Axiomatic Hardware-Software Contracts for Security

Nicholas Mosier
nmosier@stanford.edu
Stanford University
Stanford, California, USA

Hanna Lachnitt
lachnitt@stanford.edu
Stanford University
Stanford, California, USA

Hamed Nemati
hnnemati@stanford.edu
Stanford University
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany

Caroline Trippel
trippel@stanford.edu
Stanford University
Stanford, California, USA

ABSTRACT

We propose *leakage containment models* (LCMs)—novel *axiomatic* security contracts which support formally reasoning about the security guarantees of programs when they run on particular microarchitectures. Our core contribution is an axiomatic vocabulary for formalizing LCMs, derived from the established axiomatic vocabulary for formalizing processor memory consistency models. Using this vocabulary, we formalize *microarchitectural leakage*—focusing on leakage through hardware memory systems—so that it can be automatically detected in programs and provide a taxonomy for classifying said leakage by severity. To illustrate the efficacy of LCMs, we first demonstrate that our leakage definition faithfully captures a sampling of (transient and non-transient) microarchitectural attacks from the literature. Second, we develop a static analysis tool based on LCMs which automatically identifies Spectre vulnerabilities in programs and scales to analyze real-world crypto-libraries.

CCS CONCEPTS

• Computer systems organization → Architectures; **• Security and privacy → Formal security models**; **Side-channel analysis and countermeasures**; **• Software and its engineering →** *Formal software verification.*

KEYWORDS

hardware security, side-channel attacks, hardware-software contracts, Spectre, memory consistency models

ACM Reference Format:
Nicholas Mosier, Hanna Lachnitt, Hamed Nemati, and Caroline Trippel. 2022. Axiomatic Hardware-Software Contracts for Security. In *The 49th Annual International Symposium on Computer Architecture (ISCA '22), June 18–22, 2022, New York, NY, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3470496.3527412

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ISCA '22, June 18–22, 2022, New York, NY, USA
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8610-4/22/06…$15.00
https://doi.org/10.1145/3470496.3527412

1 INTRODUCTION

Microarchitectural attacks [23] are side/covert channel attacks which enable leakage/communication as a direct result of hardware optimizations. In doing so, they expose a notable deficiency in how hardware-software contracts have historically defined *software-visible state*. Rather than consisting solely of state that can be directly accessed with committed user-facing instructions (i.e., architectural state), software-visible state also includes any state that can be leaked/communicated via hardware side/covert channels.

Writing secure software in the presence of hardware side/covert channels requires new hardware-software contracts which remedy our inadequate definition of software-visibility. Specifically, ***security contracts*** should be designed which soundly abstract and expose to software the security implications of particular microarchitectures. Such contracts can support the design of automated tools that *detect* vulnerabilities in programs, *evaluate* hardware and software mitigations, and (optimally) *repair* vulnerable software to render it secure.

Hardware-software contracts for security: One established way to counter microarchitectural attacks that manifest as timing channels is with *constant-time (CT) programming*. CT programming is a paradigm that disallows the processing of secrets by *transmit instructions* [34, 79] (i.e., transmitters) which can leak their results, operands, or even data at rest in architectural structures [69] via their variable impact on execution time. However, even CT programming requires a type of security contract which precisely identifies transmitters and articulates their leakage implications. Historically, CT programming disallows secret-dependent branches and memory accesses [7, 21, 61, 62]. However, these restrictions are insufficient for modern hardware where secrets can be steered towards *transient* transmitters [37, 42, 49, 64, 67]. Also, the scope of transmitters extends beyond branch and memory instructions [69].

To address the need for security contracts, various proposals have emerged [6, 15, 16, 19, 22, 25–27, 47, 68, 76, 78, 82]. Some require hardware enhancements to explicitly track/enforce contract-level security primitives [6, 76, 78, 82]. Other contracts restrict the scope of hardware features that they consider, e.g., focusing on in-order [16, 25, 26] and single-core processor designs [15, 16, 22, 25–27, 47, 68]. The state-of-the-art security contracts solely expose transient leakage through microarchitecture to software [15, 16, 22, 25, 26, 47, 68] or highly restrict the non-transient leakage they can

capture [27]. Moreover, these contracts are typically based on operational models of processor designs. Verifying operational security contracts is challenging since a hardware designer must prove that a given microarchitecture is a sound refinement of some abstract, simplified processor model. Such proofs are not automatable by existing verification approaches and would require significant manual effort.

