' \not\in \mathcal{L}_{\ell_A} \,.\, m \simeq_{\ell'} m' \wedge E \simeq_{\ell'} E') \wedge \langle c, m', E', 0\rangle \Rrightarrow_{\ell_A} (\mathbf{x}, \mathbf{v}, \mathbf{t})\} \,|)$$

This definition uses $\mathcal{L}_{\ell_A}$ to restrict the quantification of the memory and machine environments so that we allow variations only in $\mathcal{L}_{\ell_A}$ parts of memory and machine environments. This is expressed by requiring projected equivalence (on

the second line of the definition) for all levels ℓ' not in $\mathcal{L}_{\ell_A}$. Visually, using Figure 5.4(a), this captures the flows from the gray area to the lower rectangle.

Note that the definition distinguishes flows between different levels. For example, in a three-level security lattice L $\sqsubseteq$ M $\sqsubseteq$ H and a program `sleep` (h) where h has level H, the leakage from {M} to L is zero even though flow from {H} to L is not.

5.2.3 Guarantees of the type system

The type system provides an important property: leakage from $\mathcal{L}$ to ℓ_A is bounded by the timing variation of the `mitigate` commands whose mitigation level ℓ' is in the *upward closure* of $\mathcal{L}_{\ell_A}$.

Upward closure In order to correctly approximate leakage from levels in $\mathcal{L}_{\ell_A}$, we need to account for all levels that are as restrictive as the ones in $\mathcal{L}_{\ell_A}$. For example, in a three-level lattice L $\sqsubseteq$ M $\sqsubseteq$ H, let $\mathcal{L}$ be the set {M}, and let ℓ_A = L; then $\mathcal{L}_{\ell_A}$ = {M}. Information from M can flow to H, so in order to account conservatively for leakage from {M}, we must also account for leakage from H. Our definitions therefore use the upward closure of $\mathcal{L}_{\ell_A}$, written as $\mathcal{L}_{\ell_A}\uparrow \triangleq \{\ell' \mid \exists \ell \in \mathcal{L}_{\ell_A} \wedge \ell \sqsubseteq \ell'\}$. In this example, $\mathcal{L}_{\ell_A}\uparrow$ = {M, H}. Figure 5.4(b) illustrates the relationship between $\mathcal{L}_{\ell_A}$ and its upper closure, where $\mathcal{L}_{\ell_A}\uparrow$ includes both shaded areas of gray.

Trace and projection of mitigate commands Next, we focus on the amount of time a `mitigate` command takes to execute. Recall from Section 5.1 that each

`mitigate` in a program source has an η-identifier. For brevity, we refer to the command $\texttt{mitigate}_\eta$ as M_η. Consider trace $\langle c, m, E, G\rangle \rightarrow^* \langle \texttt{stop}, m', E', G'\rangle$. We overload the notation for $\Rrightarrow$, by writing $\langle c, m, E, G\rangle \Rrightarrow (\mathbf{M}, \mathbf{t})$, where $(\mathbf{M}, \mathbf{t})$ is a vector of `mitigate` commands executed in the above trace. The vector consists of the individual tuples $(\mathbf{M}, \mathbf{t}) = (M_{\eta_1}, t_1) \ldots (M_{\eta_n}, t_n)$ where (M_{η_i}, t_i) are ordered by the time of completion, and each (M_{η_i}, t_i) corresponds to a $\texttt{mitigate}_{\eta_i}$ taking time t_i to execute.

Further, we define the *projection* of mitigate commands $(\mathbf{M}, \mathbf{t}) \upharpoonright^{\mathsf{f}}$ as the longest subsequence of $(\mathbf{M}, \mathbf{t})$, such that each (M_η, t) in the subsequence satisfies the predicate f.

Low-determinism of `mitigate` commands Consider the following well-typed program that uses `mitigate` twice.

```
mitigate1(1,H) {
  if (high)
  then mitigate2(1, H) { h:=h+1 }
  else skip; }
```

Let us write $pc(M_\eta)$ for the value of the pc-label at program point η. It is easy to see that $pc(M_1) = \mathrm{L}$, and $pc(M_2) = \mathrm{H}$. Because M_2 is nested within M_1, the timing of M_2 is accumulated in the timing of M_1. Therefore, when reasoning about the timing of the whole program, it is sufficient to only reason about the timing of M_1. In general, given a set of levels $\mathcal{L}$, an adversary level ℓ_A, and a vector $(\mathbf{M}, \mathbf{t})$, we filter high `mitigate` commands by the projection $(\mathbf{M}, \mathbf{t}) \upharpoonright^{pc(M_\eta) \not\in \mathcal{L}_{\ell_A}\uparrow}$. This projection consists of all the `mitigate` commands whose pc-label is in the white area in Figure 5.4(b).

Filtering out high `mitigate` commands rules out unrelated variations in the `mitigate` commands. It turns out that in well-typed programs, the occurrence of the remaining low `mitigate` commands is deterministic (we call these commands *low-deterministic*). This result, formalized in the following lemma, is used in the derivation of leakage bounds in Section 5.3.

Lemma 19 *(Low-determinism of* `mitigate` *commands). For all programs c such that $\Gamma \vdash c$, adversary levels ℓ_A, sets of security levels $\mathcal{L}$, and memories and environments E_1, E_2, m_1, m_2 such that $(\forall \ell' \notin \mathcal{L}_{\ell_A}\uparrow . E_1 \simeq_{\ell'} E_2 \wedge m_1 \simeq_{\ell'} m_2)$, we have*

$$\langle c, m_1, E_1, 0\rangle \Rrightarrow (\mathbf{M_1}, \mathbf{t_1}) \wedge \langle c, m_2, E_2, 0\rangle \Rrightarrow (\mathbf{M_2}, \mathbf{t_2}) \implies \mathbf{M_1}\restriction^{pc(M_\eta)\notin\mathcal{L}_{\ell_A}\uparrow} = \mathbf{M_2}\restriction^{pc(M_\eta)\notin\mathcal{L}_{\ell_A}\uparrow}$$

Note that there are no constraints on time components $\mathbf{t_1}$ and $\mathbf{t_2}$. That is, the same `mitigate` commands may take different times to execute in different traces. The proof is contained in Section 5.4.

Mitigation levels Per Section 5.1, the argument ℓ in $\texttt{mitigate}_\eta\ (e, \ell)\ c$ is an upper bound on the timing leakage of command c. Let $lev(M_\eta)$ be the label argument of $\texttt{mitigate}_\eta$ command. We call this the *mitigation level* of M_η. Note that $lev(M_\eta)$ is unrelated to $pc(M_\eta)$. For instance, in the example above, $pc(M_1)$ = L, because M_1 appears in the L-context, but $lev(M_1)$ = H.

Mitigation levels are connected to how much information an adversary at level ℓ_A may learn. For example, information at level ℓ can leak to adversary at level ℓ_A ($\ell \not\sqsubseteq \ell_A$) by a command M_η only when $\ell \sqsubseteq lev(M_\eta)$. In general, information from a set of levels $\mathcal{L}$ can be leaked by `mitigate` commands such that $lev(M_\eta) \in \mathcal{L}_{\ell_A}\uparrow$.

This leads to the definition of *timing variations*.

Definition 12 (Timing variations of `mitigate` commands) *Given a set of security levels $\mathcal{L}$, an adversary level ℓ_A, program c, memory m, and a machine environment E, let* V *be the* timing variations *of* `mitigate` *commands:*

$$\mathsf{V}(\mathcal{L}, \ell_A, c, m, E) \triangleq \{\mathbf{t}' \mid \exists\, m', E' \,.\, (\forall \ell' \notin \mathcal{L}_{\ell_A}\uparrow \,.\, m \simeq_{\ell'} m' \wedge E \simeq_{\ell'} E') \wedge \langle c, m', E', 0\rangle \Rrightarrow (\mathbf{M}, \mathbf{t}) \wedge (\mathbf{M}', \mathbf{t}') = (\mathbf{M}, \mathbf{t}) \restriction^{pc(M_\eta) \notin \mathcal{L}_{\ell_A}\uparrow \wedge lev(M_\eta) \in \mathcal{L}_{\ell_A}\uparrow}\}$$

An interesting component of this definition is the predicate used to project $(\mathbf{M}, \mathbf{t})$. In essence, we only focus on the `mitigate` commands that appear in low contexts and have high mitigation levels, such as the first `mitigate` in the example earlier. Also notice that this set counts only the distinct timing components of the `mitigate` command projection, ignoring the $\mathbf{M}'$ component. This is sufficient because for well-typed programs the $\mathbf{M}'$ components of the vectors $(\mathbf{M}', \mathbf{t}')$ are low-deterministic by Lemma 19.

In this definition, memory and machine environments are quantified differently from Definition 11, by considering variations with respect to a larger set of security levels $\mathcal{L}_{\ell_A}\uparrow$. In Figure 5.4(b), this corresponds to flows from both gray areas to the area observable by the adversary.

Leakage bounds guaranteed by the type system The type system ensures that only the execution time of `mitigate` commands within certain projections may leak information.

Theorem 6 (Bound on leakage via variations) *Given a command c, such that $\Gamma \vdash c$, and an adversary level ℓ_A, we have that for all m, E and $\mathcal{L}$ it holds that*

$$\mathsf{Q}(\mathcal{L}, \ell_A, c, m, E) \leq \log |\mathsf{V}(\mathcal{L}, \ell_A, c, m, E)|$$

The proof is included in Section 5.4. An interesting corollary of the theorem is that leakage is zero whenever a program c contains no `mitigate` command, or more generally, when all `mitigate` commands take fixed time since there is only one timing variation of `mitigate` commands in this special case.

5.3 Predictive mitigation

The predictive mitigation framework introduced in Chapter 4 removes confidential information from timing of public events by delaying them according to predefined schedules. We now build upon this framework to enable tight leakage bounds when `mitigate` commands are used.

Instead of delaying public assignments themselves, we delay the completions of `mitigate` commands that may potentially precede public events. This is sufficient for well-typed programs, because according to Theorem 6, only timing variations of `mitigate` commands carry sensitive information. The idea is that as long as the execution time of the `mitigate` command is no greater than predicted, little information is leaked. Upon a misprediction (when actual execution time is longer than predicted), a new schedule is chosen in such a way that future mispredictions are rarer.

$$\frac{\langle \texttt{update}(n,\ell), m, E, G\rangle \rightarrow}{\langle (\texttt{while}\ (t - s_\eta \geq \texttt{predict}(n,\ell))\ \texttt{do}\ (\texttt{Miss}[\ell] := \texttt{Miss}[\ell] + 1;)_{[\bot,\bot]})_{[\bot,\bot]}, m, E, G\rangle} \text{ (S-UPDATE)}$$

$$\frac{\langle \texttt{mitigate}_\eta\ (n,\ell)\ c, m, E, G\rangle \rightarrow}{\langle s_\eta := t_{[\bot,\bot]}; c; \texttt{update}(n,\ell); (\texttt{sleep}\ (\texttt{predict}(n,\ell) - t + s_\eta))_{[\bot,\bot]}, m, E, G\rangle} \text{ (S-MTGPRED)}$$

Figure 5.5: Predictive semantics for `mitigate`.

5.3.1 Mitigating semantics

Figure 5.5 shows the fragment of the small-step semantics that implements predictive mitigation. We record mispredictions in a special array `Miss`, assuming `Miss` is initialized to zeros and is otherwise unreferenced in programs. Expression `time` provides the current value of the global clock. Expression $\texttt{predict}(n,\ell) = \max(n,1) \cdot 2^{\texttt{Miss}[\ell]}$ returns the current prediction for level ℓ with initial estimate n. This prediction is the *fast doubling scheme* (Section 4.1.4) with the *local penalty policy* (Section 4.7.2); other schemes and penalty policies are possible (Chapter 4), but are not considered here.

In rule (S-MTGPRED), `mitigate` transitions to a code fragment that penalizes and delays the execution time of c. Variable $\texttt{s}_\eta$ records the time when mitigation has started. If execution of `c` takes less time ($\texttt{time} - \texttt{s}_\eta$) than predicted, command `update` does nothing; the execution idles until the predicted time. If executing `c` takes longer than predicted, `update` increments $\texttt{Miss}[\ell]$ until the new prediction is greater than the time that c has consumed.

5.3.2 Leakage analysis of the global policy

Note that all auxiliary commands in Figure 5.5 have labels $[\bot, \bot]$, ensuring that no confidential information about machine environments is leaked when executing these commands. Moreover, the execution time of the whole mitigated block is at least $\texttt{predict}(n, \ell)$. Thus, the timing variation of a single `mitigate` command is controlled by the variation of possible values of $\texttt{predict}(n, \ell)$.

The global policy (Section 4.7.2) penalizes all future `mitigate` commands after a misprediction. That is, whenever there is a misprediction, $\texttt{Miss}[\ell]$ is increased for all ℓ in the system in Rule (S-UPDATE), rather than just for the level that triggers the misprediction, as shown in Figure 5.5.

Let us analyze the variation of execution times given a sub-trace of mitigate commands with $pc(M_\eta) \notin \mathcal{L}_{\ell_A}\uparrow \wedge lev(M_\eta) \in \mathcal{L}_{\ell_A}\uparrow$. We call the period when there is no misprediction (including the mispredictions from other mitigate commands not in the trace) *epoch*.

Notice that by Definition 12 and Theorem 6, the only source of leakage is through the timing variation, which we bound next. The variation of the execution times of the sub-trace of mitigate commands depends on three factors:

1. Variation in one epoch: since all `mitigate` commands are delayed according to the schedule, variation in one epoch is bounded by the number of mitigate commands in the trace, which can be conservatively bounded by K, where K is the number of relevant `mitigate` statements in the trace, i.e., the ones that satisfy $pc(M_\eta) \notin \mathcal{L}_{\ell_A}\uparrow \wedge lev(M_\eta) \in \mathcal{L}_{\ell_A}\uparrow$, according to Theorem 6.

2. Possible schedules after misprediction: because in fast doubling scheme

there is only one possible schedule after every misprediction, this factor does not contribute to the number of variations.

3. Number of epochs: at the Nth epoch, predicted execution time is $t_\eta \times 2^{N-1}$ for all mitigate commands, where t_η is the initial prediction for M_η. Since $\forall \eta.t_\eta \geq 1$, we have $2^{N-1} \leq t_\eta \times 2^{N-1} \leq T$ with running time T. Therefore, $N \leq 1 + \log T$.

Putting them together, the total variation of execution times is bounded by $(K+1)^{1+\log T}$. This results in leakage of at most $\log(K+1) \times (1+\log T)$ bits. When K is unknown, it can be conservatively bounded by T, yielding an $O(\log^2 T)$ bound on leakage.

Number of epochs when worst-case execution time is known Note that most commands take limited time to execute. For example, command `sleep(x)` takes at most 2^8 time units to execute when $x \in [0, 2^8 - 1]$ (we assume interacting with the machine environment takes less than 1 time unit). Denote by T_w the worst-case execution time for all mitigated commands. When T_w is known, the number of epochs can be bounded by $1 + \log T_w$. Therefore, the leakage can be bounded by $\log(K+1) \cdot (1 + \log T_w)$.

5.3.3 Leakage analysis of the local policy

In contrast, the local policy (Section 4.7.2) only penalizes future `mitigate` commands with the same mitigation level. That is, a misprediction of mitigation level ℓ only increase `Miss`$[\ell]$ in Rule (S-UPDATE), which is the policy shown in Figure 5.5.

Consider level ℓ' and a sub-trace of mitigate commands with $lev(M_\eta) = \ell'$. By the nature of local penalty policy, the execution time of such commands are not affected by other `mitigate` commands executed. We refine the *epoch* as the period when there is no misprediction with level ℓ'. Similarly to the analysis of global penalty policy, the timing variation of the sub-trace is bounded by $(K+1)^{1+\log T}$.

