Title

Lifting Micro-Update Models from RTL for Formal Security Analysis

Authors
Godbole, Adwait
Cheang, Kevin
Manerkar, Yatin A
et al.

Publication Date
2024-04-27

DOI
10.1145/3620665.3640418

SemPat: From Hyperproperties to Attack Patterns for Scalable Analysis of Microarchitectural Security

Adwait Godbole (✉)
adwait@berkeley.edu
University of California, Berkeley
Berkeley, CA, USA

Yatin A. Manerkar
manerkar@umich.edu
University of Michigan
Ann Arbor, MI, USA

Sanjit A. Seshia
sseshia@berkeley.edu
University of California, Berkeley
Berkeley, CA, USA

ABSTRACT

Microarchitectural security verification of software has seen the emergence of two broad classes of approaches. The first uses non-interference-based *semantic security properties* which are verified for a given program and a given model of the hardware microarchitecture. The second is based on *attack patterns*, which, if found in a program execution, indicates the presence of an exploit. We observe that while the former uses a formal specification that can capture several gadget variants targeting the same vulnerability, it is limited by the scalability of verification. Patterns, while more scalable, must be currently constructed manually, as they are narrower in scope and sensitive to gadget-specific structure.

This work develops a technique that, given a non-interference-based semantic security hyperproperty, automatically generates attack patterns up to a certain complexity parameter (called the skeleton size). Thus, we combine the advantages of both approaches: security can be specified by a hyperproperty that uniformly captures several gadget variants, while automatically generated patterns can be used for scalable verification. We implement our approach in a tool and demonstrate the ability to generate new patterns, (e.g., for SpectreV1, SpectreV4) and improved scalability using the generated patterns over hyperproperty-based verification.

CCS CONCEPTS

• **Security and privacy → Formal security models**; **Logic and verification**; **Side-channel analysis and countermeasures**; • **Theory of computation → Verification by model checking**.

KEYWORDS

Microarchitectural Security, Hyperproperties, Attack Patterns, Verification, Side Channels

ACM Reference Format:
Adwait Godbole (✉), Yatin A. Manerkar, and Sanjit A. Seshia. 2024. SemPat: From Hyperproperties to Attack Patterns for Scalable Analysis of Microarchitectural Security. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3690214

1 INTRODUCTION

Modern processors are packed with performance-improving microarchitectural mechanisms such as caches, speculation, prefetching. These lead to subtle microarchitectural-level interactions that can be exploited by hardware execution attacks to leak sensitive data from a victim (e.g., [8, 38, 39, 45, 62]). Recently, several techniques have been proposed to detect the presence of such vulnerabilities in software and/or verify their absence (e.g. [12, 16, 17, 30, 31, 51, 59]). While sharing the goal of security verification, these approaches adopt different (hardware) platform models, security specifications, and have different strengths.

We observe the emergence of two broad classes of approaches based on the security specification they adopt: *semantic hyperproperty* (SH) based approaches (e.g., [12, 21, 30–32]) perform verification with respect to a hyperproperty-based [15] security specification, while *attack pattern* (AP) based approaches (e.g., [51, 59, 69]) perform verification using patterns that indicate the existence of an exploit. SH approaches provide advantages of uniform specification and formal guarantees, while AP approaches have better scalability. In this work, **we enable automated generation of attack patterns given a hyperproperty-based specification**, thereby combining their strengths.

SH approaches (e.g., [12, 21, 30–32]) identify a transition system-based platform model and define program security *semantically* as a hyperproperty (e.g., non-interference (NI) [15, 28]) over executions of this transition system. This hyperproperty is verified using approaches such as model checking [14] or symbolic execution [37]. By semantically characterizing vulnerabilities, hyperproperties allow *uniform specification*: i.e., specification that captures several exploit gadgets that target the same vulnerability, while differing in syntactic structure. This results in strong, high-coverage security guarantees. However, these approaches are often limited by scalability owing to microarchitectural platform model complexity.

AP approaches (e.g. [51, 59, 69]) use *attack patterns* to detect vulnerabilities. Patterns identify execution fragments that are indicative of an exploit; program executions embedding these patterns are flagged as vulnerable. Patterns can be defined (and checked) over a platform model that is more abstract than the microarchitectural models used by SH approaches. This abstraction leads to simpler verification queries which scale better with program size. However, each pattern captures *specific execution scenarios* and is sensitive to structural variability in exploit gadgets, even when the gadgets target the same microarchitectural feature. Covering the attack vector requires an enumeration of all possible patterns that it encompasses. This can be tedious to perform manually, leading to incomplete coverage.

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA.

SemPat. The above discussion motivates the question: *Can we combine the scalability of* AP *approaches with the uniform specification and guarantees of* SH *approaches?* This work aims to resolve this question in the context of microarchitectural vulnerabilities by developing algorithms to generate attack patterns given a platform model and a non-interference-based hyperproperty. **Our key insight** is to use the hyperproperty as the specification against which to check whether a candidate pattern represents an exploit. Our technique (§5) ensures that the generated patterns capture all executions in which there is a dependency-closed sub-execution of size k that violates the hyperproperty. This property, termed as k-completeness (Eq. 1), enables combining formal specification via hyperproperties with scalable verification via patterns.

Our pattern generation is guided by a *grammar* (§5.3) that identifies the space of constraints that patterns are defined over. The choice of grammar captures the tradeoff between generality and the precision (false positives) of the generated patterns. Specialized grammars result in patterns which have fewer false positives but are microarchitectural-implementation dependent. Our approach traverses the grammar-induced search space, checking candidates against the hyperproperty, a la grammar-based synthesis ([2, 36]).

Contributions. Our contributions are as follows:

(1) **Formally relating semantic hyperproperty and attack pattern based approaches:** We compare and formally relate SH and AP approaches to exploit detection. Based on the insight that hyperproperties can serve as a specification for patterns, we propose the problem of generating the latter given the former.
(2) **k-complete automatic pattern generation:** We develop a grammar-based search algorithm for automatic pattern generation that ensures that the generated patterns capture all non-interference violations up to a certain complexity parameter k (termed skeleton size).
(3) **Implementation and Evaluation - new patterns and improved verification performance:** We develop a prototype tool implementing our pattern generation technique and for bounded verification of software binaries using the generated patterns. We evaluate our approach by generating attack patterns and using them to analyze variants of Spectre-style exploits. We demonstrate: (a) the ability to generate previously unknown patterns for existing (e.g., Spectre-BCB, Spectre-STL) attacks as well as variants targeting alternative (e.g. computation-unit-based) side-channels, and (b) up to 2 orders-of-magnitude performance improvement on litmus tests with better scaling using the generated patterns compared to hyperproperty-based analysis.

Outline. In §2 we provide background on SH and AP and motivate our contribution. We provide our programming model in §3, followed by the problem formulation in §4. In §5 we describe SemPat, our approach to generate attack patterns based on a semantic platform model and a non-interference security hyperproperty. We discuss the evaluation methodology in §6 and present experimental results in §7. We discuss our approach, and its limitations in §8, related work in §9, and §10 concludes. For more details we refer to the extended version of this paper [27].

2 BACKGROUND AND MOTIVATION

2.1 Microarchitectural Execution Attacks

We provide background on microarchitectural execution attacks using the Spectre Bounds Check Bypass (BCB) vulnerability. We refer the reader to literature (e.g., [8, 24, 39]) for extensive surveys.

2.1.1 *Spectre-BCB.*

The `victimA` function in Fig. 1 shows the Spectre-BCB (also known as SpectreV1) vulnerability gadget. In Spectre-BCB, an attacker induces an invocation of the `victimA` function with an argument `i` which is out-of-bounds of array `arr1`. While architecturally this is an illegal access, branch misspeculation can allow `arr1[i]`, followed by `arr2[arr1[i] << CL_INDEX]` to be accessed *speculatively*. In particular, the second load leaves a residue in the cache which is a function of the accessed address (i.e., `arr2+arr1[i] << CL_INDEX`). This residue is preserved even after speculative rollback, and can be observed by the attacker (e.g., using a Prime+Probe primitive [46]). Thus the attacker can observe the value of `arr1[i]` for an out-of-bounds index `i`, leading to a security vulnerability. The key aspect of this vulnerability is that it leverages *microarchitectural features* - branch (mis)-speculation and cache side-channels - to leak information.

2.1.2 *Variant Vulnerabilities.*

Over the years several variants of microarchitectural hardware execution attacks have been demonstrated. These vary both at the software level (e.g., the structure of the vulnerability gadget occurring in the program library/executable) as well as at the microarchitectural level (e.g., the underlying speculative mechanism/side-channel that is exploited).

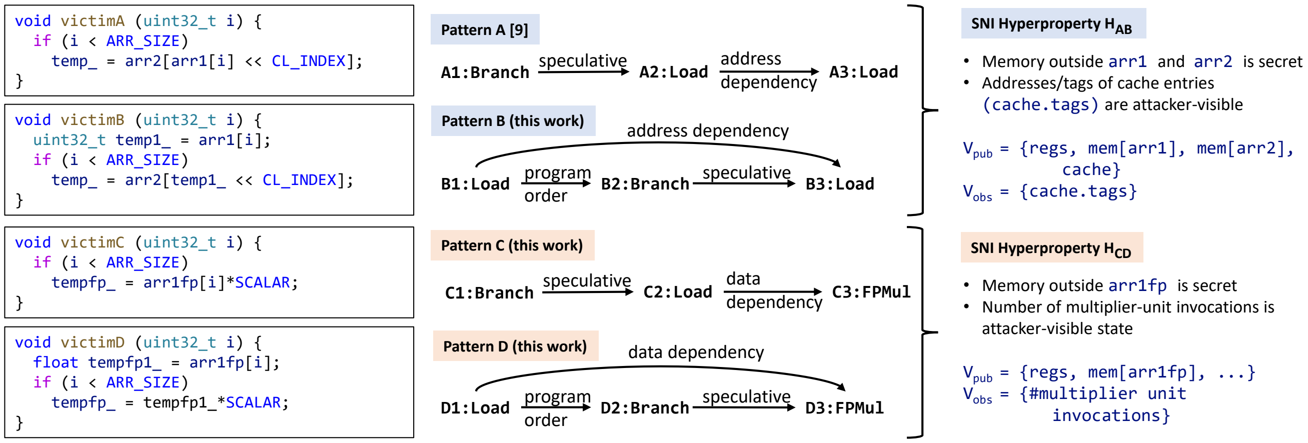
The function `victimC` in Fig. 1 shows a variant of `victimA`, that replaces a cache-based side channel with a computation unit-based side channel. Like `victimA`, `victimC` also exploits branch speculation to perform an out-of-bounds load of `arr1fp[i]` (a floating-point array). The loaded value is used in a floating-point multiply operation. If the microarchitectural implementation of this operation has data-operand dependent timing (e.g., [24, 61]) then an execution time measurement leaks the value of `arr1fp[i]`. Computation units have been targeted in this manner by previously demonstrated exploits (e.g., NetSpectre [63] uses AVX-based timing side channels) as well as conjectured vulnerabilities (e.g., [61]).

2.2 Analyzing Software for Vulnerabilities

Analyzing programs for the existence of microarchitectural vulnerabilities is an important yet challenging problem.