Our approach: Towards resolving the limitations of prior work, **our first insight** is that security contracts should explicitly account for *microarchitectural implementation details* that underpin hardware leakage. In doing so, security contracts can be directly related to, and eventually synthesized from, the microarchitectures they represent [30]. Moreover, a generic microarchitectural leakage definition can be established which encompasses a wider range of microarchitectural attacks.

Thus, we propose ***leakage containment models (LCMs)***—novel *axiomatic* hardware-software contracts designed to support automatically reasoning about microarchitectural leakage in programs when they run on particular hardware implementations. LCMs are designed to model *microarchitectural information flow*, a key contributor to microarchitectural leakage. Specifically, for every *architecture-level execution* of an instruction, there may be more than one corresponding *microarchitecture-level execution* that achieves the same software-visible effect.[1] Furthermore, *which* microarchitecture-level execution is realized when a program runs on a hardware implementation generally depends on the outcome(s) of dynamic microarchitectural information flow(s). If an attacker can distinguish one microarchitecture-level execution from another, it may infer some function of the data involved in these information flows. As an example, consider a program running on a processor with core-private L1 caches. Either an L1 cache *miss* or *hit* (i.e, one of two microarchitecture-level execution possibilities) will occur on behalf of an architecture-level load in the program. Moreover, whether a miss or hit occurs depends on the outcome of the load *microarchitecturally reading* the cache state that was *microarchitecturally written* by the last access to the same cache line (i.e., the outcome of a microarchitectural information flow). It is well known that a software-based attacker can distinguish these two microarchitecture-level execution scenarios by timing the load's execution latency [23], thereby leaking a function of the address bits involved in the culprit information flow.

Given the ingredients for hardware leakage above, **our second insight** is that LCMs can define and directly compare an *architectural semantics* and a *microarchitectural semantics* for a program to pinpoint potential hardware-induced program leaks. A program's architectural semantics encodes the software-visible ways in which it can execute; each execution possibility differs according to the architectural information flows it exhibits. A program's microarchitectural semantics encodes its distinct microarchitectural execution possibilities which differ according to their microarchitectural information flows. To define *microarchitectural leakage* based on LCMs, we first identify which microarchitectural execution of a program is *implied* by each architectural execution possibility in the absence of interference. Then, a program is examined to determine if its microarchitectural semantics can *ever* deviate from what is architecturally implied. If so, the program is susceptible to hardware leakage. LCMs also leverage a program's *speculative semantics* to reason about transient leakage [27].

[1] In this paper, *software-visibility* is used in the traditional sense to refer to observable program behavior in the absence of hardware side/covert channels.

In designing LCMs, we leverage **our third insight**—that *memory consistency models* (MCMs) [4, 39] define the same sort of architectural program semantics that LCMs require. To summarize, MCMs articulate which architecture-level information flows between shared memory operations in a parallel program are legal; distinct flows constitute distinct architecture-level (i.e., software-visible) program executions. Since well-established and formally specified MCMs already provide a key building block of LCMs—an architectural semantics for programs—we elect to derive LCMs from MCMs. A key benefit of this design choice is that security analyses built on LCMs can leverage a rich literature in MCM analysis and verification [1–4, 14, 44, 46, 75]. Moreover, recent work proposes an automated approach for synthesizing axiomatic MCM specifications directly from RTL with minimal user intervention [30]. Thus, we believe axiomatic security contracts are more amenable to automated verification approaches than their operational peers.

Overall, this paper makes the following contributions:

- **Axiomatic security contracts:** We propose LCMs—novel security contracts—and an axiomatic vocabulary for defining them, derived from axiomatic MCMs. Our formal LCM vocabulary supports advanced processor features like out-of-order and multi-core execution and captures both transient and non-transient leakage.
- **Leakage formalization:** Using our axiomatic LCM vocabulary, we formalize *microarchitectural leakage*—focusing on leakage through hardware memory systems—so that it can be automatically detected in programs.
- **Transmitter taxonomy:** We present a new taxonomy for classifying transmitters according to their severity, focusing on transmitters which facilitate cache-based leakage.
- **Leakage detection:** First, we demonstrate that our leakage definition captures a sampling of (transient and non-transient) microarchitectural attacks from the literature [15, 29, 35, 37, 69, 79]. Second, we develop a static analysis tool, Clou, based on LCMs which automatically *identifies* and *repairs* (via fence insertion) Spectre v1 [37], Spectre v1.1 [35], and Spectre v4 [31] vulnerabilities in programs. Clou offers better scalability than all state-of-the-art static analysis tools for detecting these vulnerabilities and finds new Spectre leakage in the `libsodium` [21] and `OpenSSL` [56] crypto-libraries.