For the leakage from $\mathcal{L}$, we only need to analyze the variation of mitigate commands with $lev(M_\eta) \in \mathcal{L}_{\ell_A}\uparrow$ according to Theorem 6. This gives us a bound of $|\mathcal{L}_{\ell_A}\uparrow| \times \log(K+1) \times (1+\log T)$ in total.

This bound on leakage has a nice property: the higher the information is in the lattice, the tighter is the bound. So, this policy introduces a differential leakage bound for multiple levels: information with more strict usage—labels higher in lattice—is enforced to leak less through the timing channel than less security-sensitive information—labels lower in lattice.

Number of epochs when worst-case execution time is known Similar to the analysis of the global policy, we can derive a tighter leakage bound when worst-case execution time is known. Denote by T_w the worst-case execution time for all mitigated commands. When T_w is known, the number of epochs can be bounded by $1 + \log T_w$. Therefore, the leakage can be bounded by $|\mathcal{L}_{\ell_A}\uparrow| \times \log(K+1) \cdot (1+\log T_w)$.

5.4 Proofs

Like earlier, we use a distinguished label L ("low") to define what is observable to the low observer. Since the lemmas and theorems are valid regardless of what level L is, the propositions proved hold for any label ℓ in the security lattice.

5.4.1 Extended language

Extended syntax The extended syntax is shown in Figure 5.7. We augment memories to map high variables to bracketed results. In a similar way, the syntax is augmented to include bracketed results, and bracketed commands. Intuitively, bracketed results represent values from high memory, and bracketed commands represent commands executed in a high pc context (such as in a branch with a high guard). Moreover, to keep the structure of `mitigate` commands in the small-step semantics, we introduce braced commands $\{c\}$. Braced commands are executed in `mitigate` commands.

Extended semantics The operational semantics is augmented to propagate brackets and braces, as shown in Figure 5.8, 5.9. All rules are extensions to

$$n \sim n \quad [n_1] \sim [n_2] \quad m_1 \sim m_2 \implies \forall x. m_1(x) \sim m_2(x)$$

$$c \sim c \quad \frac{c_1 \sim c_3 \qquad c_2 \sim c_4}{c_1; c_2 \sim c_3; c_4} \quad [c_1] \sim [c_2] \quad \frac{c_1 \sim c_2}{\{c_1\} \sim \{c_2\}}$$

Figure 5.6: Equivalence on memories and commands.

$$e ::= \ldots \mid [n]$$
$$c ::= \ldots \mid [c] \mid \{c\}$$

Figure 5.7: Extended syntax.

the original grammar except that (S-ASGN) is split into three rules: (S-ASGN1), (S-ASGN2), (S-ASGN3), (S-MITIGATE) is replaced with (S-MITIGATE1) which introduces braced commands. All rules with brackets and braces work the same way as the normal rules from computational perspective. Brackets and braces are just syntactic markers.

To make the proof self-contained, typing rules for expressions are shown in Figure 5.10. Most of these rules are standard, except the rule (T-BRACKETEXP) that treats bracketed expression as high. Additional typing rules in Figure 5.11 are given to support the soundness proof. A command in bracket is treated as a command conditioned on high information in the type system, but commands in braces type-check in the same way as those without braces. The rule (T-STOP) handles `stop`, which appears only during evaluation. The subsumption rule (T-SUB) is introduced to simplify the proof. The intuition is that the end-label can always be treated more conservatively without hurting security.

Equivalence on memories and commands We define the equivalence of memories and commands up to the observable level of an adversary as in Fig. 5.6. Intuitively, bracketed memory and commands are indistinguishable. Braces are syntactic, so the equivalence on a command in brace is identical to that of a command without a brace.

We write $\vdash m$ if the values of all high (and only high) variables have brackets, and say that such a memory is well-formed.

$$\langle [n], m\rangle \downarrow [n] \qquad \frac{\langle e_1, m\rangle \downarrow [v_1] \qquad \langle e_2, m\rangle \downarrow v_2 \qquad v = v_1 \ \mathsf{op}\ v_2}{\langle e_1 \ \mathsf{op}\ e_2, m\rangle \downarrow [v]}$$

$$\frac{\langle e_1, m\rangle \downarrow v_1 \qquad \langle e_2, m\rangle \downarrow [v_2] \qquad v = v_1 \ \mathsf{op}\ v_2}{\langle e_1 \ \mathsf{op}\ e_2, m\rangle \downarrow [v]} \qquad \frac{\langle e_1, m\rangle \downarrow [v_1] \qquad \langle e_2, m\rangle \downarrow [v_2] \qquad v = v_1 \ \mathsf{op}\ v_2}{\langle e_1 \ \mathsf{op}\ e_2, m\rangle \downarrow [v]}$$

Figure 5.8: Extended semantics of expressions.

S-STOP1

$$\langle [\mathtt{stop}], m\rangle \to \langle \mathtt{stop}, m\rangle$$

S-STOP2

$$\langle \{\mathtt{stop}\}, m\rangle \to \langle \mathtt{stop}, m\rangle$$

S-BRACKET

$$\frac{\langle c, m\rangle \to \langle c', m'\rangle}{\langle [c], m\rangle \to \langle [c'], m'\rangle}$$

S-BRACE

$$\frac{\langle c, m\rangle \to \langle c', m'\rangle}{\langle \{c\}, m\rangle \to \langle \{c'\}, m'\rangle}$$

S-ASGN1

$$\frac{\langle e, m\rangle \downarrow v \qquad \Gamma(x) \sqsubseteq L}{\langle x := e_{[\ell_r,\ell_w]}, m\rangle \to \langle \mathtt{stop}, m[x \mapsto v]\rangle}$$

S-ASGN2

$$\frac{\langle e, m\rangle \downarrow v \qquad \Gamma(x) \not\sqsubseteq L}{\langle x := e_{[\ell_r,\ell_w]}, m\rangle \to \langle \mathtt{stop}, m[x \mapsto [v]]\rangle}$$

S-ASGN3

$$\frac{\langle e, m\rangle \downarrow [v]}{\langle x := e_{[\ell_r,\ell_w]}, m\rangle \to \langle \mathtt{stop}, m[x \mapsto [v]]\rangle}$$

S-IF3

$$\frac{\langle e, m\rangle \downarrow [n] \qquad n \neq 0}{\langle (\mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2)_{[\ell_r,\ell_w]}, m\rangle \to \langle [c_1], m\rangle}$$

S-IF4

$$\frac{\langle e, m\rangle \downarrow [n] \qquad n = 0}{\langle (\mathtt{if}\ e\ \mathtt{then}\ c_1\ \mathtt{else}\ c_2)_{[\ell_r,\ell_w]}, m\rangle \to \langle [c_2], m\rangle}$$

S-WHILE3

$$\frac{\langle e, m\rangle \downarrow [n] \qquad n \neq 0}{\langle (\mathtt{while}\ e\ \mathtt{do}\ c)_{[\ell_r,\ell_w]}, m\rangle \to \langle [c; (\mathtt{while}\ e\ \mathtt{do}\ c)_{[\ell_r,\ell_w]}], m\rangle}$$

S-WHILE4

$$\frac{\langle e, m\rangle \downarrow [n] \qquad n = 0}{\langle (\mathtt{while}\ e\ \mathtt{do}\ c)_{[\ell_r,\ell_w]}, m\rangle \to \langle [\mathtt{stop}], m\rangle}$$

S-MITIGATE1

$$\langle \mathtt{mitigate}\ (e, \ell)\ c, m\rangle \to \langle \{c\}, m\rangle$$

Figure 5.9: Extended semantics of commands.

$$\Gamma \vdash n : \bot \ \text{T-Const} \qquad \frac{\Gamma(x) = \ell}{\Gamma \vdash x : \ell}\ \text{T-Var} \qquad \frac{\Gamma \vdash e : \ell \qquad \Gamma \vdash e' : \ell}{\Gamma \vdash e\ \mathsf{op}\ e' : \ell}\ \text{T-Op}$$

$$\frac{\Gamma \vdash e : \ell \qquad \ell \sqsubseteq \ell'}{\Gamma \vdash e : \ell'}\ \text{T-SubExp} \qquad \text{T-BracketExp}\ \frac{\ell \not\sqsubseteq L}{\Gamma \vdash [n] : \ell}$$

Figure 5.10: Typing rules: expressions.

$$\text{T-Stop}\quad \Gamma, pc, \tau \vdash \mathtt{stop} : \tau \qquad \text{T-BracketCmd}\ \frac{\Gamma, \ell, \tau \vdash c : \tau' \qquad pc \sqsubseteq \ell \qquad \ell \not\sqsubseteq L}{\Gamma, pc, \tau \vdash [c] : \tau'} \qquad \text{T-BraceCmd}\ \frac{\Gamma, pc, \tau \vdash c : \tau'}{\Gamma, pc, \tau \vdash \{c\} : \tau}$$

$$\text{T-Sub}\ \frac{\Gamma, pc, \tau \vdash c : \tau_1 \qquad \tau_1 \sqsubseteq \tau_2}{\Gamma, pc, \tau \vdash c : \tau_2}$$

Figure 5.11: Extended typing rules.

5.4.2 Notations

While ℓ_A-observable events in big-step style are already defined by $\Rrightarrow_{\ell_A}$, the corresponding event in a single step is not defined. We represent the assignment generated by a single step as

$$\langle c_1, m_1, E_1, G_1\rangle \xrightarrow{(x,v)} \langle c_1', m_1', E_1', G_1'\rangle$$

$(x, v) = \emptyset$ when evaluating c_1 does not generate an assignment. Similarly, that of multiple steps is denoted as $\langle c_1, m_1, E_1, G_1\rangle \xrightarrow{(\mathbf{x},\mathbf{v})}{}^{*} \langle c_1', m_1', E_1', G_1'\rangle$.

The ℓ-projection of observable events $(\mathbf{x}, \mathbf{v})$, denoted by $(\mathbf{x}, \mathbf{v}) \restriction_\ell$, is the longest subset such that for any $(x, v) \in (\mathbf{x}, \mathbf{v}) \restriction_\ell, \Gamma(x) \sqsubseteq \ell$. By definition, we have

$$\langle c_1, m_1, E_1, G_1\rangle \xrightarrow{(\mathbf{x},\mathbf{v})}{}^{*} \langle \texttt{stop}, m_1', E_1', G_1'\rangle \Leftrightarrow \langle c_1, m_1, E_1, G_1\rangle \Rrightarrow_{\ell_A} (\mathbf{x}, \mathbf{v}) \restriction_{\ell_A}$$

5.4.3 Completeness of the extended language

We need to show the extended semantics is complete with regard to the original semantics. Completeness means every step in the new semantics can be performed in the original semantics (maybe with removal of brackets and braces) and vice versa.

More formally, given that c is a command in the extended language, let us use the notation of $\lfloor c \rfloor$ to denote removal of all brackets and braces from c in the obvious way, yielding a command from the original language. We define $\lfloor m \rfloor$ to convert memory in a similar way. Completenesss can be expressed as the following lemma.

Lemma 20 (Completeness of extended language)

$$\vdash c \wedge \langle \lfloor c \rfloor, \lfloor m \rfloor, E, G \rangle \rightarrow^* \langle \texttt{stop}, m', E', G' \rangle$$
$$\implies \exists m''. \langle c, m, E, G \rangle \rightarrow^* \langle \texttt{stop}, m'', E', G' \rangle \ \wedge \ m' = \lfloor m'' \rfloor$$

Proof. By rule induction on each evaluation step. □

5.4.4 Useful lemmas

Lemma 21 *Low expressions always evaluate to ordinary integers (without brackets):*

$$\vdash m \wedge \Gamma \vdash e : \ell \wedge \ell \sqsubseteq L \implies \exists n. \langle e, m \rangle \downarrow n$$

Proof. By induction on the structure of expressions.

- Case $e = n$: trivial.
- Case $e = [n]$: $\Gamma \vdash [n] : \ell \wedge \ell \not\sqsubseteq L$ by the typing rule. Contradiction.
- Case $e = x$: two conditions depending on $\Gamma(x)$:
  - $\Gamma(x) \sqsubseteq L$: since m is well-formed, $m(x) = n$ for some n. Therefore $\langle e, m \rangle \downarrow n$.
  - $\Gamma(x) \not\sqsubseteq L$: contradicts the condition that $\ell \sqsubseteq L$.
- Case $e = e_1 \ \mathsf{op} \ e_2$: suppose $\Gamma \vdash e_1 : \ell_1 \wedge \ell_1 \not\sqsubseteq L$ then e cannot be typed as $\ell \sqsubseteq L$. Otherwise, we have $\ell_1 \sqsubseteq \ell$ by (T-SUBEXP) and $\ell_1 \sqsubseteq L$ by the transitivity of $\sqsubseteq$ relation. Contradiction. Similarly, suppose $\Gamma \vdash e_2 : \ell_2$, we must have $\ell_2 \sqsubseteq L$. By the induction assumption, $\exists n_1, n_2. \langle e_1, m \rangle \downarrow n_1 \wedge \langle e_2, m \rangle \downarrow n_2$. Therefore, $\langle e, m \rangle \downarrow n_1 \ \mathsf{op} \ n_2$.

□

Lemma 22 (Monotonicity of TC) *The timing end label is no less than the pc label and timing start label. That is:*

$$\Gamma, pc, \tau \vdash c : \tau' \implies pc \sqcup \tau \sqsubseteq \tau'$$

Proof. Induction on the typing derivation $\Gamma, pc, \tau \vdash c : \tau'$. □

Lemma 23 (PC Subsumption)

$$\Gamma, pc, \tau \vdash c : \tau' \wedge pc' \sqsubseteq pc \implies \Gamma, pc', \tau \vdash c : \tau'$$

Proof. Induction on the typing derivation $\Gamma, pc, \tau \vdash c : \tau_1$.

- Case (T-SKIP): from the typing rule $\Gamma, pc', \tau \vdash \texttt{skip}_{[\ell_r, \ell_w]} : \tau \sqcup \ell_r$.
- Case (T-STOP), (T-BRACECMD): by the typing rule, $\tau' = \tau$.
- Case (T-SUB): by the induction hypothesis.
- Case (T-SLEEP): from the typing rule, $\Gamma \vdash e : \ell \wedge \tau' = \tau \sqcup \ell \sqcup \ell_r$. Also, $\Gamma, pc', \tau \vdash \texttt{sleep}\ (e) : \tau \sqcup \ell \sqcup \ell_r$.
- Case (T-ASGN): from the typing rule, $\tau' = \Gamma(x)$ and $\ell \sqcup pc \sqcup \tau \sqcup \ell_r \sqsubseteq \Gamma(x)$. Since $pc' \sqsubseteq pc$, $\ell \sqcup pc' \sqcup \tau \sqcup \ell_r \sqsubseteq \Gamma(x)$. So $\Gamma, pc', \tau \vdash (x := e)_{[\ell_r, \ell_w]} : \Gamma(x)$.
- Case (T-SEQ): from the typing rule, $\Gamma, pc, \tau \vdash c_1 : \tau_1 \wedge \Gamma, pc, \tau_1 \vdash c_2 : \tau'$. By the induction hypothesis, we have $\Gamma, pc', \tau \vdash c_1 : \tau_1 \wedge \Gamma, pc', \tau_1 \vdash c_2 : \tau'$.
- Case (T-BRACKETCMD): from the typing rule, $\Gamma, \ell', \tau \vdash c : \tau' \wedge pc \sqsubseteq \ell' \wedge \ell' \not\sqsubseteq L$. Since $pc' \sqsubseteq pc \sqsubseteq \ell'$ too, $\Gamma, pc', \tau \vdash [c] : \tau'$.

