Interpretable Noninterference Measurement and its Application to Processor Designs

ZIQIAO ZHOU*, Microsoft Research, USA
MICHAEL K. REITER*, Duke University, USA

Noninterference measurement quantifies the secret information that might leak to an adversary from what the adversary can observe and influence about the computation. Static and high-fidelity noninterference measurement has been difficult to scale to complex computations, however. This paper scales a recent framework for noninterference measurement to the open-source RISC-V BOOM core as specified in Verilog, through three key innovations: logically characterizing the core's execution incrementally, applying specific optimizations between each cycle; permitting information to be declassified, to focus leakage measurement to only secret information that cannot be inferred from the declassified information; and interpreting leakage measurements for the analyst in terms of simple rules that characterize when leakage occurs. Case studies on cache-based side channels generally, and on specific instances including SPECTRE attacks, show that the resulting toolchain, called DINoMe, effectively scales to this modern processor design.

ACM Reference Format:
Ziqiao Zhou and Michael K. Reiter. 2022. Interpretable Noninterference Measurement and its Application to Processor Designs. In *Proceedings of ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '21).* ACM, New York, NY, USA, 31 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Noninterference [Goguen and Meseguer, 1982] is a classic information flow policy that, informally, requires that an attacker's view be unaffected by the values that should remain secret to it. Since systems often necessarily leak some information, however, a more practical goal is to insist that the interference be "small", which in turn requires that it be measured in some way. Various methodologies have been proposed for doing so statically (e.g., Backes et al. [2009], Phan and Malacaria [2014], Zhang et al. [2010]), though these techniques invariably must balance a tension between measurement fidelity and scalability to complex computations.

A recent advance in this domain was due to Zhou et al. [2018], which formulated noninterference measurement in terms of a *projected model counting* problem that, in turn, was amenable to relatively efficient, *approximate* model counting methods. Their measurement approach, however, scales to programs of only modest complexity, for two reasons. Computationally, their technique relies on symbolic execution to generate a logical postcondition for the computation for which noninterference is to be measured. For example, this step alone required six hours for `Smaz` and eight hours for `Gzip`, using 16 cores, for extracting postconditions to measure the risk of CRIME attacks [Kelsey, 2002] against these compression libraries. More qualitatively, while their technique

*Work performed in part at the University of North Carolina, Chapel Hill, NC, USA.

OOPSLA '21, Chicago, IL
© 2022 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

provides a measurement of interference, it provides the analyst little assistance in interpreting the measurement or focusing the analysis on particular aspects of the leakage.

While noninterference measurement for arbitrary computations remains out of reach, in this paper we adapt the approach of Zhou et al. [2018] to address the previous shortcomings within a particularly important and complex domain, namely information leaks arising in hardware processors. Leakage of software secrets due to processor optimizations have attracted massive attention in recent years, especially since the discovery of vulnerabilities arising due to the footprint of speculative executions in processor caches (Spectre [Kocher et al., 2019], Meltdown [Lipp et al., 2018], and variants). Even though many defenses (e.g., Tan et al. [2020], Wang and Lee [2007], Werner et al. [2019], Zhou et al. [2016]) have been proposed to interfere with cache-based side channels, we are aware of no measurement methodology to compare designs and evaluate their effectiveness, working directly from their Verilog specifications. Adapting a technique like Zhou et al. [2018] to do so, moreover, appears difficult: the sheer complexity of modern processor designs both necessitates greater support to help the analyst understand the factors contributing to the leakage and poses significant scaling challenges to such techniques.

In this paper, we present a methodology that does so, using three key methodological advances:

- Our methodology enables analysts to *declassify* certain information, thereby focusing the measurement on any *other* leakage that might be occurring, i.e., leakage that cannot be inferred from the declassified information. For systems as complex as modern processors, this ability is essential to permit analysts to decompose and analyze leakage in a piecemeal fashion.
- The complexity of processor designs means that once leakage is measured, the exact conditions that cause this leakage might not immediately be evident. Our methodology therefore incorporates a method of *interpreting* the leakage, i.e., providing simple rules that indicate circumstances in which leakage will (or will not) occur. These rules facilitate analyst understanding of the root causes of leakage and can guide analysts to declassify leakage that can be ignored. Each such rule is additionally accompanied by a precision and recall, so that analysts can prioritize the rules they address. These rules are expressed in terms of conditions in which leakage occurs, enabling executions to be generated that demonstrate the leakage if desired but hiding the particulars of the executions from analysts if not.
- Since generating a logical postcondition for a processor's execution of a program en masse is intractable, we devise a method to build the postcondition one cycle at a time. To build single-cycle formulas, we abandon symbolic execution, as we found that applying it to hardware designs induces significant path explosion for even one CPU cycle. Instead, we extract the single-cycle formulas without solving for feasible paths, and then leverage a number of aggressive optimizations when stitching single-cycle formulas together to build the postcondition for the processor's multi-cycle execution. In doing so, we need to be careful that these optimizations preserve the number of assignments to relevant variables across all solutions.

Due to the focus of our methodology on support for declassification and interpretability, we call our tool that realizes it DINoMe (for "Declassification and Interpretability for Noninterference Measurement").

To evaluate DINoMe, we apply it to evaluate leakage during execution on a RISC-V BOOM core [Celio et al., 2017], a state-of-the-art public domain processor design. Our improvements to generating logical postconditions for execution permit DINoMe to do so for more than 100 cycles of this core. This, in turn, permits us to evaluate leakage from cache-based side channels (Prime+Probe [Osvik et al., 2006] and Flush+Reload [Yarom and Falkner, 2014]) in various scenarios, including cryptographic key leakage in sliding-window based modular exponentiation (e.g., Aciiçmez [2007], Percival [2005]), leakage of secrets due to speculative execution, and how this leakage is (incompletely) mitigated by proposed improvements such as ScatterCache [Werner et al., 2019] and PhantomCache [Tan et al., 2020]. In each case, we not only measure interference but also generate rules to explain why the leakage occurs, and in some cases refine our view of the leakage using declassification. Our performance evaluation of DINoMe indicates that these types of analyses complete in times ranging from seconds to under 15 minutes (using horizontal scaling), after an initial phase to assemble the logical postcondition of up to (only) two hours on (only) a single core.

The rest of this paper is structured as follows. We discuss related work in Sec. 2, and provide both background on the framework on which we build Zhou et al. [2018] and our introduction of declassification to it, our first contribution, in Sec. 3. We present our method for interpreting leakage in Sec. 4. We address implementation challenges in Sec. 5, and then evaluate DINoMe through several case studies in Sec. 6. We discuss limitations in Sec. 8 and conclude in Sec. 9.

2 RELATED WORK

To our knowledge, DINoMe is the first work to measure information leakage from an executable hardware specification instantiated with a software program, in a manner that supports declassification and interpretation of its leakage results.

Timing side-channel analysis. Constant-time verification (e.g., Almeida et al. [2016], Barthe et al. [2014], Blazy et al. [2019], Gleissenthall et al. [2019], Zhang et al. [2015]) is a commonly used technique to analyze timing side channels. Software-level verification (e.g., Almeida et al. [2016], Blazy et al. [2019]) checks whether a software program runs in a constant time under specified hardware assumptions. For example, a software-level analysis [Almeida et al., 2016] might conclude that a variable leaks if it is used in a branch condition or as an address in memory access. In a different approach, hardware-level verifiers (e.g., Gleissenthall et al. [2019], Zhang et al. [2015, 2018]) can formally verify the existence of timing side channels using cycle-precise logic derived from hardware specifications. Those works either confirm a constant-time implementation or quantify the timing variation if not, but do not quantify secret leakage due to timing variations in different executions.

Hardware leakage modeling. Some works use simplified hardware models instead of real designs (e.g., Chattopadhyay et al. [2017], Doychev et al. [2013], Malacaria et al. [2018]), which makes the computation target feasible but requires more domain knowledge and manual effort to construct the model. Black-box analysis of real systems avoids the use of domain knowledge through a data-driven method that uses sampled data in a real system for estimating the leakage (e.g., Nilizadeh et al. [2019], Oleksii et al. [2020], Song et al. [2001]). In contrast, DINoMe measures leakage from hardware specifications written in a hardware design language.

Quantitative information flow. QIF (e.g., Gray [1991], Smith [2009, 2011]) represents information leakage through a numeric measurement; most mainstream QIF works (e.g., Chapman and Evans [2011], Phan and Malacaria [2014], Zhang et al. [2010]) use entropy as their measure [Seidenfeld, 1986]. The use of entropy for measuring QIF in actual systems can lead to significant costs, due to the need to compute the input preimage per output value. In addition, real implementations tend to use the most conservative min-entropy measure; e.g., QIF-Verilog [Guo et al., 2019] propagates a min-entropy label per gate and accumulates the leakage across all gates, which overestimates leakage due to its conservative leakage accumulation, especially in large, complex hardware designs (e.g., a CPU core). Entropy also does not distinguish between leaking a few bits in many executions or leaking more bits in a few cases. Alternatives to entropy-based leakage—e.g., differential privacy [Dwork et al., 2006], noninterference measurement [Zhou et al., 2018], classifier-based measurement [Chapman and Evans, 2011], and quantitative hyperproperties [Sahai et al., 2020, Yasuoka and Terauchi, 2014]—measure the attacker's ability to distinguish some secret values from others. Those metrics do not accommodate declassification or leakage interpretability, our main concerns here.

Declassification. To rule out allowed leakage and focus on targeted leakage, information flow control research supports declassification policies to specify the secret information permitted to transfer to observable variables (e.g., Banerjee et al. [2008], Chong and Myers [2004], Ferraiuolo et al. [2017], Giacobazzi and Mastroeni [2018], McCall et al. [2018], Sabelfeld and Myers [2003], Sabelfeld and Sands [2009]). However, while this work omits declassified information from its analysis, it does not quantitatively measure the remaining leakage in light of what the attacker can already infer from the declassified information. In contrast, our work adapts information leakage measurement to account for such inferences.

Leakage interpretability. To interpret quantitative leakage, domain-specific works (e.g., SPEECH-MINER [Xiao et al., 2020], CacheBar [Zhou et al., 2016]) use customized measures following a specific attack templates, forgoing general measures. Although those customized measures are more understandable when interpreting a specific attack vector, they are blind to leakage from different attacks not considered. One crucial improvement our work makes in evaluating information leakage is to generate an interpretable model to explain how leakage occurs. Already an emerging topic in machine learning (e.g., Chen et al. [2018], Molnar [2019]), interpretability is especially important in security evaluation, since it is not easy to draw a clear threshold to indicate when a system is secure enough, even with a perfect measure. Many methods for measuring leakage in software (e.g., Chattopadhyay and Roychoudhury [2018], Godefroid et al. [2012], Wang et al. [2009], Zhou et al. [2018]) generate a code path to help the analyst understand leakage. However, leakage in hardware-software joint codebases often exploits interactions between the two, which can manifest in many code-dependent paths. We are aware of no comparable work that explores an interpretable ML model to explain information-flow leakage, though the method we use to extract explanations in Sec. 4.3 builds from previous work in interpretable ML (e.g., Friedman and Popescu [2008], Ribeiro et al. [2016, 2018]).

3 NONINTERFERENCE AND DECLASSIFICATION

We begin in Sec. 3.1 by providing background on the noninterference measurement methodology of Zhou et al. [2018]. We then discuss how we extend this methodology to support declassification, our first contribution, in Sec. 3.3.

3.1 Background on noninterference measure

To analyze the leakage from a procedure *proc*[1], the procedure is modeled as having four different sets of formal parameters: a set $Vars_{\vec{s}}$ of secret input variables; a set $Vars_{\vec{c}}$ of attacker-controlled input variables; a set $Vars_{\vec{\imath}}$ of other input variables; and a set $Vars_{\vec{o}}$ of attacker-observable output variables. The actual parameter values assigned to those variables in an invocation of *proc* are given by maps $\vec{s} : Vars_{\vec{s}} \to Vals_{\vec{s}}$, $\vec{c} : Vars_{\vec{c}} \to Vals_{\vec{c}}$, and $\vec{\imath} : Vars_{\vec{\imath}} \to Vals_{\vec{\imath}}$, respectively; e.g., $\vec{\imath}(ivar) \in Vals_{\vec{\imath}}$ represents the value passed in variable $ivar \in Vars_{\vec{\imath}}$. The attacker-observable outputs of the procedure are defined by the map $\vec{o} : Vars_{\vec{o}} \to Vals_{\vec{o}}$. Accordingly, we denote the procedure

$$\vec{o} \leftarrow proc(\vec{c}, \vec{\imath}, \vec{s})$$

We assume that *proc* is deterministic; a nondeterministic *proc* can be rendered deterministic by providing the random values as inputs, say $\vec{\imath}$('coins'). A given *proc* can then be characterized by a

[1]Different from the definition used by Zhou et al. [2018], which is for a software procedure, our *proc* $(\vec{c}, \vec{\imath}, \vec{s})$ includes both the software and hardware logic.

logical postcondition $\Pi_{proc}(\vec{c}, \vec{o}, \vec{\imath}, \vec{s})$ that constrains how the values in $\vec{o}$ relate to those in $\vec{c}$, $\vec{\imath}$, and $\vec{s}$ in any execution. Without loss of generality, below we assume $Vars_{\vec{s}}$ contains a single variable *svar*, i.e., $Vars_{\vec{s}} = \{svar\}$.

The basic idea of the metric developed by Zhou et al. [2018] is to quantify the difficulty the attacker has in distinguishing between $\vec{s}(svar) \in S$ and $\vec{s}(svar) \in S'$ for random, disjoint sets S, S', based on the $\langle\vec{c}, \vec{o}\rangle$ pairs possibly available to it in the two cases, denoted Y_S, $Y_{S'}$, i.e.,

$$X_S = \left\{\langle\vec{c}, \vec{o}, \vec{\imath}\rangle \;\middle|\; \exists\vec{s} : \Pi_{proc}(\vec{c}, \vec{o}, \vec{\imath}, \vec{s}) \wedge \vec{s}(svar) \in S\right\}$$
$$Y_S = \left\{\langle\vec{c}, \vec{o}\rangle \;\middle|\; \exists\vec{\imath} : \langle\vec{c}, \vec{o}, \vec{\imath}\rangle \in X_S\right\}$$

Zhou et al. [2018] specifically explored the Jaccard distance between Y_S and $Y_{S'}$ to measure the difficulty an attacker would have in distinguishing between $\vec{s}(svar) \in S$ and $\vec{s}(svar) \in S'$. To better capture the importance of $\vec{\imath}$ in the leakage, however, they further replaced $Y_S \cap Y_{S'}$ with $\hat{X}_{S,S'}$, where[2]

$$\check{X}_{S,S'} = X_S \cup X_{S'}$$
$$\hat{X}_{S,S'} = \left\{\langle\vec{c}, \vec{o}, \vec{\imath}\rangle \;\middle|\; \langle\vec{c}, \vec{o}, \vec{\imath}\rangle \in \check{X}_{S,S'} \wedge \langle\vec{c}, \vec{o}\rangle \in Y_S \cap Y_{S'}\right\}$$

In this way, the number of values $\vec{\imath}$ for $\langle\vec{c}, \vec{o}\rangle$ exposed in $\hat{X}_{S,S'}$ serves as the "weight" of that $\langle\vec{c}, \vec{o}\rangle$ pair. When $\langle\vec{c}, \vec{o}, \vec{\imath}\rangle$ is from

$$\tilde{X}_{S,S'} = \check{X}_{S,S'} \setminus \hat{X}_{S,S'} \tag{1}$$

an attacker can distinguish if $\vec{s}(svar)$ is from S or S'. Zhou et al. [2018] thus suggested the measure $\hat{J}_n$, where

$$\hat{J}(S, S') = \left|\tilde{X}_{S,S'}\right| / \left|\check{X}_{S,S'}\right| = 1 - \left|\hat{X}_{S,S'}\right| / \left|\check{X}_{S,S'}\right| \tag{2}$$
$$\hat{J}_n = \underset{\substack{S, S' : |S| = |S'| = n \\ \wedge\ S \cap S' = \emptyset}}{\text{avg}} \hat{J}(S, S') \tag{3}$$

As discussed by Zhou et al. [2018, Sec. III], when n is small, $\hat{J}_n$ measures how frequently leakage occurs, whereas when n is large, it measures how much information about the secret leaks, when leakage occurs.

3.2 Motivating examples

To see this measure applied to simple programs, consider the two programs with a secret shown in Fig. 1(a) and Fig. 1(b). The procedure in Fig. 1(a) returns a random value between 0-7 or a fixed value 8 depending on whether $\vec{s}$('secret') mod 32 $<$ 16 if $\vec{c}$('test') mod 32 $>$ 15 and returns a fixed value 9 otherwise. The second procedure in Fig. 1(b) returns the five least significant bits of $\vec{s}$('secret') & $\vec{c}$('test'). Directly measuring the two procedures using $\hat{J}_n$ leads to different leakage measures, as it should, as shown in Fig. 1(e).