2 BACKGROUND AND MOTIVATION

LCMs define both an *architectural semantics* and a *microarchitectural semantics* for programs. A program's architectural semantics encodes the distinct software-visible ways in which it can execute; such a semantics is ISA-specific. A program's microarchitectural semantics encodes the distinct ways in which it can *microarchitecturally execute*; such a semantics is implementation-specific. Thus, LCMs are defined *per-microarchitecture*.

A key insight of this paper is that axiomatic MCM specifications [4] lay the foundation for axiomatic LCMs. First, axiomatic MCMs define an architectural semantics for programs which LCMs

use out-of-the-box. Second, axiomatic MCM primitives can be directly adapted to encode a microarchitectural semantics for programs. We provide an overview of axiomatic MCMs in this section and refer the reader to existing literature for more details [1, 4].

2.1 Axiomatic Memory Consistency Models

MCMs define the value(s) that can be legally returned by shared-memory loads in a parallel program. MCMs are central to reasoning about parallel program correctness; thus a large body of work is devoted to formalizing them [4, 9, 10, 13, 45, 52–54, 58, 60, 63, 72, 74].

An *axiomatically* formalized MCM is encoded as a *consistency predicate* which may be applied to *candidate executions* of programs. In a nutshell, candidate executions (2.1.2) are the result of taking a program and instantiating for it a particular *control flow path* and a particular set of *data-flow* relationships between shared memory instructions. Candidate executions which are consistent (resp. inconsistent) with an MCM's consistency predicate (2.1.3) are allowed (resp. disallowed) by the MCM.

2.1.1 Event Structures. A candidate execution of a program is derived from an *event structure* [1], which describes a particular *control-flow path* of the program where all branches have been resolved. For example, a program with a single conditional branch will produce two event structures: one which corresponds to the *taken* control-flow path and another which corresponds to the *not-taken* path. Precisely, an event structure $E \triangleq$ (`MemoryEvent`, `Location`, `address`, po) is defined by the following:

- `MemoryEvent`: a set of all program instructions in a particular control-flow path of the program that read or write memory. `MemoryEvent` is a subset of `Event`—a set of all program instructions in a particular control-flow path of the program.
- `Location`: a set of all architectural memory locations which are accessed by program Read/`Write` events—Read and `Write` are disjoint subsets of `MemoryEvent`.
- `address`: a binary relation that maps each `MemoryEvent` to the single shared memory `Location` it accesses; `address` is a set of (`MemoryEvent`, `Location`) tuples.
- po: a transitive binary relation that maps each `Event` to all *committed* `Events` that follow it in *program order*; po is a set of (`Event, Event`) tuples which constructs a per-thread total order on committed instructions.

2.1.2 Candidate Executions. An event structure for a program can be extended to a set of candidate executions. Each candidate execution in the set differs with respect to the shared memory interactions that are realized between instructions. Concretely, we can extend an event structure with an *execution witness* $X \triangleq$ (`rf`, co, `fr`) to produce a candidate execution. An execution witness is defined by the following relations involving *same-address* `MemoryEvents`:

- `rf` (*reads-from*): a binary relation that maps each `Write` to all same-address Reads that read from it; `rf` is a set of (`Write`, Read) tuples.
- co (*coherence-order)*: a transitive binary relation that maps each `Write` to all same-address `Writes` that follow it in coherence order; co is a set of (`Write, Write`) tuples, which constructs a per-`Location` total order on all `Writes` to said `Location`.
- `fr` (*from-reads*): a binary relation that maps each Read to all co-successors of the `Write` that it read from; `fr` is a set of (`Read, Write`) tuples, and it is derived from `rf` and co as follows: `fr=~rf.co`, where ∼ is relational transpose and . is relational join.

Collectively, `rf`, co, and `fr` comprise the com (*communication*) relation—com = `rf` + co + `fr`, where + is set union.

2.1.3 Consistency Predicates. A candidate execution is uniquely defined by an event structure E and an execution witness X. An MCM is then defined by a *consistency predicate* which renders candidate executions consistent or inconsistent with respect to it. Consistent candidate executions of a program (i.e., those for which the consistency predicate evaluates to `True`) are allowed according to the MCM, while inconsistent candidate executions are disallowed. In constructing a consistency predicate, axiomatic MCM specifications often consider a wider range of events and relations, such as:

- ppo: a binary relation that maps an `Event` to a po-later `Event` if the ISA guarantees they will be executed in order *from the perspective of all cores* in the shared memory system; ppo is a set of (`Event, Event`) tuples and a subset of po.
- `fence`: a binary relation that maps an `Event` e_0 to another `Event` e_1 if e_0 is ordered before e_1 by an explicit *synchronization event* (e.g., a fence/barrier); `fence` is a set of (`Event, Event`) tuples.