- Case (T-IF): from the typing rule, $\tau' = \ell \sqcup \tau \sqcup \tau_1 \sqcup \tau_2$ and $\Gamma, \ell \sqcup pc, \ell \sqcup \tau \sqcup \ell_r \vdash c_i : \tau_i$ where $\Gamma \vdash e : \ell$. Since $pc' \sqsubseteq pc$, $\Gamma, \ell \sqcup pc', \ell \sqcup \tau \sqcup \ell_r \vdash c_i : \tau_i$ by the induction hypothesis.
- Case (T-WHILE): from the typing rule, $\Gamma \vdash e : \ell \wedge pc \sqsubseteq \ell_w \wedge \ell \sqcup \tau \sqcup \ell_r \sqsubseteq \tau' \wedge \Gamma, \ell \sqcup pc, \tau' \vdash c : \tau'$. By the induction hypothesis, all conditions are still satisfied by replacing pc with pc'.
- Case (T-MTG): by the typing rule, $\Gamma \vdash e : \ell' \wedge pc \sqsubseteq \ell_w \wedge \Gamma, pc, \tau \sqcup \ell' \sqcup \ell_r \vdash c : \tau' \wedge \tau' \sqsubseteq \ell$. Since $pc' \sqsubseteq pc$, $pc' \sqsubseteq \ell_w$. Moreover, $\Gamma, pc', \tau \sqcup \ell' \sqcup \ell_r \vdash c : \tau'$ by the induction hypothesis.

□

Lemma 24 (TC Subsumption)

$$\Gamma, pc, \tau \vdash c : \tau_1 \wedge \tau' \sqsubseteq \tau \implies \Gamma, pc, \tau' \vdash c : \tau_1$$

Proof. Induction on the typing derivation $\Gamma, pc, \tau \vdash c : \tau_1$.

- Case (T-SKIP): from the typing rule, $\tau_1 = \tau \sqcup \ell_r$ and $\Gamma, pc, \tau' \vdash c : \tau' \sqcup \ell_r$. Since $\tau' \sqsubseteq \tau$, $\Gamma, pc, \tau' \vdash c : \tau_1$ by (T-SUB).
- Case (T-STOP): $\Gamma, pc, \tau' \vdash c : \tau'$. Since $\tau' \sqsubseteq \tau$, the result is true by (T-SUB).
- Case (T-SUB): by the induction hypothesis.
- Case (T-SLEEP): from the typing rule, $\Gamma \vdash e : \ell \wedge \tau_1 = \tau \sqcup \ell \sqcup \ell_r$. Also, $\Gamma, pc, \tau' \vdash \texttt{sleep}\ (e)_{[\ell_r, \ell_w]} : \tau' \sqcup \ell \sqcup \ell_r$. Since $\tau' \sqsubseteq \tau$, the result is true by (T-SUB).
- Case (T-ASGN): from the typing rule, $\ell \sqcup pc \sqcup \tau \sqcup \ell_r \sqsubseteq \Gamma(x)$. Since $\tau' \sqsubseteq \tau$, $\ell \sqcup pc \sqcup \tau' \sqcup \ell_r \sqsubseteq \Gamma(x)$ too. So $\Gamma, pc, \tau' \vdash c : \Gamma(x)$.

- Case (T-SEQ): from the typing rule, $\Gamma, pc, \tau \vdash c_1 : \tau_2 \wedge \Gamma, pc, \tau_2 \vdash c_2 : \tau_1$. By the induction hypothesis, we have $\Gamma, pc, \tau' \vdash c_1 : \tau_2$. Therefore, by (T-SEQ), $\Gamma, pc, \tau \vdash c_1; c_2 : \tau_1$.
- Case (T-BRACKETCMD): from the typing rule, $\Gamma, \ell', \tau \vdash c : \tau_1 \wedge pc \sqsubseteq \ell' \wedge \ell' \not\sqsubseteq L$. Since, $\tau' \sqsubseteq \tau$, $\Gamma, L', \tau' \vdash c : \tau_1$ by the induction hypothesis. By (T-BRACKETCMD), $\Gamma, pc, \tau' \vdash [c] : \tau_1$.
- Case (T-BRACECMD): by the induction hypothesis.
- Case (T-IF): from the typing rule, $\Gamma \vdash e : \ell \wedge \Gamma, \ell \sqcup pc, \ell \sqcup \tau \sqcup \ell_r \vdash c_i : \tau_i$. Since $\tau' \sqsubseteq \tau, \Gamma, \ell \sqcup pc, \ell \sqcup \tau' \sqcup \ell_r \vdash c_i : \tau_i$ by the induction hypothesis. Result is true by applying (T-IF) again.
- Case (T-WHILE): from the typing rule, $\Gamma \vdash e : \ell \wedge pc \sqsubseteq \ell_w \wedge \ell \sqcup \tau \sqcup \ell_r \sqsubseteq \tau'' \wedge \Gamma, \ell \sqcup pc, \tau'' \vdash c : \tau''$. Since $\tau' \sqsubseteq \tau$, $\ell \sqcup \tau' \sqcup \ell_r \sqsubseteq \tau''$ still holds. Other conditions are not affected by this replacement.
- Case (T-MTG): from the typing rule, $\Gamma \vdash e : \ell' \wedge \Gamma, pc, \tau \sqcup \ell' \sqcup \ell_r \vdash c : \tau'' \wedge \tau'' \sqsubseteq \ell$. Since $\tau' \sqsubseteq \tau, \Gamma, pc, \tau' \sqcup \ell' \sqcup \ell_r \vdash c : \tau''$ by the induction hypothesis. So by (T-MTG), the result is true.

□

Lemma 25 (Preservation)

$$\vdash m \wedge \Gamma, pc, \tau \vdash c : \tau' \wedge \langle c, m \rangle \rightarrow \langle c', m' \rangle \implies \vdash m' \wedge \Gamma, pc, \tau \vdash c' : \tau'$$

Proof. By rule induction on $\langle c, m \rangle \rightarrow \langle c', m' \rangle$.

- (S-SKIP, S-STOP1, S-STOP2, S-SLEEP, S-WHILE2): $\vdash m'$ is trivial since $m' = m$. Since $c' = \texttt{stop}$, $\Gamma, pc, \tau \vdash c' : \tau$. By Lemma 22, $\tau \sqsubseteq \tau'$. By (T-SUB), $\Gamma, pc, \tau \vdash c' : \tau'$.

- (S-BRACKET, S-BRACE): by the induction hypothesis.
- $\langle c_1; c_2, m\rangle \rightarrow \langle c_1'; c_2, m'\rangle$ ((S-SEQ1)): from the evaluation rule, $\langle c_1, m\rangle \rightarrow \langle c_1', m'\rangle$. By the typing rule, $\Gamma, pc, \tau \vdash c_1 : \tau_1 \wedge \Gamma, pc, \tau_1 \vdash c_2 : \tau'$. By the induction hypothesis, $\vdash m'$ and $\Gamma, pc, \tau \vdash c_1' : \tau_1$. So, $\Gamma, pc, \tau \vdash c_1'; c_2 : \tau'$.
- $\langle c_1; c_2, m\rangle \rightarrow \langle c_2, m'\rangle$ ((S-SEQ2)): by the typing rule, $\Gamma, pc, \tau \vdash c_1 : \tau_1 \wedge \Gamma, pc, \tau_1 \vdash c_2 : \tau'$. By Lemma 22, $\tau \sqsubseteq \tau_1$. By Lemma 24, $\Gamma, pc, \tau \vdash c_2 : \tau'$. By the induction hypothesis, $\vdash m'$.
- (S-ASGN1): $\Gamma, pc, \tau \vdash c' : \tau'$ is similar to (S-SKIP). Moreover, we have $\Gamma(x) \sqsubseteq L$. So changing the mapping of x to an ordinary integer in well-formed memory will result in well-formed memory too.
- (S-ASGN2) and (S-ASGN3): Similar to (S-ASGN1) except for $\vdash m'$. For rule (S-ASGN2), the type of x is high, so memory m' is still well-formed after the mapping for x is changed to a bracketed integer. For rule (S-ASGN3), $\Gamma \vdash e : \ell \wedge \ell \not\sqsubseteq L$ since otherwise, $\exists v, \langle e, m\rangle \downarrow v$ from Lemma 21. From the typing rule, $\ell \sqsubseteq \Gamma(x)$, so $\Gamma(x) \not\sqsubseteq L$. So changing the mapping of x to a bracketed integer will result in a well-formed memory too.
- (S-IF1) and (S-IF2): $\vdash m'$ since $m' = m$. From the typing rule, $\Gamma, \ell \sqcup pc, \ell \sqcup \tau \sqcup \ell_r \vdash c_i : \tau_i$. By Lemma 23 and 24, we have $\Gamma, pc, \tau \vdash c_i : \tau_i$.
- (S-IF3) and (S-IF4): $\vdash m'$ since $m' = m$. Since in either case, e evaluates to high value by evaluation rule, $\Gamma \vdash e : \ell \wedge \ell \not\sqsubseteq L$ by Lemma 21. From the typing rule, $\Gamma, \ell \sqcup pc, \ell \sqcup \tau \sqcup \ell_r \vdash c_i : \tau_i$. Thus, $\Gamma, pc, \ell \sqcup \tau \sqcup \ell_r \vdash [c_i] : \tau_i$. By Lemma 24, $\Gamma, pc, \tau \vdash [c_i] : \tau_i$. Since $\tau' = \ell \sqcup \tau \sqcup \tau_i$, $\Gamma, pc, \tau \vdash [c_i] : \tau'$ by (T-SUB).
- (S-WHILE1): $\vdash m'$ is trivial. From the typing rule, there is a τ' such that $\Gamma \vdash e : \ell \wedge \ell \sqcup \tau \sqcup \ell_r \sqsubseteq \tau' \wedge \Gamma, \ell \sqcup pc, \tau' \vdash c : \tau'$. By Lemma 23, 24, $\Gamma, pc, \tau \vdash c : \tau'$.

Also, since $\ell \sqcup \tau' \sqcup \ell_r = \tau'$, we can derive $\Gamma, pc, \tau' \vdash (\texttt{while}\ e\ \texttt{do}\ c)_{[\ell_r,\ell_w]} : \tau'$. By (T-SEQ), the result is true.

- (S-WHILE3): similar to (S-IF3), we have $\ell \not\sqsubseteq L$. Setting $\ell' = \ell \sqcup pc$ and by similar derivation as (S-WHILE1), we have $\Gamma, \ell', \tau \vdash c; (\texttt{while}\ e\ \texttt{do}\ c)_{[\ell_r,\ell_w]} : \tau'$. Thus, by (T-BRACKETCMD), the result is true.

- (S-WHILE4): similar to (S-IF3), we get $\ell \not\sqsubseteq L$. By the typing rule, checking [stop] has the following form ($\ell' = \ell \sqcup pc$):

$$\frac{\Gamma, \ell', \tau \vdash \texttt{stop} : \tau \qquad pc \sqsubseteq \ell' \qquad \ell' \not\sqsubseteq L}{\Gamma, pc, \tau \vdash [\texttt{stop}] : \tau}$$

  By Lemma 22, $\tau \sqsubseteq \tau'$. So $\Gamma, pc, \tau \vdash [\texttt{stop}] : \tau'$ by (T-SUB).

- (S-MITIGATE1): $\vdash m'$ since $m' = m$. From the typing rule, $\Gamma, pc, \tau \sqcup \ell \sqcup \ell_r \vdash c : \tau''$. By Lemma 24, $\Gamma, pc, \tau \vdash c : \tau''$. So $\Gamma, pc, \tau \vdash \{c\} : \tau$ by (T-BRACECMD). By (T-SUB), $\Gamma, pc, \tau \vdash \{c\} : \tau'$.

□

Lemma 26 (High-pc lemma) *Commands that type-check in a high-pc context neither generate low assignments nor modify low machine environment in one step.*

$$\forall pc, \tau. pc \not\sqsubseteq L \wedge \Gamma, pc, \tau \vdash c : \tau' \wedge \langle c, m, E, G\rangle \xrightarrow{(x,v)} \langle c', m', E', G'\rangle \implies (x, v) \restriction_L = \emptyset \wedge E \approx_L E'$$

Proof. First we show $E \approx_L E'$ by induction on the structure of c.

- stop: by the semantics, it does not change E.

- $c_1; c_2, \{c\}, [c]$: by the induction hypothesis.

- Other commands: all other commands have the form $c_{[\ell_r,\ell_w]}$. We have $pc \sqsubseteq \ell_w$ by the typing rule. Since $pc \not\sqsubseteq L$, $\forall \ell' \sqsubseteq L, \ell_w \not\sqsubseteq \ell'$ (otherwise, by transitivity of $\sqsubseteq$, we have $\ell_w \sqsubseteq L$). Therefore, by Property 5, we have $\forall \ell' \sqsubseteq L, E(\ell') = E'(\ell')$. That is, $E \approx_L E'$.

Next, we show $(x, v) \restriction_L = \emptyset$ by rule induction on the core semantics.

- Case (S-SKIP, S-STOP1, S-STOP2, S-SLEEP, S-IF1, S-IF2, S-IF3, S-IF4, S-WHILE1, S-WHILE2, S-WHILE3, S-WHILE4, S-MITIGATE): trivial since none of them generate an assignment in one step.
- Case (S-BRACKET): from the typing rule, $\Gamma, \ell', \tau \vdash c \wedge \ell' \not\sqsubseteq L$. So the result is true by the induction hypothesis.
- Case (S-BRACE): by the induction hypothesis.
- Case (S-SEQ1): from the typing rule, $\Gamma, pc, \tau \vdash c_1$. Since $pc \not\sqsubseteq L$, $(x, v) \restriction_L = \emptyset$ from the induction hypothesis.
- Case (S-SEQ2): similar to (S-SEQ1).
- Case (S-ASGN1): from the typing rule, $\ell \sqcup pc \sqcup \tau \sqcup \ell_r \sqsubseteq \Gamma(x)$, where $\Gamma \vdash e : \ell$. Since $pc \not\sqsubseteq L$, $\Gamma(x) \not\sqsubseteq L$. This contradicts the condition of (S-ASGN1).
- Case $\langle x := e, m, E, G\rangle \rightarrow \langle \texttt{stop}, m[x \mapsto [v]], E', G'\rangle$ ((S-ASGN2) and (S-ASGN3)): for (S-ASGN2), $\Gamma(x) \not\sqsubseteq L$. By Lemma 21 and typing rule, $\Gamma(x) \not\sqsubseteq L$ for (S-ASGN3). So $(x, v) \restriction_L = \emptyset$.

□

Lemma 27 (High-timing lemma) *Commands that type-check with high timing start label do not modify low memory or the machine environment in one step.*

$$\forall pc, \tau.\tau \not\sqsubseteq L \wedge \Gamma, pc, \tau \vdash c : \tau' \wedge \langle c, m, E, G\rangle \xrightarrow{(x,v)} \langle c', m', E', G'\rangle \implies (x, v) \restriction_L = \emptyset \wedge E \approx_L E'$$

Proof. Similar to the proof of Lemma 26 except that using the result of Lemma 26 for bracketed commands. □

Lemma 28

$$\forall \ell.m_1 \sim_\ell m_2 \wedge \Gamma \vdash e : \ell' \wedge \ell' \sqsubseteq \ell \wedge \langle e, m_1\rangle \downarrow v_1 \implies \exists v_2.\langle e, m_2\rangle \downarrow v_2 \wedge v_1 = v_2$$

Proof. By rule induction on expression evaluation. □

Lemma 29 (Unwinding)

$$\begin{aligned}\vdash m_1 \wedge \vdash m_2 \wedge \vdash c_1 \wedge \vdash c_2 \wedge m_1 \approx_L m_2 \wedge E_1 \approx_L E_2 \wedge c_1 \sim c_2 \wedge \langle c_1, m_1, E_1, G_1\rangle \xrightarrow{(x,v)} \langle c_1', m_1', E_1', G_1'\rangle \\ \implies (\exists c_2', m_2', E_2', G_2'.c_2' \sim c_1' \wedge \langle c_2, m_2, E_2, G_2\rangle \xrightarrow{(\mathbf{x},\mathbf{v})}{}^{*} \langle c_2', m_2', E_2', G_2'\rangle \\ \wedge (\mathbf{x}, \mathbf{v}) \restriction_L = (x, v) \restriction_L \wedge E_1' \approx_L E_2') \vee (\langle c_2, m_2\rangle \Uparrow \wedge \exists c.c_2 = [c])\end{aligned}$$

Proof. By rule induction on $\langle c_1, m_1, E_1, G_1\rangle \rightarrow \langle c_1', m_1', E_1', G_1'\rangle$.