2.2.1 AP*: Attack Pattern-based Analysis.*

One family of approaches, which we call AP approaches, detect vulnerabilities using *attack patterns.* An attack pattern is a small execution fragment which, if *embedded* in some (larger) program execution, indicates the presence of a vulnerability. Pattern A (Fig. 1) illustrates one such pattern from existing work [51]. Pattern A matches executions where a load (A2) is speculatively executed following a branch (A1), and where the address of a subsequent load (A3) depends on the value loaded by A2. Fig. 2 (Match A) shows the compiled `victimA`, and depicts how the instruction nodes A1, A2, A3 in pattern A match instructions in the binary execution.



```
void victimA (uint32_t i) {
  if (i < ARR_SIZE)
    temp_ = arr2[arr1[i] << CL_INDEX];
}
```

```
void victimB (uint32_t i) {
  uint32_t temp1_ = arr1[i];
  if (i < ARR_SIZE)
    temp_ = arr2[temp1_ << CL_INDEX];
}
```

```
void victimC (uint32_t i) {
  if (i < ARR_SIZE)
    tempfp_ = arr1fp[i]*SCALAR;
}
```

```
void victimD (uint32_t i) {
  float tempfp1_ = arr1fp[i];
  if (i < ARR_SIZE)
    tempfp_ = tempfp1_*SCALAR;
}
```

SNI Hyperproperty H_{AB}

- Memory outside arr1 and arr2 is secret
- Addresses/tags of cache entries (cache.tags) are attacker-visible

```
Vpub = {regs, mem[arr1], mem[arr2],
        cache}
Vobs = {cache.tags}
```

SNI Hyperproperty H_{CD}

- Memory outside arr1fp is secret
- Number of multiplier-unit invocations is attacker-visible state

```
Vpub = {regs, mem[arr1fp], ...}
Vobs = {#multiplier unit
        invocations}
```

Figure 1: Left: Variants of speculative exploits targeting branch (mis)-speculation with cache-based (victimA, victimB) and computation unit-based (victimC, victimD) side channels. Centre: While attack patterns that can detect these exploits, variants of the same (Spectre-BCB) vulnerability - victimA and victimB - require different patterns, as do victimC and victimD. In (A, C), the load arr1[i] is performed within the speculative window (after the branch), while in (B, D) it is before the branch. Right: Unlike attack patterns, a semantic hyperproperty *uniformly characterizes* several exploits aimed a particular microarchitectural vulnerability. The Speculative Non-Interference [31] (SNI) Hyperproperty H_{AB} can identify both victimA and victimB (as well as others) as exploits, while the SNI Hyperproperty H_{CD} identifies both victimC and victimD. While the hyperproperty changes with the platform's microarchitectural features (e.g., speculation and side-channels), it is robust to variances in the exploit (program) structure itself.

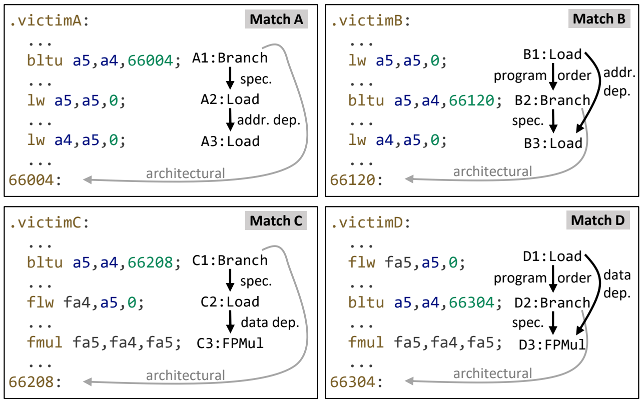
AP-approach	Execution model	Attack pattern variant
CheckMate [69]	μhb graphs [47]	*Bad execution patterns*
Cats vs. Spectre [59]	cat [1] based event structures	sec $\rightarrow^*$ ***Obs*** paths
Axiomatic HW/SW Contracts [51]	Leakage containment models (LCMs)	Transmitters

Table 1: Examples of execution modelling choices and pattern variants used in some AP-approaches.

AP *approach variants.* Existing AP-based detection approaches identify a set of such patterns, and then analyze programs for the existence of pattern embeddings. These approaches model executions and define patterns either at the architectural [51, 59] or the microarchitectural [69] level. Architecture-level approaches are augmented with sufficient microarchitectural detail to capture the vulnerability-specific features (e.g. *xstate* from Mosier et. al. [51]). Table 1 provides a summary of these differences.

Patterns defined at the architectural level *abstract away* complex microarchitectural details of the platform. This results in simpler verification queries, allowing analysis to scale to larger programs (e.g., see [51, 59]).

Patterns are tied to gadget-specific structure [1]. While attack pattern-based analysis has the benefit of greater scalability, patterns do not generalize well to other gadget variants that exploit the same underlying microarchitectural features. To illustrate this, function victimB in Fig. 1 is a modified version of victimA, where the load arr1[i] is performed non-speculatively, i.e. before the branch. Due to this load–branch inversion, pattern A does not match the execution of victimB, even though both victimA and victimB exploit the same (Spectre-BCB) mechanism. In Fig. 1 we provide pattern B that reorders the branch with the first load, and thus *can* capture victimB (depicted in Fig. 2 Match B). This highlights the fact that attack patterns are tied to gadget-specific structure.



```
.victimA:
  ...
  bltu a5,a4,66004;
  ...
  lw a5,a5,0;
  ...
  lw a4,a5,0;
  ...
66004:
```

```
.victimB:
  ...
  lw a5,a5,0;
  ...
  bltu a5,a4,66120;
  ...
  lw a4,a5,0;
  ...
66120:
```

```
.victimC:
  ...
  bltu a5,a4,66208;
  ...
  flw fa4,a5,0;
  ...
  fmul fa5,fa4,fa5;
  ...
66208:
```

```
.victimD:
  ...
  flw fa5,a5,0;
  ...
  bltu a5,a4,66304;
  ...
  fmul fa5,fa4,fa5;
  ...
66304:
```

Figure 2: Patterns A, B, C, D matched against executions of programs victimA,victimB,victimC,victimD respectively.

2.2.2 SH: *Semantic Hyperproperty-based Analysis.*

Non-interference-based hyperproperties [15] allow security specifications such as confidentiality ("secret variables should not affect public outputs") and integrity ("public inputs should not affect protected variables").

[1] Which is why we call these *attack* patterns, since they only capture specific attack executions, and not the common underlying vulnerability.

SH-approach	Non-interference (NI) variant
A Formal Approach to to Secure Speculation [12]	Trace-property dependent observational-determinism
Spectector [31]	Speculative non-interference
InSpectre [30]	Conditional non-interference
Hardware-software contracts [32]	Contract conditioned NI
Automated detection of speculative attack combinations [21]	Speculative NI

Table 2: Summary of non-interference variants used in some SH **approaches.**

SH-approaches, e.g., [12, 30, 31], formulate security semantically using hyperproperties and verify them against a platform model.

Hyperproperties allow uniform security specification. The hyperproperty H_{AB} (Fig. 1 (right)) specifies the memory *outside* (public) arrays `arr1`, `arr2` as being secret, and the cache tags as being public output. This captures a cache-based side channel where the tags are attacker-observable (e.g., using Prime+Probe [46]). `victimA` violates H_{AB}, since the cache state is tainted with the (speculatively loaded) address of the second load which is a value outside `arr1`, `arr2`. Although it has a different structure than `victimA`, `victimB` also violates H_{AB} as it too leaves a cache residue. Thus, unlike attack patterns, hyperproperties such as H_{AB} are agnostic to gadget-specific structure. By uniformly capturing several exploit-variants targetting a particular microarchitectural vulnerability, SH-approaches provide wide-scoped, strong guarantees.

Hyperproperty verification. SH-approaches base their analysis on variants of non-interference, which we summarize in Tab. 2, and discuss in more detail in §4.1. They verify this hyperproperty against a *semantic platform model*, which identifies (a) system state (variables in the system), and (b) execution semantics of operations. Fig. 3 illustrates a platform fragment with register file, memory and cache state, and `load` and `alu` operations. Non-interference-based hyperproperties can be converted into a single trace property by performing *self-composition*, i.e., composing together copies of the platform ([15, 68]). This single trace property can be checked by invoking a model checking or software verification procedure.

Self-composition-based verification against a semantic model has two drawbacks. Firstly, microarchitectural detail in the platform model results in large verification queries. (e.g., a query for H_{AB} over the platform from Fig. 3 would have to encode the cache state). In comparison, architectural level pattern-based queries (e.g., with pattern A) are smaller, enabling faster verification. Secondly, self-composition results in a further increase in the state-space (and query) and consequentially adversely affects performance.

2.3 Why convert from SH to AP?

Manual pattern generation is error-prone. While more scalable, AP approaches require creation of several patterns, due to their gadget-specificity. To avoid unsound analysis (e.g., using only pattern A on `victimB`), it is important that patterns are not missed, e.g., pattern B which we have not observed being formulated previously.

```
// System state
var regs  : [regindex_t]word_t
var cData : [index_t]word_t
var cTag  : [index_t]tag_t
var mem   : [word_t]word_t
// Load operation
operation load (rs, rd, imm) {
  addr = regs[rs] + imm;  // Compute the address
  if (cacheTag(addr_to_tag(addr)) == addr) {
    ... // Load from cache if hit
  } else {
    data = mem[addr]; // Load from memory
  }
  regs[rd] = data; // Register writeback
}

// Generic ALU Register-Register operation
operation alu (rs1, rs2, rd, op) {
  if (op == ADD) regs[rd] = regs[rs1] + regs[rs2];
  ...
}
```

Figure 3: Fragment of a platform model with state variables and `load` and `alu` operation semantics.

Moreover, patterns need to be recreated for newer microarchitectures (with newer vulnerabilities). Consider `victimC` (Fig. 1 left) which replaces a cache-based side channel (as in `victimA`) with a computation unit-based channel. Its variant `victimD` inverts the first load and the branch (as in `victimB`). These examples are inspired from [61], which hypothesizes the existence of computation-unit based side-channels on microarchitectures with data-operand-dependent timing [24, 35, 61]. Existing work, which targets cache-based side channels, misses patterns C and D (Fig. 1 center) that capture `victimC` and `victimD`. Automating pattern generation can make patterns more comprehensive and cover newer microarchitectural features.

Semantic hyperproperties as a specification for automated pattern generation. In this work, we propose a technique to automatically generate patterns for a given hyperproperty and microarchitecture. As an example we were able to automatically generate patterns C and D from the (shared) hyperproperty H_{CD}.

Our key insight is using the semantic hyperproperty as a specification to guide pattern generation. Since a hyperproperty can capture an entire class of exploits targetting a vulnerability, we can use it to determine whether a given pattern yields an exploit by checking it against the hyperproperty. By automatically checking several candidates, we can identify a comprehensive set of patterns for that hyperproperty. Thus, our technique replaces manual pattern creation with the requirement of specifying a hyperproperty and platform model. To summarize, by developing an automated conversion technique from SH-specifications to AP-based patterns, **we combine the low-overhead, uniform specifications and formal guarantees of** SH **with the superior verification scalability of** AP **to get the best of both worlds.**

3 SYSTEM MODEL

In this section, we introduce our formal model for hardware platforms, which we later use to develop our problem formulation (§4). At a high level, the hardware platform is an operational transition

Variables (arch. and march.)	v :	$\mathsf{V} = \mathsf{V_a} \cup \mathsf{V_m}$
States (variable assignments)	σ :	$\Sigma = \mathsf{V} \to \mathbb{D}$
Operation code	op :	Op
Instructions	inst :	$\mathsf{Inst} = \{\mathsf{op}(\omega)\}_{\mathsf{op},\omega}$
Full (speculative) semantics	T :	$\mathsf{Inst} \times \Sigma \to \Sigma$
Non-speculative semantics	$\mathsf{T_{ns}}$:	$\mathsf{Inst} \times \Sigma \to \Sigma$

Table 3: Platform state and operational semantics.

system over architectural and microarchitectural state variables. Instructions executed on the platform induce transitions over this state. We summarize these elements in Table 3.

3.0.1 State. The platform consists of variables V which take values from a domain $\mathbb{D}$. V includes architectural ($\mathsf{V_a}$) and microarchitectural ($\mathsf{V_m}$) variables. The platform state is an assignment to these variables, $\sigma : \mathsf{V} \to \mathbb{D}$. We denote the set of all assignments as $\Sigma = \mathsf{V} \to \mathbb{D}$.

3.0.2 Instruction semantics. The platform executes a set of instructions of form $\mathsf{inst} = \mathsf{op}(\omega)$, where op is the instruction opcode, and ω are operands. The platform assigns two *transition semantics* to each instruction: a full semantics (allowing speculation) denoted as T and a non-speculative semantics denoted as $\mathsf{T_{ns}}$. Both semantics can be viewed as functions taking an instruction and the current platform state as input and returning the next platform state (obtained after executing the instruction). The full semantics defines the behaviour when the platform *can* speculate (not necessarily enforcing speculative behaviour at all times) while the non-speculative semantics defines behaviours when speculation is disabled.