For MCMs which do not order Reads with po-later `MemoryEvents` by default (i.e., via ppo), a dep (*dependency*) relation is used to selectively enforce these orders. dep encodes syntactic dependencies between shared memory instructions *through registers*, and is comprised of the following three sub-relations:

- `addr` (*address dependency*): a binary relation that maps a Read to po-subsequent `MemoryEvent` when the `Location` accessed by the `MemoryEvent` depends syntactically on the value returned by the Read; `addr` is a set of (`Read, MemoryEvent`) tuples.
- `data` (*data dependency*): a binary relation that maps a Read to a po-subsequent `Write` when the written value depends syntactically on the value read; `data` is a set of (`Read, Write`) tuples.
- `ctrl` (*control dependency*): a binary relation that maps a Read to a po-subsequent `MemoryEvent` when the control flow decision of whether to execute the `MemoryEvent` depends syntactically on the value read; `ctrl` is a set of (`Read`, `MemoryEvent`) tuples.

An example consistency predicate is that which defines the Total Store Order (TSO) MCM used by Intel x86 processors [32]. It is composed of the conjunction of three auxiliary predicates—*sc_per_loc*, *rmw_atomicity*, and *causality* [4]. Below we define the most relevant two for the ideas presented in this paper:

- *sc_per_loc* ≜ *acyclic*(`{rf + co + fr + po_loc}`), where po_loc is the subset of po that relates same-address `MemoryEvents`.
- *causality* ≜ *acyclic*(`{rfe + co + fr + ppo + fence}`). For x86-TSO, ppo includes all (`Write, Write`) and (`Read, MemoryEvent`) tuples in po, and `rfe` (*reads-from external*) is the subset of `rf` that relates `Events` on different threads.

2.2 An Architectural Semantics for LCMs

Recall that LCMs define an ISA-specific *architectural semantics*, which encodes the various software-visible ways in which programs can execute; each execution possibility differs according to the architectural information flows it exhibits. Now consider an ISA MCM, defined axiomatically with the help of a consistency predicate. Notably, the com relation (§2.1.2) encodes architectural information flows through shared memory for a specific candidate execution. Thus, for a given program, its set of *consistent candidate executions*—i.e., those candidate executions which are consistent with the consistency predicate—constitute its architectural semantics as required by LCMs.

More precisely, consistent candidate executions comprise a program's architectural semantics *restricted to memory instructions*. In this paper, we use LCMs to model leakage on behalf of hardware memory systems optimizations, particularly cache optimizations. Hence, this restriction is appropriate.

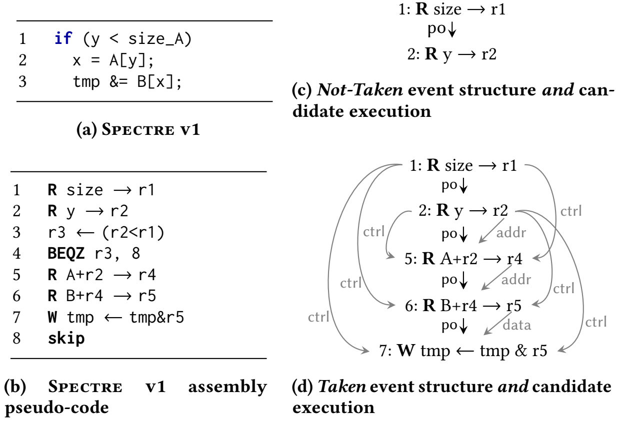
3 LEAKAGE CONTAINMENT MODELS

3.1 What Memory Models are Missing

Recent work identifies similarities between MCMs and the sorts of security contracts that software and hardware designers would benefit from [19, 22, 66]. However, MCMs themselves do not offer a complete security contract solution. To demonstrate why, consider the classic Spectre v1 [37] program in Fig. 1a and its corresponding assembly pseudo-code in Fig. 1b. Due to the branch, axiomatic MCM definitions would consider two distinct event structures for this program—one which corresponds to the *not-taken* branch outcome (Fig. 1c) and the other which corresponds to the *taken* outcome (Fig. 1d). Note that axiomatic MCM definitions facilitate modeling both event structures and candidate executions as *directed graphs*. Nodes are MemoryEvents labeled with the Location they access, as encoded in the address relation; edges denote types of "happens-before" [38] (i.e., sequencing) relationships, as encoded in relations like po, com, and dep.