Some sources of information leakage may be inevitable or intentional; e.g., a bank website may not mask the last four digits of a user's social security number when displaying it to her browser, and so the site intentionally "leaks" that portion to a malicious browser. In the context of the preceding example, now suppose the leakage of the four least significant bits of the secret is intended (similar to the SSN example). Since the $\hat{J}_n$ curve only reflects the total interference, including the portion *intended* to leak (i.e., the four least significant bits), the $\hat{J}_n$ curves shown in Fig. 1(e) mislead us to

[2]Our definition of $\hat{X}_{S,S'}$ differs from Zhou et al. [2018], which only requires $\langle\vec{c}, \vec{o}, \vec{\imath}\rangle \in X_S$. Ours has the same essential properties but is symmetric with respect to S and S' and so is easier to work with.

```
proc ($\vec{c}$, $\vec{\imath}$, $\vec{s}$)
    if ($\vec{c}$('test') mod 32 > 15)
        if ($\vec{s}$('secret') mod 32 < 16)
            $\vec{o}$('result') ← $\vec{\imath}$('random') mod 8
        else
            $\vec{o}$('result') ← 8
    else $\vec{o}$('result') ← 9
```

(a) Implicit flow

```
proc ($\vec{c}$, $\vec{\imath}$, $\vec{s}$)
    $\vec{o}$('result') ← $\vec{c}$('test') & $\vec{s}$('secret') & 0x1f
```

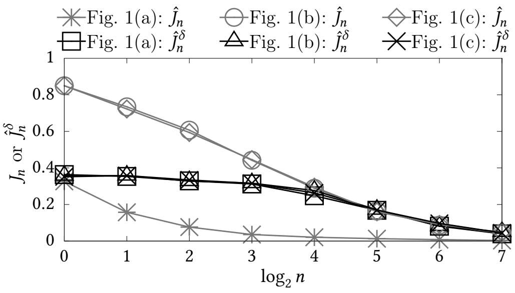
(b) Explicit flow

```
proc ($\vec{c}$, $\vec{\imath}$, $\vec{s}$)
    $\vec{o}$('result') ← $\vec{c}$('test') & $\vec{s}$('secret') & 0x2f
```

(c) Different explicit flow

```
$\delta$ ($\vec{c}$,$\vec{\imath}$,$\vec{s}$)
    $\vec{\triangle}$('info') ← $\vec{s}$('secret') & 0x0f
```

(d) Declassification policy



(e) Measurement with vs. without declassification

Fig. 1. Motivating examples for declassification (Sec. 3.3) and interpretation (Sec. 4)

conclude that Fig. 1(a) is more secure than Fig. 1(b). In truth, they both additionally leak the fifth least significant bit, which is the only leakage that matters.

3.3 Declassification

To exclude such intended leakage from the analysis, it will be helpful to provide a method to exempt some identified information leakages specified by the analyst, allowing the analysis to focus on the leakage that remains. Specifically, our methodology seeks to assess the degree to which a procedure permits secrets to be distinguished by the attacker using attacker-observable and declassified information but not by the declassified information alone.

Let $\vec{\triangle} \leftarrow \delta(\vec{c}, \vec{\imath}, \vec{s})$ denote the allowed information exposure (e.g., for a website requiring SSN, $\vec{\triangle}$ is the last four digits), and let

$$\Pi_{proc,\delta}(\vec{c}, \vec{o}, \vec{\triangle}, \vec{\imath}, \vec{s}) \leftarrow \Pi_{proc}(\vec{c}, \vec{o}, \vec{\imath}, \vec{s}) \land \Pi_{\delta}(\vec{c}, \vec{\triangle}, \vec{\imath}, \vec{s})$$

where $\Pi_{\delta}(\vec{c}, \vec{\triangle}, \vec{\imath}, \vec{s})$ is a logical postcondition for δ that relates $\vec{\triangle}$ to $\vec{c}$, $\vec{\imath}$, and $\vec{s}$. Then, we can define the attacker's accessible set Y^δ_S of $\langle \vec{c}, \vec{o}, \vec{\triangle} \rangle$ tuples and allowed accessible set D^δ_S consistent with chosen secret set S by

$$X^\delta_S = \left\{ \langle \vec{c}, \vec{o}, \vec{\triangle}, \vec{\imath} \rangle \,\middle|\, \exists \vec{s} : \vec{s}(svar) \in S \land \Pi_{proc,\delta}(\vec{c}, \vec{o}, \vec{\triangle}, \vec{\imath}, \vec{s}) \right\}$$

$$Y^\delta_S = \left\{ \langle \vec{c}, \vec{o}, \vec{\triangle} \rangle \,\middle|\, \exists \vec{\imath} : \langle \vec{c}, \vec{o}, \vec{\triangle}, \vec{\imath} \rangle \in X^\delta_S \right\}$$

$$D^\delta_S = \left\{ \langle \vec{c}, \vec{\triangle} \rangle \,\middle|\, \exists \vec{o}, \vec{\imath} : \langle \vec{c}, \vec{o}, \vec{\triangle}, \vec{\imath} \rangle \in X^\delta_S \right\}$$

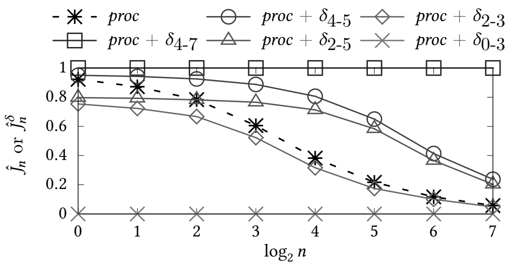
Since the declassified information is allowed to leak, we are concerned only with cases where the secret is distinguishable by $\langle \vec{c}, \vec{o}, \vec{\triangle} \rangle$ but not by $\langle \vec{c}, \vec{\triangle} \rangle$. Here, we define a set $\tilde{X}^\delta_{S,S'}$ to include the

```
$proc(\vec{c}, \vec{\imath}, \vec{s})$
    $\vec{o}(ovar) \leftarrow \vec{s}(svar)[0:3]$
```

(a) An artificial procedure

```
$\delta_{i\text{-}j}(\vec{c}, \vec{\imath}, \vec{s})$
    $\vec{\Delta}(dvar) \leftarrow \vec{s}(svar)[i:j]$
```

(b) Declassification policy



(c) Measurement with vs. without declassification

Fig. 2. Declassification example

tuples $\langle \vec{c}, \vec{o}, \vec{\Delta} \rangle$ that leak whether the secret is in S or S', assuming $\langle \vec{c}, \vec{\Delta} \rangle$ is equivalent.

$$\check{X}^{\delta}_{S,S'} = \left\{ \langle \vec{c}, \vec{o}, \vec{\Delta}, \vec{\imath} \rangle \;\middle|\; \begin{array}{l} \langle \vec{c}, \vec{o}, \vec{\Delta}, \vec{\imath} \rangle \in X^{\delta}_{S} \cup X^{\delta}_{S'} \\ \wedge \quad \langle \vec{c}, \vec{\Delta} \rangle \in D^{\delta}_{S} \cap D^{\delta}_{S'} \end{array} \right\} \tag{4}$$

$$\hat{X}^{\delta}_{S,S'} = \left\{ \langle \vec{c}, \vec{o}, \vec{\Delta}, \vec{\imath} \rangle \;\middle|\; \begin{array}{r} \langle \vec{c}, \vec{o}, \vec{\Delta}, \vec{\imath} \rangle \in \check{X}^{\delta}_{S,S'} \\ \wedge \ \langle \vec{c}, \vec{o}, \vec{\Delta} \rangle \in Y^{\delta}_{S} \cap Y^{\delta}_{S'} \end{array} \right\} \tag{5}$$

$$\tilde{X}^{\delta}_{S,S'} = \check{X}^{\delta}_{S,S'} \setminus \hat{X}^{\delta}_{S,S'} \tag{6}$$

Thus, we can use an alternative metric

$$\hat{J}^{\delta}(S,S') = \left|\tilde{X}^{\delta}_{S,S'}\right| / \left|\check{X}^{\delta}_{S,S'}\right| \tag{7}$$

$$\hat{J}^{\delta}_{n} = \operatorname*{avg}_{\substack{S,S' \,:\, |S| = |S'| = n \\ \wedge\ S \cap S' = \emptyset}} \hat{J}^{\delta}(S,S') \tag{8}$$

Returning to the examples in Fig. 1(a) and Fig. 1(b) with declassification of the four least significant bits (Fig. 1(d)), the $\hat{J}^{\delta}_{n}$ curves show the same quantitative leakage (Fig. 1(e)), as they should.

To further illustrate the the impact of declassification, consider the simple procedure shown in Fig. 2(a). In this procedure, $\vec{s}(svar)$ is an 8-bit value, and *proc* outputs the lowest 4 bits as $\vec{o}(ovar)$. The declassification policy shown in Fig. 2(b) allows the i-th to j-th bits of $\vec{s}(svar)$ to be released. We evaluate $\hat{J}^{\delta}_{n}$ with differently parameterized declassification policies in Fig. 2(c). Specifically, when the lowest 4 bits ($i = 0$, $j = 3$) are declassified, then the additional leakage from *proc* is nothing, which is demonstrated by the "$proc + \delta_{0\text{-}3}$" curve. When the declassification policy declassifies all but the lowest 4 bits ($i = 4$, $j = 7$), then the additional leakage by *proc* is maximized, as shown by the "$proc + \delta_{4\text{-}7}$" curve. Intuitively, if $\vec{o}(ovar)$ and $\vec{\Delta}(dvar)$ do not overlap (e.g., "$proc + \delta_{4\text{-}7}$" and "$proc + \delta_{4\text{-}5}$"), then the $\hat{J}^{\delta}_{n}$ curve should be higher than $\hat{J}_{n}$, whereas if $\vec{o}(ovar)$ includes all of $\vec{\Delta}(dvar)$ (e.g., "$proc + \delta_{0\text{-}3}$" and "$proc + \delta_{0\text{-}1}$"), then $\hat{J}^{\delta}_{n}$ should be lower than $\hat{J}_{n}$. A hybrid case occurs when $\vec{o}(ovar)$ includes a portion of $\vec{\Delta}(dvar)$ (e.g., "$proc + \delta_{2\text{-}5}$"), where $\hat{J}^{\delta}_{n}$ is lower than $\hat{J}_{n}$ when n is small but becomes larger when n is large. This is consistent with the interpretation that $\hat{J}^{\delta}_{n}$ with small n primarily reflects the number of secret values for which interference occurs [Zhou et al., 2018]; e.g., when $n = 1$, two secret values share bits 0–1 (and so cannot be distinguished by bits 0–3 after declassifying bits 2–5) in 25% of cases, but share bits 0–3 (and so cannot be distinguished using them) in only 6.25% of cases. Larger n, in contrast, better reflects the amount of leakage that occurs [Zhou et al., 2018]. For example, in a random partition of all 2^8 values into sets S and S' of equal size (i.e., $n = 2^7$), every value for bits 2–5 is represented in both S and S' with high probability.

In conjunction with the additional bits 0–1 output in $\vec{o}$ (yielding six bits of the secret value in total), however, these bits give the attacker greater distinguishing power than do bits 0–3 alone.

4 INTERPRETING LEAKAGE

Our metric measures the additional interference of a secret with values observable by the attacker, beyond that implied by declassified information. For this to be useful to an analyst, however, we need to explain *how* this leakage occurs. Specifically, while the conditions under which leakage occurs are already present in the procedure postcondition, it is difficult to understand the formula without further help (e.g., see Sec. 6.5).

4.1 Motivating examples for interpretation

Consider again the motivating examples in Fig. 1(a) and Fig. 1(b). The two procedures own quite different outputs but still leak the same additional information about the secret after declassification (i.e., both leak the fifth least significant bit of the secret when $\vec{c}$('test')'s fifth bit is 1 and nothing otherwise). To cut through the differences in code style and concrete values, DINoMe derives the condition when a pair of secrets are distinguishable using paired samples of input. Thus, the interference rule for both cases becomes $|\vec{s}(\text{'secret'})[4] - \vec{s}'(\text{'secret'})[4]| > 0 \land \vec{c}(\text{'test'})[4] = 1$. This rule shows the equivalence of these procedures' leakages after declassification.

In addition, interpreting leakage can differentiate cases with the same *amount* of leakage but different conditions in which that leakage occurs. For example, the procedure in Fig. 1(c), which reveals the four least significant bits and the sixth bit of the secret when the sixth bit of $\vec{c}$('test') is 1, leaks the same amount of information about a different portion of the secret under a different attack condition. A quantitative leakage measurement with the same four low-order bits declassified will not distinguish Fig. 1(c) from Fig. 1(b) (see Fig. 1(e)). Through DINoMe's interpretation, we provide a slightly different interference rule for Fig. 1(c), however: $|\vec{s}(\text{'secret'})[5]-\vec{s}'(\text{'secret'})[5]| > 0 \land \vec{c}(\text{'test'})[5] = 1$.

Though these motivating examples seem small and readable even when using different coding styles and output values, real-world code can become difficult to understand, particularly when spanning different levels of abstraction (e.g., a processor and the code it is executing). It is here we expect our interpretation of interference to simplify investigating leakage. Learning from the previous examples, our interpretation should achieve two goals. First, the interference interpretation for the same functionality should be consistent no matter how the functionality is implemented. Second, the interference interpretation should distinguish two procedures if they leak information in different ways, even when they leak the same amount.

4.2 Noninterference and interference tuples

Our first step toward providing an intuitive explanation for the leakage that occurs is to train a binary classifier to classify 4-tuples $\langle \vec{c}, \vec{\imath}, \vec{s}, \vec{s}' \rangle$ into those that illustrate leakage occurring (i.e., that permit the attacker to distinguish $\vec{s}(\mathit{svar})$ and $\vec{s}'(\mathit{svar})$ from the resulting output $\vec{o}$) and those that do not. When using declassification, the interference tuples should only include those where the secrets can be distinguished using $\vec{c}, \vec{o}, \vec{\Delta}$ but not using just $\vec{c}, \vec{\Delta}$.

More specifically, we define the interference set *IS* based on (6). That is, when the attacker chooses $\vec{c}$, if an observable value is feasible for $\langle \vec{\imath}, \vec{s} \rangle$ for some $\vec{\imath}$ but is never possible for $\langle \vec{\imath}', \vec{s}' \rangle$ for any $\vec{\imath}'$ that shares a declassification value with $\langle \vec{\imath}, \vec{s} \rangle$, then $\langle \vec{c}, \vec{\imath}, \vec{s}, \vec{s}' \rangle$ is added to *IS*:

$$IS = \left\{ \langle \vec{c}, \vec{\imath}, \vec{s}, \vec{s}' \rangle \,\middle|\, \exists \vec{o}, \vec{\Delta} : \ \langle \vec{c}, \vec{o}, \vec{\Delta}, \vec{\imath} \rangle \in X^{\delta}_{S} \land \langle \vec{c}, \vec{\Delta} \rangle \in D^{\delta}_{S'} \cap D^{\delta}_{S'} \quad \land \langle \vec{c}, \vec{o}, \vec{\Delta} \rangle \in Y^{\delta}_{S} \setminus Y^{\delta}_{S'} \right\} \tag{9}$$

where $S = \{\vec{s}(\mathit{svar})\}$ and $S' = \{\vec{s}'(\mathit{svar})\}$.

The noninterference set *NS* should include two types of tuples. For an attacker-chosen $\vec{c}$, if there is a observable value $\vec{o}$ that is feasible for an $\langle\vec{\imath},\vec{s}\rangle$ pair and an $\langle\vec{\imath}',\vec{s}'\rangle$ pair, tuple $\langle\vec{c},\vec{\imath},\vec{s},\vec{s}'\rangle$ belongs to *NS* as it is an example where no interference occurs. In addition, for an attacker-chosen $\vec{c}$, if there is a declassification value $\vec{\Delta}$ that is feasible for $\langle\vec{\imath},\vec{s}\rangle$ but not $\langle\vec{\imath}',\vec{s}'\rangle$ for any $\vec{\imath}'$, then $\langle\vec{c},\vec{\imath},\vec{s},\vec{s}'\rangle$ should also be added to *NS*, as $\vec{s}$ and $\vec{s}'$ can already be distinguished using the declassified value:

$$\begin{aligned} NS = &\left\{\langle\vec{c},\vec{\imath},\vec{s},\vec{s}'\rangle \;\middle|\; \exists\vec{o},\vec{\Delta}:\ \langle\vec{c},\vec{o},\vec{\Delta},\vec{\imath}\rangle \in X^{\delta}_{S} \wedge \langle\vec{c},\vec{o},\vec{\Delta}\rangle \in Y^{\delta}_{S} \cap Y^{\delta}_{S'} \right\} \\ \cup &\left\{\langle\vec{c},\vec{\imath},\vec{s},\vec{s}'\rangle \;\middle|\; \exists\vec{o},\vec{\Delta}:\ \langle\vec{c},\vec{o},\vec{\Delta},\vec{\imath}\rangle \in X^{\delta}_{S} \wedge \langle\vec{c},\vec{\Delta}\rangle \in D^{\delta}_{S} \setminus D^{\delta}_{S'} \right\} \end{aligned} \tag{10}$$

where $S = \{\vec{s}(svar)\}$ and $S' = \{\vec{s}'(svar)\}$.