- Case (S-SKIP, S-SLEEP, S-STOP2, S-MITIGATE1): since $c_1 \sim c_2$. Say c_2' is the result of taking one step from c_2. By the evaluation rule, we have $c_1' \sim c_2'$. $(\mathbf{x}, \mathbf{v}) \restriction_L = (x, v) \restriction_L$ since no assignments are generated by these commands. By Property 7, $E_1' \approx_L E_2'$.
- Case (S-STOP1): since $c_1 \sim c_2$, $c_2 = [c_4]$. If c_2 diverges, we are done with $c = c_4$. Otherwise, we have $\langle [c_4], m_2, E_2, G_2\rangle \rightarrow^* \langle [\texttt{stop}], m_2', E_2', G_2'\rangle \rightarrow$

$\langle \texttt{stop}, m_2', E_2', G_2' \rangle$. By the typing rule, c_4 is typable with $pc \not\sqsubseteq L$. By Lemma 26 and induction on the number of steps, we have $(\mathbf{x}, \mathbf{v}) \restriction_L = (x, v) \restriction_L = \emptyset \wedge E_1' \approx_L E_2'$. We choose $c_2' = \texttt{stop}$.

- Case $\langle [c], m_1, E_1, G_1 \rangle \rightarrow \langle [c'], m_1', E_1', G_1' \rangle$ ((S-BRACKET)): since $c_2 \sim c_1$, $c_2 = [c_4]$. By the typing rule, c is typable with $L' \not\sqsubseteq L$. By Lemma 26, we have $(x, v) \restriction_L = \emptyset \wedge E_1' \approx_L E_1 \approx_L E_2$. Therefore, we choose $c_2' = c_2$.
- Case (S-BRACE, S-SEQ1, S-SEQ2): by the induction hypothesis.
- Case (S-ASGN1, S-ASGN2, S-ASGN3): since $c_2 \sim c_1$, so $c_2 = c_1$. Thus the same evaluation rule will be applied when c_2 take one step. So $(\mathbf{x}, \mathbf{v})$ has the form (x', v'). Since $c_2 = c_1$, $x = x'$. For (S-ASGN1), we have condition $\langle e, m_1 \rangle \downarrow n$. By Lemma 28, $\langle e, m_2 \rangle \downarrow n$. So $v = v'$. For (S-ASGN2, S-ASGN3), we have $\Gamma(x) \not\sqsubseteq L$, thus $(x, v) \restriction_L = (x', v') \restriction_L = \emptyset$. From Property 7, $E_1' \approx_L E_2'$. We choose $c_2' = \texttt{stop}$.
- Case (S-IF1): assume $c_1 = (\texttt{if}\ e\ \texttt{then}\ c_3\ \texttt{else}\ c_4)_{[\ell_r, \ell_w]}$. Since $c_2 \sim c_1$, $c_2 = c_1$. By the evaluation rule, $\langle e, m_1 \rangle \downarrow n \wedge n \neq 0$. By Lemma 28, $\langle e, m_2 \rangle \downarrow n \wedge n \neq 0$. So we evaluate c_2 by one step and choose $c_2' = c_3$. Since no assignment is generated, $(x, v) \restriction_L = (\mathbf{x}, \mathbf{v}) \restriction_L = \emptyset$. $E_1' \approx_L E_2'$ by Property 7 and transitivity of $\approx_L$.
- Case (S-IF2, S-WHILE1, S-WHILE2): similar to (S-IF1).
- Case (S-IF3): assume $c_1 = (\texttt{if}\ e\ \texttt{then}\ c_3\ \texttt{else}\ c_4)_{[\ell_r, \ell_w]}$. Since $c_2 \sim c_1$, $c_2 = c_1$. By the evaluation rule, we have $\langle e, m_1 \rangle \downarrow [n]$. By Lemma 28, $\langle e, m_2 \rangle \downarrow [n]$. So c_2 may evaluate to either $[c_5]$ or $[c_6]$ by the evaluation rule. In either case, choosing c_2' to be the next step gives the desired properties.
- Case (S-IF4, S-WHILE3, S-WHILE4): similar to (S-IF3).

□

Lemma 30

$$\begin{aligned}\forall E_1, E_2, m_1, m_2, G, c, \ell \,.\, \Gamma \vdash c \wedge m_1 \sim_\ell m_2 \wedge E_1 \sim_\ell E_2 \\ \wedge \langle c, m_1, E_1, G\rangle \to^* \langle \mathtt{stop}, m_1', E_1', G_1\rangle \wedge \langle c, m_1, E_1, G\rangle \Rrightarrow_L (\mathbf{x_1}, \mathbf{v_1}, \mathbf{t_1}) \\ \wedge \langle c, m_2, E_2, G\rangle \to^* \langle \mathtt{stop}, m_2', E_2', G_2\rangle \wedge \langle c, m_2, E_2, G\rangle \Rrightarrow_L (\mathbf{x_2}, \mathbf{v_2}, \mathbf{t_2}) \\ \implies (\mathbf{x_1}, \mathbf{v_1}) = (\mathbf{x_2}, \mathbf{v_2}) \wedge E_1' \sim_\ell E_2'\end{aligned}$$

Proof. Induction on the number of steps using Lemma 25 and 29. □

Proof of Theorem 1

$$\begin{aligned}\forall E_1, E_2, m_1, m_2, G, c, \ell \,.\, \Gamma \vdash c \wedge m_1 \sim_\ell m_2 \wedge E_1 \sim_\ell E_2 \\ \wedge \langle c, m_1, E_1, G\rangle \to^* \langle \mathtt{stop}, m_1', E_1', G_1\rangle \wedge \langle c, m_2, E_2, G\rangle \to^* \langle \mathtt{stop}, m_2', E_2', G_2\rangle \\ \implies m_1' \sim_\ell m_2' \wedge E_1' \sim_\ell E_2'\end{aligned}$$

Proof. Note that memory below level L can only be modified by assignments where $\Gamma(x) \sqsubseteq L$. The result is directly implied by Lemma 30. □

Corollary 1

$$\begin{aligned}\forall E_1, E_2, m_1, m_2, G, c, \ell \,.\, \Gamma \vdash c \wedge (\forall \ell' \notin \mathcal{L}_{\ell_A}\uparrow . m_1 \simeq_{\ell'} m_2 \wedge E_1 \simeq_{\ell'} E_2) \\ \wedge \langle c, m_1, E_1, G\rangle \to^* \langle \mathtt{stop}, m_1', E_1', G_1\rangle \wedge \langle c, m_2, E_2, G\rangle \to^* \langle \mathtt{stop}, m_2', E_2', G_2\rangle \\ \implies \forall \ell' \notin \mathcal{L}_{\ell_A}\uparrow \,.\, m_1' \simeq_{\ell'} m_2' \wedge E_1' \simeq_{\ell'} E_2'\end{aligned}$$

Proof. Consider any $\ell_1 \notin \mathcal{L}_{\ell_A}\uparrow$ and any $\ell_2 \sqsubseteq \ell_1$. We have $\ell_2 \notin \mathcal{L}_{\ell_A}\uparrow$ since otherwise, by definition of upward closure, we have $\ell_1 \in \mathcal{L}_{\ell_A}\uparrow$. Contradiction. Therefore, by condition, we have $m_1 \simeq_{\ell_2} m_2 \wedge E_1 \simeq_{\ell_2} E_2$. Since this is true for all $\ell_2 \sqsubseteq \ell_1$,

$m_1 \sim_{\ell_1} m_2 \wedge E_1 \sim_{\ell_1} E_2$. By Theorem 1, we have $m_1' \sim_{\ell_1} m_2' \wedge E_1' \sim_{\ell_1} E_2'$. In particular, $m_1' \simeq_{\ell_1} m_2' \wedge E_1' \simeq_{\ell_1} E_2'$. Notice that this result applies to all $\ell_1 \notin \mathcal{L}_{\ell_A}\uparrow$, thus we get the desired result. □

5.4.5 Proof of timing properties

Lemma 31

$$\forall \ell \notin \mathcal{L}_{\ell_A}\uparrow . m_1 \sim_\ell m_2 \wedge E_1 \sim_\ell E_2 \iff \forall \ell \notin \mathcal{L}_{\ell_A}\uparrow . m_1 \simeq_\ell m_2 \wedge E_1 \simeq_\ell E_2$$

Proof. $\implies$: by definition of $\sim$.

$\impliedby$: for any level ℓ and $\ell' \sqsubseteq \ell$, we have $\ell' \notin \mathcal{L}_{\ell_A}\uparrow$ since otherwise, $\ell \in \mathcal{L}_{\ell_A}\uparrow$ by definition of upward closure. Contradiction. Therefore, $m_1 \simeq_{\ell'} m_2 \wedge E_1 \simeq_{\ell'} E_2$. Since this is true for all $\ell' \sqsubseteq \ell$, we have $m_1 \sim_\ell m_2 \wedge E_1 \sim_\ell E_2$.

□

Proof of Lemma 19 For all programs c, such that $\Gamma \vdash c$, adversary levels ℓ_A, sets of security levels $\mathcal{L}$, and memories and environments E_1, E_2, m_1, m_2 such that $(\forall \ell' \notin \mathcal{L}_{\ell_A}\uparrow . E_1 \simeq_{\ell'} E_2 \wedge m_1 \simeq_{\ell'} m_2)$, we have

$$\langle c, m_1, E_1, 0\rangle \Rrightarrow (\mathbf{M_1}, \mathbf{t_1}) \wedge \langle c, m_2, E_2, 0\rangle \Rrightarrow (\mathbf{M_2}, \mathbf{t_2}) \implies \mathbf{M_1} \restriction^{pc(M_\eta) \notin \mathcal{L}_{\ell_A}\uparrow} = \mathbf{M_2} \restriction^{pc(M_\eta) \notin \mathcal{L}_{\ell_A}\uparrow}$$

Proof. Consider any label $L \notin \mathcal{L}_{\ell_A}\uparrow$, we have $E_1 \sim_L E_2 \wedge m_1 \sim_L m_2$ by condition and Lemma 31. First consider all `mitigate` commands in the trace such that $pc(M_\eta) \sqsubseteq L$.

By the definition of environment pc, we have $\Gamma, pc(M_\eta), \tau \vdash M_\eta : \tau'$ for some Γ, τ, τ'. By the rule (T-BRACKETCMD), we know that M_η cannot appear in brackets since otherwise, we have $pc(M_\eta) \not\sqsubseteq L$ by typing rule. Therefore, similar to the proof of Lemma 29, we can show $\mathbf{M_1} \restriction^{pc(M_\eta) \sqsubseteq L} = \mathbf{M_2} \restriction^{pc(M_\eta) \sqsubseteq L}$. Since L can be an arbitrary label satisfying $L \notin \mathcal{L}_{\ell_A}\uparrow$, the whole projection is identical. □

Lemma 32

$$\forall e, \Gamma, m_1, m_2 \,.\, m_1 \approx_L m_2 \wedge \Gamma \vdash e : L \implies \forall x \in \mathbf{Vars} e.m_1(x) = m_2(x)$$

Proof. By induction on the structure of e. □

Lemma 33 (Timing determinism) *Starting from memory and machine environment that only differ in $\mathcal{L}_{\ell_A}\uparrow$, execution time of any command that type checks with end timing label that is not in $\mathcal{L}_{\ell_A}\uparrow$ is determined by low-deterministic* `mitigate` *trace s.t. $lev(M_\eta) \in \mathcal{L}_{\ell_A}\uparrow$. That is*

$$\begin{aligned}
\forall pc, \tau, c, m_1, m_2, E_1, E_2, G_1, G_2 \,.\, \Gamma, pc, \tau \vdash c : \tau' \wedge \tau' \notin \mathcal{L}_{\ell_A}\uparrow \wedge (\forall \ell' \notin \mathcal{L}_{\ell_A}\uparrow . E_1 \sim_{\ell'} E_2 \wedge m_1 \sim_{\ell'} m_2) \\
\wedge \langle c, m_1, E_1, G_0 \rangle \Rrightarrow (\mathbf{M_1}, \mathbf{t_1}) \wedge \langle c, m_2, E_2, G_0 \rangle \Rrightarrow (\mathbf{M_2}, \mathbf{t_2}) \\
\wedge (\mathbf{M_1}, \mathbf{t_1}) \restriction^{pc(M_\eta) \notin \mathcal{L}_{\ell_A}\uparrow \wedge lev(M_\eta) \in \mathcal{L}_{\ell_A}\uparrow} = (\mathbf{M_2}, \mathbf{t_2}) \restriction^{pc(M_\eta) \notin \mathcal{L}_{\ell_A}\uparrow \wedge lev(M_\eta) \in \mathcal{L}_{\ell_A}\uparrow} \\
\wedge \langle c, m_1, E_1, G_0 \rangle \to^* \langle \texttt{stop}, m_1', E_1', G_1 \rangle \wedge \langle c, m_2, E_2, G_0 \rangle \to^* \langle \texttt{stop}, m_2', E_2', G_2 \rangle \\
\implies G_1 = G_2
\end{aligned}$$

Proof. Induction on the structure of c.

- `stop`: trivial since $G_1 = G_0 = G_2$.

- $\texttt{skip}_{[\ell_r,\ell_w]}$: by the typing rule, $\ell_r \sqsubseteq \tau'$. Therefore, $\ell_r \notin \mathcal{L}_{\ell_A}\uparrow$, since otherwise $\tau' \in \mathcal{L}_{\ell_A}\uparrow$ by the definition of upward closure. Therefore, $E_1 \sim_{\ell_r} E_2$ by the condition. By Property 6, $G_1 = G_2$.

- $\texttt{sleep}\ (e)_{[\ell_r,\ell_w]}$: by the typing rule, $\Gamma \vdash e : \ell \wedge \ell \sqcup \ell_r \sqsubseteq \mathcal{L}_{\ell_A}\uparrow$. Similar to `skip`, $E_1 \sim_{\ell_r} E_2 \wedge m_1 \sim_{\ell_r} m_2$. By Lemma 32 and Property 6, $G_1 = G_2$.

- $x := e_{[\ell_r,\ell_w]}$: similar to `sleep` command.

- $[c]$: by (T-BracketCmd), $\Gamma, \ell_1, \tau \vdash c : \tau'$ for some ℓ_1. Since $\tau' \notin \mathcal{L}_{\ell_A}\uparrow$ by condition, $G_1 = G_2$ by the induction hypothesis.

- $c_1; c_2$: The evaluation must take the form $\langle c_1; c_2, m, E, G_0\rangle \rightarrow^* \langle c_2, m', E', G'\rangle \rightarrow^* \langle \texttt{stop}, m'', E'', G\rangle$. Suppose that $\langle c_1, m, E, G_0\rangle \Rrightarrow (\mathbf{M_1'}, \mathbf{t_1'})$ and $\langle c_2, m', E', G_0\rangle \Rrightarrow (\mathbf{M_2'}, \mathbf{t_2'})$. We distinguish the notations starting from $\langle m_1, E_1\rangle$ and $\langle m_2, E_2\rangle$ using subscript 1 and 2 in the obvious way. Then we have $(\mathbf{M_1}, \mathbf{t_1}) = (\mathbf{M_1'} \cdot \mathbf{M_1''}, \mathbf{t_1'} \cdot \mathbf{t_2''})$. Similarly for $(\mathbf{M_2}, \mathbf{t_2})$.

