Language-Based Control and Mitigation of Timing Channels

Danfeng Zhang
Department of Computer Science
Cornell University
Ithaca, NY 14853
zhangdf@cs.cornell.edu

Aslan Askarov *
School of Engineering and Computer Science, Harvard University
Cambridge, MA 02138
aslan@seas.harvard.edu

Andrew C. Myers
Department of Computer Science
Cornell University
Ithaca, NY 14853
andru@cs.cornell.edu

Abstract

We propose a new language-based approach to mitigating timing channels. In this language, well-typed programs provably leak only a bounded amount of information over time through external timing channels. By incorporating mechanisms for predictive mitigation of timing channels, this approach also permits a more expressive programming model. Timing channels arising from interaction with underlying hardware features such as instruction caches are controlled. Assumptions about the underlying hardware are explicitly formalized, supporting the design of hardware that efficiently controls timing channels. One such hardware design is modeled and used to show that timing channels can be controlled in some simple programs of real-world significance.

Categories and Subject Descriptors D.2.4 [*Software/Program Verification*]: Formal Methods; D.4.6 [*Security and protection*]: Information Flow Controls

General Terms Languages, Security

Keywords Timing channels, mitigation, information flow

1. Introduction

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 [8, 13, 14, 18, 22, 24, 29, 36].

Timing channels can be categorized as internal or external [28]. *Internal timing channels* exist when timing channels are converted to storage channels within a system and affect the results computed. *External timing channels* exist when the adversary can learn something from the time at which the system interacts with the outside world. In either case, confidential information transmitted through timing channels constitutes *timing leakage*.

Internal timing channels that exploit races between threads have been addressed by enforcing low determinism [16, 37] and by constraining thread scheduling [26, 28]. The focus of this paper is instead on controlling *external* timing channels, for which current methods are less satisfactory. Starting with Agat [3], program transformations have been proposed to remove external timing channels. However, these methods restrict expressiveness: for example, loop guards can depend only on public information. Further, they do not handle some realistic hardware features. External mitigation is another approach to control external timing channels, by quantitatively limiting how much information leaks via the timing of external interactions [5, 20, 38]. Since external mitigation treats computation as a black box, it cannot distinguish between benign timing variations and variations that leak information. When most timing variation is benign, this leads to a significant performance penalty.

This work introduces a more complete and effective language-based method for controlling external timing channels, with provably bounded leakage. Broadly, the new method improves control of external timing channels in three ways:

- Unlike methods based on code transformation [3], this method supports more realistic programs and hardware. For example, it can be implemented on hardware with an instruction cache.
- Another difference from code-transformation approaches is that it offers a fully expressive programming model; in particular, loops with high (confidential) guards are permitted.
- The method does not need to be as conservative as external timing mitigation because a program analysis 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.

Timing channels arise in general from the interaction of programs with the underlying implementation of the language in which the programs are written. This language implementation includes not only the compiler or interpreter used, but also the underlying hardware. Reasoning accurately about timing channels purely at the source language level is essentially impossible because language semantics, by design, prevent precise reasoning about time.

An important contribution of this paper is therefore 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 between the software (language) level and the hardware implementation.

A second contribution of this paper is a formalization of this contract. Using this formal contract, implementers may verify that their compiler and architecture designs control timing channels. We illustrate this by sketching the design of a simple memory cache architecture that avoids timing channels.

A third contribution is a new language mechanism that improves expressive power achievable while controlling timing channels. It uses *predictive timing mitigation* [5] to bound the amount of information that leaks through timing. With this mechanism, algorithms whose timing behavior *does* depend on confidential information can be implemented securely; predictive mitigation ensures that total timing leakage is bounded by a programmer-specified function.

* This work was done while the author was at Cornell University.

PLDI'12, June 11–16, 2012, Beijing, China.

To evaluate the correctness and effectiveness of our approach, we simulated hardware satisfying the hardware side of the software–hardware contract. We evaluate the use of our approach on two applications vulnerable to timing attacks. The results suggest that the combination of language-based mitigation and secure hardware works well, with only modest slowdown.

We proceed as follows. Section 2 discusses the problem of controlling timing channels on modern computer hardware, and gives an overview of the new method. Section 3 introduces a programming language designed to permit precise reasoning about timing channels. Its semantics formalize several constraints that must be satisfied by a secure implementation. Section 4 sketches how these constraints can be satisfied by both stock and specialized hardware implementations. A type system for the language that soundly controls timing channels is presented in Section 5; its novel multilevel quantitative security guarantees are explored in Section 6. Predictive mitigation of timing channels is discussed in Section 7. Section 8 presents performance results from a simulated implementation of language-based predictive mitigation. Related work is covered in Section 9, and Section 10 concludes.

2. Language-level timing mitigation

Controlling timing channels is difficult because confidential information can affect timing in many ways, and yet we want to be able to analyze these timing dependencies at the source level. However, language semantics do not and should not define timing precisely.

2.1 Timing dependencies

We call timing channels visible at the source-language level *direct timing dependencies*. In this 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. Attacks on RSA have also used control-flow-related timing channels [8, 18]. Another source of direct timing dependencies is operations whose execution time depends on parameter values, such as the `sleep` command at line 5.