Since *NS* and *IS* are large in practical scenarios, enumerating all tuples is generally infeasible. Instead, we generate samples in each set to train a machine learning model, from which explanations of the leakage will be extracted (as described below). Doing so with modern SAT solvers, however, typically results in samples that cover *NS* and *IS* unevenly, since solvers generally enumerate the next solution by simply adding a conflict constraint to block out previous solutions; as a result, the next solution found is typically close to the previous. Another drawback of using this "blocking" method to sample is that we cannot parallelize the sampling.

For this reason, we sample from *NS* and *IS* using hash-based sampling (cf., Zhou et al. [2018]). Specifically, we sample a limited number of solutions by adding a random universal hashing constraint to the formula given to the solver. Due to the hash function's universality, we can run multiple samplers in parallel to generate a large number of uniformly distributed solutions. In most cases, the sizes of the sampled sets $\hat{NS}$ and $\hat{IS}$ differ either due to differences in the sizes of *NS* and *IS* or due to the solving difficulty of one set compared to the other. We associate a sample weight to each element so the weight of each set is equal in the training process described below.

4.3 Interpretation through a rule-based method

Given $\hat{NS}$ and $\hat{IS}$—i.e., $\langle\vec{c},\vec{\imath},\vec{s},\vec{s}'\rangle$ tuples labeled according to whether they illustrate noninterference or interference—we could train an interpretable machine-learning model and then extract rules to explain to the user what gives rise to interference. A natural such model to consider is a decision tree. In a decision tree, each decision node (i.e., interior node) is a predicate on features of a $\langle\vec{c},\vec{\imath},\vec{s},\vec{s}'\rangle$ tuple, and its two children correspond to a true or false evaluation of this predicate on a tuple, respectively. A $\langle\vec{c},\vec{\imath},\vec{s},\vec{s}'\rangle$ tuple is classified by traversing the tree from its root, following the branch from each decision node corresponding to the result of evaluating the predicate at that node on the tuple. Each leaf is labeled with an estimate of the probability that a tuple constrained by the predicates' evaluations from the root to that leaf is in *IS*. We will discuss what features we include in the process of building decision trees in Sec. 4.4, but an example might be individual variables (e.g., $\vec{c}(cvar)$).

A single decision tree can easily grow to be deep and complex, and it can miss some useful combinations of predicates since each decision predicate is highly influenced by the splits above it in the tree. To make the decision tree model more powerful in finding useful predicates, we used a decision-tree ensemble called gradient boosted trees [Friedman, 2001]. This process produces m trees denoted $T_1,\ldots,T_m$, with associated weights. If we denote by $T_j(\langle\vec{c},\vec{\imath},\vec{s},\vec{s}'\rangle)$ the real number stored at the leaf to which $\langle\vec{c},\vec{\imath},\vec{s},\vec{s}'\rangle$ is assigned by T_j, then the weighted sum of $T_j(\langle\vec{c}\ ,\vec{\imath},\vec{s},\vec{s}'\rangle)$ for $j = 1,\ldots,m$ is an estimate of the probability that $\langle\vec{c},\vec{\imath},\vec{s},\vec{s}'\rangle \in IS$.

To interpret tree ensembles, rule-based classifiers (e.g., RuleFit [Friedman and Popescu, 2008], Slipper [Cohen and Singer, 1999], Pre [Fokkema, 2020]) were introduced to bridge the interpretability of a decision tree with the modeling power of a tree ensemble. Our toolchain leverages Skope-rules (https://skope-rules.readthedocs.io/) to generate logical rules from the tree ensemble. Specifically, consider any path from the root to a leaf in a tree T_j, and let $\pi_{j,1}, \ldots, \pi_{j,\ell}$ denote the predicates along that path that evaluated to true. So, for example, if the first predicate encountered in T_j, say "$\vec{c}(cvar) = 1$", evaluated to false, then $\pi_{j,1}$ = "$\vec{c}(cvar) \neq 1$". Then, SKOPE-RULES constructs a rule by conjoining $\pi_{j,1}, \ldots, \pi_{j,\ell}$, with the caveat that it limits the number of predicates included in any rule by heuristically pruning them.

Each such rule has a *precision* and *recall*, which we evaluate using a validation set held out from $\hat{NS}$ and $\hat{IS}$ during training. That is, the *recall* of a rule is the fraction of validation samples held out from $\hat{IS}$ for which the rule evaluates to true, and its *precision* is the fraction of validation samples (from $\hat{IS}$ or $\hat{NS}$) for which the rule evaluates to true that were held out from $\hat{IS}$. We further prune rules by iteratively removing conjuncts from a long rule if the precision of the resulting rule is at least 95% of the original. We then rank order rules according first to precision, and then according to recall.

4.4 Feature engineering

The utility of the rule generation described in the previous section depends critically on the features of each $\langle\vec{c}, \vec{\imath}, \vec{s}, \vec{s}'\rangle$ tuple exposed when training the tree ensemble, from which the predicates making up the decision nodes of each tree are formed. One factor that makes feature engineering especially critical here is that the SAT solver used to produce elements of $\hat{IS}$ and $\hat{NS}$ requires that the conditions defining *IS* and *NS* (i.e., conditions (9) and (10)) be presented to the SAT solver in terms of binary variables only. As such, each solution generated by the SAT solver is expressed as an assignment to these binary variables. While for some hardware logic, a binary representation of the relevant variables is most natural, for other types of logic (e.g., on integers), it is not. For this reason, we augment each binary solution returned by the SAT solver (i.e., each $\langle\vec{c}, \vec{\imath}, \vec{s}, \vec{s}'\rangle$ tuple) with additional features.

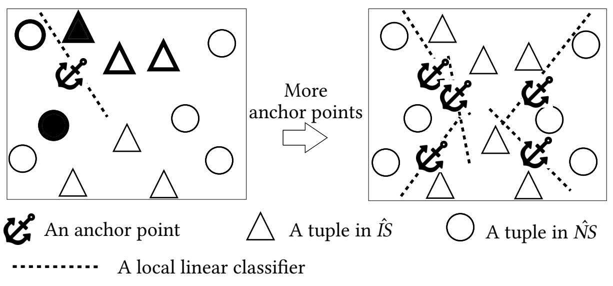


Fig. 3. Finding linear combinations of features near anchor points

- **Type-aware features**: First, we reconstruct features in a type-aware way from their binary representations. For example, if a variable was initially an integer before being reduced to a collection of binary variables in the formula presented to the SAT solver, we recover the integer value from the bit-vector solution and include it as a feature on which the tree ensemble can trained. With such type-aware features, predicates such as, e.g., $\vec{s}(svar) < 15$ can be learned in a search for simple predicates testing only a single feature, i.e., unary predicates.
- **Symmetric features**: Due to the symmetry of $\vec{s}$ and $\vec{s}'$, an interference rule could be trivially transformed to another valid interference rule by exchanging $\vec{s}$ and $\vec{s}'$. For example, when a rule is $\vec{s}(svar)[0] = 0 \wedge \vec{s}'(svar)[0] = 1$, there must be a rule $\vec{s}(svar)[0] = 1 \wedge \vec{s}'(svar)[0] = 0$. Thus, we create $|\vec{s}(svar)[i] - \vec{s}'(svar)[i]|$ for each bit i in secret.
- **Linear combinations of multiple variables**: Unary predicates, however, will be unable to naturally capture some relationships resulting in leakage. For example, if leakage happens only when $\vec{s}(svar) > \vec{c}(cvar)$, permitting only unary predicates will result in a boundary characterized point-by-point, e.g., "$\vec{s}(svar) \geq \theta \wedge \vec{c}(cvar) < \theta$" where $\theta = 1, 2, \ldots$ We thus expanded our feature

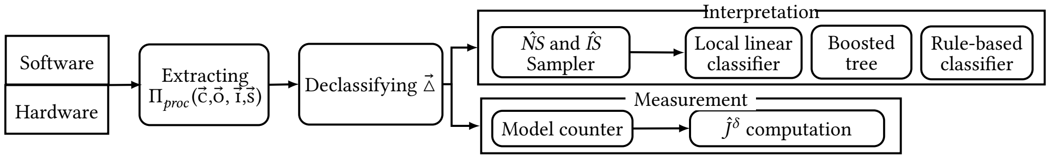

Interpretation
$\hat{NS}$ and $\hat{IS}$ Sampler
Local linear classifier
Boosted tree
Rule-based classifier
Software
Hardware
Extracting $\Pi_{proc}(\vec{c},\vec{o},\vec{\imath},\vec{s})$
Declassifying $\vec{\Delta}$
Measurement
Model counter
$\hat{J}^{\delta}$ computation

Fig. 4. DINoMe workflow

set to permit linear combinations of some features (e.g., $\vec{s}(svar) - \vec{c}(cvar)$), chosen by a linear classifier.

To accommodate branching in the procedure that results in discontinuities in the boundary between sample sets $\hat{IS}$ and $\hat{NS}$, we opted for a *local* linear classifier (e.g., Fan [1993], Ribeiro et al. [2018]). That is, we pick *anchor points*, around each of which we train a local classifier that best separates the *nearby* samples in $\hat{IS}$ and $\hat{NS}$. (See Fig. 3.) To select anchor points, we first find pairs of $\langle\vec{c}, \vec{\imath}, \vec{s}, \vec{s}'\rangle$ tuples, one from $\hat{IS}$ and one from $\hat{NS}$, that are *neighbors* in one feature (i.e., after ranking all tuples by this feature, the pair are adjacent in the ranking) and then take the pair's *midpoint* tuple as their per-feature means. We select anchors uniformly at random from these midpoints. For each anchor, we train a linear classifier using the tuples in $\hat{IS}$ and $\hat{NS}$ that are within a threshold Euclidean distance from the anchor. The linear combination of features used in this linear classifier is then added as another feature to each $\langle\vec{c}, \vec{\imath}, \vec{s}, \vec{s}'\rangle$ tuple.

5 IMPLEMENTATION

We developed DINoMe[3] for evaluating and interpreting leakage, described in Sec. 3–4, with an eye toward applying it to evaluate and understand leakage from hardware designs. Though our declassification and interpretation methodologies are not limited to hardware designs, we believe they will be most useful in complicated cases where developers need to understand the interactions between low-level and high-level code. To capture such cases, we define the procedure *proc* to be a hardware design, say written in Verilog, in its initial state but with a predefined program stored in its memory. DINoMe enables the user to annotate the configuration by marking components of the hardware state as attacker-controlled (i.e., in $Vars_{\vec{c}}$), attacker-observable (in $Vars_{\vec{o}}$), secret (in $Vars_{\vec{s}}$), or otherwise unknown to the attacker (in $Vars_{\vec{\imath}}$). DINoMe workflow for analyzing this "procedure" is illustrated in Fig. 4. Our system converts this "procedure," which we continue to denote *proc*, to a cycle-accurate logical formula Π_{proc} that characterizes hardware execution of the program and that relates $\vec{c}$, $\vec{o}$, $\vec{\imath}$, and $\vec{s}$. The user can also declare a declassification function δ that operates on the hardware state of the system (we will give examples below), from which DINoMe similarly produces a logical formula Π_{δ} that characterizes how the declassified information $\vec{\Delta}$ relates to inputs $\vec{c}$, $\vec{\imath}$, and $\vec{s}$ in the execution of *proc*. From Π_{proc} and Π_{δ}, DINoMe generates $\hat{J}^{\delta}_{n}$ for varying n (see (8)) and, if requested, sample sets $\hat{IS}$ and $\hat{NS}$ from *IS* (see (9)) and *NS* (see (10)), respectively. These sets seed the generation of the rules for interpreting leakage, as discussed in Sec. 4.

Below we discuss particular challenges we encountered when building DINoMe and how we overcame them. We focus on how to extract $\Pi_{proc}(\vec{c}, \vec{o}, \vec{\imath}, \vec{s})$ in Sec. 5.1. In Sec. 10.1, we describe simplification techniques we leverage as a preprocessing step before performing projected model counting, described in Sec. 5.2. Finally, we discuss our technique for sampling to create $\hat{IS}$ and $\hat{NS}$ in Sec. 5.3.

[3]https://github.com/DINoMe-Project/DINoMe

5.1 Extracting $\Pi_{proc}(\vec{c}, \vec{o}, \vec{\imath}, \vec{s})$

To analyze the leakage from *proc*, we need an accurate postcondition $\Pi_{proc}(\vec{c}, \vec{o}, \vec{\imath}, \vec{s})$ for *proc*. In practice, generating a postcondition for an arbitrary procedure is not trivial. Especially here, where our concern is detecting leakage from a processor implementation when running an application—i.e., the procedure *proc* includes numerous cycles of a cycle-accurate implementation of the processor logic as well as the software logic—the postcondition will be quite large.

Our general strategy to construct $\Pi_{proc}(\vec{c}, \vec{o}, \vec{\imath}, \vec{s})$ in these circumstances is to assemble it one cycle at a time. Yosys [Wolf] provides a framework to convert the Verilog code for a processor design to its internal register-transfer level (RTL) intermediate language, optimize or modify the design using a series of passes, and finally translate the design to targeted formula through its back-end pass. The SMT2 back-end pass defines a data structure for each hardware module representing the module's temporary hardware state, a function to implement the module's state transition from one cycle to the next, and an initialization function to initialize the module's state. To incorporate the software logic of *proc*, we compile the software to its hardware-readable assembly and load the assembly into the instruction memory unit.

To mark the symbolic variables, the analyst defines a configuration file to mark as symbolic each input parameter of *proc* (in this case, *svar*, *ivar*, and *cvar*), which can be a software variable located at a fixed location in the memory unit or a wire/register inside the hardware module. Our modified SMT2 backend pass in Yosys then tracks the constraints associated with this symbolic data throughout a cycle execution. Specifically, it outputs a logical postcondition $\tau_{proc}(\vec{H}^{t-1}, \vec{H}^{t})$ that relates fully symbolized hardware state $\vec{H}^{t-1} : Vars_{\vec{H}} \rightarrow Vals_{\vec{H}}$ at the end of cycle $t-1$ to the hardware state $\vec{H}^{t}$ that results from executing cycle t. Since the hardware state includes memory units, registers, etc., τ_{proc} $(\vec{H}^{t-1}, \vec{H}^{t})$ with fully symbolized $\vec{H}^{t-1}$ is too large to naively extend to cover multiple cycles. We also use the pass to generate initialization logic $\Psi^{0}_{proc}(\vec{c}, \vec{\imath}, \vec{s}, \vec{H}^{0})$ that concretely characterizes the first-cycle starting state $\vec{H}^{0}$ (upon a reset) except for the configured symbolic inputs *svar*, *ivar*, and *cvar*.

Using the transition logic, we construct a cycle-accurate postcondition Ψ^{T}_{proc} representing the logic between symbolic inputs and its internal hardware state one cycle at a time, leveraging the entire hardware state as an "observable" output of the cycle.

$$\Psi^{T}_{proc}(\vec{c}, \vec{\imath}, \vec{s}, \vec{H}^{T}) \leftarrow \Psi^{0}_{proc}(\vec{c}, \vec{\imath}, \vec{s}, \vec{H}^{0}) \wedge \bigwedge_{t=1}^{T} \tau_{proc}(\vec{H}^{t-1}, \vec{H}^{t})$$

We finally define $\Pi_{proc}(\vec{c}, \vec{o}, \vec{\imath}, \vec{s})$ by defining $\vec{o}$ in terms of the sequence of hardware states $\langle \vec{H}^{t} \rangle_{t=0}^{T}$ using a formula $\Gamma(\langle \vec{H}^{t} \rangle_{t=0}^{T}, \vec{o})$.

$$\Pi_{proc}(\vec{c}, \vec{o}, \vec{\imath}, \vec{s}) \leftarrow \Psi^{T}_{proc}(\vec{c}, \vec{\imath}, \vec{s}, \vec{H}^{T}) \wedge \Gamma(\langle \vec{H}^{t} \rangle_{t=0}^{T}, \vec{o}) \tag{11}$$

For example, in cache-based side channels, the observable parameters are whether there is a cache hit/miss during the execution, which is constructed using the values of the *s2_hit* register across the execution (as demonstrated in Sec. 6.2).

Applying a correct combination of techniques to simplify $\Pi_{proc}(\vec{c}, \vec{o}, \vec{\imath}, \vec{s})$ is critical to scaling the sampling of *IS* and *NS* to create $\hat{IS}$ and $\hat{NS}$ and to count $\left|\tilde{X}^{\delta}_{S,S'}\right|$ and $\left|\check{X}^{\delta}_{S,S'}\right|$ to compute $\hat{J}^{\delta}_{n}$. We defer discussion of these simplifications to Appendix 10.1.