Each event structure in Fig. 1 can be extended to *exactly one* candidate execution. Thus, there are two possible candidate executions for Spectre v1. This is because every memory access in the Spectre v1 program touches a *distinct* memory location; thus, only one instantiation of the com relation is possible for each event structure. Specifically, all Read events read from the *initial state* of memory—by convention, no rf edges are explicitly drawn since initialization writes are not explicitly modeled. Further, the sole Write event is coherence-ordered after the last *initialization write* to the same memory location—by convention, no co edges are drawn. Without rf and co edges, there are no fr edges (§2.1.2). Figs. 1c and 1d thus *also* constitute candidate executions. Moreover, they constitute *consistent candidate executions* according to TSO (§2.1.3) making them valid architectural execution possibilities on Intel processors. Fig. 1d uses gray edges to depict instances of the dep relation, although it is not a distinguishing feature of event structures or candidate executions.

As is known, the program in Fig. 1a exhibits a variety of hardware-induced leakage when run on modern processors. First, the *addresses* accessed by instructions 1 and 2 in Fig. 1c and instructions



(a) Spectre v1

(b) Spectre v1 assembly pseudo-code

(c) *Not-Taken* event structure *and* candidate execution

(d) *Taken* event structure *and* candidate execution

Figure 1: Spectre v1 in (a), produces two event structures (§2.1.1)—(c) and (d). Each event structure can be completed with a single execution witness (§2.1.2). The resulting two candidate executions look identical to the event structures, since no explicit com edges are instantiated (§3).

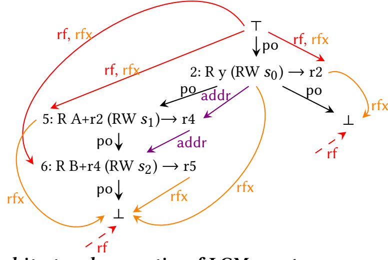
1, 2, 5, 6, and 7 in Fig. 1d may be leaked to an attacker via a simple cache side-channel attack. Second, the *data* returned by read instructions 2 and 5 in Fig. 1d can be leaked. This is because the addr dependency from instruction 2 (resp. 5) to 5 (resp. 6) indicates that the data returned by 2 (resp. 5) is supplied as the address operand of 5 (resp. 6), an instruction which we established can leak its address operand. Third, the outcome of the branch can be leaked. Moreover, the program in Fig. 1a exhibits speculative leakage. None of this leakage potential can be discerned directly from Figs. 1c and 1d. In summary, MCMs cannot directly capture microarchitectural leakage out-of-the-box.

3.2 A Microarchitectural Semantics for LCMs

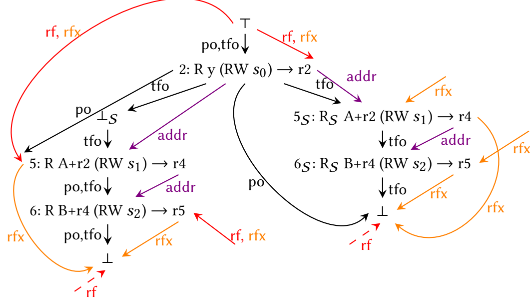
LCMs facilitate reasoning about hardware-induced leakage in programs by augmenting the architectural semantics provided by axiomatic MCMs with a *microarchitectural semantics* that describes the various ways in which a program can microarchitecturally execute. Each execution possibility differs according to microarchitectural information flows it exhibits.

In defining a microarchitectural semantics for LCMs we leverage *two key building blocks*, featured in Fig. 2a. Fig. 2a effectively merges together the two Spectre v1 candidate executions (Figs. 1c and 1d) into a single graph and adds some new nodes and edges. Some instructions are also omitted for clarity, but numeric instruction labels are retained.

In Fig. 2a, $\top$ represents *explicitly* the set of architectural/microarchitectural writes that initialize relevant architectural/microarchitectural state. $\perp$ represents a set of ***observer*** accesses that observe aspects of final architectural/microarchitectural state after the program runs to completion. In this paper, we assume that the observer ($\perp$) does not share memory with the executing program, and thus it *cannot* interact with the program architecturally (i.e. via com). $\perp$ may only be involved in a com relation with $\top$. However, $\perp$ *can* interact with the program microarchitecturally, such as by probing



(a) The *microarchitectural semantics* of LCMs captures communication between instructions via xstate (s_0, s_1, and s_2).