Example 1. *For the platform model in Fig. 3, the architectural and microarchitectural variables are* $\mathsf{V_a} = \{\mathtt{mem}, \mathtt{regs}\}$ *and* $\mathsf{V_m} = \{\mathtt{cacheTag}, \mathtt{cacheData}\}$*, with* $\mathsf{V} = \mathsf{V_a} \cup \mathsf{V_m}$*. The semantics of the load instruction updates the register file* `regs` *and the cache state variables (*`cacheData`*,* `cacheTag`*) as defined in Fig. 3.*

3.0.3 Modelling speculation. Instructions, e.g., branches or loads (store-to-load forwarding), signal that they are initiating speculation by setting a variable $spec \in \mathsf{V}$. Internally, the full semantics T defines instruction behaviour by conditioning on *spec*: *spec* being set implies that the platform is *currently* speculating. Indeed, *spec* can be set only in the full semantics (T) and not in $\mathsf{T_{ns}}$. Prior work considers specialized semantics that define when *spec* is set (e.g., oracle-based semantics [31]). Since we adopt a hardware-oriented model, we assume that this is explicitly defined in the transition semantics of the speculating instruction. Despeculation is assumed to be similarly defined (this time, however, by unsetting *spec*).

In our current implementation, we restrict speculation to a single frame, and do not support nested speculation (e.g., speculative loads within a branch speculation context). However, this is not a fundamental limitation of our approach; extension to nested speculation is possible by defining a stack of frames storing architectural state.

3.0.4 Executions. The platform consumes a stream of instructions, and transitions on them, thereby producing a trace of states. Then, an input instruction stream $C = \mathsf{inst}_0, \mathsf{inst}_1, \ldots, \mathsf{inst}_n$, starting in state σ_0 leads to a sequence of states $\pi = \sigma_0\ \sigma_1\ \ldots\ \sigma_{n+1}$, where $\sigma_{i+1} = \mathsf{T}(\mathsf{inst}_i, \sigma_i)$ (under the full semantics) and $\sigma_{i+1} = \mathsf{T_{ns}}(\mathsf{inst}_i, \sigma_i)$ (under the non-speculative semantics). The execution generated by instruction stream C from initial state σ is denoted as $\llbracket C \rrbracket(\sigma)$. We similarly define non-speculative executions $\llbracket C \rrbracket_{\mathsf{ns}}(\sigma)$ in which instructions follow the $\mathsf{T_{ns}}$ transition relation.

4 SPECIFICATIONS AND PROBLEM FORMULATION

In this section we formalize hyperproperty specifications (§4.1) and the notion of attack patterns that our approach aims to generate (§4.2). We then discuss a technical limitation of pattern-based approaches in §4.3, and formulate the problem statement in §4.4.

4.1 Hyperproperty-based Security Specification

We follow existing work ([15, 31]) to formalize non-interference-based security specifications.

Non-interference ([15]) states that any pair of executions which begin in states with equivalent values of *public (non-secret) variables* ($\mathsf{V_{pub}}$) continue to have states with equivalent values of *observable variables* ($\mathsf{V_{obs}}$): [2]

$$C \models \mathsf{NI}(\Sigma_{\mathsf{init}}, \mathsf{V_{pub}}, \mathsf{V_{obs}}) \overset{\Delta}{=} \forall \sigma_1, \sigma_2 \in \Sigma_{\mathsf{init}}.$$
$$\sigma_1 \equiv_{\mathsf{V_{pub}}} \sigma_2 \implies \llbracket C \rrbracket(\sigma_1) \equiv_{\mathsf{V_{obs}}} \llbracket C \rrbracket(\sigma_2)$$

This property is parameterized by the choice of Σ_{init}, $\mathsf{V_{pub}}$ and $\mathsf{V_{obs}}$ and intuitively says that the observable variables are not affected by the secret ($\mathsf{V_{sec}} = \mathsf{V} \setminus \mathsf{V_{pub}}$) variables. Non-interference expresses security against an attacker that tries to infer the values of $\mathsf{V_{sec}}$ by observing $\mathsf{V_{obs}}$.

Speculative non-interference enforces non-interference *only if* the program is non-interfering under non-speculative semantics:

$$C \models \mathsf{SNI}(\Sigma_{\mathsf{init}}, \mathsf{V_{pub}}, \mathsf{V_{obs}}) \overset{\Delta}{=} \forall \sigma_1, \sigma_2 \in \Sigma_{\mathsf{init}}.$$
$$(\sigma_1 \equiv_{\mathsf{V_{pub}}} \sigma_2 \land \llbracket C \rrbracket_{\mathsf{ns}}(\sigma_1) \equiv_{\mathsf{V_{obs}}} \llbracket C \rrbracket_{\mathsf{ns}}(\sigma_2)) \implies$$
$$\llbracket C \rrbracket(\sigma_1) \equiv_{\mathsf{V_{obs}}} \llbracket C \rrbracket(\sigma_2)$$

Intuitively, SNI restricts the scope of non-interference enforcement, we refer the reader to [12, 31] for more details.

Conditional/contract-based non-interference [32] is another variant which also restricts the scope of non-interference using an architectural semantics. It requires that a program be non-interfering in the full semantics if it is non-interfering in the architectural semantics. While we focus our presentation on non-interference our technique also applies to these variants, as demonstrated in §7.

4.2 Attack Pattern-based Security

4.2.1 Patterns. A pattern p is a pair (w, ϕ) of a template w and a constraint (ϕ), i.e., a boolean formula. The template is a sequence over opcodes $w = \mathsf{op}_0 \cdot \mathsf{op}_1 \cdots \mathsf{op}_k$ that *structurally* restricts executions the pattern can be embedded in, to those with an opcode

[2] Here, $\sigma_1 \equiv_{\mathsf{V}'} \sigma_2$ for $\mathsf{V}' \subseteq \mathsf{V}$ means that $\sigma_1(v) = \sigma_2(v)$ for all $v \in \mathsf{V}'$ (the assignments agree on all variables in V'). For traces π_1 and π_2, $\pi_1 \equiv_{\mathsf{V}'} \pi_2$ holds if $\pi_1[i] \equiv_{\mathsf{V}'} \pi_2[i]$ for all i.

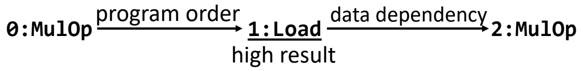


Figure 4: A pattern for a computation-based side channel.

subsequence matching the template. The constraint (ϕ) further *semantically* constrains matching executions to those satisfying it.

Example 2 ($\mathtt{MulOp} - \mathtt{LdOp} - \mathtt{MulOp}$ pattern). *The pattern from Fig. 4 is formalized as (w, ϕ) where, $w = (0 : \mathtt{MulOp}) \cdot (1 : \mathtt{LdOp}) \cdot (2 : \mathtt{MulOp})$ and the constraint is $\phi \equiv \mathsf{datadep}(1 : \mathtt{LdOp}, 2 : \mathtt{MulOp}) \wedge \mathsf{highresult}(1 : \mathtt{LdOp})$, i.e., there is a data dependency between the load and the second multiplication operation, and a the loaded result is high (secret dependent). Intuitively, this pattern matches executions satisfying ϕ with a $\mathtt{MulOp} - \mathtt{LdOp} - \mathtt{MulOp}$ instruction subsequence.*

4.2.2 Execution-embedding. Now we formalize when a pattern embeds (matches) an execution, which determines which programs the pattern flags as exploits. Consider pattern $\mathsf{p} = (w, \phi)$ with template $w = \mathsf{op}_1^* \cdots \mathsf{op}_k^*$, and an execution $\pi = \sigma_0 \cdot \sigma_1 \cdots \sigma_n$. Let the sequence of opcodes in π be $\mathsf{op}_1 \cdots \mathsf{op}_n$, i.e., the transition from σ_i to σ_{i+1} is performed by executing an instruction with opcode op_{i+1}. The pattern p embeds in execution π at a subsequence given by indices $(i_1 < \cdots < i_k)$ if the corresponding opcodes match the template w: $\mathsf{op}_{i_j} = \mathsf{op}_j^*$ for $j \in [1 \cdots k]$, and the execution π satisfies the constraint ϕ. We denote the fact that p embeds at indices $(i_1, \cdots, i_k)$ in trace π as $\pi \models_{(i_1, \cdots, i_k)} (w, \phi)$. Execution π embeds (w, ϕ) if there is a matching subsequence:

$$\pi \models (w, \phi) \overset{\Delta}{=} \exists i_1, \cdots, i_k.\ \pi \models_{(i_1, \cdots, i_k)} (w, \phi)$$

Attack pattern $\mathsf{p} = (w, \phi)$ matches instruction sequence C if there is some execution of C that embeds it:

$$C \models \mathsf{p} \overset{\Delta}{=} \exists \sigma \in \Sigma_{\mathsf{init}}.\ [\![C]\!](\sigma) \models \mathsf{p}$$

Fig. 2 provides examples of patterns matching instructions.

4.3 Non-interference Violation Skeleton

Given a hyperproperty, we aim to generate a set of patterns such that any hyperproperty violation is detected by atleast one of the patterns in this set. However, as illustrated in Example 3 the fact that patterns have fixed length is a fundamental limitation in the violations they can detect.

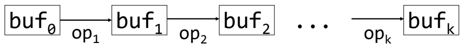


Figure 5: Buffer chain in PlatSynth(k) with operations.

Example 3 (Large skeletons: the PlatSynth(k) platform). *Consider the pedagogical example microarchitecture illustrated in Fig. 5 with $k + 1$ state variables (e.g., buffers): $\mathtt{buf}_0, \cdots, \mathtt{buf}_k \in \mathsf{V}$, and k corresponding operations $\mathsf{op}_1, \mathsf{op}_2, \cdots, \mathsf{op}_k \in \mathsf{Op}$. Operation op_i moves data from $\mathtt{buf}_{i-1}$ to $\mathtt{buf}_i$. Now, consider a non-interference property with $\mathsf{V}_{\mathsf{pub}} = \{\mathtt{buf}_1, \cdots, \mathtt{buf}_k\}$, and $\mathsf{V}_{\mathsf{obs}} = \{\mathtt{buf}_k\}$. That is we want to identify whether the secret input $\mathtt{buf}_0$ affects the observable output $\mathtt{buf}_k$. While the operation sequence $\mathsf{op}_1 \cdots \mathsf{op}_k$ violates this property (it moves data from $\mathtt{buf}_0$ to $\mathtt{buf}_k$), any sequence of length $k - 1$ or less does not.*

Given the possibility of large non-interference violations that would be greater than the size of *any* fixed set of patterns, we qualify our problem statement to detecting those violations with small *skeletons*, as we now define.

Consider an instruction sequence $C = \mathsf{inst}_0\ \mathsf{inst}_1\ \cdots$, and a corresponding execution trace $\pi = [\![C]\!](\sigma)$. For a trace index i, we denote the variables that inst_i depends on as $\mathsf{dep}_\pi(i) \subseteq \mathsf{V}$. We denote the last writer of a variable $v \in \mathsf{V}$ at index i, denoted as $\mathsf{lw}_\pi(v, i) \in \{\mathsf{inst}_0, \cdots, \mathsf{inst}_{i-1}\}$ as the last instruction that writes to v before inst_i. Finally, the set of all (instruction) dependencies of an instruction inst_i is the union of the last writers of all variables it depends on: $\mathsf{idep}_\pi(i) = \bigcup_{v \in \mathsf{dep}_\pi(i)} \mathsf{lw}_\pi(v, i)$. For a trace π, we say that $i_1, \cdots, i_k$ is a subsequence that is *closed under dependencies* if for all $j \in [1 \cdots k]$, $\mathsf{idep}(i_j) \in \{i_1, \cdots, i_k\}$.

Definition 1 (Skeleton). *Suppose the sequence of instructions $\mathsf{inst}_1 \cdots \mathsf{inst}_n$ violates a non-interference property NI. Then, C has an NI-violation skeleton of size k, denoted as $C \not\models_k \mathsf{NI}$ if there exist traces π_1, π_2 with subsequences of length k which are closed under dependencies and also violate NI.*

Discussion: The skeleton identifies instruction sub-sequences that cause the non-interference violation. A length k instruction stream that violates the non-interference property can have multiple skeletons, each of size $\leq k$. Thus, fixing a skeleton size does not bound the program size itself, i.e., large programs can have small skeletons.

4.4 Formal Problem Statement

Formally, given (a) a platform model (with state V and semantics T), and (b) a non-interference security specification $\mathsf{NI}(\Sigma_{\mathsf{init}}, \mathsf{V}_{\mathsf{pub}}$, and $\mathsf{V}_{\mathsf{obs}})$, and (c) a given skeleton size k, we generate a set of patterns P such that any instruction sequence C that violates the non-interference property with a skeleton of size k is detected by one of the patterns in P:

$$\forall C.\ C \not\models_k \mathsf{NI}(\Sigma_{\mathsf{init}}, \mathsf{V}_{\mathsf{pub}}, \mathsf{V}_{\mathsf{obs}}) \implies \exists \mathsf{p} \in \mathsf{P}.\ C \models \mathsf{p} \qquad (1)$$

We refer to Eq. 1 as the k-completeness property.