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 [14, 24] also 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 [1], branch predictors and branch target buffers [2], and shared functional units [34].

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 language level—the semantics of languages and even of instruction set architectures (ISAs) hide information about execution time by abstracting away low-level implementation details. For instance, we do not know that the timing of line 5 depends on `h1` without knowing how the data cache works in the example above.

It is worth noting that we assume a strong adversary that is particularly interesting with the rise of cloud computing: an adversary coresident on the system, controlling concurrent threads that can read low memory locations. The adversary can therefore time when low memory locations change. Further, the adversary can probe timing using the shared cache. This is a more powerful adversary than that considered in much previous work on timing channels, including prior attempts to control decryption side channels [5, 20, 38]. The prior methods are not effective against this adversary, who can efficiently learn secret keys using timing side channels [24].

2.2 Representing indirect timing dependencies abstractly

Recent work in the architecture community has aimed for a hardware-based solution to timing channels. Their hardware designs implicitly rely on assumptions about how software uses the hardware, but these assumptions have not been rigorously defined. For example, the cache design by Wang and Lee [35] works only under the assumption that the AES lookup table is preloaded into cache and that the load time is not observable to the adversary [19].

Timing channels cannot be controlled effectively purely at the source code or the hardware level. Hardware mechanisms can help, but do a poor job of controlling language-level leaks such as implicit flows. 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 approaches, by representing the machine environment abstractly at source level.

As is standard in information flow type systems [27], we associate all information with a *security label* that in this case describes 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. We assume there are at least two distinct labels L (low) and H (high) such that $L \sqsubseteq H \not\sqsubseteq L$. The label of public information is L; that of secret is H. As is standard, we denote by $\top$ the most restrictive label, and by $\bot$, the least restrictive one.

We assume that different components of the machine environment have security labels as well. For example, different partitions in a partitioned cache [35] can be associated with different labels.

To track how information flows into the machine environment, but without concretely representing the hardware state, we associate 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 $\ell_r = L$ 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. It ensures that the labels of hardware state reflect the confidentiality of information that has flowed into that state.

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 previous example 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]
```

The example on the left 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 [11] 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* [11], 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 cache is to deactivate it in high contexts. A generalization of this idea is to partition the cache into two partitions, low and high. Cache misses in a high context then cause only the high cache partition to be updated.

With ℓ_r and ℓ_w abstracting the timing behavior of hardware, timing channel security can be statically checked at the language level, according to the type system described in Sec. 5. Moreover, these timing labels could be inferred automatically according to the type system, reducing the burden on programmers.

2.3 Language-level mitigation of timing channels

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

To increase expressiveness, we introduce a new command `mitigate` to the language. Command `mitigate` (e,ℓ) c executes the command c while ensuring that timing leakage is bounded. The expression e computes an initial prediction for the execution time of c. The label ℓ bounds what information that can 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. This property is enforced by the type system of Sec. 5. Moreover, the type system ensures that timing leakage can be bounded using the variation in the execution time of `mitigate` commands.

To provide a strict bound on the execution time of `mitigate` commands while providing practical performance, we introduce the use of predictive timing mitigation [5, 38] as a language-level mechanism. The idea is that given a prediction of how long executing c will take (the e in `mitigate` command), the `mitigate` command ensures that at least that much time is consumed by simply waiting if necessary. In the case of a misprediction (that is, 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 uses of `mitigate`.

For example, we can use `mitigate` to limit timing leakage from the command `sleep(h)`, as in this program:

```
1   mitigate(1,H){sleep(h)[H,H]}
```

The possible execution times of this program will not be arbitrary; they might, for example, be forced by `mitigate` to be the powers of 2. Limiting the possible execution times bounds the timing leakage from `sleep`. We explore the details of the mitigation mechanism more fully in Sec. 7 and evaluate its performance in Sec. 8.

Previous work has shown that predictive timing mitigation can bound timing leakage to a function that is sublinear (in fact, polylogarithmic) in time. But this is the first work that provides similar, quantitative bounds on timing leakage at the language level.

3. A language for controlling timing channels

Fig. 1 gives the syntax for a simple imperative language extended with our mechanism. All the novel elements—read and write labels, and the `mitigate` and `sleep` commands—have already been introduced. Notice that the sequential composition command itself needs no timing labels. As a technical convenience, each `mitigate` in the source has a unique identifier η. These identifiers are mainly used in Sec. 6; they are omitted where they are not essential.

We present our semantics in a series of modular steps. We start with a *core semantics*, a largely standard semantics for a 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.

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

Figure 1: Syntax of the language

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

$$\langle (\texttt{mitigate } (e,\ell) \ c)_{[\ell_r,\ell_w]}, m\rangle \to \langle c, m\rangle$$

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

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

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

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

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

Figure 2: Core semantics of commands (unmitigated)

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 (Fig. 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. For `mitigate` we give an identity semantics for now: `mitigate` (e,ℓ) c simply evaluates to c. Since time is not part of the core semantics, `sleep` behaves like `skip`.

3.2 Abstracted full language semantics

The core semantics 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 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 (Sec. 3.5) are straightforward; the security requirements (Sec. 3.6) are more subtle.

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.

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 cache 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. Sec. 4.1).

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 associate 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. Recall that we are defending against a strong, coresident adversary. Therefore, 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:

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

3.5 Faithfulness requirements for the full semantics

The faithfulness requirements for the full semantics comprise four properties: adequacy, deterministic execution, sequential composition, and 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). $\forall m, c, c', E, G$.
$(\exists E', G' \,.\, \langle c, m, E, G\rangle \rightarrow \langle c', m', E', G'\rangle) \Leftrightarrow \langle c, m\rangle \rightarrow \langle c', m'\rangle$

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). $\forall m, c, E, G$.

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

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 because it is used for timing mitigation. 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.

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

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

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* $c_{[\ell_r,\ell_w]}$,

$$\begin{aligned}\forall m_1,m_2,E_1,E_2,G\,.\,&(\forall x \in vars_1(c_{[\ell_r,\ell_w]})\,.\,m_1(x)=m_2(x))\\ &\wedge E_1 \sim_{\ell_r} E_2\\ &\wedge \langle c_{[\ell_r,\ell_w]},m_1,E_1,G\rangle \to \langle c_1,m_1',E_1',G_1\rangle\\ &\wedge \langle c_{[\ell_r,\ell_w]},m_2,E_2,G\rangle \to \langle c_2,m_2',E_2',G_2\rangle \implies G_1=G_2\end{aligned}$$

PROPERTY 7 (Single-step machine-environment noninterference). *Given any labeled command* $c_{[\ell_r,\ell_w]}$, *and any level* ℓ,

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

Figure 3: Security requirements

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`, `while`, or `mitigate`, this time does not include time spent in subcommands.

Property 6 in Fig. 3 formalizes the requirement that read labels accurately capture the influences of the machine environment. This formalization uses the $vars_1$ 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 $vars_1$ because parts of the memory can also affect timing, such as e in `sleep` (e). A simple syntactic definition of $vars_1$ 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, `while` e `do` c, and `mitigate` (e,ℓ) 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 Fig. 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` $(h)_{[L,\ell_w]}$ with read-label $\ell_r = L$, 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 $L \sqsubseteq M \sqsubseteq H$, and a command $(x := y)_{[M,M]}$, 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 Fig. 3. Note that here level ℓ is independent of read or write labels.

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. For simplicity, we assume 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 [35].

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 always take the same time in equal environments, and no more precise. Including state that has 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. 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'`. 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 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 it. Yet this insight is important for performance: otherwise, code with a low timing label cannot access high variables using cache.

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 $\ell_w = H$, 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 $\ell_w = L$, this property is vacuously true since there is no ℓ such that $L \not\sqsubseteq \ell$. Commands with $\ell_w = H$ 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 $\forall x \in vars_1(c_{[\ell_r,\ell_w]}).m_1(x) = m_2(x)$ 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 $\texttt{sleep}\ (e)_{[\ell_r,\ell_w]}$. When $\ell_w = H$, 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.

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. As discussed in Sec. 8, we have implemented a simulation of this design; here we focus on the correctness of hardware design.

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 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 $L \not\sqsubseteq \ell$. When the write label is H, a new entry is installed only in the H partition, so $E \sim_L E'$.

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, $E_1 \sim_H E_2$ 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. One realization exists in Caisson [21], which enforces a version of noninterference that is both memory and timing-sensitive. Property 7 requires only timing-*insensitive* noninterference, so Caisson arguably tackles an unnecessarily difficult problem. A similar implementation that satisfies Property 7 more exactly might be more efficient.