  By Lemma 19, $\mathbf{M_1'} \restriction^{pc(M_\eta)\notin\mathcal{L}_{\ell_A}\uparrow} = \mathbf{M_2'} \restriction^{pc(M_\eta)\notin\mathcal{L}_{\ell_A}\uparrow}$. Therefore, we have

  $$(\mathbf{M_1'}, \mathbf{t_1'}) \restriction^{pc(M_\eta)\notin\mathcal{L}_{\ell_A}\uparrow\wedge lev(M_\eta)\in\mathcal{L}_{\ell_A}\uparrow} = (\mathbf{M_2'}, \mathbf{t_2'}) \restriction^{pc(M_\eta)\notin\mathcal{L}_{\ell_A}\uparrow\wedge lev(M_\eta)\in\mathcal{L}_{\ell_A}\uparrow}$$

  and

  $$(\mathbf{M_1''}, \mathbf{t_1''}) \restriction^{pc(M_\eta)\notin\mathcal{L}_{\ell_A}\uparrow\wedge lev(M_\eta)\in\mathcal{L}_{\ell_A}\uparrow} = (\mathbf{M_2''}, \mathbf{t_2''}) \restriction^{pc(M_\eta)\notin\mathcal{L}_{\ell_A}\uparrow\wedge lev(M_\eta)\in\mathcal{L}_{\ell_A}\uparrow}$$

  By the typing rule, we have $\Gamma, pc, \tau \vdash c_1 : \tau_1$ and $\Gamma, pc, \tau_1 \vdash c_2 : \tau'$. By Lemma 22, $\tau_1 \sqsubseteq \tau'$. Since $\tau' \notin \mathcal{L}_{\ell_A}\uparrow$, we have $\tau_1 \notin \mathcal{L}_{\ell_A}\uparrow$. Thus by the induction hypothesis on c_1, we have $G_1' = G_2'$. By Corollary 1, we have $\forall \ell'' \notin \mathcal{L}_{\ell_A}\uparrow. m_1' \simeq_{\ell''} m_2' \wedge E_1' \simeq_{\ell''} E_2'$. By the induction hypothesis on c_2, we have $G_1 = G_2$.

- $(\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2)_{[\ell_r,\ell_w]}$: suppose $i = 1, 2$. We have $\Gamma \vdash e : \ell \wedge \Gamma, \ell \sqcup pc, \ell \sqcup \tau \sqcup \ell_r \vdash c_i : \tau_i \wedge \tau_1 \sqcup \tau_2 \notin \mathcal{L}_{\ell_A}\uparrow$ by (T-If). Therefore, $\ell_r \notin \mathcal{L}_{\ell_A}\uparrow \wedge \ell \notin \mathcal{L}_{\ell_A}\uparrow$. Thus, $m_1 \sim_\ell m_2$ by the condition.

By Lemma 28, e evaluates to the same value whether in $\langle m_1, E_1\rangle$ or $\langle m_2, E_2\rangle$. So the same rule must be applied. Without losing generality, let us continue rule (S-IF) when $e \neq 0$. The evaluation must take this form:

$$\langle \texttt{if } e \texttt{ then } c_1 \texttt{ else } c_2, m_i, E_i, G_0\rangle \to \langle c_1, m_i, E_i'', G_i''\rangle \to^* \langle \texttt{stop}, m_i', E_i', G_i\rangle$$

Since $\ell_r \not\in \mathcal{L}_{\ell_A}\uparrow$, we have $E_1 \sim_{\ell_r} E_2$. As $m_1 \sim_\ell m_2$, we have $G_1'' = G_2''$ by Property 6. Since the first step produces no mitigate command event, the second part produces identical `mitigate` command projections. By Property 7 and condition, we have $\forall \ell'' \not\in \mathcal{L}_{\ell_A}\uparrow. E_1'' \simeq_{\ell'} E_2''$. By Lemma 25, $\Gamma, pc, \tau \vdash c_1 : \tau'$. Therefore, $G_1 = G_2$ by the induction hypothesis.

- $(\texttt{while } e \texttt{ do } c)_{[\ell_r,\ell_w]}$: by the typing rule, we have $\Gamma \vdash e : \ell_1 \wedge \ell_1 \sqcup \tau \sqcup \ell_r \sqsubseteq \tau'$. So $\ell_1 \sqsubseteq \tau' \wedge \ell_r \sqsubseteq \tau'$. Therefore, we have $\ell_1 \not\in \mathcal{L}_{\ell_A}\uparrow \wedge \ell_r \not\in \mathcal{L}_{\ell_A}\uparrow$ since $\tau' \not\in \mathcal{L}_{\ell_A}\uparrow$. Thus $m_1 \sim_{\ell_1} m_2$ and $E_1 \sim_{\ell_r} E_2$ by the assumption.

  Similarly to the case of an `if` command, the same evaluation rule can be applied. We proceed with (S-WHILE1) and (S-WHILE2) since (S-WHILE3) is similar to (S-WHILE1) and (S-WHILE4) is similar to (S-WHILE2). We proceed by induction on the evaluation rule.

  - (S-WHILE2): since $m_1 \sim_{\ell_1} m_2$ and $E_1 \sim_{\ell_r} E_2$, $G_1 = G_2$ by Lemma 32 and the Property 6.
  - (S-WHILE1): denote $(\texttt{while } e \texttt{ do } c)_{[\ell_r,\ell_w]}$ as W. Evaluation has the form $\langle W, m_i, E_i, G_0\rangle \to \langle c; W, m_i, E_i'', G_i''\rangle \to^* \langle W, m_i''', E_i''', G_i'''\rangle \to^* \langle \texttt{stop}, m_i', E_i', G_i'\rangle$, where $i = 1, 2$.

    Similarly to the proof for `if` command, we can show $\forall \ell'' \not\in \mathcal{L}_{\ell_A}\uparrow. E_1'' \simeq_{\ell''} E_2'' \wedge G_1'' = G_2''$. Moreover, as in the proof for composition $(c_1; c_2)$, we have

$$\forall \ell'' \not\in \mathcal{L}_{\ell_A}\uparrow. m_1''' \sim_{\ell''} m_2''' \wedge E_1''' \sim_{\ell''} E_2''' \wedge G_1''' = G_2'''$$

and

$$(\mathbf{M_1'}, \mathbf{t_1'}) \restriction^{pc(M_\eta) \notin \mathcal{L}_{\ell_A}\uparrow \wedge lev(M_\eta) \in \mathcal{L}_{\ell_A}\uparrow} = (\mathbf{M_2'}, \mathbf{t_2'}) \restriction^{pc(M_\eta) \notin \mathcal{L}_{\ell_A}\uparrow \wedge lev(M_\eta) \in \mathcal{L}_{\ell_A}\uparrow}$$

where $\langle W, m_i''', E_i''', G_i''' \rangle \Rrightarrow (\mathbf{M_i'}, \mathbf{t_i'})$. By the induction hypothesis on the induction rule, we get $G_1 = G_2$.

- $(\texttt{mitigate}_\eta\ (e, \ell)\ c)_{[\ell_r, \ell_w]}$: first consider $\ell \in \mathcal{L}_{\ell_A}\uparrow$. We have $\Gamma, pc, \tau \vdash (\texttt{mitigate}\ (e, \ell)\ c)_{[\ell_r, \ell_w]} : \tau' \wedge \tau' \notin \mathcal{L}_{\ell_A}\uparrow$ by the condition. Therefore, we have $pc \notin \mathcal{L}_{\ell_A}\uparrow$ since otherwise, we have $\tau' \in \mathcal{L}_{\ell_A}\uparrow$ because $pc \sqsubseteq \tau'$. Contradiction. By definition, (η, G_i) is the last element in both projections. Since these two projections are equal, we have $G_1 = G_2$.

  Otherwise (when $\ell \notin \mathcal{L}_{\ell_A}\uparrow$), The evaluation must take the form

$$\langle (\texttt{mitigate}_\eta\ (e, \ell)\ c)_{[\ell_r, \ell_w]}, m_i, E_i, G_0 \rangle \rightarrow \langle c, m_i, E_i'', G_i'' \rangle \rightarrow^* \langle \texttt{stop}, m_i', E_i', G_i \rangle$$

  Since $\ell \notin \mathcal{L}_{\ell_A}\uparrow$ and by the typing rule $\Gamma \vdash e : \ell' \wedge \ell' \sqcup \ell_r \sqsubseteq \ell$, we have $\ell' \notin \mathcal{L}_{\ell_A}\uparrow \wedge \ell_r \notin \mathcal{L}_{\ell_A}\uparrow$, and thus $m_1 \sim_\ell m_2$ and $E_1 \sim_{\ell_r} E_2$. So, by Lemma 32 and the Property 6, we have $G_1'' = G_2''$. Since the first step produces no `mitigate` command event to the trace (only the end of a `mitigate` command may add to a trace), the second part produces the same trace projection.

  For all $\ell'' \notin \mathcal{L}_{\ell_A}\uparrow$, we can infer that $E_1 \sim_{\ell''} E_2$ by condition. By Property 7, $E_1'' \sim_{\ell''} E_2''$. So we have $\forall \ell'' \notin \mathcal{L}_{\ell_A}\uparrow. E_1'' \sim_{\ell''} E_2''$. By the typing rule (T-MTG), we have $\Gamma \vdash e : \ell' \wedge \Gamma, pc, \tau \sqcup \ell' \sqcup \ell_r \vdash c : \tau' \wedge \tau' \sqsubseteq \ell$. Since $\ell \notin \mathcal{L}_{\ell_A}\uparrow$, $\tau' \notin \mathcal{L}_{\ell_A}\uparrow$ too. Therefore, $G_1 = G_2$ by the induction hypothesis.

- $\{c\}$: braced command does not appear in the source code. Structural induction does not rely on this case.

□

Lemma 34 (Determinism of L-observable assignment events) *Focusing on $\ell_A = L$ for any label L, starting from memory and machine environment that only differ in $\mathcal{L}_L\uparrow$, L-observable assignment events that are generated by well-typed commands are determined by low-deterministic* `mitigate` *trace s.t. $lev(M_\eta) \in \mathcal{L}_L\uparrow$. That is*

$$\begin{aligned}\forall pc, \tau, \tau', c, m_1, m_2, E_1, E_2 \,.\ & \wedge \Gamma, pc, \tau \vdash c : \tau' \wedge (\forall \ell' \notin \mathcal{L}_L\uparrow \,.\, m_1 \sim_{\ell'} m_2 \wedge E_1 \sim_{\ell'} E_2) \\ & \wedge \langle c, m_1, E_1, G_0\rangle \Rrightarrow (\mathbf{M_1}, \mathbf{t_1}) \wedge \langle c, m_2, E_2, G_0\rangle \Rrightarrow (\mathbf{M_2}, \mathbf{t_2}) \\ & \wedge (\mathbf{M_1}, \mathbf{t_1}) \restriction^{pc(M_\eta)\notin\mathcal{L}_L\uparrow \wedge lev(M_\eta)\in\mathcal{L}_L\uparrow} = (\mathbf{M_2}, \mathbf{t_2}) \restriction^{pc(M_\eta)\notin\mathcal{L}_L\uparrow \wedge lev(M_\eta)\in\mathcal{L}_L\uparrow} \\ & \wedge \langle c, m_1, E_1, G_0\rangle \Rrightarrow_L (\mathbf{x_1}, \mathbf{v_1}, \mathbf{t_1}) \wedge \langle c, m_2, E_2, G_0\rangle \Rrightarrow_L (\mathbf{x_2}, \mathbf{v_2}, \mathbf{t_2}) \\ & \Longrightarrow (\mathbf{x_1}, \mathbf{v_1}, \mathbf{t_1}) = (\mathbf{x_2}, \mathbf{v_2}, \mathbf{t_2})\end{aligned}$$

Proof. Induction on the structure of c.

- `sleep` $(e)_{[\ell_r,\ell_w]}$, $\texttt{skip}_{[\ell_r,\ell_w]}$, `stop`: trivial since they produce no L-observable assignments.
- $[c]$: by Lemma 26, it produces no side effects either.
- $x := e_{[\ell_r,\ell_w]}$: when $\Gamma(x) \not\sqsubseteq L$, this command does not produce any L-observable assignment. Otherwise, we have $\Gamma, pc, \tau \vdash x := e_{[\ell_r,\ell_w]} : L$ by (T-Asgn) and (T-Sub). Also, we have $\Gamma \vdash e : \ell \wedge \ell \sqsubseteq L$. By Lemma 28, $\mathbf{v_1} = \mathbf{v_2}$.

  Suppose $\langle c, m_i, E_i, G_0\rangle \to \langle \texttt{stop}, m_i', E_i', G_i'\rangle$ where $i = 1, 2$. By definition, we have

  $\langle c, m_i, E_i, G_0\rangle \Rrightarrow_L (x_i, v_i, t_i)$ where $t_i = G_i' - G_0$. Notice that $L \notin \mathcal{L}_L\uparrow$ by definition, $G_1' = G_2'$ by Lemma 33. So $t_1 = t_2$.
- $c_1; c_2$: by the typing rule, $\Gamma, pc, \tau \vdash c_1 : \tau_1$ and $\Gamma, pc, \tau_1 \vdash c_2 : \tau'$. Moreover, the evaluation has the form $\langle c_1; c_2, m, E, G\rangle \to^* \langle c_2, m'', E'', G''\rangle \to^*$

$\langle \texttt{stop}, m', E', G' \rangle$. By the induction hypothesis, the first part produces same L-observable events. When $\tau_1 \not\sqsubseteq L$, by Lemma 27, the second part produces no L-observable events, so we are done. Otherwise, by Lemma 33, $G_1'' = G_2''$. By Corollary 1, $\forall \ell' \notin \mathcal{L}_L\uparrow. m_1'' \sim_{\ell'} m_2'' \wedge E_1'' \sim_{\ell'} E_2''$. By the induction hypothesis for c_2, the second part produces the same L-observable effects too.

- $(\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2)_{[\ell_r,\ell_w]}$: by the typing rule, $\Gamma \vdash e : \ell' \wedge \Gamma, pc \sqcup \ell', \tau \sqcup \ell' \sqcup \ell_r \vdash c_i : \tau_i$. When $\tau \sqcup \ell' \sqcup \ell_r \not\sqsubseteq L$, by Lemma 27, c_i produces no L-observable events. We are done. Otherwise, by Lemma 28, the same branch is taken. Without loss of generality, assume c_1 is executed. Then the evaluation must have this form:

$$\langle \texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2, m_i, E_i, G_0 \rangle \rightarrow \langle c_1, m_i, E_i'', G_i'' \rangle \rightarrow^* \langle \texttt{stop}, m_i', E_i', G_i \rangle$$

  By Property 6, we have $G_1'' = G_2''$. By Property 7, $E_1'' \approx_L E_2''$. Therefore, the result is true by the induction hypothesis on c_1.

- $(\texttt{while}\ e\ \texttt{do}\ c)_{[\ell_r,\ell_w]}$: by the typing rule, there is some τ' such that $\Gamma, pc, \tau' \vdash c : \tau'$. When $\tau' \not\sqsubseteq L$, c can produce no L-observable events by Lemma 27. We are done. Otherwise, we have $\Gamma \vdash e : \ell' \wedge \ell' \sqcup \tau \sqcup \ell_r \sqsubseteq \tau' \sqsubseteq L$ by the typing rule (T-WHILE). So only rule (S-WHILE1) and (S-WHILE2) can be applied and same rule is applied. We proceed by induction on the evaluation rule.