To correctly measure leakage, the postcondition for *proc* must be complete and sound. Completeness means that if $\langle \vec{c}, \vec{\imath}, \vec{s}, \vec{o} \rangle$ is feasible for *proc*, then $\langle \vec{c}, \vec{\imath}, \vec{s}, \vec{o} \rangle$ satisfies $\Pi_{proc}(\vec{c}, \vec{o}, \vec{\imath}, \vec{s})$. Soundness means that if $\langle \vec{c}, \vec{\imath}, \vec{s}, \vec{o} \rangle$ is infeasible for *proc*, then $\langle \vec{c}, \vec{\imath}, \vec{s}, \vec{o} \rangle$ does not satisfy $\Pi_{proc}(\vec{c}, \vec{o}, \vec{\imath}, \vec{s})$. Here,

$\Pi_{proc}(\vec{c}, \vec{o}, \vec{\imath}, \vec{s})$ is derived from the hardware transition logic τ_{proc}. Since τ_{proc} represents how the next hardware state is derived from the previous hardware state[4] and is derived from the actual hardware design, our postcondition is consistent with the real verilog code, provided that the Yosys SMT2 backend pass is correct.

In our experiments, we selected T to ensure the termination of the execution, based on our knowledge gained by studying the CPU. A more conservative method would be to track the CPU pipeline and call the SAT solver each cycle to check whether the last instruction has certainly committed. We have confirmed that adding more cycles after the termination of the execution does not affect Π_{proc} meaningfully, as the additional cycles do not process any valid opcodes and so only trivially change the hardware state.

5.2 Measurement with declassification using projected model counting

Using `CryptoMiniSAT 5.0` as the basic solver, we implemented a counter to estimate the numerator and the denominator in the measurement $\hat{J}^{\delta}(S, S')$ in (8).

5.2.1 Computing $\hat{J}^{\delta}(S, S')$. To compute $\hat{J}^{\delta}(S, S')$, we need to count the sizes of $\tilde{X}^{\delta}_{S,S'}$ and $\check{X}^{\delta}_{S,S'}$. Directly counting $\tilde{X}^{\delta}_{S,S'}$ is not easy as the set difference operation introduces a "forall" quantifier. Fortunately, since $\left|\tilde{X}^{\delta}_{S,S'}\right| = \left|\check{X}^{\delta}_{S,S'}\right| - \left|\hat{X}^{\delta}_{S,S'}\right|$, it suffices to count $\check{X}^{\delta}_{S,S'}$ and $\hat{X}^{\delta}_{S,S'}$ for each sample pair S, S'. Intuitively, counting $\check{X}^{\delta}_{S,S'}$ could be expressed as a projected model counting task [Aziz et al., 2015] over $\langle \vec{c}, \vec{o}, \vec{\imath}, \vec{s}\rangle$ in a quantifier-free SAT problem with two copies of Π_{proc} shown in $\check{F}$ below. $\check{F}$ is translated to a CNF proposition where it uses v bit variables to represent $\langle \vec{c}, \vec{o}, \vec{\Delta}, \vec{\imath}\rangle$ and others to represent $\langle \vec{s}, \vec{s}', \vec{\imath}, \vec{\imath}'\rangle$ and auxiliary variables.

$$\begin{aligned} \check{F} \leftarrow \quad & \Big(\Pi_{proc}(\vec{c}, \vec{o}, \vec{\imath}, \vec{s}) \vee \Pi_{proc}(\vec{c}, \vec{o}, \vec{\imath}', \vec{s}')\Big) \wedge \Pi_{\delta}(\vec{c}, \vec{\Delta}, \vec{\imath}, \vec{s}) \wedge \Pi_{\delta}(\vec{c}, \vec{\Delta}, \vec{\imath}', \vec{s}') \\ & \wedge \big((\vec{s}(svar) \in S \wedge \vec{s}'(svar) \in S') \vee (\vec{s}'(svar) \in S \wedge \vec{s}(svar) \in S')\big) \end{aligned} \tag{12}$$

Following Zhou et al. [2018], two random, disjoint sets S and S' of expected size n are specified with distinct strings $p, \hat{p} \in \{0, 1\}^b$ where $n = |\mathbb{S}|/2^b$, and specifically with the constraint that for a fixed hash function, the hash of each $s \in S$ is p and the hash of each $s' \in S'$ is $\hat{p}$.

For $\hat{X}^{\delta}_{S,S'}$, we can define another projected model counting task over $\langle \vec{c}, \vec{o}, \vec{\Delta}, \vec{\imath}\rangle$ in a quantifier-free SAT problem $\hat{F}$ shown below. $\hat{F}$ uses the logical postcondition Π_{proc} twice, where the first copy is for the execution with a secret $\vec{s}(svar) \in S$ and the second checks for existence of a secret $\vec{s}'(svar) \in S'$ leading to a result $\vec{o}$ also possible with $\vec{s}$. $\hat{F}$ also checks the existence of some secret (denoted by $\vec{s}''(svar)$) in the secret set S' leading to the equivalent declassification value $\vec{\Delta}$ so that we can ensure the $\vec{s}$ and $\vec{s}'$ cannot be distinguished by $\vec{\Delta}$.

$$\begin{aligned} \hat{F} \leftarrow \quad & \Pi_{proc,\delta}(\vec{c}, \vec{o}, \vec{\Delta}, \vec{\imath}, \vec{s}) \wedge \vec{s}(svar) \in S \\ & \wedge \Pi_{proc}(\vec{c}, \vec{o}, \vec{\imath}', \vec{s}') \wedge \vec{s}'(svar) \in S' \\ & \wedge \Pi_{\delta}(\vec{c}, \vec{\Delta}, \vec{\imath}'', \vec{s}'') \wedge \vec{s}''(svar) \in S' \end{aligned} \tag{13}$$

5.2.2 Optimizations for counting $\tilde{X}^{\delta}_{S,S'}$ and $\check{X}^{\delta}_{S,S'}$. Enumerating all solutions to (12) and (13) using a solver is intractable. To estimate the number of solutions to each instead, we used the approximate model counting technique due to Chakraborty et al. [2013], specifically the approach taken by Soos

[4]Unlike software, hardware code (e.g., verilog) does not use do-while loops within one cycle for which the number of iterations is determined dynamically. In our case studies, we found that the one-cycle logic for BOOM is correspondingly simple, enabling the completeness and soundness of τ_{proc}.

$$\text{E-Solver with } H \text{ and } p \text{ generates } \langle\vec{c}, \vec{\imath}, \vec{s}, \vec{s}', \vec{o}, \vec{\Delta}\rangle \text{ satisfying} \\ \Pi_{proc,\delta}(\vec{c}, \vec{o}, \vec{\Delta}, \vec{\imath}, \vec{s}) \wedge \Pi_{proc,\delta}(\vec{c}, \vec{o}', \vec{\Delta}, \vec{\imath}', \vec{s}') \wedge \vec{o} \neq \vec{o}' \wedge H(\langle\vec{c}, \vec{\imath}, \vec{s}, \vec{s}'\rangle) = p \tag{16}$$

$$\text{F-Solver cancels } \langle\vec{c}, \vec{\imath}, \vec{s}, \vec{s}', \vec{o}, \vec{\Delta}\rangle \text{ satisfying (16) if there is some } \vec{\imath}'' \text{ satisfying} \\ \Pi_{proc,\delta}(\vec{c},\vec{o},\vec{\Delta},\vec{\imath}'',\vec{s}') \tag{17}$$

Fig. 5. Generating examples in $\hat{IS}$ using EF-solver

and Meel [2019]. That is, by specifying a randomly selected hash function $\hat{H}^{\hat{b}} : \{0,1\}^v \to \{0,1\}^{\hat{b}}$ and an output $\hat{p} \in \{0,1\}^b$ as an additional constraint, we can estimate $\left|\hat{X}^{\delta}_{S,S'}\right|$ using the average value of multiple estimations of $\left|\hat{Z}^{\hat{p}}_{S,S'}\right|$ with some error ϵ and confidence δ (i.e., $\left|\hat{X}^{\delta}_{S,S'}\right| \approx \left|\hat{Z}^{\hat{p}}_{S,S'}\right| \times 2^{\hat{b}}$). Similarly, we could estimate $\left|\check{X}^{\delta}_{S,S'}\right|$ using $\check{Z}^{\check{p}}_{S,S'}$.

$$\check{Z}^{\check{p}}_{S,S'} = \left\{\langle\vec{c}, \vec{o}, \vec{\Delta}, \vec{\imath}\rangle \;\middle|\; \langle\vec{c}, \vec{o}, \vec{\Delta}, \vec{\imath}\rangle \in \check{X}^{\delta}_{S,S'} \wedge \check{H}^{\check{b}}(\langle\vec{c}, \vec{o}, \vec{\Delta}, \vec{\imath}\rangle) = \check{p} \right\} \tag{14}$$

$$\hat{Z}^{\hat{p}}_{S,S'} = \left\{\langle\vec{c}, \vec{o}, \vec{\Delta}, \vec{\imath}\rangle \;\middle|\; \langle\vec{c}, \vec{o}, \vec{\Delta}, \vec{\imath}\rangle \in \hat{X}^{\delta}_{S,S'} \wedge \hat{H}^{\hat{b}}(\langle\vec{c}, \vec{o}, \vec{\Delta}, \vec{\imath}\rangle) = \hat{p} \right\} \tag{15}$$

This optimization for model counting will limit the number of calls to the SAT solver by constraining the number of solutions available, and thus make the counting more scalable for large set size. Thus, $\hat{J}^{\delta}$ (S, S') is estimated using the average value of $1 - \left|\hat{Z}^{\hat{p}}_{S,S'}\right| / \left|\check{Z}^{\check{p}}_{S,S'}\right|$ for various $\hat{p}, \check{p}$.

Our primary departure from the implementation by Soos and Meel [2019] lies in utilizing task-specific properties in our counting tasks to reduce redundant effort in solution searching. Specifically, since $\hat{X}^{\delta}_{S,S'} \subseteq \check{X}^{\delta}_{S,S'}$, we ensure that $\hat{X}^{\delta}_{S,S'} \cap \check{Z}^{\check{p}}_{S,S'} \subseteq \hat{Z}^{\hat{p}}_{S,S'}$ in our counting by defining $\hat{H}^{\hat{b}}(\langle\vec{c}, \vec{o}, \vec{\Delta}, \vec{\imath}\rangle)$ to be the $\hat{b}$-bit prefix of $\check{H}^{\check{b}}(\langle\vec{c}, \vec{o}, \vec{\Delta}, \vec{\imath}\rangle)$ for $\hat{b} \leq \check{b}$. Then once we have generated solutions in $\check{Z}^{\check{p}}_{S,S'}$, we speed up finding solutions in $\hat{Z}^{\hat{p}}_{S,S'}$ for $\hat{b} = \check{b}$ (and so $\hat{p} = \check{p}$) by first checking each solution in $\check{Z}^{\check{p}}_{S,S'}$ to see if it satisfies $\hat{F}$ (i.e., is in $\hat{X}^{\delta}_{S,S'} \cap \check{Z}^{\check{p}}_{S,S'}$). Only if insufficient solutions are found with $\hat{b} = \check{b}$ is $\hat{b}$ reduced and the solver used to generate additional solutions in $\hat{Z}^{\hat{p}}_{S,S'}$ for $\hat{p}$ a $\hat{b}$-bit prefix of $\check{p}$.

In Sec. 6, we set the error $\epsilon = 0.4$ and confidence $\delta = 0.9$ in this method to estimate the sizes of $\hat{X}^{\delta}_{S,S'}$ and $\check{X}^{\delta}_{S,S'}$, from which $\hat{J}^{\delta}(S, S')$ is estimated using (8). For each set size n, we compute $\hat{J}^{\delta}_{n}$ using ≥ 100 hash functions, i.e., implicit selection of pairs S, S' of expected size n.

5.3 Sampling $\hat{NS}$ and $\hat{IS}$ for interpretable learning

Similar to the counting process, to construct $\hat{NS}$ and $\hat{IS}$, the sampler will select hash functions H randomly from a family and output values p randomly from its range to solve for tuples $\langle\vec{c}, \vec{\imath}, \vec{s}, \vec{s}'\rangle$ for which $H(\langle\vec{c}, \vec{\imath}, \vec{s}, \vec{s}'\rangle) = p$ (and are in NS or IS, respectively). In the following experiments, we will generate up to 100,000 solutions for each of $\hat{NS}$ and $\hat{IS}$, where 70% used for training and 30% used for validation.

We cannot directly encode set difference, used in (9) and (10), using an equivalent quantifier-free formula. To implement a sampler to generate solutions in the set difference, we will use one solver ("E-Solver") to search for candidate solutions and another ("F-Solver") cancel candidates; this is a commonly used algorithm for an SMT solver to solve exist-forall problems (e.g., see Dutertre [2015]).

Here, we will illustrate sampling *IS*, while sampling *NS* is similar. The sampler first uses the E-Solver to generate feasible solutions $\langle \vec{c}, \vec{i}, \vec{s}, \vec{s}' \rangle$ (see (16)) that guarantee, for an attacker's chosen $\vec{c}$, the observable value $\vec{o}$ derived from $\vec{s}$ with $\vec{i}$ could be different from an observable $\vec{o}'$ generated by $\vec{s}'$ with some $\vec{i}'$ when the declassified value $\vec{\Delta}$ is the same. However, it does not guarantee the $\vec{o}$ is never feasible for $\vec{s}$. To further test whether the $\langle \vec{c}, \vec{i}, \vec{s}, \vec{s}' \rangle$ is in $\hat{IS}$, we use the F-Solver to test whether $\langle \vec{s}', \vec{i}'' \rangle$ for some $\vec{i}''$ could generate $\vec{o}$ with $\langle \vec{s}, \vec{i} \rangle$ when they share the declassification value $\vec{\Delta}$, to check whether we need to cancel the solution. That is, $\langle \vec{c}, \vec{i}, \vec{s}, \vec{s}' \rangle$ satisfying (16) but not (17) will be included in $\hat{IS}$.

After generating enough $\langle \vec{c}, \vec{i}, \vec{s}, \vec{s}' \rangle$ tuples in $\hat{NS}$ and $\hat{IS}$, the interpretation module trains local support vector machine (SVM) classifiers [Fan et al., 2008] around each of 50 anchor points, after ruling out data whose normalized Euclidean distance (i.e., after scaling each attribute to a value between 0 and 1, use Euclidean distance divided by the number of attributes) is more than 0.2 from the anchor. Then a logistic regression model for *NS* and *IS* is learned using a gradient boosted tree implementation *xgboost* [Chen and Guestrin, 2016]. To generate the interpretable models, we implemented the rule learner using Skope-rules.

6 CASE STUDIES

In this section, we illustrate DINoMe by describing its application to the BOOM core (https://github.com/riscv-boom/riscv-boom), an open-source RISC-V core that is susceptible to cache-based side channels and Spectre attacks. The goal of these case studies is to illustrate our methodology and to show how it can be useful to system analysts. Our method is also applicable to other side channels, not only cache-based ones. Analysts can specify the secret to protect and define their side channels using attacker-controlled and attacker-observable variables but, critically, not the specific attacker algorithm.

- We applied DINoMe to evaluate cache-based side-channel leakage due to secret-dependent memory accesses. With different BOOM configurations (i.e., number of cache ways w and whether to share memory), the case studies shows how $\hat{J}_n^\delta$ curves reveal the effects of the configurations on the leakage. We also implemented and evaluated two possible mitigations, namely ScatterCache [Werner et al., 2019] and PhantomCache [Tan et al., 2020], which reduce but not eliminate the cache leakage. Our measurements using $\hat{J}_n^\delta$ illustrate which mitigation is better for a specific BOOM setting.
- We used DINoMe to assess leakage via cache-based side channels from a modular exponentiation function commonly used in cryptographic algorithms. The rule-based interpretation explains how to choose attacker-controlled variables and which portion of the secret is leaked.
- We evaluated software code snippets causing speculative execution. It demonstrates how to use declassification to focus on leakage caused by speculative execution (i.e., by declassifying other leakage to reveal it) and how to generate efficient interpretable rule set. We found that some software with a short speculation window is insufficient to cause memory leakage in the latest version of BOOM.