5. A type system for controlling timing channels

Next, we present the security type system for our language. This section focuses on the non-quantitative guarantees that the type system provides, assuming Properties 1–7 hold. We show that the type system isolates the places where timing needs to be controlled externally. These places are where `mitigate` commands are needed.

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 [27] and we omit them here. Typing rules for commands, in Fig. 4, have the form $\Gamma, pc, \tau \vdash c : \tau'$. Here pc is the usual program-counter label [27], 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. We write $\Gamma \vdash c$ to denote $\Gamma, \bot, \bot \vdash c : \tau'$ for some τ'.

All rules enforce the constraint $\tau \sqsubseteq \tau'$ because timing dependencies accumulate as the program executes. Every rule except (T-MTG) 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.

The most interesting rule is (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.

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.

5.2 Machine-environment noninterference

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

THEOREM 1 (Memory and machine-environment noninterference).

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

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

$$\frac{\Gamma \vdash e : \ell \qquad pc \sqsubseteq \ell_w \quad \Gamma, \ell \sqcup pc, \ell \sqcup \tau \sqcup \ell_r \vdash c_i : \tau_i \quad i = 1,2}{\Gamma, pc, \tau \vdash (\texttt{if}\ e\ \texttt{then}\ c_1\ \texttt{else}\ c_2)_{[\ell_r,\ell_w]} : \tau_1 \sqcup \tau_2}\ \text{T-IF} \qquad \frac{\Gamma \vdash e : \ell \quad pc \sqsubseteq \ell_w \quad \ell \sqcup \tau \sqcup \ell_r \sqsubseteq \tau' \quad \Gamma, \ell \sqcup pc, \tau' \vdash c : \tau'}{\Gamma, pc, \tau \vdash (\texttt{while}\ e\ \texttt{do}\ c)_{[\ell_r,\ell_w]} : \tau'}\ \text{T-WHILE}$$

$$\frac{\Gamma, pc, \tau \vdash c_1 : \tau_1 \qquad \Gamma, pc, \tau_1 \vdash c_2 : \tau_2}{\Gamma, pc, \tau \vdash c_1; c_2 : \tau_2}\ \text{T-SEQ} \qquad \frac{\Gamma \vdash e : \ell \qquad pc \sqsubseteq \ell_w \qquad \Gamma, pc, \tau \sqcup \ell \sqcup \ell_r \vdash c : \tau' \quad \tau' \sqsubseteq \ell'}{\Gamma, pc, \tau \vdash (\texttt{mitigate}\ (e, \ell')\ c)_{[\ell_r,\ell_w]} : \ell \sqcup \tau \sqcup \ell_r}\ \text{T-MTG}$$

Figure 4: Typing rules: commands

Theorem 1 guarantees the adversary obtains no information by observing public parts of the memory and machine environments. Any confidential information the adversary obtains must be via timing. The proof is provided in the corresponding technical report [39].

Theorem 1 does *not* guarantee that information is not leaked through timing, that is, by observation of G. However, such a guarantee holds if the program contains no `mitigate` commands. This stronger guarantee is not proved here because it is a corollary of more general results presented in the next section.

A note on termination. The definition of memory and machine noninterference in Theorem 1 is presented in the batch-style termination-insensitive form [4]. 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.

6. Quantitative properties of the type system

The type system identifies potential timing channels in the program. We now introduce a quantitative measure of leakage for multilevel systems, and show that the type system 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.

6.1 Adversary observations

As discussed earlier in Sec. 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 $\alpha \in \{(x,v,t), \varepsilon\}$ 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: $\langle c, m, E, G\rangle \Rrightarrow_H (l_1, v_1, t_1), (h_1, v_2, t_2)$ for some v_1, t_1, v_2 and t_2. For the L-adversary, we have $\langle c, m, E, G\rangle \Rrightarrow_L (l_1, v_1, t_1)$, which does not include the assignment to h_1.

6.2 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 $\mathscr{L}$ 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 $\mathscr{L}_{\ell_A}$ be the subset of $\mathscr{L}$ that excludes all levels observable to ℓ_A, that is $\mathscr{L}_{\ell_A} \triangleq \{\ell' \mid \ell' \in \mathscr{L} \wedge \ell' \not\sqsubseteq \ell_A\}$. For example, for a three-level lattice $L \sqsubseteq M \sqsubseteq H$, with $\ell_A = M$, if $\mathscr{L} = \{M, H\}$ then $\mathscr{L}_{\ell_A} = \{H\}$.

Fig. 5a 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 $\mathscr{L}$ is represented by the dashed rectangle (though in general this set does not have to be contiguous). The gray area corresponds to the security levels that are in $\mathscr{L}_{\ell_A}$.

Leakage from $\mathscr{L}$ 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 [38], this measure bounds those of Shannon entropy and min-entropy, used in the literature [11, 22, 31].

DEFINITION 1 (Quantitative leakage from $\mathscr{L}$ to ℓ_A). *Given any ℓ_A, m, and E, the leakage of program c from levels $\mathscr{L}$ to level ℓ_A, denoted by $\mathsf{Q}(\mathscr{L}, \ell_A, c, m, E)$ is defined as follows*

$$\mathsf{Q}(\mathscr{L}, \ell_A, c, m, E) \triangleq \log(|\ \{(\mathbf{x}, \mathbf{v}, \mathbf{t})\ \mid \exists m', E'\ .\ (\forall \ell'\ .\ \ell' \not\in \mathscr{L}_{\ell_A}\ .\ m \simeq_{\ell'} m' \wedge E \simeq_{\ell'} E') \wedge \langle c, m', E', 0\rangle \Rrightarrow_{\ell_A} (\mathbf{x}, \mathbf{v}, \mathbf{t})\}\ |)$$

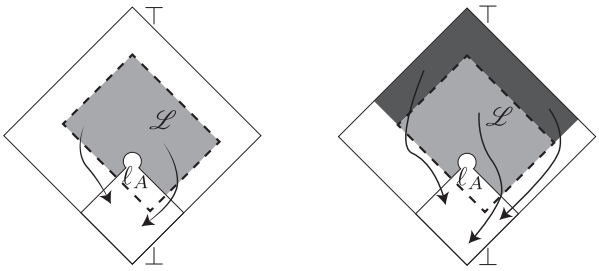
This definition uses $\mathscr{L}_{\ell_A}$ to restrict the quantification of the memory and machine environments so that we allow variations only in $\mathscr{L}_{\ell_A}$ parts of memory and machine environments. This is expressed by requiring projected equivalence (on the second line of the definition) for all levels ℓ' not in $\mathscr{L}_{\ell_A}$. Visually, using Fig. 5a, 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.

6.3 Guarantees of the type system

The type system provides an important property: leakage from $\mathscr{L}$ to ℓ_A is bounded by the timing variation of the `mitigate` commands whose mitigation level ℓ' is in the *upward closure* of $\mathscr{L}_{\ell_A}$.

Upward closure. In order to correctly approximate leakage from levels in $\mathscr{L}_{\ell_A}$, we need to account for all levels that are as restrictive as the ones in $\mathscr{L}_{\ell_A}$. For example, in a three-level lattice $L \sqsubseteq M \sqsubseteq H$, let $\mathscr{L}$ be the set $\{M\}$, and let $\ell_A = L$; then $\mathscr{L}_{\ell_A} = \{M\}$. 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 $\mathscr{L}_{\ell_A}$, written as



(a) Leakage from $\mathscr{L}$ to ℓ_A *(b)* Variations with $\mathscr{L}^{\sqsubseteq}_{\ell_A}$

Figure 5: Quantitative leakage

$\mathscr{L}_{\ell_A}\uparrow \triangleq \{\ell' \mid \exists \ell \in \mathscr{L}_{\ell_A} \wedge \ell \sqsubseteq \ell'\}$. In this example, $\mathscr{L}_{\ell_A}\uparrow = \{M, H\}$. Fig. 5b illustrates the relationship between $\mathscr{L}_{\ell_A}$ and its upper closure, where $\mathscr{L}_{\ell_A}\uparrow$ includes both shades of gray.

Trace and projection of mitigate commands. Next, we focus on the amount of time a `mitigate` command takes to execute. Recall from Sec. 3 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, 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 predicate f.

Low-determinism of `mitigate` commands. Consider the following well-typed program that uses `mitigate` twice.