5 THE SEMPAT APPROACH

In this section we discuss our approach to automatically generate attack patterns satisfying k-completeness (Eq. 1). We provide an overview in §5.1, followed by details in §5.2 and §5.4.

5.1 Pattern Generation Overview

Operation. Figure 6 provides a high-level overview of our approach. We take in a transition system-based platform model (§3), a non-interference specification (§4.1), a constraint grammar G and depth d. We generate a set of patterns P up to depth d, with constraints sourced from G (discussed in §5.3). Our approach has two components: template generation (§5.2) and grammar-based specialization (§5.4). We explain these elements using a running example.

Running Example: Computation Reuse. We use the running example of the generation of the $\mathtt{MulOp} - \mathtt{LdOp} - \mathtt{MulOp}$ pattern from Fig. 4. We generate this pattern based on the computation-reuse platform model (PlatCR), an excerpt of which is provided in Fig. 7. The PlatCR microarchitecture includes a reuse buffer (`reuse_buf`) that

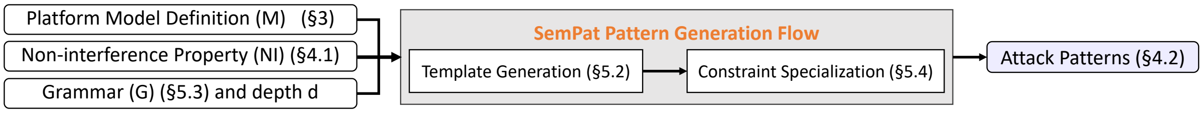

Platform Model Definition (M) (§3)
Non-interference Property (NI) (§4.1)
Grammar (G) (§5.3) and depth d
SemPat Pattern Generation Flow
Template Generation (§5.2)
Constraint Specialization (§5.4)
Attack Patterns (§4.2)

Figure 6: The SemPat approach.

```
type rentry_t = record { op1: word_t, op2: word_t,
    result: word_t };
var reuse_buf : [instr_ind_t]rentry_t;

operation mulop (rd, rs1, rs2) {
  op1 = regfile[rs1]; op2 = regfile[rs2];
  // Check for reuse
  if (∃ i. reuse_buf[i].op1 == op1 &&
      reuse_buf[i].op2 == op2) {
    result = reuse_buf[i].result; // Reuse result
  } else {  // Otherwise invoke multiplier unit
    result = multiplier(op1, op2);
    mulcount = mulcount + 1;
  }
  // Replace reuse_buf entry (at index ind_)
  reuse_buf[ind_] = rentry_t {op1, op2, result};
  regfile[rd] = result;
}
```

Figure 7: PlatCR: Fragment of the computation reuse platform model.

stores the operands and results of previous multiplication (`mul`) instructions. Future `mul` instructions matching operands of previous instructions, can reuse results from the buffer instead of reinvoking the multiplier. We encode the security property that secret data from the memory should not affect the count of multiplier invocations as the property $\mathsf{NI}_{\text{PLATCR}}(\mathsf{init}_{\mathsf{reuse\_buf}}, \mathsf{mem}, \mathsf{mulcount})$ where $\mathsf{init}_{\mathsf{reuse\_buf}}$ constrains all entries in the `reuse_buf` to be initially invalid and $\mathsf{mulcount} \in \mathrm{V}$ counts multiplier invocations.

```
Algorithm 1: GenerateTemplates($M$, NI, $d$)
  Input: Semantic platform definition $M$, non-interference property
         $\mathsf{NI}(\mathrm{V}_{\mathrm{pub}}, \mathrm{V}_{\mathrm{obs}}, \Sigma_{\mathrm{init}})$, depth $d \in \mathbb{N}$
  Output: A set of pattern templates
  Data: acc: the accumulated set of pattern templates
  Function TemplateHelper($w$):
      /* Search depth not reached?                              */
      if $|w| < d$ then
          for op $\in$ Op do
              if op $\cdot$ $w$ propagates taint from $\mathrm{V}^{C}_{\mathrm{pub}}$ to $\mathrm{V}_{\mathrm{obs}}$ then
                  if op $\cdot$ $w \not\models$ NI then acc.append(op $\cdot$ $w$)
              TemplateHelper(op $\cdot$ $w$)
  /* Search over depth $d$ templates                            */
  TemplateHelper($\epsilon$)
  return acc
```

5.2 Template Generation

The first phase of our approach uses an overapproximate analysis to generate templates up to the user-specified depth d. This is performed by the **GenerateTemplates** procedure (Alg. 1) which we discuss with an example. **GenerateTemplates** scans over all templates starting with size 1 (single operations) up to size d. For each template, we first perform (overapproximate) taint analysis to check whether the template propagates taint from the secret inputs ($\mathrm{V}_{\mathrm{sec}}$) to the public outputs ($\mathrm{V}_{\mathrm{obs}}$). **This static syntactic taint analysis** requires an imperatively defined platform semantics.

No taint propagation implies that the template does not violate the non-interference property. On the other hand, if the template propagates taint, we check if it *semantically* violates the non-interference property. We do this by reducing non-interference ($C \models \mathsf{NI}$) to a safety query (e.g., [15, 60]), and solving this query using SMT-based model checking [6, 49].

Example 4 (Template generation for $\mathsf{MulOp} - \mathsf{LdOp} - \mathsf{MulOp}$). *Consider invoking **GenerateTemplates** for the PlatCR platform (Fig. 7) with NI_{PLATCR} as the non-interference property and a search depth of 3. **GenerateTemplates** first considers single operation templates. However, none of them propagate taint from* `mem` *to* `mulcount`*. Subsequently, **GenerateTemplates** finds the two-operation template* $\mathsf{LdOp} - \mathsf{MulOp}$ *does propagate taint from* `mem` *to* `mulcount`*. However, this template does not (semantically) violate non-interference as the* `reuse_buf` *is initially empty, and the multiplier must be invoked in all executions. Eventually, **GenerateTemplates** considers the size 3 template* $\mathsf{MulOp} - \mathsf{LdOp} - \mathsf{MulOp}$*. This template both, propagates taint and semantically violates* NI_{PLATCR}*. This is because the first* `MulOp` *'primes' the reuse buffer, and the second* `MulOp` *uses the primed buffer entry. This will lead to two executions with different* `mulcount` *values, i.e., a non-interference violation.*

GenerateTemplates generates all operation sequences (up to length d) that violate the non-interference property:

Lemma 1. *For $k \leq d$, if $\mathsf{inst}_1 \cdots \mathsf{inst}_k \not\models \mathsf{NI}$ where instruction $\mathsf{inst}_i = \mathsf{op}_i(\omega_i)$, then, $\mathsf{op}_1 \cdots \mathsf{op}_k \in$ **GenerateTemplates**(M, NI, d).*

Proof Sketch. The proof of Lemma 1 is a direct consequence of the algorithm **GenerateTemplates** and the soundness of a taint-based overapproximation. We rely on the fact that, if a template w does not propagate taint from $\mathrm{V}^{C}_{\mathrm{pub}}$ to $\mathrm{V}_{\mathrm{obs}}$, then it also does not violate the non-interference property $\mathsf{NI}(\mathrm{V}_{\mathrm{pub}}, \mathrm{V}_{\mathrm{obs}}, \Sigma_{\mathrm{init}})$. □

5.3 Conjunction-based Pattern Grammar

While templates alone are too overapproximate to be useful, augmenting (specializing) them with constraints (ϕ) leads to more precise patterns with fewer false positives. Specialization is performed using the user provided grammar G that identifies the space of these constraints.

```
Algorithm 2: ConstraintSpecialize($M$, NI, $w$, $G$)
  Input: Semantic platform definition $M$, non-interference property
         NI, pattern template $w$, and a grammar $G$
  Output: A set of patterns
  Data: acc: an accumulated set of patterns
1 Function ConsHelper($\phi, i, L$):
      /* Exhausted all atomic predicates?                        */
2     if $i > |L|$ then acc.append($(w, \phi)$)
      /* Does adding $\neg L[i]$ eliminate all violations?      */
3     else if $\forall \mathsf{inst}_1, \cdots, \mathsf{inst}_{|w|}.\ \mathsf{inst}_1 \cdots \mathsf{inst}_{|w|} \models$
          $(w, \phi \wedge \neg L[i]) \implies \mathsf{inst}_1, \cdots, \mathsf{inst}_{|w|} \models \mathsf{NI}$
4     then ConsHelper($\phi \wedge L[i], i+1, L$)      /* add $L[i]$ */
5     else ConsHelper($\phi, i+1, L$);          /* skip over $L[i]$ */
6 $L$ = ApplyPredicates($w, G$)       /* Create all atoms in $L$ */
7 ConsHelper(true, 0, $L$)            /* Counterfact. addition */
8 return acc
```

5.3.1 Conjunction of Predicate Atoms. We consider constraints which are conjunctions of atoms.[3] That is, ϕ has form: $\phi = \bigwedge_i f_i$, where each f_i is an atom. These atoms (f_i) are generated by applying predicates from a grammar G, such as the one in Table 4, which we use as the default. Each predicate from the grammar is applied to some number of instructions from the pattern, as indicated by its arity (e.g., datadep is an arity-2 (binary) predicate). To identify the instructions a predicate is applied to, we distinguish apart identical opcodes using their position (e.g., the datadep predicate is applied to $(0 : \texttt{MulOp})$, $(2 : \texttt{MulOp})$ in Example 2).

5.3.2 Precision vs. Robustness Tradeoff. A grammar that only allows high-level (architectural) predicates (e.g., Tab. 4) leads to patterns which are less sensitive to microarchitectural implementation details, but have more false positives. Conversely, a grammar that exposes low-level microarchitectural constraints leads to more precise patterns (with fewer false positives). However, these patterns are then specific to the platform microarchitecture. Thus, the pattern grammar exposes a tradeoff between the robustness and expressivity/precision of generated patterns. While our specialization technique (§5.4) requires a conjunction-based grammar (§5.3.1), we are not fundamentally limited to the predicates from Table 4. We explore this further in §7.4 where we augment the default grammar with additional predicates to improve precision.

5.4 Template Specialization with Predicates

The goal of template specialization is making patterns as precise as possible using the pattern grammar (§5.3), while ensuring that they do not miss any violating executions (as required by Eq. 1). Specialization is performed by invoking the **ConstraintSpecialize** procedure (Alg. 2) on every pattern template generated by **GenerateTemplates**. At a high level, starting with the `true` constraint (line 0), **ConstraintSpecialize** continues adding predicate atoms to the constraint.

5.4.1 Candidate predicate atoms. As introduced in §5.3, the pattern constraint is a conjunction of predicate atoms from grammar G. We apply each predicate to operations from the template to get a set of atoms. For example, the binary datadep predicate with the $(0 : \texttt{MulOp}) - (1 : \texttt{LdOp}) - (2 : \texttt{MulOp})$ template results in three atoms: $\text{datadep}(0 : \texttt{MulOp}, 1 : \texttt{LdOp})$, $\text{datadep}(1 : \texttt{LdOp}, 2 : \texttt{MulOp})$ or $\text{datadep}(0 : \texttt{MulOp}, 2 : \texttt{MulOp})$ (we ignore backwards dependencies). **ConstraintSpecialize** first collects all such atoms in a list L using the ApplyPredicates helper function (line 6).

5.4.2 Counterfactual-based atom addition. Adding an atom strengthens the pattern constraint, resulting in it capturing fewer executions. To ensure that the generated pattern does not miss any non-interference violations (and thereby violate Eq. 1) we use *counterfactual atom addition.* Counterfactual addition adds an atom only if adding the negation of the atom leads to only non-violating executions. Intuitively, if the negation leads to only non-violating executions, then adding the atom preserves all violating executions:

Observation 1. *Consider a pattern (w, ϕ), where $|w| = k$, and an atom f. We have, for all $C = \mathsf{inst}_1 \cdots \mathsf{inst}_k$:*

$$(C \models (w, \phi \wedge \neg f) \implies C \models \mathsf{NI}) \implies$$
$$((C \models (w, \phi) \wedge C \not\models \mathsf{NI}) \implies C \models (w, \phi \wedge f))$$

For each atom in L, we check (Alg. 2 line 3) if it satisfies the counterfactual addition condition, specializing the pattern (line 4) if so. If not, we skip over it (line 5) and move to the next atom in L.