6.1 BOOM configurations

In the following experiments, we used pocket-size hardware modules to replace the modules in the BOOM v2.2.3 configuration. A simplified diagram is shown in Fig. 6. Analyzing artificially small but otherwise faithful configurations of a system is not uncommon in model checking, for example (e.g., Ball et al. [2004], Pnueli et al. [2002]). Specifically, we set the cache line size to *bbytes* = 64B and the total L1 data cache size to 1KB (16 cache lines in total). We then varied the cache ways w and sets c (i.e., subject to $w \times c = 16$) in Sec. 6.2 but used a fixed setting $c = 2$, $c = 8$ for other

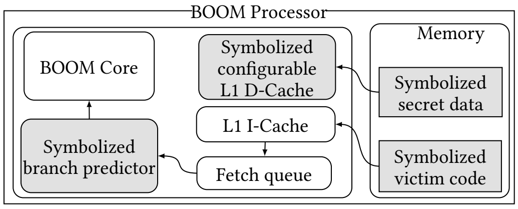


Fig. 6. BOOM configuration

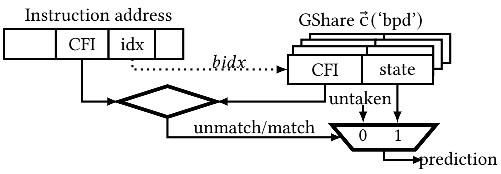


Fig. 7. GShare branch predictor's logical architecture

evaluations. BOOM only provides a configurable associative L1 cache module using a random replacement policy. To compare different cache designs, we implemented two side-channel-resistant cache modules, as described in Sec. 6.3. For the main memory, we set the memory size to 4KB and thus a memory address is only 12 bits. For evaluation purposes, we used the upper half of the memory address space as instruction memory and the lower half as data memory. To simplify the following analysis, we removed the page table walker module and assumed virtual addresses were the same as physical addresses. For the instruction fetch, we set the fetch width to 4 and configured the L1 instruction cache to a 1KB, 8-set, 2-way cache with a customized prefetching module that preloaded the software workload at the first cycle. One feature of BOOM is that it supports speculative execution, with which we will experiment in Sec. 6.5. Speculative execution leverages a *branch predictor*, for which we used the GShare branch predictor. The logical structure of GShare is shown in Fig. 7. When a prediction request arrives for a branch instruction, the GShare predictor derives a value *bidx* from the certain bits (denoted 'idx' in Fig. 7) in the instruction address and an instruction history register and then uses *bidx* to index into a table to which we refer as 'bpd'. Each entry of the 'bpd' table includes a label called 'CFI' and a 2-bit 'state', of which one bit indicates whether the entry holds a strong or weak prediction and the other bit holds that prediction (i.e., whether the branch will be taken or not). If the 'bpd{*bidx*}.CFI' value matches the 'CFI' portion of the instruction address, then the predictor uses the 'bpd{*bidx*}.state' value to make a branch prediction. The GShare predictor will globally tune entries based on executions in any user's domain. Thus, an attacker can easily affect the 'bpd' table before victim's execution, and so we include 'bpd' in $Vars_{\vec{c}}$. In our evaluation, we fix the number of 'bpd' entries to 4 so that only 2 bits in the instruction address are used as 'idx' while another 2 bits (=$\log_2$(fetch width)) are used as its 'CFI' label.

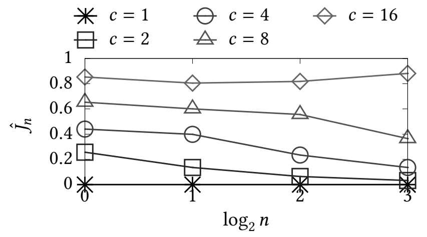
In the following case studies, we added the 'bpd' table in the GShare module to $Vars_{\vec{c}}$ and registers in the L1 data cache module including the cache metadata, the replacement state (i.e., the linear-feedback shift register (LFSR) for the random replacement policy), and the memory-to-cache mapping (if using a nonfixed mapping) to $Vars_{\vec{I}}$.

In cache-based side channel attacks, $\vec{c}$ and $\vec{o}$ are not directly represented in the hardware state or in victim's code, and so it is necessary to define them through an adversary model. We assume that the adversary has access to 16 memory blocks $block_1$, $block_2$, …, $block_\ell$, …, $block_{16}$ aligned to cache lines, which is sufficient to control the cache as our L1 data cache consists of only 16 cache lines in our experiments. Specifically, $\vec{c}$('load')$[\ell]$ indicates whether the adversary loads (1) or flushes (0) $block_\ell$, while $\vec{o}$('hit')$[\ell]$ indicates whether the adversary observes a cache hit (1) or miss (0) when accessing $block_\ell$. Appendix 10.2 illustrates how to automatically construct them.

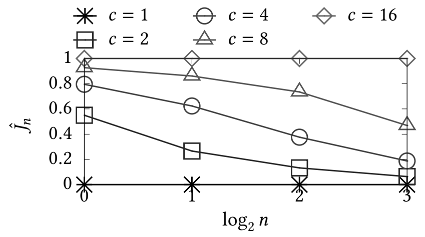
6.2 Cache-based side channels

In this section, we evaluate cache-based side channels under different memory isolation and cache configurations.

6.2.1 Without shared memory.



(a) Symbolic $\vec{c}$('load')



(b) $\forall \ell : \vec{c}(\text{'load'})[\ell] = 1$

Fig. 8. $\hat{J}_n$ for Prime+Probe attacks

```
proc (c⃗, ı⃗, s⃗)
  li s0, 0x2000010
  add s1, s0, s⃗('secret')
  sll s1, s1, 6
  lbu a2, 0(s1)
```

Here, we target a victim's RISC-V assembly *proc* to access a secret-indexed memory block not shared with the attacker, by setting the base address in `s0` to a value `0x2000010`, in contrast to the one used in attacker's process *acc* (see Appendix 10.2). We experimented with different numbers of cache sets c including $c = 1$ (i.e., 16-way, 1-set, fully associative), $c = 2$ (i.e., 8-way, 2-set), $c = 4$ (i.e., 4-way, 4-set), $c = 8$ (2-way, 8-set), and $c = 16$ (i.e., 1-way, 16-set, direct-mapped). As shown in Fig. 8(a), $\hat{J}_n$ increases when the number of sets increases. Specifically, there is no leakage ($\hat{J}_n = 0$ for all n) when $c = 1$. Using fewer cache sets, each cache set is shared by more memory blocks, and so an attacker will have more difficulty distinguishing one execution from others. When $1 < c < 16$, $\hat{J}_n$ decreases as n grows, since the attacker can learn only $log_2(c)$ bits about the secret and thus may be unable to distinguish secrets in large sets (i.e., large n).

An example *interference* rule for *IS* generated as described in Sec. 4 with the highest precision (1.00) and a recall ≈ 0.04 in a 2-way, 8-set cache is:

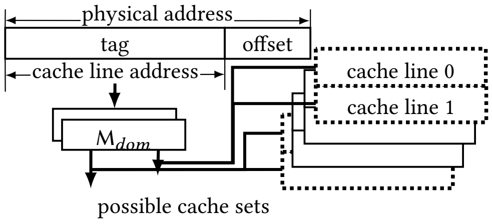
$$\left\{ \begin{array}{lcl} & \vec{s}(\text{'secret'})[2] \geq 1 & \wedge \quad \vec{s}(\text{'secret'})[1] < 1 \\ \wedge & \vec{s}(\text{'secret'})[0] \geq 1 & \wedge \quad \vec{s}'(\text{'secret'})[1] \geq 1 \end{array} \right\} \wedge \left\{ \begin{array}{ll} & \vec{c}(\text{'load'})[5] \geq 1 \\ \wedge & \vec{c}(\text{'load'})[13] \geq 1 \end{array} \right\} \tag{18}$$

In this rule, the $\vec{s}$ and $\vec{s}'$ conjuncts concretize the least significant 3 bits of $\vec{s}$('secret') (i.e., $\vec{s}(\text{'secret'}) \equiv 5 \bmod 8$) and the lowest bit of $\vec{s}'$('secret') (i.e., $\vec{s}'(\text{'secret'}) \equiv 0 \bmod 2$). The $\vec{c}$ conjuncts are $\vec{c}(\text{'load'})[5] \geq 1$ and $\vec{c}(\text{'load'})[13] \geq 1$; note that $13 \equiv 5 \bmod 8$. That is, an attacker could load all blocks $block_\ell$ with $\ell \equiv 5 \bmod 8$ into cache to distinguish a secret $\vec{s}(\text{'secret'}) \equiv 5 \bmod 8$ from $\vec{s}'(\text{'secret'}) \bmod 8 \in \{0, 2, 4, 6\}$.

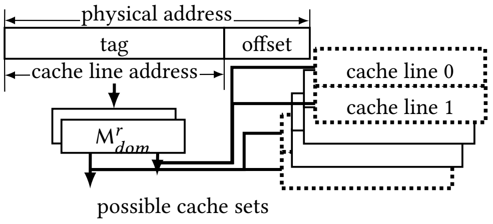
Our approach could not directly represent $\vec{c}(\text{'load'})[\ell] \equiv \vec{s}(\text{'secret'}) \bmod c$. So, the trees in the model split the dataset based on the cache set index. As such, there were many other top-ranking rules similar to (18), each focusing on one residue class of the secret value modulo c where $c = 8$ and constraining $\vec{c}(\text{'load'})[\ell] = 1$ for all ℓ with that residue class modulo c. Each such rule works for $\frac{1}{8}$ of $\vec{s}$'s domain and $\frac{1}{2}$ of $\vec{s}'$'s domain, thus only for $\frac{1}{8} \times \frac{1}{2} \approx 0.06$ of secret pairs. The recall rate $0.04 < 0.06$ indicates that priming the corresponding cache set ensures (i.e., precision = 1.0) the interference but is not necessary to cause it.

Analogously, we can generate rules for the *noninterference* set *NS*, as well. One example with precision 1.0 (i.e., that ensures noninterference) and recall 0.11 constrains the secret's least-significant 3 bits to be the same for $\vec{s}$ and $\vec{s}'$:

$$\begin{array}{ll} & |\vec{s}(\text{'secret'})[2] - \vec{s}'(\text{'secret'})[2]| < 1 \\ \wedge & |\vec{s}(\text{'secret'})[1] - \vec{s}'(\text{'secret'})[1]| < 1 \\ \wedge & |\vec{s}(\text{'secret'})[0] - \vec{s}'(\text{'secret'})[0]| < 1 \end{array} \tag{19}$$



(a) ScatterCache



(b) PhantomCache ($r = 2$)

Fig. 10. Cache modules in 2-way, 4-set configure

This analysis illustrates that an attacker can easily distinguish $\vec{s}$('secret') and $\vec{s}'$('secret') when priming a cache set used by $\vec{s}$('secret') or $\vec{s}'$('secret') but not both. It is therefore safe to assume that the attacker will Prime the cache using all its controlled memory blocks to maximize the chances for leakage. The $\hat{J}_n$ measure under this specific attack is shown in Fig. 8(b). The worst case will leak all of the 4-bit secret when using high-granularity memory-to-cache mapping, i.e., where $c = 16$.

6.2.2 With shared memory. To evaluate the leakage due to shared memory (i.e., with Flush+Reload attacks), we allow the attacker to control and observe all memory blocks used by the victim by setting the base to `0x2000000` in *proc* instead of to `0x2000010`. The $\hat{J}_n$ curves are similar and close to 1 for all settings, indicating that the leakage does not have much correlation with w. An example rule for interference derived using the methodology of Sec. 4, having a precision of 1.0 and recall of ≈ 0.04, is

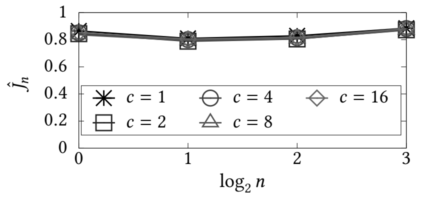


Fig. 9. $\hat{J}_n$ for Flush+Reload attacks with symbolic $\vec{c}$('load')

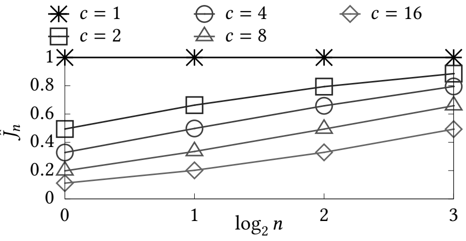
$$\vec{s}'(\text{'secret'}) < 2 \wedge \vec{s}'(\text{'secret'}) \geq 1 \wedge \vec{c}(\text{'load'})[1] < 1 \tag{20}$$

That is, if $\vec{s}'$('secret') = 1 then $\vec{c}$('load')[1] = 0 results in interference. Indeed, the other top-ranked rules for this example (not shown) were roughly 32 similar rules, each one setting $\vec{c}$('load')[ℓ]=0 for a specific secret value $\vec{s}$('secret') = ℓ or $\vec{s}'$('secret') = ℓ. The intuition behind these rules is that an attacker can precisely detect if $\vec{s}$('secret') = ℓ by setting $\vec{c}$('load')[ℓ] = 0 (i.e., Flushing $block_\ell$ so he can later Reload it), and similarly for $\vec{s}'$('secret'). Going further, if an attacker sets $\vec{c}$('load')[ℓ] = 0 for all ℓ, he can detect the victim's access to any $block_\ell$, where $\hat{J}_n = 1$ for all n.

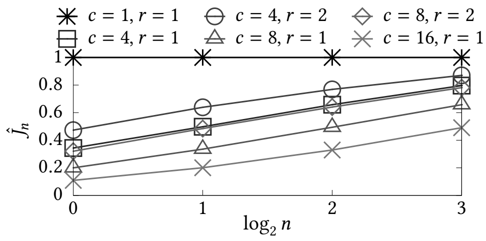
6.3 Side-channel-resistant cache designs

To demonstrate the power of DINoMe in comparing different implementations, we evaluate two cache designs for mitigating side channels, namely ScatterCache [Werner et al., 2019] and PhantomCache [Tan et al., 2020]. Unfortunately, Verilog specifications of these are unavailable, and so we implemented two simplified cache modules (which we continue to refer to as ScatterCache and PhantomCache) in BOOM following their paper designs.

ScatterCache maps a memory block to a cache line using a cryptographic index derivation function computed using the block's physical address and a private key. As shown in Fig. 10(a), to simulate this index derivation without choosing a concrete function, we use a symbolic look-up table denoted by M_{dom} per security domain dom ($dom = 0$ denotes the victim's domain and $dom = 1$ denotes the attacker's) to store the mapping from memory address to cache line. For security domain dom, its access to memory contents at physical address $paddr$ and so with block address $baddr = \lfloor paddr/bbytes \rfloor$ is mapped to cache lines with way index k and set index $j = M_{dom}\{baddr\}\{k\}$ for



(a) ScatterCache



(b) PhantomCache

Fig. 11. Memory sharing enabled with $\forall \ell : \vec{c}(\text{'load'})[\ell] = 0$ (Flush+Reload attack)

$k = 0, 1, \ldots, w-1$. Similarly, for PhantomCache, we used a domain-specific memory-to-cache mapping (shown in Fig. 10(b)) represented by M^r_{dom} to allow a memory block to use cache lines in *up to r* cache sets indexed by $M^r_{dom}\{baddr\}\{k\}$ for $k = 0, 1, \ldots, r$.[5] In the following evaluation, we have $M_{dom}, M^r_{dom} \in Vars_{\vec{1}}$.

6.3.1 Random memory-to-cache mappings. First, we experimented without memory sharing when assuming the memory-to-cache mapping is completely unknown to the attacker. We ended up with $\hat{J}_n = 0$ for all n in both ScatterCache and PhantomCache. The attacker cannot tell which memory blocks are accessed by the victim, as a memory block could be mapped to any cache line if the mapping is unknown. Thus, we focused on the leakage analysis when memory sharing is enabled.

Intuitively, Flush+Reload is the best attacker strategy for a normal cache design when memory sharing is enabled. However, for a new cache design, it may not be clear that it is still the best. Our leakage rules provide some insight for ScatterCache and PhantomCache. For example, one top-ranking rule for ScatterCache, with precision ≥ 0.80 and recall of ≈ 0.02, is:

$$\begin{aligned} &\vec{s}(\text{'secret'})[3] \geq 1 \;\wedge\; \vec{s}(\text{'secret'})[2] < 1 \;\wedge\; \vec{s}(\text{'secret'})[1] < 1 \;\wedge\; \vec{s}(\text{'secret'})[0] < 1 \\ \wedge\; &\vec{1}(M_0\{8\}\{1\}) \geq 5 \;\wedge\; \vec{1}(M_1\{8\}\{1\}) \geq 5 \;\wedge\; \vec{c}(\text{'load'})[8] < 1 \end{aligned} \tag{21}$$

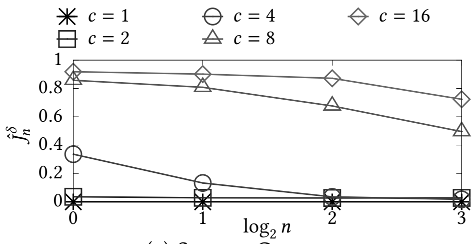
This rule is similar to (20) but with some additional predicates about M_0. Specifically, (21) adds $\vec{1}(M_0\{8\}\{1\}) \geq 5 \wedge \vec{1}(M_1\{8\}\{1\}) \geq 5$ to the rule when setting $\vec{c}(\text{'load'})[8] = 0$ (i.e., attacker Flushes $block_8$) and $\vec{s}(\text{'secret'}) = 8$, which indicates that the $block_8$ should occupy line $k = 1$ in set $j = 5$ in both the victim's and attacker's domains to ensure leakage about whether $\vec{s}(\text{'secret'}) = 8$ when the attacker Reloads $block_8$.

Thus, an attacker should Flush+Reload all blocks that could share cache lines between victim's and attacker's domain to cause more leakage. Since the memory-to-cache mapping is unknown, an attacker may Flush+Reload all shared memory blocks. The resulting $\hat{J}_n$ is shown in Fig. 11(a) for ScatterCache and Fig. 11(b) for PhantomCache. $\hat{J}_n$ is high when n is large, indicating the attacker can precisely determine $\vec{s}(\text{'secret'})$ when leakage occurs. Our results indicate that lower cache set granularity leaks more: In Fig. 11(a), $c = 1$ leaks the most, which is similar to the normal cache. When $c > 1$, the leakage is reduced.