```
1  mitigate_1(1,H) {
2    if (high)
3    then mitigate_2(1, H) { h:=h+1 }
4    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 $pc(M_1) = L$, and $pc(M_2) = H$. 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 $\mathscr{L}$, an adversary level ℓ_A, and a vector $(\mathbf{M}, \mathbf{t})$, we filter high `mitigate` commands by the projection $(\mathbf{M}, \mathbf{t}) \upharpoonright^{pc(M_\eta) \notin \mathscr{L}_{\ell_A}\uparrow}$. This projection consists of all the `mitigate` commands whose pc-label is in the white area in Fig. 5b.

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 Sec. 7.

LEMMA 1. *(Low-determinism of* `mitigate` *commands). For all programs c such that $\Gamma \vdash c$, adversary levels ℓ_A, sets of security levels $\mathscr{L}$, and memories and environments E_1, E_2, m_1, m_2 such that $(\forall \ell' \notin \mathscr{L}_{\ell_A}\uparrow \,.\, E_1 \simeq_{\ell'} E_2 \wedge m_1 \simeq_{\ell'} m_2)$, we have*

$$\langle c, m_1, E_1, 0\rangle \Rrightarrow (\mathbf{M_1}, \mathbf{t_1}) \wedge \langle c, m_2, E_2, 0\rangle \Rrightarrow (\mathbf{M_2}, \mathbf{t_2}) \implies \mathbf{M_1} \upharpoonright^{pc(M_\eta) \notin \mathscr{L}_{\ell_A}\uparrow} = \mathbf{M_2} \upharpoonright^{pc(M_\eta) \notin \mathscr{L}_{\ell_A}\uparrow}$$

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 the corresponding technical report [39].

Mitigation levels. Per Sec. 3, 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 $\mathscr{L}$ can be leaked by `mitigate` commands such that $lev(M_\eta) \in \mathscr{L}_{\ell_A}\uparrow$.

This leads to the definition of *timing variations*.

DEFINITION 2 (Timing variations of `mitigate` commands). *Given a set of security levels $\mathscr{L}$, an adversary level ℓ_A, program c, memory m, and a machine environment E, let V be the* timing variations *of* `mitigate` *commands:*

$$\mathsf{V}(\mathscr{L}, \ell_A, c, m, E) \triangleq \{\mathbf{t}' \mid \exists\, m', E' \,.\, (\forall \ell' \notin \mathscr{L}_{\ell_A}\uparrow \,.\, m \simeq_{\ell'} m' \wedge E \simeq_{\ell'} E') \wedge \langle c, m', E', 0\rangle \Rrightarrow (\mathbf{M}, \mathbf{t}) \wedge (\mathbf{M}', \mathbf{t}') = (\mathbf{M}, \mathbf{t}) \upharpoonright^{pc(M_\eta) \notin \mathscr{L}_{\ell_A}\uparrow \wedge lev(M_\eta) \in \mathscr{L}_{\ell_A}\uparrow}\}$$

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 1.

In this definition, memory and machine environments are quantified differently from Definition 1, by considering variations with respect to a larger set of security levels $\mathscr{L}_{\ell_A}\uparrow$. In Fig. 5b, 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 2 (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 $\mathscr{L}$ it holds that*

$$\mathsf{Q}(\mathscr{L}, \ell_A, c, m, E) \leq \log |\mathsf{V}(\mathscr{L}, \ell_A, c, m, E)|$$

The proof is included in the corresponding technical report [39]. 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.

7. Predictive mitigation

Predictive mitigation [5, 38] removes confidential information from timing of public events by delaying them according to predefined schedules. We build upon this prior work [5, 38], but unlike the earlier work, our results improve precision for multilevel security, enabling better tradeoffs between security and performance.