Example 5. *For the $(0 : \texttt{MulOp}) - (1 : \texttt{LdOp}) - (2 : \texttt{MulOp})$ template, **ConstraintSpecialize** adds the $f = \text{datadep}(1 : \texttt{LdOp}, 2 : \texttt{MulOp})$ atom to the constraint, since if $(2 : \texttt{MulOp})$ does not depend on $(1 : \texttt{LdOp})$, then its operands are not secret, and the non-interference property is not violated. In a further iteration, **ConstraintSpecialize** adds the atom $\text{highresult}(1 : \texttt{LdOp})$ which says that the loaded value is secret-dependent. Once again, the negation of this would lead to a non-violating execution. This gives us the final pattern with $w = (0 : \texttt{MulOp}) - (1 : \texttt{LdOp}) - (2 : \texttt{MulOp})$ and $\phi = \text{datadep}(1 : \texttt{LdOp}, 2 : \texttt{MulOp}) \wedge \text{highresult}(1 : \texttt{LdOp})$.*

5.4.3 Multiple counterfactuals and branching. Counterfactual addition relies on a strong condition which may not hold for single atoms. In such cases, we consider multiple counterfactual atoms:

Observation 2 (Multiple counterfactuals). *For pattern (w, ϕ), atoms $\{f_i\}_i$, and for any $C = \mathsf{inst}_1 \cdots \mathsf{inst}_{|w|}$:*

$$(C \models (w, \phi \wedge \bigwedge_i \neg f_i) \implies C \models \mathsf{NI}) \implies$$
$$((C \models (w, \phi) \wedge C \not\models \mathsf{NI}) \implies \bigvee_i (C \models (w, \phi \wedge f_i)))$$

We only prove Observation 2 since it generalizes Observation 1.

Proof of Observation 2. Consider a partially specialized template (w, ϕ), and a set of atoms $\{f_i\}_i$. We define sets of programs: $S = \{C \mid C \models (w, \phi)\}$, for each f_i, let $S_i = \{C \mid C \models (w, \phi \wedge f_i)\}$, and finally, $T = \{C \mid C \models \mathsf{NI}\}$. Then, the set $U = S \setminus \bigcup_i S_i$ is of programs satifying,

$$C \models (w, \phi \wedge \bigwedge_i \neg f_i)$$

Now, suppose that the claim was not true. Then there exists a program C, such that (a) $C \in S \cap \overline{T}$, and (b) $C \notin \bigcup_i S_i$. Thus, $C \in (S \cap \overline{T} \cap \overline{\bigcup_i S_i})$, implying (by the definition of U), $C \in U \cap \overline{T}$. However, this means that the antecedent of the claim (which is $\overline{U} \cup T$) does not hold, leading to a contradiction. □

[3] In formal logic, an atom (atomic formula) is a single (indivisible) logical proposition.

Predicate Atom	Meaning	Encoding (assuming RISC-V ISA)
$\mathsf{datadep}(\mathsf{inst}_1, \mathsf{inst}_2)$	Data operand of inst_2 depends on result of inst_1	Last write to inst_2.`rs1` or inst_2.`rs2` is by inst_1
$\mathsf{addrdep}(\mathsf{inst}_1, \mathsf{inst}_2)$	Address operand of inst_2 depends on result of inst_1	Last write to inst_2.`rs1` is by inst_1
$\mathsf{sameaddr}(\mathsf{inst}_1, \mathsf{inst}_2)$	Address operands of inst_1 and inst_2 are the same	inst_2.`addr` = inst_1.`addr` (for memory operations)
$\mathsf{diffaddr}(\mathsf{inst}_1, \mathsf{inst}_2)$	Address operands of inst_1 and inst_2 are different	inst_2.`addr` $\neq$ inst_1.`addr` (for memory operations)
$\mathsf{srcdata}_{\mathtt{reg}}(\mathsf{inst}_1)$	Data operand is read from register `reg`	e.g., inst_1.`rs1` = `reg`, inst_1.`rs2` = `reg` (depends on opcode)
$\mathsf{srcaddr}_{\mathtt{reg}}(\mathsf{inst}_1)$	Address operand is read from register `reg`	inst_1.`rs1` = `reg` (for memory operations)
$\mathsf{destreg}_{\mathtt{reg}}(\mathsf{inst}_1)$	Result is written to a register `reg`	inst_1.`rd` = `reg`
$\mathsf{speculative}(\mathsf{inst})$	Instruction inst initiates speculation	inst sets the *spec* variable
$\mathsf{lowoperands}(\mathsf{inst}_1)$	Operands of inst_1 is independent of $\mathrm{V}_{\mathrm{sec}}$ (= V \ $\mathrm{V}_{\mathrm{pub}}$)	e.g., $\sigma_1(\mathsf{regfile}[\mathtt{rs1}]) = \sigma_2(\mathsf{regfile}[\mathtt{rs1}])$ (depends on opcode)
$\mathsf{lowresult}(\mathsf{inst}_1)$	Result of inst_1 is independent of $\mathrm{V}_{\mathrm{sec}}$ (= V \ $\mathrm{V}_{\mathrm{pub}}$)	$\sigma_1(\mathsf{regfile}[\mathtt{rd}]) = \sigma_2(\mathsf{regfile}[\mathtt{rd}])$

Table 4: Pattern predicate grammar: we generate patterns with constraints as conjunctions of these predicate atoms.

Considering multiple counterfactuals $\{f_i\}$ results in a disjunctive branching (bolded $\bigvee_i$) over atoms. **ConstraintSpecialize** recursively invokes `ConsHelper` on $(w, \phi \wedge f_i)$ for each i. This is sound as the collection of patterns together continue to cover all violating executions. We do not include multi-counterfactuals in Alg. 2 pseudocode for brevity; **please refer to [26] for more details**.

Example 6 (Multiple counterfactuals for PlatSynth). *Consider the platform* PlatSynth(k) *from Ex. 3, with parameter* $k = 2$ *and grammar depth* $d = 3$. ***GenerateTemplates*** *returns the template* $(0 : \mathsf{op}_0) - (1 : \mathsf{op}_0) - (2 : \mathsf{op}_1)$ *for which neither atom* $\mathsf{datadep}(0 : \mathsf{op}_0, 2 : \mathsf{op}_1)$ *nor* $\mathsf{datadep}(1 : \mathsf{op}_0, 2 : \mathsf{op}_1)$ *can be added alone. This is because, even if one is negated, the other might be true, leading to a non-interference violation. However, on adding both negations, the violation is blocked. Our approach then will branch and specialize each case further.*

By Lem. 1 and Obs. 1,2, we have k-completeness of our approach up to skeleton size $k = d$. This is formalized in Theorem 1.

Theorem 1. *Let* W = ***GenerateTemplates***(M, NI, d) *be the templates for depth* d*, and let* P_i = ***ConstraintSpecialize***(M, NI, w_i, G) *be the specialized patterns for each* $w_i \in W$*. Then, for all instruction sequences* $C = \mathsf{inst}_1 \cdots \mathsf{inst}_k$*, with* $k \leq d$*, we have:*

$$C \not\models \mathsf{NI} \implies \exists \mathsf{p} \in \bigcup_i \mathsf{P}_i.\ C \models \mathsf{p}$$

Proof of Theorem 1. Theorem 1 is proven by induction on the number of specialization iterations. Suppose **ConstraintSpecialize** is invoked with arguments (w, NI, M, G). We claim that in Algorithm 2, at each recursive call of ConsHelper with arguments (ϕ, i, L), the following property holds:

$$\begin{aligned}&\forall C.\ (C \models (w, \mathsf{true}) \wedge C \not\models \mathsf{NI})\\ &\quad\implies \exists (w, \phi') \in \mathsf{acc} \cup \{(w, \phi)\}.\ C \models (w, \phi')\end{aligned}$$

I.e., the partially specialized patterns maintained in the acc queue form an overapproximation of the violating programs for skeleton w.

Base case: $\mathsf{ConsHelper}(\mathsf{true}, 0, L)$ trivially implies the property.

Inductive step: Let the property hold for $\mathsf{ConsHelper}(\phi, i, L)$. Then, either we (a) add (w, ϕ) to acc, (b) we skip over $L[i]$ or (c) reinvoke ConsHelper with an incremented i. The property holds immediately in cases (a) and (c). In case (b), we note that the set of programs C that satisfying (w, ϕ) and violating NI is identical to those that violate $(w, \phi \wedge L[i])$ due to Observation 1. A similar argument holds for the multi-counterfactual case (using Obs. 2).

Therefore, by induction, the property holds for all recursive calls of ConsHelper, and hence, the final set of patterns in acc (when there are no more calls to ConsHelper) is a k-complete set of patterns for w. □

5.4.4 **Atom addition order**.

6 EVALUATION METHODOLOGY

6.1 Tool Prototype

We implement our approach in a prototype tool SECANT. SECANT allows us to generate patterns, and analyze program binaries using these patterns. For pattern generation, SECANT allows the user to specify the platform description (§3), and the non-interference specification (§4.1). By default pattern generation uses the predicate grammar from Tab. 4. However, we also allow the user specify custom predicates. Based on these inputs the tool generates a set of patterns as described in §5. The resulting patterns can be inspected by the user, and used for binary analysis. Analysis can either be performed using patterns or the hyperproperty specification. We discuss the details of this analysis in §6.3.

SECANT uses the UCLID5 [58] verification engine as the backend model checker for both, the non-interference checks involved in pattern generation as well as binary analysis. UCLID5 internally compiles model checking queries into SMT [5, 6] queries, and invokes an SMT solver (e.g. Z3 [20], CVC5 [4]) on them.

6.2 Platform Designs

While users can specify their own platform models, we describe the models considered in our experimentation.

6.2.1 PlatCR: Computation Reuse. The computation reuse platform (PlatCR, Fig. 7) was introduced in §5.1. It is based on microarchitectural optimizations that enable dynamic reuse of results of previous high-latency computations. Our model includes a *reuse buffer* (`reuse_buf`), which records operands and results of multiply operations, which are then reused by future operations with matching operands. Dynamic (in-hardware) optimization schemes have been proposed in the literature (e.g., [11, 52, 66]) and also have been hypothesized to be vulnerable to computational side-channel attacks [61]. Our model is based on the scheme proposed in [66].

6.2.2 PlatSS: Store Optimization. The store optimization platform (PlatSS) models microarchitectural optimizations related to the

```
type lsqc_entry_t = record { valid: boolean,
  is_load: boolean, addr: word_t, data: word_t }
var lsqc : [lsqc_index_t]lsqc_entry_t;
var lptr : lsqc_index_t; // Pointer into LSQC
var lscount : int; // Number of memory accesses

operation store (rs2, rs1, imm) {
  addr = addrgen(regfile[rs1], imm);
  data = regfile[rs2];
  if (lsqc[lptr].valid && lsqc[lptr].addr == addr
      && lsqc[lptr].data == data) {
      // Store squashing (silent store)
  } else {              // Perform memory access
      mem[addr] = data; lscount += 1;
      lsqc = ...        // Update the lsqc
  }
}
operation load (rd, rs1, imm) {
  addr = addrgen(regfile[rs1], imm)
  if (lsqc[lptr].addr == addr && lsqc[lptr].valid){
    data = lsqc[lptr].data; // Data forwarding
  } else {              // Perform memory access
    data = mem[data]; lscount += 1;
    lsqc = ...          // Update the LSQC
  }
  regfile[rd] = data;
}
```

Figure 8: PlatSS: Excerpt of the abstract platform modelling load-store optimizations in the microarchitecture.

store unit in the memory hierarchy. PlatSS models the silent stores [42, 43] and store-to-load forwarding optimizations. The former squashes (makes *silent*) stores that would rewrite the same value to the memory. Store-to-load forwarding serves store values to subsequent loads on the same address. While these optimizations have several implementation variants (e.g. [64, 65, 67]), we base our model on the relatively simple *LSQ cache* (load-store queue cache) [44] which allows combining these optimizations.

We illustrate the LSQ cache through an excerpt of our model in Fig. 8. The LSQ cache is a FIFO buffer that records load and store requests. On a *load*, the buffer is checked for store requests to the same address. If there exists such a request (with no subsequent stores to the same address), then the load can be sourced with this store. On a *store* too, the buffer is checked for a request, and the store is squashed (silenced) if the request payload matches the value written by the store. If a valid entry is not found, the new load/store performs a full memory request as usual and the LSQ cache is updated.