(b) The *speculative semantics* of LCMs demonstrates that leakage can involve speculatively-executed source instructions, denoted with subscript S.

Figure 2: LCMs extend MCMs with a *microarchitectural semantics,* as in (a)—to modeling microarchitectural leakage—and a *speculative semantics,* as in (b), to model transient leakage.

cache sate. Thus, ⊤ can be involved in `comx` (§3.2.2) relations with program instructions. Each straight-line path through `po` edges from ⊤ to ⊥, together with the `com` relation (restricted to those instructions involved in the `po` path in question), denotes a distinct candidate execution. In Fig. 2a, the `com` relation has been explicitly drawn—i.e., note the presence of `rf` edges involving ⊤ in contrast to Figs. 1c and 1d which do not model initialization writes. Edges missing a source node are implicitly related to ⊤.

3.2.1 Modeling Microarchitectural State. The microarchitectural semantics defined by LCMs explicitly considers *microarchitectural state*, effectively denoting *which* state elements in a processor are accessed on behalf of architectural program instructions and *how* they are accessed. We refer to said state as *extra-architectural state* (or `xstate`), meaning that it can consist of any *non-architectural* state in a microarchitecture.[2] Fig. 2a illustrates that `xstate` elements s_0, s_1, and s_2 are accessed on behalf of `Read` instructions 2, 5, and 6, respectively. Furthermore, all three `xstate` accesses are *microarchitectural* read-modify-write operations, denoted by "RW" before the `xstate` identifier in the figure. In other words, *R y (RW* s_0*)* $\longrightarrow$ *r2* means that architectural `Read` event *R*, which accesses architectural `Location` *y*, *induces* a microarchitectural read-modify-write of `xstate` element s_0.

[2]The term *extra-architectural state* was coined in prior work [43]; however, we assign it a different meaning in this paper.

The `xstate` identifiers used by LCMs, such as those featured in Fig. 2a, may represent a *set* of hardware state elements in a microarchitecture. Furthermore, an instruction can access a *vector* of `xstate` rather than a single `xstate` element. The key idea is that instructions which access common `xstate` elements are capable of *communicating* microarchitecturally—modeled by a new communication relation, `comx` (§3.2.2). In fact, the sole reason for modeling `xstate` is to establish `comx` for a given candidate execution. Instructions may also access different `xstate` elements in different ways depending on execution context, as described below.

In this paper, we seek to model hardware leakage due to memory systems optimizations, particularly cache optimizations. Thus, we consider `xstate` accessed on behalf of architectural *memory instructions* only. In particular, the `xstate` elements we model in this paper are intended to capture the ways in which *same-core memory instructions* can communicate *microarchitecturally*—said `xstate` then effectively represents the core-private cache lines and load-store queue (LSQ) entries that are accessed on behalf of architectural memory instructions.

To understand what the above `xstate` modeling choice means for *how* memory instructions access these abstract `xstate` elements, consider the following. In general, (cacheable) architectural read instructions either microarchitecturally read a cache line (a cache hit) or microarchitecturally read-modify-write a cache line (a cache miss). With respect to a local LSQ, architectural reads *may* microarchitecturally read (i.e., forward) data from a pending store. Similarly, (cacheable) architectural writes always behave as cache line read-modify-writes, unless they are executing on a microarchitecture with a no-write-allocate cache policy. With respect to LSQ state, architectural writes always behave as microarchitectural writes. Given `xstate` elements which collectively represent the core-private cache line and LSQ entries accessed on behalf of an architectural memory instruction: *read hits* read `xstate` (from the cache or from a pending store in the LSQ), *read misses* read-modify-write `xstate` (namely a cache line), and *writes* read-modify-write `xstate` (namely a cache line which subsumes the LSQ write).

3.2.2 Modeling Microarchitectural Information Flow. Fig. 2a shows that LCMs define a `comx` relation which lifts `com` [4] to `xstate` accesses; `com` relates same-`address` operations, while `comx` relates same-`xstate` operations. Recall that LCMs use the `com` relation to encode the architectural information flows that distinguish program executions according to their architectural semantics. Likewise, LCMs use the `comx` relation to encode microarchitectural information flows that distinguish program executions according to their microarchitectural semantics. Just as a consistency predicate was used to rule out illegal instantiations of `com`, a similar *confidentiality predicate* must be defined to rule out illegal instantiations of `comx` according to a specific hardware implementation. §4.2 discusses features of confidentiality predicates that are required to capture different sorts of known hardware optimizations.