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 2, 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.

$$\langle \texttt{update}(n,\ell), m, E, G\rangle \to \langle (\texttt{while}\ (\texttt{time} - s_\eta \geq \texttt{predict}(n,\ell))\ \texttt{do}\ (\texttt{Miss}[\ell] := \texttt{Miss}[\ell]+1;)_{[\bot,\bot]})_{[\bot,\bot]}, m, E, G\rangle \text{ (S-UPDATE)}$$

$$\langle \texttt{mitigate}_\eta\ (n,\ell)\ c, m, E, G\rangle \to \langle s_\eta := \texttt{time}_{[\bot,\bot]}; c; \texttt{update}(n,\ell); (\texttt{sleep}\ (\texttt{predict}(n,\ell) - \texttt{time} + s_\eta))_{[\bot,\bot]}, m, E, G\rangle \text{ (S-MTGPRED)}$$

Figure 6: Predictive semantics for `mitigate`

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 1: Machine environment parameters

Mitigating semantics. Fig. 6 shows the fragment of 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* [5] with the *local penalty policy* [38] ; other schemes and penalty policies are possible [5, 38], 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.

Leakage analysis of the mitigating semantics. Note that all auxiliary commands in Fig. 6 have labels $[\bot,\bot]$, ensuring 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 $\texttt{predict}(n,\ell)$.

We can show that leakage is at most $|\mathscr{L}_{\ell_A}\uparrow|\cdot\log(K+1)\cdot(1+\log T)$. Here T is the elapsed time and K is the number of relevant `mitigate` statements in the trace: the ones satisfying $pc(M_\eta) \notin \mathscr{L}_{\ell_A}\uparrow \wedge lev(M_\eta) \in \mathscr{L}_{\ell_A}\uparrow$. When `mitigate` is not used, $K=0$, so no timing leakage occurs. When K is unknown, it can be conservatively bounded by T, yielding an $O(\log^2 T)$ leakage bound. Note that the bound is proportional to the size of the set $\mathscr{L}_{\ell_A}\uparrow$. A detailed analysis and derivation can be found in the technical report [39].

8. Implementation

We implemented a simulation of the partitioned cache design described in Sec. 4.3 so we could evaluate our approach on real C programs. As case studies, we chose two applications previously shown to be vulnerable to timing attacks. The results suggest the new mechanism is sound and has reasonable performance.

8.1 Hardware implementation

We developed a detailed, dynamically scheduled processor model supporting two-level data and instruction caches, data and instruction TLBs, and speculative execution. Table 1 summarizes the features of the machine environment. We implemented this processor design by modifying the SimpleScalar simulator, v.3.0e [9].

A new register is added 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.

As discussed in Sec. 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.

8.2 Compilation

We use the `gcc` compiler in the SimpleScalar tool set to run C applications on the simulator. Sensitive data in applications are labeled, and timing labels are then inferred as the least restrictive labels satisfying the typing rules from Fig. 4 (transferring the rules from Sec. 5 to C is straightforward). To inform the hardware of the current timing label, assembly code setting the timing-label register is inserted before and after command blocks.

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, setting 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.

8.3 Web login case study

Web applications have been shown vulnerable to timing channel attacks. For example, Bortz and Boneh [7] 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 on the right. 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.

```
1  Hashmap m:=loadusers()
2  while true
3    (user, pass):=input()
4    uhash:=MD5(user)
5    if uhash in m
6      hash:=m.get(uhash)
7      phash:=MD5(pass)
8      if phash=hash
9        state:=success
10   state:=fail
11   response:=1
```

The 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 Fig. 7. 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.

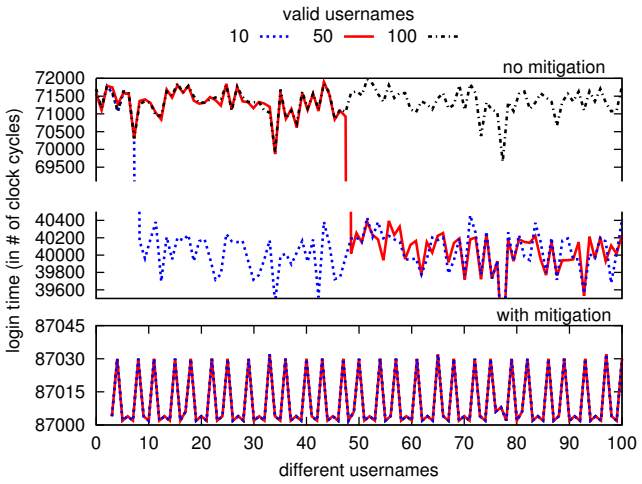


Figure 7: Login time with various secrets

	nopar	moff	mon
ave. time (valid)	70618	78610	86132
ave. time (invalid)	39593	43756	86147
overhead (valid)	1	1.11	1.22

Table 2: Login time with various usernames and options (in clock cycles)

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 2 shows the execution time of the main loop with various options, including both valid and invalid usernames, hardware with no partitions (nopar), and secure hardware both without (moff) and with (mon) mitigation.

As in Fig. 7, 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 2 shows that partitioned hardware is slower by about 11%. On valid usernames, language-based mitigation adds 10% slowdown; slowdown with combined software/hardware mitigation is about 22%.