6.2.3 Speculation primitives. In addition to the above optimizations, our consider platform models have branch and store-to-load forwarding speculative features. We model speculation non-deterministically, abstracting away from the speculative choice mechanism (e.g. branch predictor). An instruction that speculates moves the platform into speculative mode by setting the *spec* variable (described in §3). Our current implementation only supports a single speculative frame. For example, we do not allow speculating on loads within a branch speculation context. However, this is not a fundamental limitation of our approach.

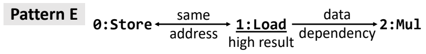


Figure 9: Pattern generated for the STL platform (§7.1).

```
void test1 (uint32_t idx) { // INSECURE
  // Bounds-check-bypass
  if (idx < arr_size)
-   temp &= arr2[arr [idx] << CL_WIDTH];
+   temp &= arr[idx] * SCALAR;
}
```

```
void test2 (uint32_t idx) { // INSECURE
  idx = idx & (arr_size - 1);
  /* Access overwritten secret */
- temp &= arr2[arr[idx] << CL_WIDTH];
+ temp &= arr[idx] * SCALAR;
}
```

Figure 10: Modification of SpectreV1 (left) and SpectreV4 (right) litmus tests from [16] to target a computation-unit side channel (instead of cache-based side channel).

6.3 Binary Analysis

We disassemble the binary using `riscv64-unknown-elf-objdump` and obtain the control flow graph (CFG). We unroll the CFG up to a fixed depth, starting from the function entry point. We instrument the unrolling at the branch instructions with assumptions corresponding to the branch condition, allowing bypassing the condition when speculating (i.e., if *spec* is set). For example "`if (x == 0) stmt`" is instrumented as "`assume(x == 0 || ` *spec*`); stmt`". Each unrolling is analyzed independently w.r.t. a pattern/hyperproperty.

Pattern-based analysis. Given an unrolling, the pattern-based analysis implements the pattern check described in §2.2.1. That is, we first identify all subsequences in that unrolling that match the pattern template. For each such subsequence, we formulate an SMT query that checks whether the subsequence satisfies the pattern constraint, and solve it using the UCLID5 model checker [58].

Hyperproperty-based analysis. The hyperproperty-based analysis follows existing approaches (e.g., [12, 31]). We encode the hyperproperty as a safety property over a self-composition ([15, 68]) of the platform model executing the (identical) unrolled programs. We then use UCLID5 to formulate and solve the resulting query.

7 EXPERIMENTAL RESULTS

Experimental Setup. We perform experiments on a machine with an Intel i9-10900X CPU with 64GB RAM running Ubuntu 20.04. As discussed in §6, we use the UCLID5 model checker [58] and Z3 (version 4.8.7) [20] as the back-end SMT solver for the queries that UCLID5 generates. We mention timeouts for each experiment in its respective section.

Research Questions. We aim to answer the following questions through the evaluation: (a) Can our approach generate new patterns, can these patterns be used to detect attacks, do they provide performance advantages? (b) How well does our approach scale with platform complexity, and the pattern generation search depth? (c) How does the choice of grammar affect false positives?

7.1 RQ1: Can we generate patterns for new vulnerability variants?

To answer **RQ1**, we considered two extensions to the computation reuse platform (PlatCR) introduced in §6. In these extensions we modeled branch speculation and store-to-load forwarding respectively (as discussed in §6.2.3). We then generated patterns for

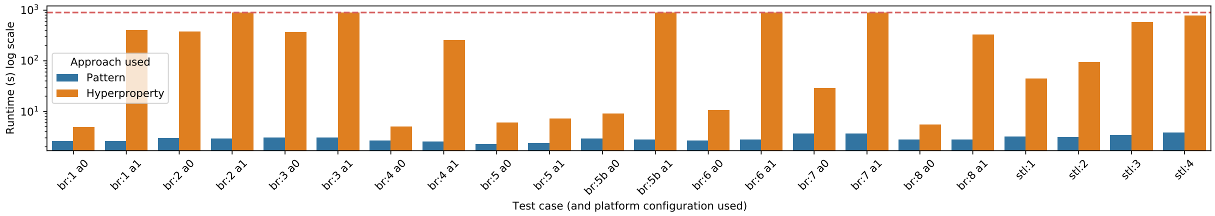


Figure 11: Verification of Branch and STL speculation litmus tests (see Fig. 10) with hyperproperties and generated patterns.

these platforms based on the speculative non-interference property H_{CD} (§2). We used a `reuse_buf` of size 4 (i.e., 4 buffer entries), and grammar search depth $d = 3$.

Platform	Gen. time (ww=16)	Gen. time (ww=20)	Gen. time (ww=24)	Gen. time (ww=32)
Branch + CR	–			69s
STL + CR	92s	334s	TO (1 hour)	

Table 5: Generation times for branch speculation and store-to-load forwarding patterns.

Our approach was able to generate patterns C and D (Fig. 1) for branch speculation with the computation-unit side channel. For store-to-load forwarding, we obtain pattern E (Fig. 9). This pattern corresponds to the situation where the load address matches the store. However, it is served from memory instead, while the store is in the store buffer and has not propagated to memory.

We provide generation run times in Table 5. The STL+CR generation times out on one non-interference query with a platform word width of 32 bits, hence, we use lower word widths. We believe that this is an underlying issue with the back-end Z3 solver. The same hyperproperty (H_{CD}) was used for both (Branch and STL) cases, demonstrating that hyperproperties allow uniform vulnerability specification. *This demonstrates the ability of our approach to generate new patterns.*

7.2 RQ2: Can the generated patterns efficiently find exploits or prove their absence?

To answer **RQ2**, we evaluate whether the generated patterns provide advantages over hyperproperty-based detection. For this, we considered modified versions of Spectre V1 and V4 litmus tests from [16, 38]. For each test, we replaced the secret-dependent load instruction with a multiply instruction that targets the computation-unit side channel. We call these examples SpectreV1-CR and SpectreV4-CR respectively, and illustrate two examples (one for V1 and one for V4) in Fig. 10. We consider 9 such tests (8 unsafe, 1 safe) for SpectreV1-CR and 4 tests (all unsafe) for SpectreV4-CR.

We then evaluated both hyperproperty-based and pattern-based detection approaches on these tests with a timeout of 15 minutes per test. We provide the results in Fig. 11. For each SpectreV1-CR variant test, we considered two platform configurations: one with a non-associative cache, (denoted as `br:<n> a0`) and one with an associative cache (these are marked as `br:<n> a1`). The pattern-based approach verifies all test cases correctly within (~5 seconds).

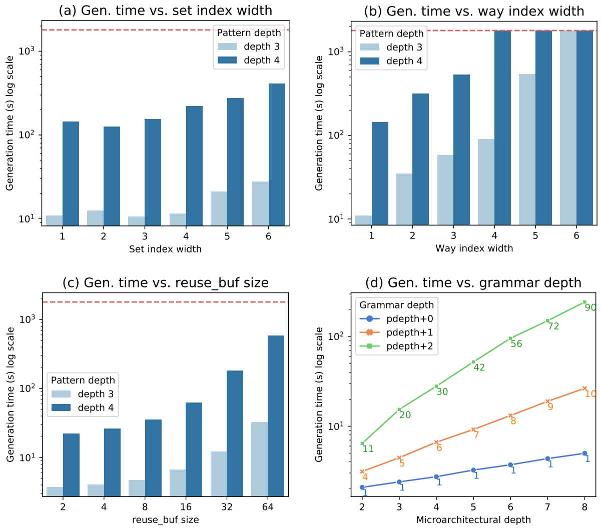


Figure 12: Scalability of pattern generation vs. platform complexity and grammar depth: (a, b) varying set and way index width of the LSQ Cache (Fig. 8), (c) varying size of the reuse buffer (`reuse_buf` in Fig. 7), and (d) generation time and number of generated patterns for PlatSynth with varying microarchitectural (pdep) and grammar (gdep) depths (Ex. 3).

In contrast, the hyperproperty-based approach often takes upwards of 2 minutes, with some timeouts (e.g., `br:2`, `br:3`).

In certain cases (e.g., `br:1`, `br:4`) the hyperproperty-based check run time differs between the two cache configurations. We observe convergence with the non-associative cache (`a0`), while timing out with the associative cache (`a1`). Since the pattern-based approach operates over architectural state, it is resilient to these (microarchitectural) differences. This experiment demonstrates how, by abstracting away complex platform microarchitecture, *patterns enable security verification that scales much better that hyperproperties.*

7.3 RQ3: How well does pattern generation scale?

We now explore the scalability of our approach with the complexity of the platform and the depth of the pattern grammar.

7.3.1 Scalability with model parameters. To evaluate this, we experiment on the PlatSS (§6.2.2) and PlatCR (§6.2.1) platforms. For PlatSS, we specify a non-interference hyperproperty with

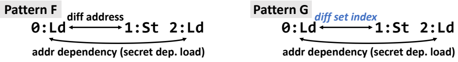
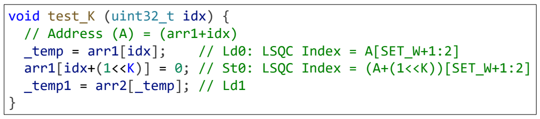


Figure 13: (F) Pattern using default grammar (Tab. 4). (G) More precise pattern after adding the diffindex **predicate.**



```
void test_K (uint32_t idx) {
  // Address (A) = (arr1+idx)
  _temp = arr1[idx];    // Ld0: LSQC Index = A[SET_W+1:2]
  arr1[idx+(1<<K)] = 0; // St0: LSQC Index = (A+(1<<K))[SET_W+1:2]
  _temp1 = arr2[_temp]; // Ld1
}
```

Figure 14: Example illustrating false positives with patterns.

V_{sec} = {mem}, as the secret input, and V_{obs} = {lscount} (number of memory operations propagating to memory) as the public output. For PlatCR, we specify a non-interference property with V_{sec} = {mem}, and V_{obs} = {mulcount}, i.e., number of multiplier invocations as the public output. We vary size parameters of the LSQ cache (§6.2.2) and reuse buffer (§6.2.1) components respectively. For the LSQ cache we sweep through the set index width (same associativity, larger number of sets), and the way index width (same sets, larger associativity) and present results in Fig. 12 (a, b). For the reuse buffer, we sweep through different buffer sizes and present the results in Fig. 12 (c). We perform generation with depths 3 and 4, and a timeout of 30 minutes.

Observations: In Fig. 12(a, b), we observe a more rapid increase in generation time with way index width as compared to the set index width. Our hypothesis is that a larger number of sets increases the width of indexing bitvectors, while larger associativity increases the number case-splits (conditionals) since ways are iterated over. The former increase is slower due to the use of word-level reasoning in SMT solvers, while the latter has an explicit if-then-else encoding, and hence a larger formula. We observe a similar increase for the reuse buffer. In general, the run time depends on how efficiently the back-end SMT solver reasons about the microarchitectural structure. While the run time increases exponentially, our approach generates patterns on the order of minutes for realistic sizes.

7.3.2 Scalability with search depth. In order to investigate the scalability of our approach with the depth of the grammar, we perform experimentation with the PlatSynth(k) platform (Ex. 3). This platform is parameterized by a *platform depth* pdep($= k$), which is the number of microarchitectural buffers. The non-interference property specifies buf_0 as the secret input, and $\text{buf}_{\text{pdep}-1}$ as public output. We apply our approach to this platform for grammar depths gdep ∈ {pdep, pdep + 1, pdep + 2} for varying values of pdep.

Fig. 12(d) presents the run times as well as the number of patterns generated. For the gdep = pdep case, we generate only one pattern with skeleton $\text{op}_1 \cdots \text{op}_{\text{pdep}}$ and constraint $\wedge_i \text{datadep}(\text{op}_i, \text{op}_{i+1})$ signifying data propagation through these operations. With a higher gdep, multiple instruction sequences can lead to a violation due to redundancy. Capturing all violations requires branching with multiple counterfactuals (Ex. 6). We need 2-counterfactuals for gdep = pdep + 1 and 3-counterfactuals for gdep = pdep + 2. Branching leads to a run time blowup as evidenced in Fig. 12(d).

Check	Result with `test_K` (Fig. 14) and `SET_W` set index	
	K > SET_W + 2	K ≤ SET_W + 2
Hyperproperty	SAFE	UNSAFE
Pat. F (Fig. 13)	UNSAFE	UNSAFE
Pat. G (Fig. 13)	SAFE	UNSAFE

Table 6: Precision of patterns on `test_K` (Fig. 14).