  - (S-WHILE2): trivial since no L-observable event is produced.
  - (S-WHILE1): denote $(\texttt{while}\ e\ \texttt{do}\ c)_{[\ell_r,\ell_w]}$ as W, evaluation has the form $\langle W, m_i, E_i, G_0 \rangle \rightarrow \langle c; W, m_i, E_i'', G_i'' \rangle \rightarrow^* \langle W, m_i''', E_i''', G_i''' \rangle \rightarrow^* \langle \texttt{stop}, m_i', E_i', G_i' \rangle$, where $i = 1, 2$. Since $\ell' \sqcup \ell_r \sqsubseteq L$, $G_1'' = G_2''$ by Property 6. Also, $E_1'' \sim_L E_2''$ by Property 7. By the induction hypothesis on command $c; W$, the second part of the evaluation produces the

same L-observable events. Moreover, since $\tau' \sqsubseteq L$ and $\Gamma, pc, \tau' \vdash c : \tau'$ from (T-WHILE), $\tau' \notin \mathcal{L}_L\uparrow$. Because otherwise, we have $L \in \mathcal{L}_L\uparrow$, which contradicts the definition of upward closure. By Lemma 33, $G_1''' = G_2'''$. By Corollary 1, $\forall \ell' \notin \mathcal{L}_L\uparrow . m_1''' \sim_{\ell'} m_2''' \wedge E_1''' \sim_{\ell'} E_2'''$. Therefore, by the induction hypothesis of the evaluation rule of (S-WHILE1) and (S-WHILE2), the last part in the evaluation trace produces same L-observable events too.

- $(\texttt{mitigate}\ (e, \ell)\ c)_{[\ell_r, \ell_w]}$: By the typing rule, we have $\Gamma \vdash e : \ell' \wedge \Gamma, pc, \tau \sqcup \ell \sqcup \ell_r \vdash c : \tau'$. When $\tau \sqcup \ell \sqcup \ell_r \not\sqsubseteq L$, c produces no L-observable events by Lemma 27. We are done.

  Otherwise, the evaluation must take the form

  $$\langle (\texttt{mitigate}\ (e, \ell)\ c)_{[\ell_r, \ell_w]}, m_i, E_i, G_0 \rangle \rightarrow \langle c, m_i, E_i'', G_i'' \rangle \rightarrow^* \langle \texttt{stop}, m_i', E_i', G_i \rangle$$

  Since $\ell' \sqcup \ell_r \sqsubseteq L$, we have $G_1'' = G_2''$ by Lemma 32 and Property 6. By Property 7, $E_1'' = E_2''$. Therefore, the result is true by the induction hypothesis on the second step.

- $\{c\}$: braced command does not appear in the source code. Structural induction does not rely on this case.

□

Proof of Theorem 6 Given a command c such that $\Gamma \vdash c$, we have that for all m, E, $\mathcal{L}$, ℓ and ℓ_A:

$$\mathsf{Q}(\mathcal{L}, \ell_A, c, m, E) \leq \log |\mathsf{V}(\mathcal{L}, \ell_A, c, m, E)|$$

Proof. For any $\mathcal{L}$, memory m and machine environment E, consider the case $\ell_A = L$, where L is an arbitrary label. We use a larger set Q', which is same as Q except that more parts of memory are allowed to vary, to bound Q:

$$\mathsf{Q}'(\mathcal{L}, L, c, m, E) \triangleq \log(|\ \{(\mathbf{x}, \mathbf{v}, \mathbf{t})\ \mid \exists m', E'\ .\ (\forall \ell'\ .\ \ell' \notin \mathcal{L}_L\uparrow\ .\ m \sim_{\ell'} m' \wedge E \sim_{\ell'} E')) \wedge \langle c, m', E', 0\rangle \Rrightarrow_L (\mathbf{x}, \mathbf{v}, \mathbf{t})\}\ |)$$

Consider the following set

$$\mathsf{V}'(\mathcal{L}, L, c, m, E) \triangleq \{(\mathbf{M}', \mathbf{t}') \restriction^{pc(\eta)\notin\mathcal{L}_L\uparrow\wedge lev(M_\eta)\in\mathcal{L}_L\uparrow}_{|} \exists\, m', E'\ .\ (\forall \ell' \notin \mathcal{L}_L\uparrow\ .\ m \sim_{\ell'} m' \wedge E \sim_{\ell'} E') \wedge \langle c, m', E', 0\rangle \Rrightarrow (\mathbf{M}', \mathbf{t}')\}$$

Notice that memory is quantified using $\sim$ instead of $\simeq$ because of Lemma 31.

By Lemma 19, we have $\mathsf{V}'(\mathcal{L}, L, c, m, E) = \mathsf{V}(\mathcal{L}, L, c, m, E)$.

Given any element $v \in \mathsf{V}'(\mathcal{L}, L, c, m, E)$, consider this set of memory and machine environments:

$$(\mathbf{m}, \mathbf{E}) = \{(m', E') | (\forall \ell' \notin \mathcal{L}_L\uparrow\ .\ m' \sim_{\ell'} m \wedge E' \sim_{\ell'} E) \wedge \langle c, m', E', 0\rangle \Rrightarrow (\mathbf{M}', \mathbf{t}') \wedge (\mathbf{M}', \mathbf{t}') \restriction^{pc(M_\eta)\notin\mathcal{L}_L\uparrow\wedge lev(M_\eta)\in\mathcal{L}_L\uparrow} = v\}$$

Pick any $(m_1, E_1) \in (\mathbf{m}, \mathbf{E})$, say $\langle c, m_1, E_1, 0\rangle \Rrightarrow_L (\mathbf{x_1}, \mathbf{v_1}, \mathbf{t_1})$. Then by Lemma 34, we have $\forall (m', E') \in (\mathbf{m}, \mathbf{E}), \langle c, m', E', 0\rangle \Rrightarrow_L (\mathbf{x_1}, \mathbf{v_1}, \mathbf{t_1})$.

Therefore, for all memory and machine environments that give the same element in V', they give the same element in Q'. By definition, both V' and Q' are quantifying over the same space of m, E, so we have

$$\mathsf{Q}'(\mathcal{L}, L, c, m, E) \leq \log |\mathsf{V}'(\mathcal{L}, L, c, m, E)|$$

Since the proof above works for any $L, \mathcal{L}, m, E, \ell$ and by the fact that $\mathsf{V} = \mathsf{V}'$, we get

$$\forall m, E, \mathcal{L}, \ell, \ell_A . \mathsf{Q}'(\mathcal{L}, \ell_A, c, m, E) \leq \log |\mathsf{V}(\mathcal{L}, \ell_A, c, m, E)|$$

Since $\mathsf{Q}(\mathcal{L}, \ell_A, c, m, E) \leq \mathsf{Q}'(\mathcal{L}, \ell_A, c, m, E)$, so

$$\forall m, E, \mathcal{L}, \ell, \ell_A . \mathsf{Q}(\mathcal{L}, \ell_A, c, m, E) \leq \log |\mathsf{V}(\mathcal{L}, \ell_A, c, m, E)|$$

□

Proof of Theorem 2

$$\forall E_1, E_2, m_1, m_2, G, c, \ell \,.\, \Gamma \vdash c \wedge c \text{ has no } \texttt{mitigate} \text{ commands} \wedge m_1 \sim_\ell m_2 \wedge E_1 \sim_\ell E_2$$
$$\wedge \langle c, m_1, E_1, G \rangle \Rrightarrow_\ell (\mathbf{x_1}, \mathbf{v_1}, \mathbf{t_1}) \wedge \langle c, m_2, E_2, G \rangle \Rrightarrow_\ell (\mathbf{x_2}, \mathbf{v_2}, \mathbf{t_2})$$
$$\implies \mathbf{x_1} = \mathbf{x_2} \wedge \mathbf{v_1} = \mathbf{v_2} \wedge \mathbf{t_1} = \mathbf{t_2}$$

Proof. Direct implication of Theorem 6, since c contains no `mitigate` commands. □

CHAPTER 6

EVALUATION

To evaluate our approach on real-world applications, we implemented a simulation of the partitioned cache design described in Section 2.4.3, as well as a formally verified MIPS processor using SecVerilog. As case studies, we chose two applications previously shown to be vulnerable to timing attacks, as well as security benchmarks. The results suggest that the approach proposed in this dissertation is sound and has reasonable performance.

6.1 Compilation

We use a modified GCC compiler to compile C applications with timing annotations. Sensitive data in applications are labeled, and timing labels are then inferred as the least restrictive labels satisfying the typing rules from Figure 3.5 (transferring the rules from Section 2.5 to C is straightforward).

To inform the hardware of the current timing label, a new register is added in hardware as an interface to communicate the timing label from the software to the hardware. Simply encoding the timing labels into instructions does not work, since labels may be required before the instruction is fetched and decoded: for example, to guide instruction cache behavior. Labels are also propagated along the pipeline to restrict the behavior of hardware. Assembly code setting the timing-label register is inserted before and after command blocks with same labels.

Name	# of sets	issue	block size	latency
L1 Data Cache	128	4-way	32 byte	1 cycle
L2 Data Cache	1024	4-way	64 byte	6 cycles
L1 Inst. Cache	512	1-way	32 byte	1 cycle
L2 Inst. Cache	1024	4-way	64 byte	6 cycles
Data TLB	16	4-way	4KB	30 cycles
Instruction TLB	32	4-way	4KB	30 cycles

Table 6.1: Machine environment parameters.

Selecting the initial prediction With the doubling policy, the slowdown of mitigation is at most twice the worst-case time. To improve performance, we can sample the running time of mitigated commands, and then set the initial prediction to be a little higher than the average. In the experiments, we used 110% of average running time, measured with randomly generated secrets, as the initial prediction.

6.2 Partitioned cache simulation

We developed a detailed, dynamically scheduled processor model supporting two-level data and instruction caches, data and instruction TLBs, and speculative execution. Table 6.1 summarizes the features of the machine environment. We implemented this processor design by modifying the SimpleScalar simulator, version 3.0e [14].

As discussed in Section 2.5.1, commodity cache designs require $\ell_r = \ell_w$. In our implementation, we treat this requirement as an extra side condition in the type system.

6.2.1 Web login case study

Web applications have been shown vulnerable to timing channel attacks. For example, Bortz and Boneh [11] have shown that adversaries can probe for valid usernames using a timing channel in the login process. This is unfortunate since usernames can be abused for spam, advertising, and phishing.

The pseudo-code for a simple web-application login procedure is shown below. The variable `response` and user inputs `user`, `pass` are public to users. Contents of the preloaded hashmap `m` (MD5 digests of valid usernames and corresponding passwords), password digest `hash` and the login status `state` are secrets. The final assignment to public variable `response` is always 1 on purpose in order to avoid the storage channel arising from the response. However, the timing of this assignment might create a timing channel.

```
Hashmap m:=loadusers()
while true
  (user, pass):=input()
  uhash:=MD5(user)
  if uhash in m
    hash:=m.get(uhash)
    phash:=MD5(pass)
    if phash=hash
      state:=success
  state:=fail
  response:=1
```

The information leakage is explicit when all confidential data (`m`, `hash` and `state`) are labeled `H`. The type system forces line 1 and line 5–10 to have high timing labels, so without a `mitigate` command, type checking fails at line 11. We secure this code by separately mitigating both line 1 and lines 5–10. The code then type-checks.

Correctness In each of our experiments, we measured the time needed to perform a login attempt using 100 different usernames. Since valid usernames (the hashmap `m`) are secrets in this case study, we varied the number of these usernames that were valid among 10, 50, and 100. The resulting measurements are shown as three curves in the upper part of Figure 6.1. The horizontal axis shows which login attempt was measured and the vertical axis is time.

The data for 10 and 50 valid usernames show that an adversary can easily distinguish invalid and valid usernames using login time. There is also measurable variation in timing even among different valid usernames. It is not clear what a clever adversary could learn from this, but since passwords are used in the computation, it seems likely that something about them is leaked too.

The lower part of the figure shows the timing of the same experiments with timing channel mitigation in use. With mitigation enabled, execution time does not depend on secrets, and therefore all three curves coincide. This result validates the soundness of our approach. The roughly 30-cycle timing difference between different requests does not represent a security vulnerability because it is unaffected by secrets; it is influenced only by public information such as the position in the request sequence.

Performance Table 6.2 shows the execution time of the main loop with various options, including both valid/invalid usernames, hardware with no partitions (nopar), and secure hardware both without (moff) and with (mon) mitigation.

As shown in Figure 6.1, for unmitigated logins, valid and invalid usernames can be easily distinguished, but mitigation prevents this (we also verified that the tiny difference is unaffected by secrets). Table 6.2 shows that partitioned

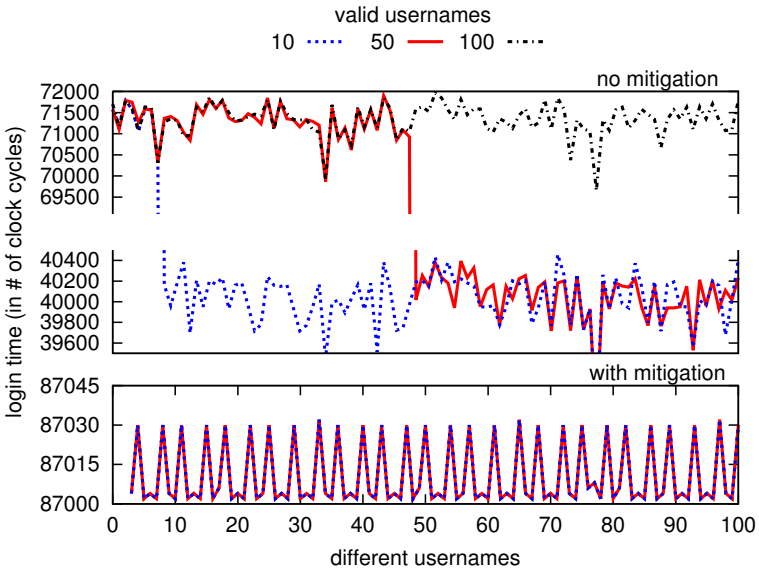


Figure 6.1: Login time with various secrets.

	nopar	moff	mon
ave. time (valid)	70618	78610	86132
ave. time (invalid)	39593	43756	86147
overhead (valid)	1	1.11	1.22

Table 6.2: Login time with various options (in clock cycles).

hardware is slower by about 11%. On valid usernames, language-based mitigation adds 10% slowdown; slowdown with combined software/hardware mitigation is about 22%.

6.2.2 RSA case study

The timing of efficient RSA implementations depends on the private key, creating a vulnerability to timing attacks [43, 13]. Using the RSA reference implementation, we demonstrate that its timing channels can be mitigated when decrypting a multi-block message.

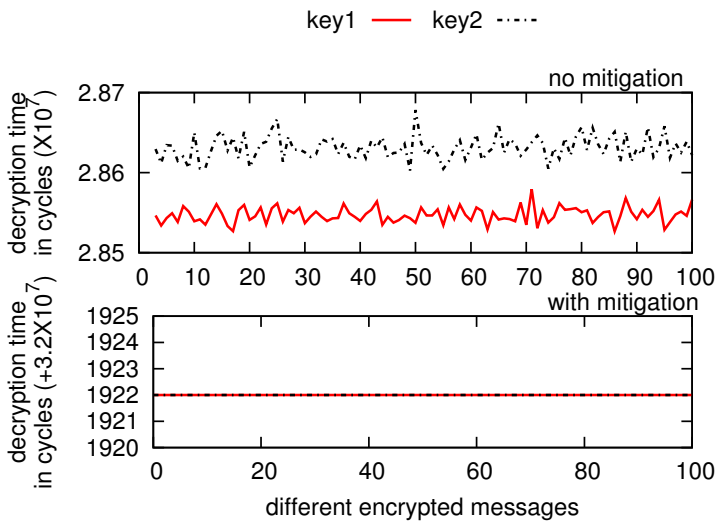


Figure 6.2: Decryption time with various secrets.

In the pseudo-code below, only the fourth line uses the confidential variable `key`. Therefore, source code corresponding to this line is labeled as high. Both "preprocess" and "postprocess" include low assignments whose timing is observable to the adversary.

```
text:=readText()
for each block b in text
  ...preprocess...
  compute (p:=b^key mod n)
  ...postprocess...
  write(output, plain)
```

Correctness We use 100 encrypted messages and two different private keys to measure whether secrets affect timing. The upper plot in Figure 6.2 shows that different private keys have different decryption times, so decryption time does leak information about the private key. The lower plot shows that mitigated time is exactly 32,001,922 cycles regardless of the private key. Timing channel leakage is successfully mitigated.