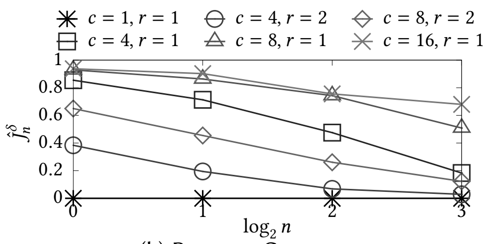
Overall, with same cache set granularity, $\hat{J}_n$ is higher with PhantomCache with $r = 2$ than PhantomCache with $r = 1$ and ScatterCache. This is because setting $r = 2$ allows one physical address to be mapped to more cache sets and so gains more chance to share cache lines across domains.

We also see that $\hat{J}_n$ for '$c = 8, r = 2$' is close to that for '$c = 4, r = 1$', as randomly mapping to 2 out of 8 sets is similar to mapping to 1 out of 4 cache sets. Our evaluation results suggests

[5]In contrast to the original paper [Tan et al., 2020], we do not force each memory block to map to *r unique* cache sets, i.e., we do not constrain $M^r_{dom}\{baddr\}\{k\} \neq M^r_{dom}\{baddr\}\{k'\}$ for $k \neq k'$.

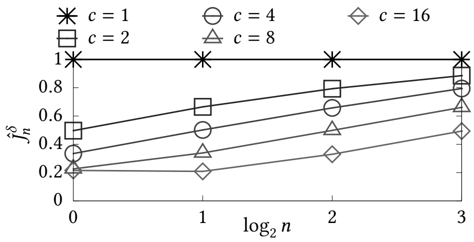


(a) ScatterCache

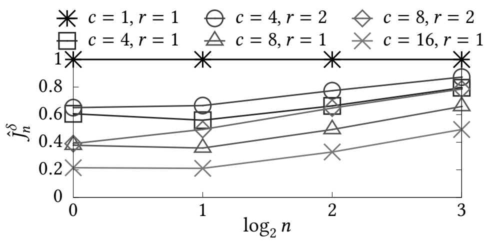


(b) PhantomCache

Fig. 12. Memory sharing disabled (Prime+Probe attack), $\vec{\triangle}$('info') $\leftarrow \vec{\imath}(\mathsf{M})$ (or $\vec{\imath}(\mathsf{M}^r)$)



(a) ScatterCache



(b) PhantomCache

Fig. 13. Memory sharing enabled (Flush+Reload attack), $\vec{\triangle}$('info') $\leftarrow \vec{\imath}(\mathsf{M})$ (or $\vec{\imath}(\mathsf{M}^r)$)

that ScatterCache and PhantomCache eliminate side-channel leakage when there is no shared memory and largely restrict it when there is shared memory, if the address-to-cache mapping is random and remains unknown to the attacker.

6.3.2 Declassifying the memory-to-cache mapping. When $\vec{\imath}(\mathsf{M})$ is unknown to the attacker, our previous analysis shows that cache-based side channels are mitigated. Werner et al. [2019] also discussed the possibility of this mapping being disclosed to the attacker, however, through a profiling procedure. If we declassify $\vec{\imath}(\mathsf{M})$, the interference $\hat{J}_n^\delta$ will increase: Fig. 12(a) shows $\hat{J}_n^\delta$ due to Prime+Probe attacks in this case, and Fig. 13(a) shows the impact of this declassification on Flush+Reload attacks.

Similarly, using $\vec{\triangle}$('info') $\leftarrow \vec{\imath}(\mathsf{M}^r)$, we evaluate PhantomCache's leakage when the random mapping is declassified; results are shown in Fig. 12(b) and Fig. 13(b). Comparing Fig. 12(b) and Fig. 12(a), PhantomCache's leakage (measured by $\hat{J}_n^\delta$) for unshared memory is higher than ScatterCache's when $r = 1$. The strength of PhantomCache is revealed when r increases, since it allows memory blocks to map to more than one cache set. Specifically, the leakage for ScatterCache's '$c = 4$' is much less than PhantomCache's '$c = 4$, $r = 1$' but is similar to PhantomCache's '$c = 4$, $r = 2$'. However, PhantomCache with $r = 2$ provides weaker protection for Flush+Reload than PhantomCache with $r = 1$ and ScatterCache.

6.4 Leaking exponent in modular exponentiation

The evaluations in Sec. 6.2 and Sec. 6.3 focused on whether the adversary could detect the victim's access to a particular memory block, which is a well-known vector of information leakage. To further demonstrate the utility of our framework in measuring this type of leakage, here we consider a classic example whereby the secret is not a memory address, but rather is a cryptographic secret that, due to the algorithm in use, can influence the victim's cache footprint.

The particular example we evaluate here is modular exponentiation as used in algorithms such as RSA. A textbook implementation of modular exponentiation uses a sliding-window method that is known to leak information in caches [Bernstein et al., 2017, Zhang et al., 2012]. As shown in Fig. 14(a), the algorithm leverages some small powers $b[k]$ of a base b (where $k < 2^W - 1$) to compute a larger power. Accesses to those precomputed powers is determined by the window-sized segment d_i of the private key d in each loop iteration i. First, this procedure will leak via the cache whether d_i is zero. Second, since the precomputed elements are addressed by d_i, an attacker may identify up to $\log_2 c$ bits about d_i if those precomputed powers map to different cache sets.

To evaluate the one-round leakage of Fig. 14(a), we used the RISC-V assembly shown in Fig. 14(b) in BOOM with a 2-way, 8-set cache ($c = 8$). The $\hat{J}_n$ measure shown in Fig. 15(a) indicates that the amount of leakage for one loop iteration i is limited, when $W \leq 4$ and so the precomputed b only uses up to $4 \times 2^4 = 64$ bytes (i.e., one cache line). When $4 < W < 8$, the side channel will leak more about d_i when W increases. Thus, choosing $W = 4$ is the best choice to protect the secret in our cache configuration.

```
function MODEXP(b,d)
    e ← 1
    for i ← n to 1 do
        e ← e × e mod M
        if d_i ≠ 0 then
            e ← e × b[d_i]
        end if
    end for
return e
end function
```

(a) Algorithm

```
proc(c⃗, i⃗, s⃗)
  li sp, 0x80000400
  li  a0,1
  li a2, M
  li a3, s⃗(d_i)
.oneIteration:
  mulw  a0,a0,a0
  remw  a0,a0,a2
  beqz  a3,.NextIteration
  sll  a5,a3,2
  add  a5,sp,a5
  lw  a5,0(a5)
  mulw  a0,a0,a5
  remw  a0,a0,a2
```

(b) Assembly for one iteration

Fig. 14. Sliding window modular exponentiation. d is the private key where each d_i ($i = 1, \ldots, n$) is a W-bit value.

To further diagnose the cause of leakage, we generated the interference rules for $W = 1$, $W = 4$, and $W = 8$. When $W = 1$, we obtain a single rule with precision and recall of 1.0, namely

$$\vec{c}(\text{'load'})[0] \geq 1 \wedge \vec{c}(\text{'load'})[8] \geq 1$$

This has no $\vec{s}$ or $\vec{s}'$ related conjuncts, indicating that the 1-bit secret d_i is fully leaked when an attacker PRIMES one cache set. In contrast, when $W = 4$, the top rules (precision of 1.0, recall ≥ 0.5) include some $\vec{s}$ or $\vec{s}'$ related conjuncts, constraining the secret value to be zero, e.g.,

$$\vec{s}(d_i) < 1 \wedge \vec{c}(\text{'load'})[0] \geq 1 \wedge \vec{c}(\text{'load'})[8] \geq 1$$

That is, it only leaks whether it is zero or not for a 4-bit secret.

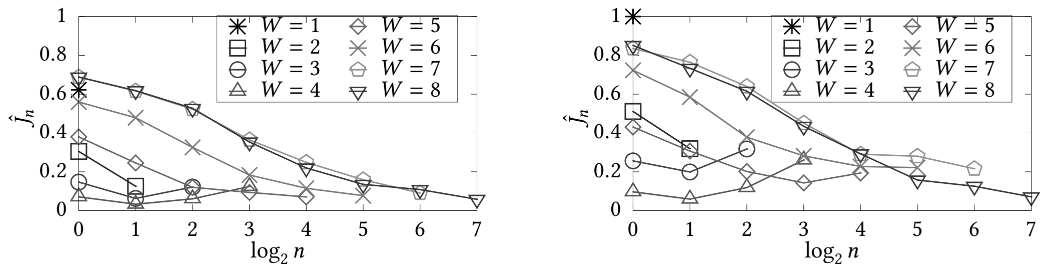
When $W > 4$, however, the most important cause of leakage changes from whether a memory access happens to which cache set is used by d_i. For example, when $W = 8$, one highly ranked rule (precision of 1.0, recall ≥ 0.04) is

$$\begin{aligned} &\vec{s}'(d_i)[6] < 1 \quad \wedge \quad \vec{s}'(d_i)[5] \geq 1 \quad \wedge \quad \vec{s}'(d_i)[4] < 1 \quad \wedge \quad \vec{s}(d_i)[4] \geq 1 \\ \wedge \quad &\vec{c}(\text{'load'})[10] \geq 1 \quad \wedge \quad \vec{c}(\text{'load'})[2] \geq 1 \end{aligned} \tag{22}$$

which indicates that the attacker can distinguish an $\vec{s}'(d_i)$ with $\vec{s}'(d_i)[4:6] = 2$ from an $\vec{s}(d_i)$ with $\vec{s}(d_i)[4:6] \in \{1, 3, 5, 7\}$ if the attacker PRIMES cache set 2. Similar to the analysis in Sec. 6.2.1, rules for $W = 8$ illustrate that an attacker can reveal the cache set used by the victim (e.g., secret bits 4-6) when priming all cache sets.

6.5 Cache-based side channels in speculative execution

SPECTRE and its variants have received widespread attention in recent years. In a SPECTRE attack, a CPU predicts the outcome of a conditional branch and executes instructions based on that prediction to reduce delays incurred by those instructions if its prediction was correct. However, even if the prediction is incorrect, then some changes to the hardware state caused by speculative execution will persist even after the mispredicted computations have been discarded. These changes propagate information to exploitable cache-based side channels, allowing the attacker to steal it.



(a) Symbolic $\vec{c}$('load') (b) $\forall \ell : \vec{c}(\text{'load'})[\ell] = 1$

Fig. 15. $\hat{J}_n$ for MODEXP in 2-way, 8-set cache

```
conditionalAccess(offset, arr1.size)
  if (offset < arr1.size)
    tmp ← arr2[(arr1[offset] × 64) & 1023]
    declassify(arr1[offset])
```

(a) Conditional memory access

```
victimFunc(offset,secret,arr1.size)
  arr1[offset] ← secret
  arr1.size← (arr1.size × 257) mod 256
  arr1.size← (arr1.size × 257) mod 256
  conditionalAccess(offset, arr1.size)
```

(b) Bounds check with long dependency

```
victimFunc(offset,secret)
  arr1[offset] ← secret
  read arr1.size from memory;
  conditionalAccess(offset, arr1.size)
```

(c) Bounds check with short dependency

```
1 .shortDependency:
2   lbu  a0, 0x100(t3)
```

(d) Short speculation

```
1  proc(c⃗, i⃗, s⃗)
2  .prepareData:
3    li a0, i⃗('arr1.size')
4    li a1, c⃗('offset')
5    li a2, s⃗('secret')
6    //t3 ← arr1.addr
7    //t4 ← arr2.addr
8    add a3, t3, a1
9    sb s2, 0(a3)
10 .complexDependency:
11   li   t1,0x101
12   li   t2,0x100
13   mul   a4,a0,t1
14   remuw  a4,a4,t2
15   mul   a4,a4,t1
16   remuw  a0,a4,t2
17 .conditionalAccess:
18   bleu  a0,a1,.end
19   add  t3,t3,a1
20   lbu  a3,0x0(t3)
21   sll  a3,a3,6
22   and  a3,a3,0x3ff
23   add  a3,t4,a3
24   lbu  a4,0(a3)
```

(e) Long speculation

Fig. 16. Speculative execution example. Assembly in (e) is snippet from compilation of pseudocode in (b). Replacing lines 10–16 with (d) gives the analogous assembly for the pseudocode in (c).

To explore such leaks using our framework, we used the software pseudocode in Fig. 16(b) and Fig. 16(c), each of which accesses an element of array arr2 at a secret index arr1[offset]. The bounds check on offset is dependent on a complex sequence of computations in Fig. 16(b) and on reading arr1.size from memory in Fig. 16(c). Theoretically, speculative execution may leak arr1[offset] through cache-based side channels in both cases if the dependency is not resolved before speculative execution, i.e., by bringing arr2[(arr1[offset] × 64) & 1023] into cache. Fig. 16(e) shows an important snippet of RISC-V assembly for Fig. 16(b) running on BOOM with a 2-way, 8-set cache. To evaluate the software snippet in Fig. 16(c), we change the block denoted by `.complexDependency` (Lines 10–16) with the `.shortDependency` in Fig. 16(d). Furthermore, we evaluated a mitigation similar to **`lfence`** [Int, 2018], by adding a RISC-V instruction '**`fence`** `r,r`' just after Line 18 in Fig. 16(e).

We assume the attacker can control the offset value $\vec{c}$('offset'), train the *GShare* branch predictor $\vec{c}$('bpd') shown in Fig. 7, and use FLUSH+RELOAD to observe $\vec{o}$('hit'). The attacker can use the FLUSH+RELOAD-style attacks to precisely determine the index into arr2 if arr2 is shared and thus four bits of arr1[offset]. Note that the secret value $\vec{\mathrm{s}}$(‘secret’) is assigned to arr1[offset] as the first step of Fig. 16(c) and Fig. 16(b). We presume that $\vec{\mathrm{i}}$(‘arr1.size’) is an attacker-known but not controlled variable; thus, we include it as one output parameters as well, i.e., $\vec{\mathrm{o}}$(‘arr1.size’) $\longleftarrow$ $\vec{\mathrm{i}}$(‘arr1.size’).

As shown in Fig. 17, the $\hat{J}_n$ measures for ‘ShortSpec’ (denoting Fig. 16(d)) and ‘Fence’ are somewhat similar to that for ‘LongSpec’ (denoting Fig. 16(e))—contrary to what intuition would suggest. This counterintuitive result is due to the fact that leakage from *in-bounds* array accesses is also being counted. By declassifying in-bounds array elements (i.e., declassifying arr1[offset] if $\vec{\mathrm{c}}$(‘offset’) $<$ $\vec{\mathrm{i}}$(‘arr1.size’)), we obtain a better picture of when leakage occurs. Specifically, when measuring the leakage with declassification of in-bounds array elements, $\hat{J}_n^\delta$ indicates that both *proc* with the short dependency (‘ShortSpec+δ’) and *proc* with the **fence** mitigation (‘Fence+δ’) do not leak out-of-boundary memory contents, while the *proc* with the longer dependency (‘LongSpec+δ’) continues to leak secret data and indeed, is just slightly lower than ‘complexDepend’.

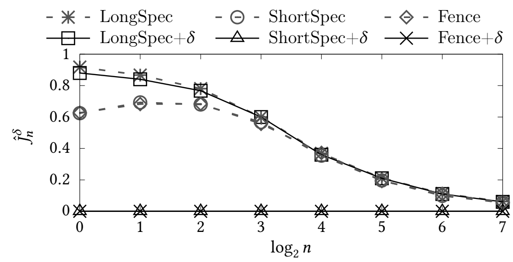


Fig. 17. $\hat{J}_n^\delta$ for Spectre in different procedures

In generating interference rules for *proc* with a long speculation (Fig. 16(e)), the linear feature

$$L_0 = 0.005 \times \vec{\mathrm{s}}(\text{‘secret’}) - 0.003 \times \vec{\mathrm{s}}'(\text{‘secret’}) - 0.494 \times \vec{\mathrm{c}}(\text{‘offset’}) + 0.496 \times \vec{\mathrm{i}}(\text{‘arr1.size’}) \quad (23)$$

$$\approx 0.5 \times \vec{\mathrm{i}}(\text{‘arr1.size’}) - 0.5 \times \vec{\mathrm{c}}(\text{‘offset’}) \quad (24)$$

and specifically the conjunct $L_0 < 1$ appears in many of the top ranked rules. Using the approximation of L_0 above, $L_0 < 1$ implies that $\vec{\mathrm{i}}$(‘arr1.size’) $<$ $\vec{\mathrm{c}}$(‘offset’) + 2, and so the offset is indeed out-of-bounds.