8.4 RSA case study

The timing of efficient RSA implementations depends on the private key, creating a vulnerability to timing attacks [8, 18]. Using the RSA reference implementation, we demonstrate that its timing channels can be mitigated when decrypting a multi-block message.

```
1  text:=readText()
2  for each block b in text
3    ...preprocess...
4    compute (p:=b^key mod n)
5    ...postprocess...
6    write(output, plain)
```

In the pseudo-code on the left, only line 4 uses confidential data. 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.

Correctness. We use 100 encrypted messages and two different private keys to measure whether secrets affect timing. The upper plot in Fig. 8 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.

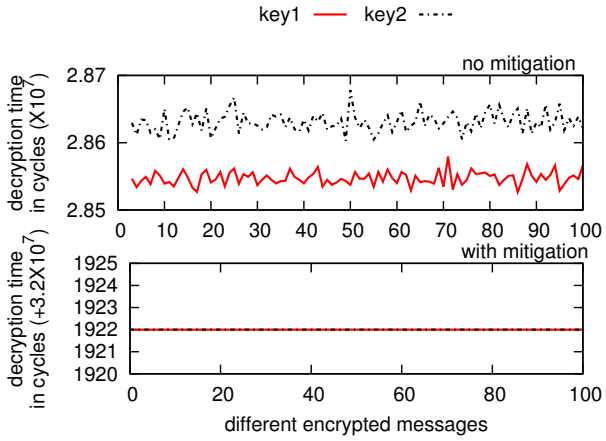


Figure 8: Decryption time with various secrets

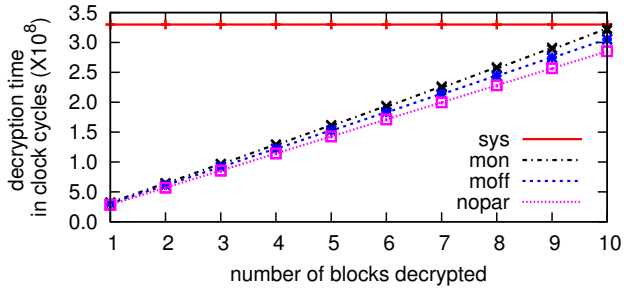


Figure 9: Language-level vs. system-level mitigation

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 system-level predictive mitigation [5], 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 Fig. 9 show that fine-grained language-based mitigation is faster because it does not try to mitigate the timing variation due to the number of decrypted blocks.

9. 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 [16, 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 [33], but are very restrictive. Often (e.g., [28, 30, 32, 33]) 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 [3, 6, 15]), or limit the

number of loop iterations [10, 23]. 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 [1, 2, 34].

Secure multi-execution [12, 17] 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 [25] and dynamic [35] 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. [19] show vulnerabilities in Wang's cache design [35]. Recent work by Li et al. [21] introduces a statically verifiable hardware description language for building hardware that is information-flow secure by construction. This work could complement our own.

10. Conclusions

Timing channels have long been considered one of the toughest challenges in computer security. They have become more of a concern as different computing systems are more tightly intermeshed and code from different trust domains is executed on the same hardware (e.g., cloud computing servers and web browsers).

Solving the timing channel problem requires work at both the hardware level and the software level. Neither level has enough information to allow accurate reasoning about timing channels, because timing is a property that crosses abstraction boundaries.

The new abstraction of read and write labels makes a useful step toward allowing timing channels to be controlled effectively at the language level. The corresponding security properties help guide the design of hardware secure against timing attacks.

For programs where timing channels cannot be blocked entirely, predictive mitigation can be incorporated at the language level. The security guarantees of this language-level enforcement have been proved formally; the performance characteristics of the enforcement mechanism have been studied experimentally and appear promising.

Acknowledgments

We thank Owen Arden, Dan Ports, and Nate Foster for their helpful suggestions. This work has been supported by a grant from the Office of Naval Research (ONR N000140910652), by two grants from the NSF: 0424422 (the TRUST center), and 0964409, and by MURI grant FA9550-12-1-0400, administered by the US Air Force. This research is also sponsored by the Air Force Research Laboratory.

References

[1] O. Acıiçmez. Yet another microarchitectural attack: Exploiting I-cache. In *Proceedings of the ACM Workshop on Computer Security Architecture (CSAW '07)*, pages 11–18, 2007.

[2] O. Acıiçmez, C. Koç, and J. Seifert. On the power of simple branch prediction analysis. In *ASIACCS*, pages 312–320, 2007.

[3] J. Agat. Transforming out timing leaks. In *Proc. 27th ACM Symp. on Principles of Programming Languages (POPL)*, pages 40–53, Boston, MA, January 2000.

[4] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *ESORICS*, pages 333–348, October 2008.

[5] A. Askarov, D. Zhang, and A. C. Myers. Predictive black-box mitigation of timing channels. In *ACM Conf. on Computer and Communications Security (CCS)*, pages 297–307, October 2010.

[6] G. Barthe, T. Rezk, and M. Warnier. Preventing timing leaks through transactional branching instructions. *Electronic Notes in Theoretical Computer Science*, 153(2):33–55, 2006.