7.4 RQ4: How does the choice of grammar affect false positives?

To explore this, we consider a variant of the PlatSS platform. As in the original model (Fig. 8), a load is sourced from the LSQ cache if there exists a valid entry with the same address. Otherwise, it is sourced from memory and updates the LSQ cache. However, in this variant, a store invalidates all LSQ entries with the same set index (to avoid loading stale values). We perform pattern generation with the memory access count (lscount) as public output, with grammar depth 3. We get pattern F (Fig. 13) with an address dependency between (0:Ld) and (2:Ld), (as in SpectreV1), and where the intervening store (1:St) *does not* invalidate the LSQ entry (captured by diffaddr). If 1:St invalidated the LSQ entry, 2:Ld would access the memory in all executions, leading to no violating behaviors.

This pattern flags the code in Fig. 14 due to a match on instructions Ld0, St0, Ld1. While the addresses of the store and the first load (Fig. 14) are different, their set indices are a slice (sub-word) of the address and hence can be identical. Thus, under the condition that K > SET_W + 2, the store will be in the same set as the first load, and the LSQ entry will be invalidated. Hence, under this condition, test_K is in fact safe, and pattern F produces a false positive.

To address this, we can add a new predicate diffindex to the grammar. diffindex checks if the set indices (defined as an address slice) of two memory instructions are different. Pattern generation with this augmented grammar results in pattern G (Fig. 13). This pattern only flags test_K if K ≤ SET_W+2, and is hence more precise. We summarize these observations in Tab. 6. Thus, grammars over (high-level) architectural state result in patterns with more false positives. The choice of grammar *exposes a precision-complexity tradeoff - more precise patterns can be generated at the cost of a carefully tailored, microarchitecture-specific grammar.*

8 DISCUSSION AND LIMITATIONS

Applicability and Scope. We have applied pattern generation to speculative (e.g., §7.1, Fig. 9) and non-speculative (e.g., §7.3, Fig. 13) execution, and multiple side channels (§6.2.1, §6.2.2). Crucially, this is possible due to the power of non-interference-based specifications: they can express varied attacker scenarios (e.g., secure-programming, constant-time [9, 32]) and platform semantics. While our approach is general, it requires formulating non-interference specifications that accurately capture the threat model. While this requires much care and expertise, the effort in developing these specifications can be amortized over patterns generated, and subsequently, programs verified/exploits found.

Platform complexity. Our evaluation (§7) considers abstract platform models (ref. §6) that are similar to those adopted in previous SH approaches (e.g., [12, 31, 78]). While these abstract models expose security relevant microarchitectural detail, they are much simpler than full processor RTL. Larger designs will result in more costly analysis (for both, taint analysis and model-checking required in the non-interference checks).

In this paper our focus has been identifying core theory and techniques for pattern generation. The main requirement of our approach is a platform model that is amenable to static taint analysis and symbolic model-checking. These can be performed on RTL using off-the-shelf verification tools (e.g., JasperGold, Yosys/SymbiYosys [13, 77]). Applying our approach to RTL hardware is an impactful direction for future work. Further, techniques extracting abstract models from RTL (e.g., [26, 78]) could aid in this direction.

Pattern Grammar and False Positives. While patterns using architecture level predicates are more robust/microarchitecture-independent, they lead to more false positives. As demonstrated in §7.4, using specialized grammars that expose microarchitectural execution details (e.g., cache-indexing, replacement/prediction policies) helps reduce false positives. However, this is at the cost of (a) less performant analysis (it is abstraction that makes pattern-based approaches scale) and (b) more microarchitecture-dependent patterns. Opinion in literature is divided on whether or not to expose microarchitectural detail to software analyses (e.g., [10]). We view the grammar as a tradeoff: a generic grammar leads to abstract and thus efficient analysis at the cost of more false positives, while a specialized grammar is more precise at the cost of being more microarchitecture-specific. While our current approach requires a user-specified grammar (Fig. 6), future work could automate this, e.g., by using counter-example guided predicate discovery [3, 18].

9 RELATED WORK

Microarchitectural optimizations and vulnerabilities. Our work targets hardware execution vulnerabilities, from Spectre and Meltdown and their variants [39, 40, 45, 63] to more recent attacks (e.g., targeting store/line-fill buffers [8, 48, 62, 72]). Good overviews of these vulnerabilities may be found in [7, 24, 34]. These vulnerabilities are possible because of microarchitectural optimizations such as instruction reuse [66] and silent stores [42, 44] which we have used in our experiments. Attacks exploiting these mechanisms were hypothesized in [61] among which some have recently been demonstrated on actual hardware (e.g., [50]).

Software Analysis. Several approaches perform software analysis, adopting different specifications, models, and techniques.

Semantic hyperproperty-based verification. Semantic hyperproperty-based approaches (e.g., [12, 21, 30, 31]) formulate security as a hyperproperty [15, 41, 60, 68] over executions of the program on an abstract platform model. These serve as inputs to our approach.

Symbolic software analysis. While the earlier works develop high-level hyperproperties that capture varied microarchitectural mechanisms, other approaches adopt more specific security models (e.g., constant-time execution). This allows them to develop specialized and more efficient analysis techniques (see [25] for a systematic evaluation and comparison), based on symbolic execution (SE) [19, 33, 37, 74] or *relational* symbolic execution (RelSE) [16, 17, 23]. Our work is orthogonal to these; our main goal is pattern generation, not binary analysis. Our approach is also not limited to a specific (e.g., constant-time) leakage model. We can perform generation for new vulnerabilities/leakage models as long as they are representable using a non-interference property.

Pattern-based detection. Previous work performing pattern-based detection (e.g., [51, 59, 69]) manually defines patterns. We develop an approach to systematically generate patterns that are complete (up to a skeleton size), thus complementing these approaches.

Microarchitectural verification/abstraction. The patterns that we generate can be thought of as software-side abstractions extracted based on our semantic platform analysis.

Contract-based abstractions. Approaches (e.g., [32, 75]) develop security contracts at the hardware/software interface. By proving that (a) the hardware refines the contract and (b) software satisfies the contract, these approaches guarantee software security. While the contract can also be viewed as a hardware/software abstraction, it is of a different nature than the patterns we generate.

Security verification/side-channel analysis. Several approaches directly verify hardware RTL w.r.t. security properties either formally (using symbolic techniques) [22, 71, 73] or using faster but incomplete techniques (e.g., fuzzing) [54–56, 70]. Finally, there are approaches that perform automated extraction/validation of side-channels with white/black box designs [29, 53, 57, 76]. While this too can be seen as extracting security relevant hardware abstractions, it needs to be paired with software analysis techniques to be useful for software security verification.

10 CONCLUSION

In this work, we presented an approach to convert a given platform model and non-interference-based security hyperproperty into a set of attack patterns. Our automatic generation approach improves on manual pattern creation (which can result in missed patterns) by guaranteeing that the generated patterns capture all non-interference violations up to a certain size. We implemented our approach in a prototype tool. Our evaluation resulted in the identification of, to our knowledge, previously unknown patterns for Spectre BCB and Spectre STL, and other vulnerability variants. We also demonstrated improved verification performance using generated patterns as compared to the original hyperproperty. By providing a systematic way to generate patterns, our work combines the best of both worlds: formal guarantees of hyperproperty-based specification and scalability of pattern-based verification.

REFERENCES

[1] Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Trans. Program. Lang. Syst.* 36, 2, Article 7 (Jul 2014), 74 pages. https://doi.org/10.1145/2627752

[2] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo M. K. Martin, Mukund Raghothaman, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. 2013. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*. IEEE, 1–8. https://doi.org/10.1109/fmcad.2013.6679385

[3] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. 2001. Automatic predicate abstraction of C programs. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation* (Snowbird, Utah, USA) *(PLDI '01)*. Association for Computing Machinery, New York, NY, USA, 203–213. https://doi.org/10.1145/378795.378846

[4] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems.*

Springer International Publishing, 415–442. https://doi.org/10.1007/978-3-030-99524-9_24

[5] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. 2016. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`.

[6] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. 2009. Satisfiability Modulo Theories. In *Handbook of Satisfiability*.

[7] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A Systematic Evaluation of Transient Execution Attacks and Defenses. (Aug. 2019), 249–266. https://www.usenix.org/conference/usenixsecurity19/presentation/canella

[8] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. 2019. Fallout: Leaking Data on Meltdown-resistant CPUs. *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (2019).

[9] Sunjay Cauligi, Craig Disselkoen, Klaus v. Gleissenthall, Dean Tullsen, Deian Stefan, Tamara Rezk, and Gilles Barthe. 2020. Constant-time foundations for the new spectre era. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20)*. ACM. https://doi.org/10.1145/3385412.3385970

[10] Sunjay Cauligi, Craig Disselkoen, Daniel Moghimi, Gilles Barthe, and Deian Stefan. 2022. SoK: Practical Foundations for Software Spectre Defenses. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 666–680. https://doi.org/10.1109/sp46214.2022.9833707

[11] Desiree Charles, Ali R. Hurson, and Narayanan Vijaykrishnan. 2002. Improving ILP with instruction-reuse cache hierarchy. *Fifth International Conference on Algorithms and Architectures for Parallel Processing, 2002. Proceedings.* (2002), 206–213.

[12] Kevin Cheang, Cameron Rasmussen, Sanjit Seshia, and Pramod Subramanyan. 2019. A Formal Approach to Secure Speculation. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*. IEEE, 288–28815. https://doi.org/10.1109/csf.2019.00027

[13] Claire Wolf, et. al. 2022. SymbiYosys. https://github.com/YosysHQ/sby.

[14] Edmund M. Clarke, Orna Grumberg, and David E. Long. 1994. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.* 16, 5 (Sep. 1994), 1512–1542. https://doi.org/10.1145/186025.186051

[15] Michael R. Clarkson and Fred B. Schneider. 2008. Hyperproperties. *2008 21st IEEE Computer Security Foundations Symposium* (2008), 51–65.

[16] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. 2021. Hunting the Haunter - Efficient Relational Symbolic Execution for Spectre with Haunted RelSE. *Proceedings 2021 Network and Distributed System Security Symposium* (2021). https://api.semanticscholar.org/CorpusID:231878700

[17] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. 2023. Binsec/Rel: Symbolic Binary Analyzer for Security with Applications to Constant-Time and Secret-Erasure. *ACM Trans. Priv. Secur.* 26, 2, Article 11 (Apr 2023), 42 pages. https://doi.org/10.1145/3563037

[18] Satyaki Das and David L. Dill. 2002. Counter-Example Based Predicate Discovery in Predicate Abstraction. In *Formal Methods in Computer-Aided Design*, Mark D. Aagaard and John W. O'Leary (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 19–32. https://doi.org/10.1145/378795.378846

[19] Robin David, Sébastien Bardin, Thanh Dinh Ta, Laurent Mounier, Josselin Feist, Marie-Laure Potet, and Jean-Yves Marion. 2016. BINSEC/SE: A Dynamic Symbolic Execution Toolkit for Binary-Level Analysis. *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)* 1 (2016), 653–656. https://api.semanticscholar.org/CorpusID:7488274

[20] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*.

[21] Xaver Fabian, Marco Guarnieri, and Marco Patrignani. 2022. Automatic Detection of Speculative Execution Combinations. *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (2022).

[22] Mohammad Rahmani Fadiheh, Dominik Stoffel, Clark W. Barrett, Subhasish Mitra, and Wolfgang Kunz. 2018. Processor Hardware Security Vulnerabilities and their Detection by Unique Program Execution Checking. *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)* (2018), 994–999.

[23] Gian Pietro Farina, Stephen Chong, and Marco Gaboardi. 2019. Relational Symbolic Execution. In *Proceedings of the 21st International Symposium on Principles and Practice of Declarative Programming (PPDP '19)*. ACM. https://doi.org/10.1145/3354166.3354175

[24] Qian Ge, Yuval Yarom, David A. Cock, and Gernot Heiser. 2018. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *J. Cryptogr. Eng.* 8, 1 (2018), 1–27. https://doi.org/10.1007/S13389-016-0141-6

[25] Antoine Geimer, Mathéo Vergnolle, Frédéric Recoules, Lesly-Ann Daniel, Sébastien Bardin, and Clémentine Maurice. 2023. A Systematic Evaluation of Automated Tools for Side-Channel Vulnerabilities Detection in Cryptographic Libraries. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (Copenhagen, Denmark) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 1690–1704. https://doi.org/10.1145/3576915.3623112

[26] Adwait Godbole, Kevin Cheang, Yatin A. Manerkar, and Sanjit A. Seshia. 2024. Lifting Micro-Update Models from RTL for Formal Security Analysis. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 631–648. https://doi.org/10.1145/3620665.3640418

[27] Adwait Godbole, Yatin A. Manerkar, and Sanjit A. Seshia. 2024. SemPat: Using Hyperproperty-based Semantic Analysis to Generate Microarchitectural Attack Patterns. *arXiv preprint arXiv: 2406.05403* (2024).