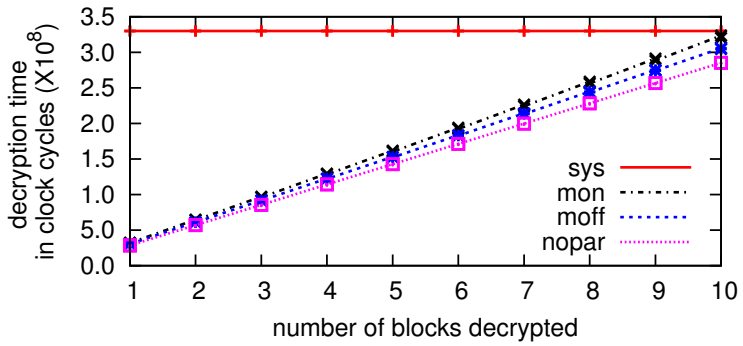


Figure 6.3: Language-level vs. system-level mitigation.

Performance To evaluate how mitigation affects decryption time, we use 10 encrypted secret messages whose size ranges from 1 to 10 blocks; the size is treated as public. We also compared the performance of language-level mitigation with the black-box mitigation (Section 4.1.1), where the entire decryption computation is mitigated, even though system-level mitigation is not effective against the strong, coresident attacker. To simulate system-level mitigation, the entire code body was wrapped in a single `mitigate` command. The results in Figure 6.3 show that fine-grained language-based mitigation is faster because it does not have to mitigate the timing variation due to the number of decrypted blocks, which is public information.

6.3 Formally verified MIPS processor

We used SecVerilog (Chapter 3) to design and verify a secure MIPS processor. We sketch the processor design, and show how SecVerilog helped avoid security vulnerabilities, including some not identified in prior work. We then provide results on the overhead of SecVerilog and timing channel protection. Overall, we found that the capability to statically control information flow at a fine gran-

ularity enables efficient secure hardware designs, and that the SecVerilog type system only requires a small number of changes to the Verilog code with no added run-time overhead.

6.3.1 A secure MIPS processor design

We implemented a SecVerilog compiler based on Icarus Verilog [1]. The constraints generated by the type system are solved by Z3 [58]. Using this implementation, we designed a complete MIPS processor that enforces the timing label contract discussed in Section 3.1.3. Our processor is based on a classic 5-stage in-order pipeline with separate instruction and data caches, both of which are 32kB and 4-way associative. The processor also includes typical pipelining techniques, such as data hazard detection, stalling and data bypassing, as well as a floating point unit (FPU) that we constructed using the Synopsys DesignWare library.

The Verilog code for our processor has more than 1700 LOC excluding the FPU, as shown in Table 6.3. LOC for the FPU is not reported because the source for the DesignWare library component is not available. Table 6.4 summarizes the processor's ISA, which is rich enough that we can compile a recent OpenSSL release with an off-the-shelf GCC compiler. The ISA is at least comparable to the ISAs of prior processors with formally verified security (e.g., [48]). New instructions `setr` and `setw` are used to set timing labels.

Our secure processor design supports fine-grained sharing of hardware resources between different security levels. For example, the design allows both high and low cache partitions to be securely used by a single program. This

Module Name	LOC
Fetch	60
Decode + Register File	465
Execute + ALU	218
FPU	N/A
Memory + Cache	537
Write Back	20
Control Logic + Forwarding + Stalling	419
Total w/o FPU	1719

Table 6.3: Lines of Code (LOC) for each processor component.

Instruction type	Instructions
Additive Arithmetic	add, addi, addiu, addu, sub, subu
Binary Arithmetic	and, or, xor, nor, srl, sra, sll, sllv srlv, srav, slt, sltu, slti, sltiu, andi, ori, xori
Multiply/divide	mult, multu, div, divu
Floating point	add.s, sub.s, mul.s, div.s, neg.s, abs.s mov.s, cvt.s.w, cvt.w.s, c.lt.s, c.le.s
Branch and jump	bne, beq, blez, bgtz, jr, jalr, j, jal
Memory operation	lw, lhu, lh, lbu, lb, sw, sh, sb, swc1, lwc1
Others	mfhi, mflo, lui, mtc1, mfc1 syscall, break
Security-related	setr, setw

Table 6.4: Complete ISA of our MIPS processor.

effectively increases the cache size and improves performance for applications with multiple security levels.

To implement such a rich policy, we divide a 4-way cache into a low partition and a high partition. When the timing label is H, both low and high partitions can be used securely. When the timing label is L, both the low and high partitions are still searched. However, to ensure that timing can be affected only by the low cache partition, a cache access is treated as a miss even when there is a hit in the high partition. To avoid the problem of data duplication, the cache line moves from the high partition to the low partition when the

data arrives from memory, achieving functional correctness without violating the timing constraint. Since cache states have static labels, they are not zeroed out when timing label changes.

The pipeline, on the other hand, is dynamically partitioned using the timing label. When the timing label changes, the pipeline is flushed to avoid leaking information. A pipeline that interleaves high and low instructions without flushing is indeed insecure, since high instructions may stall low ones.

We found that implementing such a complex policy securely would be difficult without using SecVerilog. For example, the SecVerilog type checker caught a security flaw not foreseen by us: the dirty bit copied from the high partition to the low partition created a potential timing channel. Our solution is to set the dirty bit for every cache line immediately after it is fetched. This change still allows store hits to write directly to cache line without writing to memory.

Another security issue caught by the SecVerilog type checker is a stall at the instruction fetch stage affecting the memory stage: in our pipeline implementation, a load miss in an instruction could stall instructions in later pipeline stages. Thus, when the timing label changes, an instruction with timing label H can stall another instruction with label L, breaking the timing-label contract. To make the design type-check, the pipeline is flushed at every timing-label switch.

6.3.2 Overhead of SecVerilog

SecVerilog may require designers to add additional branches to establish invariants needed to convince the type system of the security of the design. For

instance, the type system fails to infer in our cache design that variable `way` can only be 2 or 3 at a particular point in the code. In this case, the design needs to include an if-statement establishing this fact. These added branches represents the overhead of using SecVerilog.

To measure this overhead, we compare our secure MIPS processor written in SecVerilog ("Verified") with another secure design written in Verilog ("Unverified"). The Unverified design is essentially the same as Verified, including the same timing channel protections. Because it eliminates the if-statements necessary for type checking, it cannot be verified.

Designer effort and verification time The Unverified MIPS processor comprises 1692 lines of Verilog code. Converting this design to the Verified processor requires adding only 27 lines of extra code to the cache module, in order to establish necessary invariants to convince the type checker, suggesting that very little overhead is imposed by imprecision of the type system.

The current implementation of SecVerilog requires a programmer to explicitly write down one security label for each variable declaration, unless the variable has the default label L. However, most labels can be automatically inferred via adding type inference (e.g., as in [60, 17, 91]) to the SecVerilog compiler, which we leave as future work.

The verification process is fast. For our processor design, it takes a total of two seconds to both generate all obligations and then discharge them with Z3.

Delay, area and power We synthesized the processor designs using the Synopsys synthesis flow, using the 90nm `saed90nm_max` digital standard cell library.

For all designs, we increased the frequency of the processors to the maximum achievable to see what overhead the Verified design adds to the critical path. The synthesis results are shown in Table 6.5. Here "Insecure" represents the baseline, unmodified MIPS processor without timing channel protection. We discuss the baseline result in the next subsection.

The Verified design only adds 27 lines of code to Unverified, so we found that the delay, area, and power consumption of the two designs are almost identical. For example, area overhead is only 0.16% even without including cache SRAM, which is identical for all designs. This overhead is much lower than that of other secure design techniques, as reported in [48]: GLIFT, 660%; Caisson, 100%; and Sapper, 4%. Power consumption of the two designs is identical. Critical path delay is slightly lower for Verified, likely due to randomness in synthesis. The results show the benefit of sharing hardware across security levels and of controlling information flow at design time, without run-time checks.

Performance The Verified design does not add any performance overhead over the Unverified design because the added logic does not change cycle-by-cycle behavior.

6.3.3 Overhead of timing channel protection

The timing channel protection mechanisms in our processor ("Verified") adds overheads compared to the unmodified and unprotected baseline ("Baseline").

	Baseline	Unverified	Verified
Delay w/ FPU (ns)	4.20	4.20	4.20
Delay w/o FPU (ns)	1.64	1.67	1.66
Area (μm^2)	399400	401420	402079
Power (mW)	575.5	575.6	575.6

Table 6.5: Comparing processor designs.

Delay, area and power When an FPU is included, we found that the critical path delay is identical for both Verified and Baseline, as shown in Table 6.5. This is because the critical path of the processor lies in the FPU, which is largely unmodified for secure designs. To more meaningfully evaluate the impact of secure design, we also measured the maximum achievable frequency without an FPU. Nevertheless, the delay overhead is still only 1.22%. The area overhead of 0.67% is also quite low, and power overhead is almost negligible. Because SecVerilog allows hardware resources to be shared across security levels while properly restricting their allocations, timing channel protection mostly does not require duplicating or adding hardware.

Performance The timing channel protection in our secure processor design imposes restrictions on cache usage and results in additional pipeline flushes and cache write-backs. We measured the performance overhead of the Verified processor and tested its correctness on two security benchmarks.

Our benchmarks include three security programs (blowfish, rijndael, SHA-1) from MiBench, a popular embedded benchmark suite for architectural designs[1] [33], as well as ciphers and hash functions in a recent release (version 1.0.1g) of OpenSSL, a widely used open-source SSL library.

[1]The only benchmark omitted is PGP, which requires a full-featured OS.

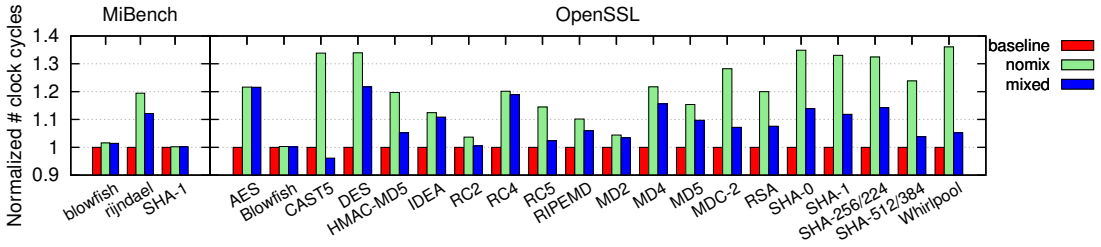


Figure 6.4: Performance overhead of timing channel protection.

Thanks to the rich ISA of our MIPS processor, compiling and running these benchmarks requires only modest effort. We use an off-the-shelf GCC compiler to cross-compile the benchmarks to the MIPS 1 platform. We use Cadence NCVerilog to simulate our processor design running these binaries. Because we lack an operating system on the processor, system calls (e.g., open, read, close, time) are emulated by Programming Language Interface (PLI) routines. Dynamic memory allocation is implemented by simple code using preallocated static memory.

Most test programs in these benchmarks were used as is. The only exceptions are a few tests in OpenSSL that take a long time to simulate. To make evaluation feasible on these tests, we replace long inputs with shorter ones.

We evaluate two security policies: "nomix", a coarse-grained policy where the entire program is labeled H, corresponding to the security policy targeted by previous secure hardware design methods, and "mixed", a fine-grained policy allowing mixed H and L instructions, enabled by the new features of SecVerilog. In the latter case, we use a simple policy to decide timing labels: for ciphers (e.g., AES, RSA), the encryption and decryption functions are marked as H; for secure hash functions (e.g., MD4, SHA512), we pretend part of the input is secret, and mark the hash functions on these inputs as H. Performance results for a single

run of each test are shown in Figure 6.4. Multiple runs are unnecessary for our evaluation since the simulation is deterministic.

From the MiBench suite, only rijndael shows noticeable performance overhead, at 19.6%. Overhead is reduced to 12.2% when the fine-grained model with mixed labels is used. Overhead on OpenSSL ranges from 0.3% (Blowfish) to 34.9% (SHA-0), with an average of 21.0%. For the fine-grained model, the overhead on OpenSSL ranges from −3.9% (CAST5) to 21.7% (DES), with an average of 8.8%. CAST5 runs faster with the partitioned cache because H instructions cannot evict frequently used data in the L partition.

The results clearly show the benefit of fine-grained information flow control within a single application. Most slowdown comes from the restriction that H instructions cannot write to the low cache partition. Allowing mixed H and L instructions in a single program improves performance because the restrictions only apply to a subset of program instructions.

We could not compare our performance overhead with prior work [84, 49, 48] because they do not report the overhead over a baseline design with unpartitioned cache[2].

[2]The previous method [48] calls a secure but unverified design "insecure", and reports the overhead of verified vs. unverified as we do in Section 6.3.2.

CHAPTER 7

CONCLUSIONS

This dissertation presents sound and practical methods for full-system timing channel control. The proposed approach consists of a new software-hardware timing contract, as well as control mechanisms present at both the software level and the hardware level.

Solving the timing channel problem requires work at both the hardware level and the software level. Neither level has enough information to allow accurate and complete reasoning about timing channels, because timing is a property that crosses abstraction boundaries. This dissertation introduces a new timing contract of read and write labels. The new contract makes it possible to control timing channels completely and effectively across abstraction boundaries.

At the software level, the timing contract provides just enough information for programming languages to accurately and completely control timing channels, assuming the underlying hardware obeys the contract. In particular, this dissertation proposes a novel type system that uses read and write labels to control timing channels. It is formally proved that any well-typed program has no timing channel leakage, assuming that the hardware obeys the timing contract.

At the hardware level, this dissertation introduces SecVerilog, a new hardware design language for statically verifying timing-channel-free hardware designs. SecVerilog makes it possible to design complex and efficient hardware where most resources are shared across security domains. This is enabled by novel features such as type-valued functions for dependent labels, the ability to soundly and precisely use mutable variables within labels, and the modular

incorporation of program analyses to improve precision. Moreover, SecVerilog comes with strong security assurance; we formally prove that all forms of information flow, including explicit flows, implicit flows, and flows via timing channels, are controlled in a well-typed hardware design.

For applications where entirely blocking timing channels is too restrictive, this dissertation proposes a general framework, called predictive mitigation, to improve the tradeoff between security and performance. Predictive mitigation offers the possibility of mitigating timing channels for general computations, while ensuring rigorous leakage bound: timing channel leakage is provably bounded by a programmer-specified function. Experiments show that predictive mitigation successfully defends against several published timing channel attacks, with an acceptable performance overhead. Moreover, this dissertation integrates predictive mitigation into the aforementioned restrictive software language which provably eliminates all timing channels. The result is a permissive programming model which improves the tradeoff between security and performance for many real-world applications.

Finally, we implement the proposed approach and apply it to real-world security-sensitive applications. Notably, using SecVerilog, we design and formally verify a reasonably complex MIPS processor which satisfies the proposed timing contract. Applications with read and write labels are run on the secure processor via a modified compiler. The results suggest that the mechanisms present at both the software level and the hardware level together control timing channels in these applications. Moreover, the verified processor has overheads of only about 1% in chip area, delay and power consumption. The application performance overhead is reasonable, with an average of about 20%.

BIBLIOGRAPHY

[1] Icarus Verilog. http://iverilog.icarus.com/.

[2] Onur Acıiçmez. Yet another microarchitectural attack: Exploiting I-cache. In *Proc. ACM Workshop on Computer Security Architecture (CSAW '07)*, pages 11–18, 2007.

[3] Onur Acıiçmez, Çetin K. Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. In *Proc. 2nd ACM Symposium on Information, Computer and Communications Security (ASIACCS'07)*, pages 312–320, 2007.