An example rule with precision 1.0 and recall 0.30 is

$$L_0 < 1 \wedge \vec{\mathrm{c}}(\text{‘bpd\{0\}.state’})[1] < 1 \wedge |\vec{\mathrm{s}}(\text{‘secret’})[2] - \vec{\mathrm{s}}'(\text{‘secret’})[2]| \geq 1 \quad (25)$$

This rule indicates that an attacker can determine the third bit of the secret when the second bit of the state of the prediction entry $\vec{\mathrm{c}}$(‘bpd{0}.state’) is 0 (‘strongly untaken’) or 1 (‘weakly untaken’). Analogous rules appear in the list for each of bits 0-2 and 4 of the secret. Other highly ranked rules (also with precision 1.0 and recall 0.30) are

$$L_0 < 1 \wedge \vec{\mathrm{c}}(\text{‘bpd\{0\}.CFI’})[0] \geq 1 \wedge |\vec{\mathrm{s}}(\text{‘secret’})[0] - \vec{\mathrm{s}}'(\text{‘secret’})[0]| \geq 1 \quad (26)$$

$$L_0 < 1 \wedge \vec{\mathrm{c}}(\text{‘bpd\{0\}.CFI’})[1] < 1 \wedge |\vec{\mathrm{s}}(\text{‘secret’})[3] - \vec{\mathrm{s}}'(\text{‘secret’})[3]| \geq 1 \quad (27)$$

Rule (26) leaks the first bit of the secret when the ‘CFI’ value (i.e., $\vec{\mathrm{c}}$(‘bpd{0}.CFI’)) in the prediction entry is 1 or 3, and (27) leaks the fourth bit when the ‘CFI’ value is 0 or 1. In these cases, the ‘CFI’ value does not match the CFI portion of the instruction address (i.e., the address of Line 18 in Fig. 16(e)), which was `0x800000800` + `0x44` (= `0b0 10 00100`), yielding a CFI portion of `0b 10` and *bidx* of `0b00`. Because of the mismatch on CFI value, $\vec{\mathrm{c}}$(‘bpd{0}.state’) is ignored and so speculation will not execute Lines 19–24. Though (26) and (27) are specific to the first or fourth bit of the secret, respectively, analogous rules appear for each of bits 0-3.

The simplicity of these rules stands in stark contrast to the complexity of the Yosys-generated per-cycle transition logic $\tau_{proc}(\vec{\mathrm{H}}^{t-1}, \vec{\mathrm{H}}^{t})$, which includes 459,170 bit variables and 1,922,229 clauses in CNF, or the postcondition Π_{proc}, which still includes 5,413 bit variables and 41,940 clauses. Clearly, our interpretation rules are vastly simpler for the analyst to consider than these alternatives.

Fig. 18(a) shows the cumulative precision and recall for all leakage rules in this case study. However, we do not need to use all rules for interpretation, since most rules do not help much with the cumulative recall. For example, considering only rules that improve cumulative recall by $\geq 1\%$ gives 12 rules that achieve 0.97 precision and 0.98 recall (Fig. 18(b)).

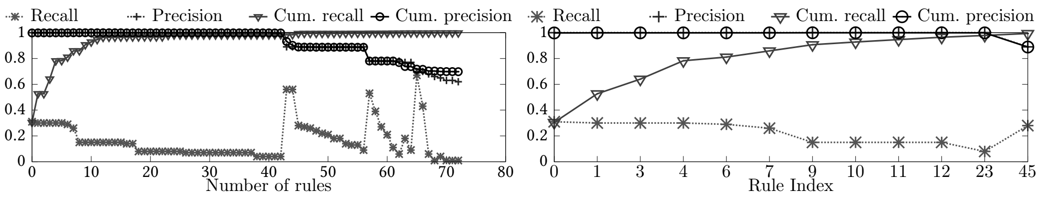
We have performed this evaluation using earlier BOOM versions and noticed that the out-of-bounds leakage was partially eliminated in version 2.2.3.[6] Since version 2.2.1, the miss handling (MSHR) module of the L1 cache tracks branch prediction results and discards the pending cache refill request if a misprediction is detected before the refill commit.

7 PERFORMANCE

In this section, we discuss the runtime performance of DINoMe on the case studies described in Sec. 6. In DINoMe, we have four important components: an automated logical formula generator (Sec. 5.1), a model counter (Sec. 5.2), a sampler (Sec. 5.3), and a rule learner (Sec. 4.3). This section reports the time costs in the first three stages for all case studies we have evaluated. We performed those experiments on a DELL PowerEdge R815 server with 2.3GHz AMD Opteron 6376 processors and 128GB memory.

The time to generate and simplify the logical postcondition is primarily influenced by the number of RISC-V BOOM cycles represented by that postcondition, as we incrementally compose the formula

[6]In BOOM version 2.2.1, the victim program described in Fig. 16(c) also suffers the out-of-bounds leakage and thus has 'ShortSpec' close to 'LongSpec' and 'ShortSpec+δ' close to 'LongSpec+δ'.



(a) Sorted by precision first and then recall (b) Dropping rules that improve cum. recall by < 1% (rule index is based on Fig. 18(a).

Fig. 18. Cumulative precision and recall vs. rules

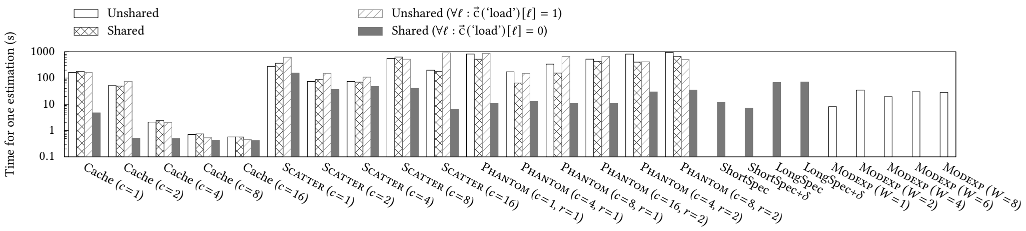


Fig. 19. Time used in one estimation of $\hat{J}^{\delta}(S, S')$

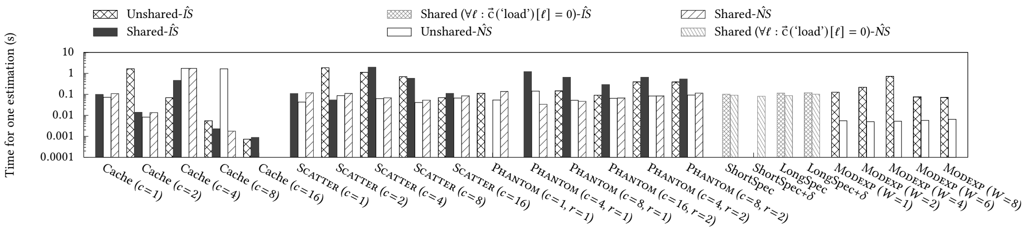


Fig. 20. Time used in generating one tuple in $\hat{NS}$ or $\hat{IS}$

cycle by cycle. Computing Π_{proc} required 20-40 minutes for the memory accessing experiments (100 cycles) in Sec. 6.2 and Sec. 6.3; 45 minutes for the modular exponentiation experiments (120 cycles) in Sec. 6.4; and around 2 hours for the SPECTRE experiments (150 cycles) in Sec. 6.5. Different from Zhou et al. [2018], DINoMe assembles the postcondition without path splitting per branch (and so avoids path explosion) and defers its solving task to a simplification step and final cycle, which reduces the complexity dramatically.

Fig. 19 shows the runtime to compute *one* estimate of $\hat{J}(S, S')$ or $\hat{J}^{\delta}(S, S')$ in the model counting process; note the logarithmic y-axis. Specifically, counting for cache-based side channels in SCATTERCACHE and PHANTOMCACHE are much more expensive than others, where one estimate requires up to 16 minutes. The difficulty in counting for SCATTERCACHE (denoted by 'SCATTER') and PHANTOMCACHE (denoted by 'PHANTOM') is due to the large size of their counting variables. For SCATTERCACHE, the memory-to-cache mapping uses $log_2(c) \times w$ bits per domain per memory block for 32 memory blocks. Specifically, the 8-way 2-set SCATTERCACHE (denoted by 'SCATTER ($c = 2$)'), uses 512 bits to represent $\vec{\imath}(\mathsf{M})$, which means the counting process would add hundreds of XOR constraints to compute one estimate, which greatly increases the difficulty to find a feasible solution. To obtain the sample sets $\hat{IS}$ and $\hat{NS}$, the sampling process generates a tuple in $\hat{IS}$ or $\hat{NS}$ within seconds, as illustrated in Fig. 20.

Our reported results reflect estimations of $\hat{J}(S, S')$ or $\hat{J}^{\delta}(S, S')$ for at least 100 S, S' pairs per n, and we sampled up to 100,000 tuples in $\hat{IS}$ and $\hat{NS}$. These estimations and samplings are trivially parallelizable and so, with horizontal scaling, can be performed in total times approaching those in Fig. 19 and Fig. 20 to the extent budget allows.

8 LIMITATIONS

Despite the scalability represented by DINoMe specifically for analyzing processor designs, it still has limitations. First, due to the complexity of hardware logic, generating the postcondition $\Pi_{proc}(\vec{c}, \vec{o}, \vec{\imath}, \vec{s})$ for a *proc* representing both the OS and the application would require more CPU cycles than the number to which we have been able to scale DINoMe thus far. The DINoMe workloads described in this paper represent a tradeoff, using a sequence of opcodes with concretized operations and selected symbolic operands above a partially symbolic hardware specification. To evaluate with more complicated software, a possible solution is to highly concretize the initial hardware state (especially for the memory and cache states) or highly concretize the software, at the cost of possibly missing some potential leakage that remains hidden due to this concretization.

A second limitation of DINoMe, and specifically of its generation of interpretation rules to explain leakage, is that the interpretation rules may not be complete, for two reasons. First, the interpretation rules might skip a rule that covers few leakage samples (i.e., with low recall). A possible way to address this source of incompleteness is to declassify the sources of leakage exposed in the inference rules that *are* learned, and then rerun the learning process again. Second, the conditions that result in leakage might be more complicated than can be learned using decision trees built using local linear classifiers. Alternative learning methods might be tried, though doing so while retaining interpretability will be a challenge.

9 CONCLUSION

Scaling high-fidelity, static noninterference measurement to complex computations has been a challenge since the introduction of noninterference in the 1980s [Goguen and Meseguer, 1982]. We believe that we have advanced the state-of-the-art in this area both generally and specifically for its application to processor designs. Certain innovations in our DINoMe framework, such as the cycle-by-cycle construction of the logical postcondition for processor execution, are specific to processor designs. Others, such as our methods for declassification and interpreting leakage results, are not. Together, however, they permit the measurement of leakage in complex scenarios, as we demonstrated through using DINoMe to analyze leakage due to speculative execution in the BOOM core and of published defenses to mitigate it. Our analysis enables comparisons between defenses to discover, e.g., the processor and defense parameterizations where one defense outperforms the other. Though the performance of DINoMe suggests that static measurement of noninterference for processors is still too time-intensive for highly interactive use, it is fast enough to permit multiple analysis iterations per day in many cases. And through its improvements in declassification and interpretability, it substantially facilitates human understanding of its measurement results.

REFERENCES

Intel analysis of speculative execution side channels. Technical report, Intel Corp., Jan 2018. URL https://www.intel.com/content/www/us/en/architecture-and-technology/intel-analysis-of-speculative-execution-side-channels-paper.html.

O. Aciiçmez. Yet another microarchitectural attack: Exploiting I-cache. In *ACM Workshop on Computer Security Architecture*, pages 11–18, 2007.

J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi. Verifying constant-time implementations. In *25th USENIX Security Symposium*, pages 53–70, 2016.

R. A. Aziz, G. Chu, C. Muise, and P. Stuckey. #∃SAT: Projected model counting. In *18th International Conference on Theory and Applications of Satisfiability Testing*, number 9340 in LNCS, pages 121–137, 2015.

M. Backes, B. Kopf, and A. Rybalchenko. Automatic discovery and quantification of information leaks. In *30th IEEE Symposium on Security and Privacy*, pages 141–153, 2009.

T. Ball, B. Cook, V. Levin, and S. K. Rajamani. SLAM and static driver verifier: Technology transfer of formal methods inside Microsoft. In *4th International Conference on Integrated Formal Methods*, volume 2999 of *LNCS*, pages 1–20, 2004.

A. Banerjee, D. A. Naumann, and S. Rosenberg. Expressive declassification policies and modular static enforcement. In *29th IEEE Symposium on Security and Privacy*, pages 339–353, 2008.

G. Barthe, G. Betarte, J. Campo, C. Luna, and D. Pichardie. System-level non-interference for constant-time cryptography. In *21st ACM Conference on Computer and Communications Security*, pages 1267–1279, 2014.

D. J. Bernstein, J. Breitner, D. Genkin, L. G. Bruinderink, N. Heninger, T. Lange, C. V. Vredendaal, and T. Yarom. Sliding right into disaster: Left-to-right sliding windows leak. In *19th International Conference on Cryptographic Hardware and Embedded Systems*, volume 10529 of *LNCS*, pages 555–576, 2017.

S. Blazy, D. Pichardie, and A. Trieu. Verifying constant-time implementations by abstract interpretation. *Journal of Computer Security*, 27(1):137–163, 2019.

C. Celio, P. Chiu, B. Nikolic, D. A. Patterson, and K. Asanovic. BOOMv2: an open-source out-of-order RISC-V core. In *1st Workshop on Computer Architecture Research with RISC-V (CARRV)*, 2017.

S. Chakraborty, K. S. Meel, and M. Y. Vardi. A scalable approximate model counter. In *Principles and Practice of Constraint Programming*, volume 8124 of *LNCS*, pages 200–216, 2013.

P. Chapman and D. Evans. Automated black-box detection of side-channel vulnerabilities in web applications. In *18th ACM Conference on Computer and Communications Security*, pages 263–274, 2011.

S. Chattopadhyay and A. Roychoudhury. Symbolic verification of cache side-channel freedom. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2812–2823, 2018.

S. Chattopadhyay, M. Beck, A. Rezine, and A. Zeller. Quantifying the information leak in cache attacks via symbolic execution. In *15th ACM International Conference on Formal Methods and Models for System Design*, pages 25–35, New York, NY, USA, 2017. ISBN 978-1-4503-5093-8.

C. Chen, K. Lin, C. Rudin, Y. Shaposhnik, S. Wang, and T. Wang. An interpretable model with globally consistent explanations for credit risk, 2018.

T. Chen and C. Guestrin. Xgboost: A scalable tree boosting system. In *22rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 785–794, 2016.

S. Chong and A. C. Myers. Security policies for downgrading. In *11th ACM conference on Computer and communications security*, pages 198–209, 2004.

W. W. Cohen and Y. Singer. A simple, fast, and effective rule learner. *16th AAAI Conference on Artificial Intelligence*, 99: 335–342, 1999.

G. Doychev, B. Köpf, L. Mauborgne, and J. Reineke. CacheAudit: A tool for the static analysis of cache side channels. In *22nd USENIX Security Symposium*, pages 431–446, 2013.

B. Dutertre. Solving exists/forall problems with yices. In *Workshop on satisfiability modulo theories*, 2015.

C. Dwork, F. McSherry, K. Nissim, and A. Smith. Calibrating noise to sensitivity in private data analysis. In *3rd Theory of Cryptography Conference*, volume 3876 of *LNCS*, page 265–284, 2006.

J. Fan. Local linear regression smoothers and their minimax efficiencies. *The Annals of Statistics*, pages 196–216, 1993.

R. Fan, K. Chang, C. Hsieh, X. Wang, and C. Lin. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research*, 9(Aug):1871–1874, 2008.

A. Ferraiuolo, R. Xu, D. Zhang, A. C. Myers, and G. Suh. Verification of a practical hardware security architecture through static information flow analysis. In *22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 555–568, 2017.

M. Fokkema. Fitting prediction rule ensembles with R package pre. *Journal of Statistical Software*, 92(12):1–30, 2020. ISSN 1548-7660.

J. H. Friedman. Greedy function approximation: a gradient boosting machine. *Annals of Statistics*, pages 1189–1232, 2001.

J. H. Friedman and B. E. Popescu. Predictive learning via rule ensembles. *The Annals of Applied Statistics*, 2(3):916–954, 2008.

R. Giacobazzi and I. Mastroeni. Abstract non-interference: Parameterizing non-interference by abstract interpretation. *ACM SIGPLAN Notices*, 39(1):186–197, 2004.

R. Giacobazzi and I. Mastroeni. Abstract non-interference: a unifying framework for weakening information-flow. *ACM Transactions on Privacy and Security (TOPS)*, 21(2):1–31, 2018.

K. V. Gleissenthall, R. G. Kıcı, D. Stefan, and R. Jhala. IODINE: Verifying constant-time execution of hardware. In *28th USENIX Security Symposium*, pages 1411–1428, 2019.

P. Godefroid, M. Y. Levin, and D. Molnar. SAGE: Whitebox fuzzing for security testing. *Queue*, 10(1):20–27, 2012. ISSN 1542-7730.

J. A. Goguen and J. Meseguer. Security policies and security models. In *3rd IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

J. W. Gray. Toward a mathematical foundation for information flow security. In *12nd IEEE Symposium on Security and Privacy*, pages 21–34, 1991.

X. Guo, R. G. Dutta, J. He, M. M. Tehranipoor, and Y. Jin. Qif-verilog: Quantitative information-flow based hardware description languages for pre-silicon security assessment. In *IEEE International Symposium on Hardware Oriented Security and Trust*, pages 91–100, 2019.