[28] Joseph A. Goguen and José Meseguer. 1984. Unwinding and Inference Control. *1984 IEEE Symposium on Security and Privacy* (1984), 75–75.

[29] Ben Gras, Cristiano Giuffrida, Michael Kurth, Herbert Bos, and Kaveh Razavi. 2020. ABSynthe: Automatic Blackbox Side-channel Synthesis on Commodity Microarchitectures. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020.* The Internet Society. https://www.ndss-symposium.org/ndss-paper/absynthe-automatic-blackbox-side-channel-synthesis-on-commodity-microarchitectures/

[30] Roberto Guanciale, Musard Balliu, and Mads Dam. 2020. InSpectre: Breaking and Fixing Microarchitectural Vulnerabilities by Formal Analysis. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20)*. ACM. https://doi.org/10.1145/3372297.3417246

[31] Marco Guarnieri, Boris Köpf, José Francisco Morales, Jan Reineke, and Andrés Sánchez. 2020. Spectector: Principled Detection of Speculative Information Flows. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1–19. https://doi.org/10.1109/sp40000.2020.00011

[32] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. 2021. Hardware-Software Contracts for Secure Speculation. *2021 IEEE Symposium on Security and Privacy (SP)* (2021), 1868–1883.

[33] Shengjian Guo, Yueqi Chen, Peng Li, Yueqiang Cheng, Huibo Wang, Meng Wu, and Zhiqiang Zuo. 2019. SPECUSYM: Speculative Symbolic Execution for Cache Timing Leak Detection. *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)* (2019), 1235–1247. https://api.semanticscholar.org/CorpusID:207869919

[34] Guangyuan Hu, Zecheng He, and Ruby B. Lee. 2021. SoK: Hardware Defenses Against Speculative Execution Attacks. In *2021 International Symposium on Secure and Private Execution Environment Design (SEED)*. IEEE, 108–120. https://doi.org/10.1109/seed51797.2021.00023

[35] Intel. 2023. Data Operand Independent Timing Instruction Set Architecture (ISA) Guidance. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/data-operand-independent-timing-isa-guidance.html Accessed: 2023-11-23.

[36] Susmit Jha and Sanjit A. Seshia. 2015. A theory of formal synthesis via inductive learning. *Acta Informatica* 54 (2015), 693–726.

[37] James C. King. 1976. Symbolic execution and program testing. *Commun. ACM* 19, 7 (Jul 1976), 385–394. https://doi.org/10.1145/360248.360252

[38] Paul Kocher. 2018. *Spectre Mitigations in Microsoft's C/C++ Compiler.* https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html

[39] Paul C. Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Michael Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. *2019 IEEE Symposium on Security and Privacy (SP)* (2019), 1–19.

[40] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2024. Spectre Returns! Speculation Attacks Using the Return Stack Buffer. *IEEE Design & Test* 41, 2, 47–55. https://doi.org/10.1109/mdat.2024.3352537

[41] Elisavet Kozyri, Stephen Chong, and Andrew C. Myers. 2022. Expressing Information Flow Properties. *Found. Trends Priv. Secur.* 3 (2022), 1–102.

[42] Kevin M. Lepak, Gordon B. Bell, and Mikko H. Lipasti. 2001. Silent Stores and Store Value Locality. *IEEE Trans. Computers* 50 (2001), 1174–1190.

[43] Kevin M. Lepak and Mikko H. Lipasti. 2000. On the value locality of store instructions. *Proceedings of 27th International Symposium on Computer Architecture (IEEE Cat. No.RS00201)* (2000), 182–191.

[44] Kevin M. Lepak and Mikko H. Lipasti. 2000. Silent stores for free. *Proceedings 33rd Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-33 2000* (2000), 22–31.

[45] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul C. Kocher, Daniel Genkin, Yuval Yarom, and Michael Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium*.

[46] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks are Practical. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 605–622. https://doi.org/10.1109/sp.2015.43

[47] Daniel Lustig, Geet Sethi, Margaret Martonosi, and Abhishek Bhattacharjee. 2016. COATCheck: Verifying Memory Ordering at the Hardware-OS Interface. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, Georgia, USA) *(ASPLOS '16)*. Association for Computing Machinery, New York, NY, USA, 233–247. https://doi.org/10.1145/2872362.2872399

[48] Giorgi Maisuradze and Christian Rossow. 2018. ret2spec: Speculative Execution Using Return Stack Buffers. *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018). https://api.semanticscholar.org/CorpusID:51804116

[49] Kenneth L. McMillan. 1993. *Symbolic model checking*. Kluwer. https://doi.org/10.1007/978-1-4615-3190-6_3

[50] Daniel Moghimi. 2023. Downfall: Exploiting Speculative Data Gathering. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 7179–7193. https://www.usenix.org/conference/usenixsecurity23/presentation/moghimi

[51] Nicholas Mosier, Hanna Lachnitt, Hamed Nemati, and Caroline Trippel. 2022. Axiomatic hardware-software contracts for security. In *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA '22)*. ACM. https://doi.org/10.1145/3470496.3527412

[52] Onur Mutlu, Hyesoon Kim, Jared Stark, and Yale N. Patt. 2005. On Reusing the Results of Pre-Executed Instructions in a Runahead Execution Processor. *IEEE Computer Architecture Letters* 4 (2005), 2–2.

[53] Hamed Nemati, Pablo Buiras, Andreas Lindner, Roberto Guanciale, and Swen Jacobs. 2020. Validation of Abstract Side-Channel Models for Computer Architectures. *Computer Aided Verification* 12224 (2020), 225 – 248. https://api.semanticscholar.org/CorpusID:211567271

[54] Shirin Nilizadeh, Yannic Noller, and Corina S. Pasareanu. 2018. DifFuzz: Differential Fuzzing for Side-Channel Analysis. *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)* (2018), 176–187.

[55] Yannic Noller and Saeid Tizpaz-Niari. 2021. QFuzz: quantitative fuzzing for side channels. *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis* (2021).

[56] Oleksii Oleksenko, Christof Fetzer, Boris Köpf, and Mark Silberstein. 2022. Revizor: testing black-box CPUs against speculation contracts. (Feb. 2022). https://doi.org/10.1145/3503222.3507729

[57] Oleksii Oleksenko, Marco Guarnieri, Boris Köpf, and Mark Silberstein. 2023. Hide and Seek with Spectres: Efficient discovery of speculative information leaks with random testing. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE. https://doi.org/10.1109/sp46215.2023.10179391

[58] Elizabeth Polgreen, Kevin Cheang, Pranav Gaddamadugu, Adwait Godbole, Kevin Laeufer, Shaokai Lin, Yatin A. Manerkar, Federico Mora, and Sanjit A. Seshia. 2022. UCLID5: Multi-modal Formal Modeling, Verification, and Synthesis. In *34th International Conference on Computer Aided Verification (CAV) (Lecture Notes in Computer Science, Vol. 13371)*. Springer, 538–551.

[59] Hernán Ponce-de Leon and Johannes Kinder. 2022. Cats vs. Spectre: An Axiomatic Approach to Modeling Speculative Execution Attacks. (May 2022). https://doi.org/10.1109/sp46214.2022.9833774

[60] John Rushby. 1992. *Noninterference, Transitivity, and Channel-Control Security Policies*. Technical Report. http://www.csl.sri.com/papers/csl-92-2/

[61] Jose Rodrigo Sanchez Vicarte, Pradyumna Shome, Nandeeka Nayak, Caroline Trippel, Adam Morrison, David Kohlbrenner, and Christopher W. Fletcher. 2021. Opening Pandora's Box: A Systematic Study of New Ways Microarchitecture Can Leak Private Data. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 347–360. https://doi.org/10.1109/isca52012.2021.00035

[62] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19)*. ACM. https://doi.org/10.1145/3319535.3354252

[63] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. 2019. NetSpectre: Read Arbitrary Memory over Network. *ArXiv* abs/1807.10535 (2019).

[64] Tingting Sha, Milo M.K. Martin, and Amir Roth. 2006. NoSQ: Store-Load Communication without a Store Queue. *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)* 27, 1, 285–296. https://doi.org/10.1109/mm.2007.17

[65] Tingting Sha, Milo M. K. Martin, and Amir Roth. 2005. Scalable store-load forwarding via store queue index prediction. *38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)* (2005), 12 pp.–170.

[66] Avinash Sodani and Gurindar S. Sohi. 1997. Dynamic Instruction Reuse. *Conference Proceedings. The 24th Annual International Symposium on Computer Architecture* (1997), 194–205.

[67] Samantika Subramaniam and Gabriel Loh. 2006. Fire-and-Forget: Load/Store Scheduling with No Store Queue at All. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. IEEE, 273–284. https://doi.org/10.1109/micro.2006.26

[68] Tachio Terauchi and Alex Aiken. 2005. Secure Information Flow as a Safety Problem. In *Proceedings of the 12th International Conference on Static Analysis* (London, UK) *(SAS'05)*. Springer-Verlag, Berlin, Heidelberg, 352–367. https://doi.org/10.1007/11547662_24

[69] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. 2019. Security Verification via Automatic Hardware-Aware Exploit Synthesis: The CheckMate Approach. *IEEE Micro* 39 (2019), 84–93.

[70] Aakash Tyagi, Addison Crump, Ahmad-Reza Sadeghi, Garrett Persyn, Jeyavijayan Rajendran, Patrick Jauernig, and Rahul Kande. 2022. TheHuzz: Instruction Fuzzing of Processors Using Golden-Reference Models for Finding Software-Exploitable Vulnerabilities. *ArXiv* abs/2201.09941 (2022).

[71] Klaus v. Gleissenthall, Rami Gökhan Kıcı, Deian Stefan, and Ranjit Jhala. 2021. Solver-Aided Constant-Time Hardware Verification. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 429–444. https://doi.org/10.1145/3460120.3484810

[72] Stephan van Schaik, Alyssa Milburn, Sebastian Osterlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue In-Flight Data Load. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 88–105. https://doi.org/10.1109/sp.2019.00087

[73] Klaus von Gleissenthall, Rami Gökhan Kici, Deian Stefan, and Ranjit Jhala. 2019. IODINE: Verifying Constant-Time Execution of Hardware. *ArXiv* abs/1910.03111 (2019). https://api.semanticscholar.org/CorpusID:197672843

[74] Guanhua Wang, Sudipta Chattopadhyay, Arnab Kumar Biswas, Tulika Mitra, and Abhik Roychoudhury. 2020. KLEESpectre: Detecting Information Leakage through Speculative Cache Attacks via Symbolic Execution. *ACM Transactions on Software Engineering and Methodology* 29, 3 (June 2020), 1–31. https://doi.org/10.1145/3385897

[75] Zilong Wang, Gideon Mohr, Klaus von Gleissenthall, Jan Reineke, and Marco Guarnieri. 2023. Specification and Verification of Side-channel Security for Open-source Processors via Leakage Contracts. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (, Copenhagen, Denmark,) *(CCS '23)*. Association for Computing Machinery, New York, NY, USA, 2128–2142. https://doi.org/10.1145/3576915.3623192

[76] Daniel Weber, Ahmad Ibrahim, Hamed Nemati, Michael Schwarz, and Christian Rossow. 2021. Osiris: Automated Discovery of Microarchitectural Side Channels. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 1415–1432. https://www.usenix.org/conference/usenixsecurity21/presentation/weber

[77] Clifford Wolf, Johann Glaser, and Johannes Kepler. 2013. Yosys-A Free Verilog Synthesis Suite.

[78] Yu Zeng, Aarti Gupta, and Sharad Malik. 2022. Automatic generation of architecture-level models from RTL designs for processors and accelerators. In *Proceedings of the 2022 Conference & Exhibition on Design, Automation & Test in Europe* (Antwerp, Belgium) *(DATE '22)*. European Design and Automation Association, Leuven, BEL, 460–465.