[7] A. Bortz and D. Boneh. Exposing private information by timing web applications. In *Proc. 16th Int'l World-Wide Web Conf.*, May 2007.

[8] D. Brumley and D. Boneh. Remote timing attacks are practical. *Computer Networks*, January 2005.

[9] D. C. Burger and T. M. Austin. The SimpleScalar tool set, version 3.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[10] B. Coppens, I. Verbauwhede, K. D. Bosschere, and B. D. Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. *IEEE Symposium on Security and Privacy*, pages 45–60, 2009.

[11] D. E. Denning. *Cryptography and Data Security*. Addison-Wesley, Reading, Massachusetts, 1982.

[12] D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *IEEE Symposium on Security and Privacy*, pages 109–124, May 2010.

[13] J. Giffin, R. Greenstadt, P. Litwack, and R. Tibbetts. Covert messaging through TCP timestamps. *Privacy Enhancing Technologies, Lecture Notes in Computer Science*, 2482(2003):189–193, 2003.

[14] D. Gullasch, E. Bangerter, and S. Krenn. Cache games—bringing access-based cache attacks on AES to practice. In *IEEE Symposium on Security and Privacy*, pages 490–505, 2011.

[15] D. Hedin and D. Sands. Timing aware information flow security for a JavaCard-like bytecode. *Electronic Notes in Theoretical Computer Science*, 141(1):163–182, 2005.

[16] M. Huisman, P. Worah, and K. Sunesen. A temporal logic characterisation of observational determinism. In *Proc. 19th IEEE Computer Security Foundations Workshop*, 2006.

[17] V. Kashyap, B. Wiedermann, and B. Hardekopf. Timing- and termination-sensitive secure information flow: Exploring a new approach. In *IEEE Symposium on Security and Privacy*, pages 413–430, May 2011.

[18] P. Kocher. Timing attacks on implementations of Diffie–Hellman, RSA, DSS, and other systems. In *Advances in Cryptology—CRYPTO'96*, August 1996.

[19] J. Kong, O. Acıiçmez, J.-P. Seifert, and H. Zhou. Deconstructing new cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 2nd ACM Workshop on Computer Security Architectures*, pages 25–34, 2008.

[20] B. Köpf and M. Dürmuth. A provably secure and efficient countermeasure against timing attacks. In *2009 IEEE Computer Security Foundations*, July 2009.

[21] X. Li, M. Tiwari, J. Oberg, V. Kashyap, F. Chong, T. Sherwood, and B. Hardekopf. Caisson: a hardware description language for secure information flow. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 109–120, 2011.

[22] J. K. Millen. Covert channel capacity. In *Proc. IEEE Symposium on Security and Privacy*, Oakland, CA, April 1987.

[23] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner. The program counter security model: automatic detection and removal of control-flow side channel attacks. *Cryptology ePrint archive: report 2005/368*, 2005.

[24] D. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: the case of AES. *Topics in Cryptology–CT-RSA 2006*, January 2006.

[25] D. Page. Partitioned cache architecture as a side-channel defense mechanism. In *Cryptology ePrint Archive, Report 2005/280*, 2005.

[26] A. Russo and A. Sabelfeld. Securing interaction between threads and the scheduler. In *Proc. 19th IEEE Computer Security Foundations Workshop*, 2006.

[27] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, January 2003.

[28] A. Sabelfeld and D. Sands. Probabilistic noninterference for multi-threaded programs. In *Proc. 13th IEEE Computer Security Foundations Workshop*, pages 200–214. IEEE Computer Society Press, July 2000.

[29] S. Sellke, C. Wang, and S. Bagchi. TCP/IP timing channels: Theory to implementation. In *Proc. INFOCOM 2009*, pages 2204–2212, January 2009.

[30] G. Smith. A new type system for secure information flow. In *Proc. 14th IEEE Computer Security Foundations Workshop*, pages 115–125, June 2001.

[31] G. Smith. On the foundations of quantitative information flow. *Foundations of Software Science and Computational Structures*, 5504:288–302, 2009.

[32] G. Smith and D. 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.

[33] D. Volpano and G. Smith. Eliminating covert flows with minimum typings. In *Proc. 10th IEEE Computer Security Foundations Workshop*, pages 156–168, 1997.

[34] Z. Wang and R. Lee. Covert and side channels due to processor architecture. In *ACSAC '06*, pages 473–482, 2006.

[35] Z. Wang and R. Lee. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th annual international symposium on computer architecture (ISCA '07)*, pages 494–505, 2007.

[36] J. C. Wray. An analysis of covert timing channels. In *Proc. IEEE Symposium on Security and Privacy*, pages 2–7, 1991.

[37] S. Zdancewic and A. C. Myers. Observational determinism for concurrent program security. In *Proc. 16th IEEE Computer Security Foundations Workshop*, pages 29–43, June 2003.

[38] D. Zhang, A. Askarov, and A. C. Myers. Predictive mitigation of timing channels in interactive systems. In *ACM Conf. on Computer and Communications Security (CCS)*, pages 563–574, October 2011.

[39] D. Zhang, A. Askarov, and A. C. Myers. Language mechanisms for controlling and mitigating timing channels. Technical report, Cornell University, March 2012. `http://hdl.handle.net/1813/28635`.