J. Kelsey. Compression and information leakage of plaintext. In *9th International Workshop on Fast Software Encryption*, pages 263–276, 2002.

V. Klebanov, N. Manthey, and C. Muise. SAT-based analysis and quantification of information flow in programs. In *10th International Conference on Quantitative Evaluation of Systems*, volume 8054 of *LNCS*, pages 177–192, 2013.

P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, and T. Prescher. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy*, pages 1–19, 2019.

J. Lagniez and P. Marquis. On preprocessing techniques and their impact on propositional model counting. *Journal of Automated Reasoning*, 58(4):413–481, 2017.

J. Lagniez, E. Lonca, and P. Marquis. Improving model counting by leveraging definability. In *25th International Joint Conference on Artificial Intelligence*, pages 751–757, 2016.

M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, S. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium*, pages 973–990, 2018.

P. Malacaria, M. Khouzani, C. S. Pasareanu, Q. Phan, and K. Luckow. Symbolic side-channel analysis for probabilistic programs. In *31st IEEE Computer Security Foundations Symposium*, pages 313–327, 2018.

N. Manthey. Coprocessor 2.0 – a flexible CNF simplifier. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 436–441, 2012.

M. McCall, H. Zhang, and L. Jia. Knowledge-based security of dynamic secrets for reactive programs. In *31st IEEE Computer Security Foundations Symposium*, pages 175–188, 2018.

C. Molnar. *Interpretable Machine Learning*. 2019. https://christophm.github.io/interpretable-ml-book/.

S. Nilizadeh, Y. Noller, and C. S. Păsăreanu. Diffuzz: Differential fuzzing for side-channel analysis. In *41st International Conference on Software Engineering*, ICSE '19, page 176–187, 2019.

O. Oleksii, T. Bohdan, S. Mark, and F. Christof. SpecFuzz: Bringing spectre-type vulnerabilities to the surface. In *29th USENIX Security Symposium*, 2020.

D. A. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: The case of AES. In *Topics in Cryptology – CT-RSA*, volume 3860 of *LNCS*, pages 1–20, 2006.

C. Percival. Cache missing for fun and profit. In *BSDCan 2005*, 2005.

Q. Phan and P. Malacaria. Abstract model counting: A novel approach for quantification of information leaks. In *9th ACM Symposium on Information, Computer and Communications Security*, pages 283–292, 2014.

A. Pnueli, Y. Rodeh, O. Strichman, and M. Siegel. The small model property: How small can it be? *Information and Computation*, 178(1):279–293, 2002.

M. T. Ribeiro, S. Singh, and C. Guestrin. “why should I trust you?”: Explaining the predictions of any classifier. In *22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1135–1144, 2016.

M. T. Ribeiro, S. Singh, and C. Guestrin. Anchors: High-precision model-agnostic explanations. In *32rd AAAI Conference on Artificial Intelligence*, pages 1527–1535, 2018.

A. Sabelfeld and A. C. Myers. A model for delimited information release. In *2nd International Symposium on Software Security – Theories and Systems*, volume 3233 of *LNCS*, pages 174–191, 2003.

A. Sabelfeld and D. Sands. Declassification: Dimensions and principles. *Journal of Computer Security*, pages 517–548, 2009.

S. Sahai, P. Subramanyan, and R. Sinha. Verification of quantitative hyperproperties using trace enumeration relations. In *32nd International Conference on Computer Aided Verification*, volume 12224 of *LNCS*, pages 201–224, 2020.

T. Seidenfeld. Entropy and uncertainty. *Philosophy of Science*, 53(4):467–491, 1986.

G. Smith. On the foundations of quantitative information flow. In *12th International Conference on Foundations of Software Science and Computational Structures*, volume 5504 of *LNCS*, pages 288–302, 2009.

G. Smith. Quantifying information flow using min-entropy. In *8th International Conference on Quantitative Evaluation of Systems*, pages 159–167, 2011. doi: 10.1109/QEST.2011.31.

D. X. Song, D. Wagner, and X. Tian. Timing analysis of keystrokes and timing attacks on SSH. In *10th USENIX Security Symposium*, Aug. 2001.

M. Soos and K. S. Meel. BIRD: Engineering an efficient CNF-XOR SAT solver and its applications to approximate model counting. In *36th AAAI Conference on Artificial Intelligence*, pages 1592–1599, 2019.

Q. Tan, Z. Zeng, K. Bu, and K. Ren. PhantomCache: Obfuscating cache conflicts with localized randomization. In *27th Network and Distributed System Security Symposium*, 2020.

T. Wang, T. Wei, L. Z, and W. Zou. IntScope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *16th Network and Distributed System Security Symposium*, 2009.

Z. Wang and R. B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *34th International Symposium on Computer Architecture*, pages 494–505, 2007.

M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard. ScatterCache: Thwarting cache attacks via cache set randomization. In *28th USENIX Security Symposium*, pages 675–692, Santa Clara, CA, 2019. ISBN 978-1-939133-06-9.

C. Wolf. Yosys open synthesis suite. http://www.clifford.at/yosys/.

Y. Xiao, Y. Zhang, and R. Teodorescu. SPEECHMINER: A framework for investigating and measuring speculative execution vulnerabilities. In *27th Network and Distributed System Security Symposium*, 2020.

Y. Yarom and K. E. Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security Symposium*, pages 719–732, 2014.

H. Yasuoka and T. Terauchi. Quantitative information flow as safety and liveness hyperproperties. *Theoretical Computer Science*, 538:167–182, 2014.

D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers. A hardware design language for timing-sensitive information-flow security. In *20th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 503—-516, New York, NY, USA, 2015. Association for Computing Machinery. doi: 10.1145/2694344.2694372.

K. Zhang, Z. Li, R. Wang, X. Wang, and S. Chen. Sidebuster: Automated detection and quantification of side-channel leaks in web application development. In *17th ACM Conference on Computer and Communications Security*, pages 595–606, 2010.

R. Zhang, C. Deutschbein, P. Huang, and C. Sturton. End-to-end automated exploit generation for validating the security of processor designs. In *51st IEEE/ACM International Symposium on Microarchitecture*, pages 815-–827, 2018.

Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. Cross-VM side channels and their use to extract private keys. In *19th ACM Conference on Computer and Communications Security*, pages 305–316, 2012.

Z. Zhou, M. K. Reiter, and Y. Zhang. A software approach to defeating side channels in last-level caches. In *23rd ACM Conference on Computer and Communications Security*, pages 871–882, 2016.

Z. Zhou, Z. Qian, M. K. Reiter, and Y. Zhang. Static evaluation of noninterference using approximate model counting. In *39th IEEE Symposium on Security and Privacy*, pages 514–528, 2018.

10 APPENDICES

10.1 Preprocessing $\Pi_{proc}(\vec{c}, \vec{o}, \vec{\imath}, \vec{s})$ for #∃SAT

Applying a correct combination of techniques to simplify $\Pi_{proc}(\vec{c}, \vec{o}, \vec{\imath}, \vec{s})$ is critical to scaling its use in DINoMe. Lagniez and Marquis [2017] provides a summary of the options. As defined in Sec. 3.3 and Sec. 4.2, our uses of $\Pi_{proc}(\vec{c}, \vec{o}, \vec{\imath}, \vec{s})$ are instances of *projected model counting* (#∃SAT) [Aziz et al.,

	cycles						
	5	10	15	20	25	30	35
No simplification	49.1	126.6	177.0	304.2	459.0	667.6	867.6
Simplified	1.10	1.54	1.98	2.03	89.9	167.3	389.0
#∃SAT-preprocess	1.10	1.41	1.45	1.65	1.71	1.77	1.88

Table 1. CNF file size (MB) for logic formulas extracted from the RISC-V BOOM core configured with a small application program, starting from an initial state with some symbolic memory blocks and cache states (see Sec. 6.2). The CNF file size for a one-cycle execution with completely symbolic initial state is 40MB. Only computations that terminated within 10 minutes are represented.

2015], which counts feasible assignments of selected variables in a propositional formula. The complexity of the counting problem is #NP-hard.

To simplify the logical formula, we use preprocessing (similar to that used in, e.g., Klebanov et al. [2013], Manthey [2012]) that fully applies equivalence-preserving simplification techniques (e.g., vivification and occurrence reduction), and then partially applies SAT-preserving simplifications (e.g., literal eliminations, variable eliminations and clause eliminations) targeting variables not in our counting target (i.e., not marked as *svar*, *ivar*, *cvar*, or *ovar*). For example, the partially applied blocked-clause elimination will remove a clause if it contains a variable not in the counting target such that every resolvent obtained by resolving on it is a tautology. Especially for hardware designs where the number of possible states increases with the cycle count but many registers are not modified in some cycles, preprocessing the formula can substantially reduce the redundancy between the initial and final cycle formulas.

Although we only need the logic $\Pi_{proc}(\vec{c}, \vec{o}, \vec{\imath}, \vec{s})$ to describe the relationship between the attacker-observable outputs $\vec{o}$ and inputs $\vec{c}, \vec{s}, \vec{\imath}$, the translated conjunctive-normal-form (CNF) propositional formula F produced by the commonly used Tseitin algorithm would have numerous auxiliary variables. The number of auxiliary variables and clauses increases quickly with the number of cycles. To reduce the use of auxiliary variables, we applied a state-of-art preprocessing technique for *model counting* called B+E proposed by Lagniez et al. [2016], who also discussed a possible application of this method to *projected model counting*. In our modified version of B+E, we partition the variables in the CNF formula representing Π_{proc} into two disjoint variable sets: *Sup* containing the variables in $Vars_{\vec{\imath}}$, $Vars_{\vec{c}}$, $Vars_{\vec{s}}$, and $Vars_{\vec{o}}$, and *Dep* containing all other variables. For each variable v in *Dep* and pair of clauses $\bar{v} \vee C_j$ and $v \vee C'_i$, we then resolve on v and replace the clauses with their resolvent $C_j \vee C'_i$. We do this *only* for variables $v \in Dep$, since this ensures:

$$\left\{ \langle \vec{c}, \vec{\imath}, \vec{s}, \vec{o} \rangle \;\middle|\; \bigwedge_{i=0}^{a} (v \vee C'_i) \wedge \bigwedge_{j=0}^{b} (\bar{v} \vee C_j) \right\} = \left\{ \langle \vec{c}, \vec{\imath}, \vec{s}, \vec{o} \rangle \;\middle|\; \bigwedge_{i=0}^{a} \bigwedge_{j=0}^{b} (C'_i \vee C_j) \right\}$$

Using this algorithm, we eliminate variables in *Dep* to the extent possible, so that the final formula uses only *svar*, *ivar*, *cvar*, and *ovar* with few auxiliary variables. This does have the side effect of introducing more complicated clauses, however, and so we avoid eliminating $v \in Dep$ when v is present in numerous clauses (e.g., $b \cdot a > 500$).

In Table 1, we present example sizes of CNF formulas generated using different simplification options, for our case studies in Sec. 6. The row denoted by "No simplification" represents the size of a directly translated CNF formula from the multi-cycle SMT formula, which linearly increases with the number of cycles. The "Simplified" row is generated by directly applying the CNF simplifications provided by `CryptoMiniSAT 5.0` to the formula in "Original", while the row

"#∃SAT-preprocess" is obtained by incrementally applying the preprocessor described in this section to each $\Psi^{T}_{proc}(\vec{c}, \vec{\imath}, \vec{s}, \vec{H}^{T})$ before using it to build $\Psi^{T+1}_{proc}(\vec{c}, \vec{\imath}, \vec{s}, \vec{H}^{T+1})$.

10.2 Defining $\vec{c}$ and $\vec{o}$ for cache-based side channels

The most common cache-based side-channel attacks are Prime+Probe, Flush+Reload, and their variants (e.g., see Yarom and Falkner [2014], Zhang et al. [2012]). In a Prime+Probe attack, the attacker loads memory blocks to fill (Prime) cache sets, permits the victim computation to run for a Prime+Probe interval, and then reads (Probes) these same blocks to determine which were evicted by the victim computation during the Prime+Probe interval. In a Flush+Reload attack, the attacker Flushes a shared-memory block from cache and then, after a Flush+Reload interval, accesses (Reloads) the block to determine whether the block was brought back into the cache by the victim computation.

To model side channel attacks in our framework, it is necessary to model the effects on the cache of the phases before victim execution (the Prime and Flush steps) and to define $\vec{o}$ to include the results of the phases after victim execution (the Probe and Reload steps). To do so, we assume that the adversary has access to memory blocks $block_1$, $block_2$, ..., $block_m$ aligned to cache lines, and we define the RISC-V assembly routine *acc* (see above) by which the adversary can access the block with index $\ell = \hat{\vec{c}}$('blockIdx') and empty $\hat{\vec{s}}$.

acc $(\hat{\vec{c}}, \hat{\vec{\imath}}, \hat{\vec{s}})$
```
  li s0, 0x2000000
  add s1, s0, ℓ
  sll s1, s1, 6
  lbu a2, 0(s1)
```

Starting from hardware state $\hat{\vec{H}}^{0}_{\ell}$ $(= \hat{\vec{\imath}})$ that is completely symbolic, we generate the per-cycle logical postcondition $\tau_{acc}(\hat{\vec{H}}^{t-1}_{\ell}, \hat{\vec{H}}^{t}_{\ell})$ for each $0 < t \leq \hat{T}$ as in Sec. 5.1, where we empirically choose $\hat{T} = 45$.

We use these postconditions in two ways. First, we use them to extract a constraint $\Gamma(\langle\vec{H}^{t}\rangle^{T}_{t=1}, \vec{o})$ that defines the attacker's observations $\vec{o}$ in terms of the hardware states $\langle\vec{H}^{t}\rangle^{T}_{t=1}$ induced by the execution (see (11)). A naive attempt to do so would be to simply include in $\vec{o}$ the metadata for each cache line at every step of the execution. However, this would grant too much power to an attacker, who should not be given access to the tag values and the exact locations of blocks inside a set. Instead, we permit only a weaker attacker (cf., abstract noninterference [Giacobazzi and Mastroeni, 2004]) by defining the constraint $\Gamma(\langle\vec{H}^{t}\rangle^{T}_{t=1}, \vec{o})$ that represents the view of cache hits and misses immediately observable by the adversary, by:

$$\vec{o}(\text{'hit'})[\ell] = \left(\begin{array}{l} (\hat{\vec{H}}^{0}_{\ell} = \vec{H}^{T}) \wedge \left(\bigwedge_{t=1}^{\hat{T}} \tau_{acc}(\hat{\vec{H}}^{t-1}_{\ell}, \hat{\vec{H}}^{t}_{\ell}) \right) \\ \wedge \left(1 - \bigvee_{t=1}^{\hat{T}} \textsc{CacheMiss}(\hat{\vec{H}}^{t}_{\ell}, block_{\ell}) \right) \end{array} \right)$$

for $\ell = \hat{\vec{c}}$('blockIdx'). Here, CacheMiss is a BOOM-defined Verilog code snippet that, intuitively, checks a set of cache lines where $block_\ell$ might reside and returns 1 (in a register called `s2_hits`) if none of those cache lines has a valid tag matched with $block_\ell$ (and returns 0 otherwise). In this way, we characterize the procedure *acc* using a logical postcondition without manually modeling CacheMiss.

Second, we permit the attacker to control which of its blocks are loaded into the cache before the victim runs. Specifically, the predicate $\Psi^{0}_{proc}(\vec{c}, \vec{\imath}, \vec{s}, \vec{H}^{0})$ that controls the initial hardware state from which the victim executes is modified to constrain which of the attacker's blocks are present in cache, as communicated through a reserved variable 'load' $\in Vars_{\vec{c}}$, for which the $\vec{c}$('load') is a bit vector of length m. That is, attacker block $block_\ell$ should be loaded before the victim runs if and only if $\vec{\text{c}}(\text{‘load’})[\ell] = 1$. To effect this in $\Psi^0_{proc}(\vec{\text{c}}, \vec{\text{i}}, \vec{\text{s}}, \vec{\text{H}}^0)$, we construct $\Psi^0_{proc}(\vec{\text{c}}, \vec{\text{i}}, \vec{\text{s}}, \vec{\text{H}}^0)$ to include

$$\vec{\text{c}}(\text{‘load’})[\ell] = \begin{pmatrix} (\hat{\vec{\text{H}}}^0_\ell = \vec{\text{H}}^0) \wedge \left(\bigwedge_{t=1}^{\hat{T}} \tau_{acc}(\hat{\vec{\text{H}}}^{t-1}_\ell, \hat{\vec{\text{H}}}^t_\ell)\right) \\ \wedge \left(1 - \bigvee_{t=1}^{\hat{T}} \textsc{CacheMiss}(\hat{\vec{\text{H}}}^t_\ell, \mathit{block}_\ell)\right) \end{pmatrix}$$

Of course, we rename variables to ensure no conflicts between copies of $\hat{\vec{\text{H}}}^t_\ell$ included within the $\vec{\text{c}}(\text{‘load’})[\ell]$ and $\vec{\text{o}}(\text{‘hit’})[\ell]$ constraints.