3.2.3 Modeling Microarchitectural Leakage. LCMs formalize *microarchitectural leakage* by (1) determining which microarchitectural semantics (`comx` edges) are implied by a given architectural semantics (`com` edges) when microarchitectural *non-interference* [24] holds, and (2) detecting when a program's microarchitectural semantics deviates from architectural expectation.

As an example, consider an `rf` edge which relates a write to a *same-core* read that it sources—called `rf`-internal or `rfi` [4]. Consider also our `xstate` of interest which corresponds to core-private processor cache lines or LSQ entries. In the absence of interference, an `rfi` edge which relates some `Write` w to some `Read` r, implies a consistent `rfx` edge—an `rfx` edge which relates w to r. In other words, if microarchitectural non-interference holds, a read r which *architecturally* reads from a same-core write w will further *microarchitecturally* read from the core-private cache line or LSQ entry populated by w. If r reads from a cache line populated by a different instruction, this means that the cache line was evicted by an interfering access in between w's and r's cache accesses. If r forwards data from an interfering store residing in the LSQ or the most recent write to memory rather than reading from w, then it exhibits memory address mis-speculation (§3.3) which will be eventually rolled back.

Fig. 2a contains two instances of the program's microarchitectural semantics deviating from what is architecturally implied—the two dashed `rf` edges are lacking consistent `rfx` edges. The endpoints of these culprit `com` edges—⊥ for both—constitute ***receivers*** of microarchitectural leakage.

3.2.4 A Taxonomy for Cache `xstate` *Transmitters.* In this paper, we define three classes of ***transmitters*** that can microarchitecturally convey information to a receiver, described as follows and summarized in Table 1.

First, `xstate` *transmitters* are instructions which source (i.e., convey information to) a receiver via an `rfx` edge. In other words, `xstate` transmitters communicate some function of their accessed `xstate` to a receiver via microarchitectural information flows. In this paper, where `xstate` consists of core-private cache lines or LSQ state which facilitate microarchitectural communication between same-address memory accesses, `xstate` transmitters are in reality ***address transmitters***. Address transmitters transmit a function of their address operand.

Second, ***data transmitters*** (resp. ***control-flow transmitters***) are address transmitters which are the target of an `addr` (resp. `ctrl`) dependency, originating at a `Read` r. Both data transmitters and control transmitters leak a function of the data returned by r, where r is referred to as the ***access instruction***. However, we consider data transmitters more dangerous since control transmitters leak the outcome of a branch condition involving r's return value rather than the return value itself.

Third, ***universal data transmitters*** (resp. ***universal control-flow transmitters***) are data transmitters (resp. control-flow transmitters) whose access instruction is the target of an `addr` dependency, originating at some `Read` r', called the ***index instruction***. For example, the relevant transmitter patterns in Table 1 indicate that the *memory location* accessed by *access* depends on the data returned by *index*. If an adversary can control the contents of the memory location referenced by *index*, it can possibly leak arbitrary memory [47]. In Fig. 2a, instructions 2, 5, and 6 are address transmitters, 5 and 6 are data transmitters, and 6 is a *candidate* universal data transmitter. The execution semantics of the program in Fig. 2a, which features a bounds check on index y restricts the leakage scope of instruction 6. However, §4.2 explains how transient execution of instruction 6 unlocks true universal data leakage.

Transmitter Type	Leakage Pattern
address	$\textit{transmit} \xrightarrow{\text{rfx}} \textit{receiver}$
data	$\textit{access} \xrightarrow{\text{addr}} \textit{transmit} \xrightarrow{\text{rfx}} \textit{receiver}$
control	$\textit{access} \xrightarrow{\text{ctrl}} \textit{transmit} \xrightarrow{\text{rfx}} \textit{receiver}$
universal data	$\textit{index} \xrightarrow{\text{addr}} \textit{access} \xrightarrow{\text{addr}} \textit{transmit} \xrightarrow{\text{rfx}} \textit{receiver}$
universal control	$\textit{index} \xrightarrow{\text{addr}} \textit{access} \xrightarrow{\text{ctrl}} \textit{transmit} \xrightarrow{\text{rfx}} \textit{receiver}$

Table 1: Transmitter taxonomy for `xstate` considered in this paper (§3.2.1). The partial order on transmitter severity is $\text{AT} < \text{CT} < \{\text{DT}, \text{UCT}\} < \text{UDT}$.

3.3 A Speculative Semantics for LCMs

It is crucial that LCMs account for all instructions capable of accessing `xstate`—and thus all instructions capable of impacting the `comx` relation—including those that *transiently* execute. Thus, LCMs extend MCMs with a *speculative semantics* [15, 26, 55], as illustrated in Fig. 2b.