[4] Johan Agat. Transforming out timing leaks. In *Proc. 27th ACM Symp. on Principles of Programming Languages (POPL)*, pages 40–53, January 2000.

[5] Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proc. 13th European Symp. on Research in Computer Security (ESORICS)*, pages 333–348, October 2008.

[6] Aslan Askarov, Danfeng Zhang, and Andrew C. Myers. Predictive black-box mitigation of timing channels. In *Proc. 17th ACM Conf. on Computer and Communications Security (CCS)*, pages 297–307, October 2010.

[7] Lennart Augustsson. Cayenne—a language with dependent types. In *Proc. 3rd ACM SIGPLAN Int'l Conf. on Functional Programming (ICFP)*, pages 239–250, 1998.

[8] Thomas H. Austin and Cormac Flanagan. Efficient purely-dynamic information flow analysis. In *Proc. 4th ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 113–124, 2009.

[9] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison Wesley, April 2003. ISBN 0321136160.

[10] Gilles Barthe, Tamara Rezk, and Martijn Warnier. Preventing timing leaks through transactional branching instructions. *Electronic Notes in Theoretical Computer Science*, 153(2):33–55, 2006.

[11] Andrew Bortz and Dan Boneh. Exposing private information by timing web applications. In *Proc. 16th Int'l World-Wide Web Conf.*, May 2007.

[12] Randy Browne. Mode security: An infrastructure for covert channel suppression. In *IEEE Symposium on Research in Security and Privacy*, pages 39–55, May 1994.

[13] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, January 2005.

[14] Doug Burger and Todd M. Austin. The SimpleScalar tool set, version 3.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[15] David Chaum. Blind signatures for untraceable payments. In *CRYPTO*, pages 199–203, 1982.

[16] Michael R. Clarkson, Andrew C. Myers, and Fred B. Schneider. Quantifying information flow with beliefs. *Journal of Computer Security*, 17(5):655–701, October 2009.

[17] Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C. Necula. Dependent types for low-level programming. In *Proc. European Symposium on Programming (ESOP)*, pages 520–535, 2007.

[18] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *Proc. 30th IEEE Symp. on Security and Privacy (S&P)*, pages 45–60, 2009.

[19] Don Coppersmith. Small solutions to polynomial equations, and low exponent RSA vulnerabilities. *Journal of Cryptology*, 10(4), December 1997.

[20] Thomas M. Cover and Joy A. Thomas. *Elements of information theory*. Wiley, 2006.

[21] Dorothy E. Denning. *Cryptography and Data Security*. Addison-Wesley, Reading, Massachusetts, 1982.

[22] Dominique Devriese and Frank Piessens. Noninterference through secure multi-execution. In *Proc. 31st IEEE Symp. on Security and Privacy (S&P)*, pages 109–124, May 2010.

[23] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *CACM*, 18(8):453–457, August 1975.

[24] Jordan Dimitrov. Operational semantics for Verilog. In *Proc. 8th Asia-Pacific Software Engineering Conference*, pages 161–168, 2001.

[25] Robert W. Floyd. Assigning meanings to programs. In *Proc. Sympos. Appl. Math.*, volume XIX, pages 19–32, 1967.

[26] Robert G. Gallagher. Basic limits on protocol information in data communication networks. *IEEE Transactions on Information Theory*, 22(4), July 1976.

[27] James R. Giles and Bruce Hajek. An information-theoretic and game-theoretic study of timing channels. *IEEE Transactions on Information Theory*, 48(9):2455–2477, 2002.

[28] Joseph A. Goguen and Jose Meseguer. Security policies and security models. In *Proc. IEEE Symp. on Security and Privacy*, pages 11–20, April 1982.

[29] David M. Goldschlag. Several secure store and forward devices. In *ACM Conf. on Computer and Communications Security (CCS)*, pages 129–137, March 1996.

[30] Mike Gordon. The semantic challenge of Verilog HDL. In *Proc. Logic in Computer Science*, pages 136–145, 1995.

[31] Robert Grabowski and Lennart Beringer. Noninterference with dynamic security domains and policies. In *Advances in Computer Science – ASIAN 2009. Information Security and Privacy*, pages 54–68, 2009. LNCS 5913.

[32] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games—bringing access-based cache attacks on AES to practice. In *Proc. IEEE Symp. on Security and Privacy (S&P)*, pages 490–505, 2011.

[33] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proc. IEEE International Workshop on Workload Characterization (WWC)*, pages 3–14, 2001.

[34] Daniel Hedin and David Sands. Timing aware information flow security for a JavaCard-like bytecode. *Electronic Notes in Theoretical Computer Science*, 141(1):163–182, 2005.

[35] C. A. R. Hoare. An axiomatic basis for computer programming. *CACM*, 12(10):576–580, October 1969.

[36] Wei-Ming Hu. Reducing timing channels with fuzzy time. In *Proc. IEEE Symp. on Security and Privacy (S&P)*, pages 8 – 20, 1991.

[37] Marieke Huisman, Pratik Worah, and Kim Sunesen. A temporal logic characterisation of observational determinism. In *Proc. 19th IEEE Computer Security Foundations Workshop*, 2006.

[38] Sebastian Hunt and David Sands. On flow-sensitive security types. In *Proc. 33rd ACM Symp. on Principles of Programming Languages (POPL)*, pages 79–90, 2006.

[39] Limin Jia, Jeffrey A. Vaughan, Karl Mazurak, Jianzhou Zhao, Luke Zarko, Joseph Schorr, and Steve Zdancewic. Aura: A programming language for authorization and audit. In *Proc. 13th ACM SIGPLAN Int'l Conf. on Functional Programming (ICFP)*, pages 27–38, 2008.

[40] Myong H. Kang and Ira S. Moskowitz. A pump for rapid, reliable, secure communication. In *Proc. ACM Conf. on Computer and Communications Security (CCS)*, pages 119–129, 1993.

[41] Myong H. Kang, Ira S. Moskowitz, and Daniel C. Lee. A network pump. *IEEE Transactions on Software Engineering*, 22:329–338, 1996.

[42] Vineeth Kashyap, Ben Wiedermann, and Ben Hardekopf. Timing- and termination-sensitive secure information flow: Exploring a new approach. In *Proc. IEEE Symp. on Security and Privacy (S&P)*, pages 413–430, May 2011.

[43] Paul C. Kocher. Timing attacks on implementations of Diffie–Hellman, RSA, DSS, and other systems. In *Advances in Cryptology—CRYPTO'96*, August 1996.

[44] Jingfei Kong, Onur Acıiçmez, Jean-Pierre Seifert, and Huiyang Zhou. Deconstructing new cache designs for thwarting software cache-based side channel attacks. In *Proc. 2nd ACM Workshop on Computer Security Architectures*, pages 25–34, 2008.

[45] Boris Köpf and Markus Dürmuth. A provably secure and efficient countermeasure against timing attacks. In *Proc. IEEE Computer Security Foundations (CSF)*, pages 324–335, July 2009.

[46] Boris Köpf and Geoffrey Smith. Vulnerability bounds and leakage re-

silience of blinded cryptography under timing attacks. In *Proc. IEEE Computer Security Foundations (CSF)*, pages 44–56, July 2010.

[47] Butler W. Lampson. A note on the confinement problem. *Comm. of the ACM*, 16(10):613–615, October 1973.

[48] Xun Li, Vineeth Kashyap, Jason K. Oberg, Mohit Tiwari, Vasanth Ram Rajarathinam, Ryan Kastner, Timothy Sherwood, Ben Hardekopf, and Frederic T. Chong. Sapper: A language for hardware-level security policy enforcement. In *Proc. 19th Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 97–112, 2014.

[49] Xun Li, Mohit Tiwari, Jason K. Oberg, Vineeth Kashyap, Frederic T. Chong, Timothy Sherwood, and Ben Hardekopf. Caisson: a hardware description language for secure information flow. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 109–120, 2011.

[50] Fangfei Liu and Ruby B. Lee. Random fill cache architecture. In *Proc. 47th Annual IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, pages 203–215, 2014.

[51] Yali Liu, Dipak Ghosal, Frederik Armknecht, Ahmad-Reza Sadeghi, Steffen Schulz, and Stefan Katzenbeisser. Hide and seek in time—robust covert timing channels. In *Proc. European Symp. on Research in Computer Security (ESORICS)*, pages 120–135, 2009.

[52] Luísa Lourenço and Luís Caires. Dependent information flow types. *FCT/UNL Technical Report*, October 2013.

[53] Gavin Lowe. Quantifying information flow. In *Proc. IEEE Computer Security Foundations Workshop (CSFW)*, pages 18–31, June 2002.

[54] Jonathan K. Millen. Covert channel capacity. In *Proc. IEEE Symp. on Security and Privacy*, Oakland, CA, April 1987.

[55] Jonathan K. Millen. Finite-state noiseless covert channels. In *Proc. 2nd IEEE Computer Security Foundations Workshop*, pages 11–14, June 1989.

[56] 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 *Proc. 8th International Conference on Information Security and Cryptology*, pages 156–168, 2006.

[57] Ira S. Moskowitz and Allen R. Miller. The channel capacity of a certain noisy timing channel. *IEEE Trans. on Information Theory*, 38(4):1339–1344.

[58] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proc. Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[59] Andrew C. Myers. JFlow: Practical mostly-static information flow control. In *Proc. 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 228–241, January 1999.

[60] Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. Jif 3.0: Java information flow. Software release, `http://www.cs.cornell.edu/jif`, July 2006.

[61] Aleksandar Nanevski, Anindya Banerjee, and Deepak Garg. Verification of information flow and access control policies with dependent types. In *Proc. IEEE Symp. on Security and Privacy*, pages 165–179, 2011.

[62] Jason Oberg, Wei Hu, Ali Irturk, Mohit Tiwari, Timothy Sherwood, and Ryan Kastner. Theoretical analysis of gate level information flow tracking. In *Proc. 47th Design Automation Conference*, pages 244–247, 2010.

[63] Jason Oberg, Wei Hu, Ali Irturk, Mohit Tiwari, Timothy Sherwood, and Ryan Kastner. Information flow isolation in I2C and USB. In *Proc. 48th Design Automation Conference*, pages 254–259, 2011.

[64] Michael A. Olson, Keith Bostic, and Margo Seltzer. Berkeley DB. In *Proc. USENIX Annual Technical Conference*, 1999.

[65] Dag A. Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. *Topics in Cryptology–CT-RSA 2006*, pages 1–20, January 2006.

[66] M. A. Padlipsky and D. W. Snow. Limitations of end-to-end encryption in secure computer networks. Technical Report ESD TR-78-158, Mitre Corp., 1978.

[67] Dan Page. Partitioned cache architecture as a side-channel defense mechanism. In *Cryptology ePrint Archive, Report 2005/280*, 2005.

[68] Colin Percival. Cache missing for fun and profit. In *BSDCan*, 2005.

[69] Alessandra Di Pierro, Chris Hankin, and Herbert Wiklicky. Quantifying timing leaks and cost optimisation. *Information and Communications Security*, pages 81–96, 2010.

[70] A. W. Roscoe. CSP and determinism in security modelling. In *Proc. IEEE Symp. on Security and Privacy*, pages 114–127, May 1995.

[71] Alejandro Russo and Andrei Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proc. 23rd IEEE Computer Security Foundations (CSF)*, CSF '10, pages 186–199, 2010.

[72] Andrei Sabelfeld and Andrew C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[73] Andrei Sabelfeld and David Sands. Probabilistic noninterference for multi-threaded programs. In *Proc. 13th IEEE Computer Security Foundations Workshop*, pages 200–214. IEEE Computer Society Press, July 2000.

[74] Gaurav Shah, Andres Molina, and Matt Blaze. Keyboards and covert channels. *Proc. 15th USENIX Security Symp.*, August 2006.

[75] Vincent Simonet. The Flow Caml System: documentation and user's manual. Technical Report 0282, Institut National de Recherche en Informatique et en Automatique (INRIA), July 2003.

[76] Geoffrey Smith. A new type system for secure information flow. In *Proc. IEEE Computer Security Foundations Workshop (CSFW)*, pages 115–125, June 2001.

[77] Geoffrey Smith. On the foundations of quantitative information flow. *Foundations of Software Science and Computational Structures*, 5504:288–302, 2009.

[78] Geoffrey Smith and Dennis Volpano. Secure information flow in a multi-threaded imperative language. In *Proc. 25th ACM Symp. on Principles of Programming Languages (POPL)*, pages 355–364, January 1998.

[79] Nikhil Swamy, Juan Chen, and Ravi Chugh. Enforcing stateful authorization and information flow policies in Fine. In *Proc. European Symposium on Programming (ESOP)*, pages 529–549, 2010.

[80] Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan

Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. In *Proc. 16th ACM SIGPLAN Int'l Conf. on Functional Programming (ICFP)*, pages 266–278, 2011.

[81] Nikhil Swamy, Brian J. Corcoran, and Michael Hicks. Fable: A language for enforcing user-defined security policies. In *Proc. IEEE Symp. on Security and Privacy (S&P)*, pages 369–383, 2008.

[82] Mohit Tiwari, Xun Li, Hassan M. G. Wassel, Frederic T. Chong, and Timothy Sherwood. Execution leases: A hardware-supported mechanism for enforcing strong non-interference. In *Proc. Annual IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, December 2009.

[83] Mohit Tiwari, Jason Oberg, Xun Li, Jonathan K. Valamehr, Timothy Levin, Ben Hardekopf, Ryan Kastner, Frederic T. Chong, and Timothy Sherwood. Crafting a usable microkernel, processor, and I/O system with strict and provable information flow security. In *Proc. Annual International Symp. on Computer Architecture (ISCA)*, pages 189–200, June 2011.

[84] Mohit Tiwari, Hassan M.G. Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T. Chong, and Timothy Sherwood. Complete information flow tracking from the gates up. In *Proc. Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 109–120, 2009.

[85] Stephen Tse and Steve Zdancewic. Run-time principals in information-flow type systems. *ACM Trans. on Programming Languages and Systems*, 30(1):6, 2007.

[86] Dennis Volpano and Geoffrey Smith. Eliminating covert flows with minimum typings. In *Proc. 10th IEEE Computer Security Foundations Workshop*, pages 156–168, 1997.

[87] Zhenghong Wang and Ruby B. Lee. Covert and side channels due to processor architecture. In *Proc. Annual Computer Security Applications Conference (ACSAC)*, pages 473–482, 2006.

[88] Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proc. Annual International Symp. on Computer Architecture (ISCA)*, pages 494–505, 2007.

[89] Zhenghong Wang and Ruby B. Lee. A novel cache architecture with en-

hanced performance and security. In *Proc. 41st Annual IEEE/ACM Int'l Symp. on Microarchitecture (MICRO)*, pages 83–93, 2008.

[90] John C. Wray. An analysis of covert timing channels. In *Proc. IEEE Symp. on Security and Privacy (S&P)*, pages 2–7, 1991.

[91] Hongwei Xi. Imperative programming with dependent types. In *Proc. IEEE Symposium on Logic in Computer Science*, pages 375–387, 2000.

[92] Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proc. ACM Symp. on Principles of Programming Languages (POPL)*, pages 214–227, 1999.

[93] Steve Zdancewic and Andrew C. Myers. Observational determinism for concurrent program security. In *Proc. 16th IEEE Computer Security Foundations Workshop*, pages 29–43, June 2003.

[94] Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Predictive mitigation of timing channels in interactive systems. In *Proc. 18th ACM Conf. on Computer and Communications Security (CCS)*, pages 563—574, October 2011.

[95] Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. Language-based control and mitigation of timing channels. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, pages 99–110, June 2012.

[96] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. A hardware design language for timing-sensitive information-flow security. In *Proc. 20th Int'l Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 503–516, 2015.

[97] Lantian Zheng and Andrew C. Myers. Dynamic security labels and static information flow control. *Intl' J. of Information Security*, 6(2–3), March 2007.