The speculative semantics of LCMs leverages a new *transient fetch order* (`tfo`) relation to construct a per-thread total order on all instructions that are *fetched* from instruction memory. `po` is a subset of `tfo`, and *instructions ordered by* `tfo` *but not* `po` *are considered transient*; `po` relates committed instructions only. Transient instructions can interact with other transient or committed instructions via `xstate` and ultimately construct new opportunities for program-level information leakage by impacting a program's microarchitectural semantics. We consider two types of hardware speculation in this paper—***control-flow speculation*** and ***address speculation***.

To model control-flow speculation, at each control-flow instruction where the architectural semantics considers both possible committed branch paths (i.e., both possible event structures), the speculative semantics additionally considers a window of speculative instructions along each branch path according to a user-defined speculation depth. In this way, formal analyses that leverage LCMs consider the worst-case attacker who can poison the prediction of any branch [26, 27]. Fig. 2b demonstrates this idea with a speculation depth of two. The left (i.e., taken) branch speculatively jumps to the end of the program ($\perp_S$) before rolling back speculation and executing the body of the branch. The right (i.e., not-taken) branch speculatively executes the body of the branch (5_S and 6_S) before rolling back speculation and jumping to the end of the program.

To model address speculation, we consider two types—***store forwarding*** and ***alias prediction***. Both enable an architectural `Read` instruction to induce a window of speculation. Furthermore, they *relax* the placement of `rfx` edges, and thus the derived `frx` edges, in legal candidate executions. Figs. 4a and 4b in §4 give examples of data leakage which results from store forwarding and alias prediction, respectively.

Store forwarding permits a read to forward data from the most-recent older store to the same address among stores whose addresses have been resolved. However, *all* older stores need not resolve their addresses before forwarding can occur. Thus, while a load will always read from the *correct address*, it may be forwarded *stale data* speculatively. Alias prediction permits a load to forward

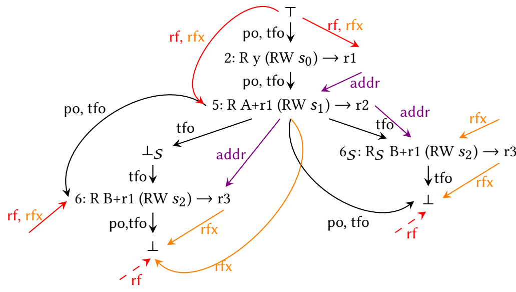


Figure 3: Spectre v1 variant [27, 79]. LCMs detect a transient transmitter and non-transient access.

from a store with a potentially *mismatching address*—i.e., a load may forward data from a store even if its own address has not resolved.

3.4 The subrosa Toolkit

We mechanize the LCM vocabulary in a toolkit built in Alloy [33], called subrosa,[3] which supports the design and formal analysis of custom LCM specifications using our axiomatic vocabulary. We plan to leverage subrosa as a starting point for automatically comparing LCMs across microarchitectures and evaluating the effectiveness of microarchitectural attack mitigations in future work.

4 DETECTING LEAKAGE IN REAL-WORLD EXAMPLES

In this section, we use §3's axiomatic LCM vocabulary to formalize *microarchitectural leakage*. Our leakage definition is extensible, but focuses on formalizing hardware leakage on behalf of processor memory systems optimizations.

4.1 Formalizing Microarchitectural Leakage

To construct our leakage definition, we define three *non-interference predicates* (based on `rf`, `co`, and `fr`, respectively) which can be used to detect violations of microarchitectural non-interference in programs. We say that given state machine M, and subjects S and S', S is *microarchitecturally non-interfering* with S' if the actions of S do not affect the microarchitectural observations (via `comx`) of S' [83]. Our non-interference predicates are based on mappings from the building blocks of an LCMs's architectural semantics—`rf`, `co`, and `fr`—to the building blocks of its microarchitectural semantics—`rfx`, `cox`, and `frx`. Our mappings assume that architectural memory events access a single `xstate` location which collectively represents a core-local cache line and LSQ entry. While we could model these state elements as a pair of `xstate` locations, we choose to merge them since they both facilitate microarchitectural information flow between architectural memory instructions. We also limit our mappings below to a *single-core* execution setting where caches are *direct-mapped* (see §5.2) and feature a *write-allocate* cache policy.

If two writes are ordered by `co`, $w_0 \xrightarrow{\texttt{co}} w_1$, they should be similarly ordered by `cox` and `frx`. This is because two same-address writes behave as microarchitectural read-modify-writes (§3.2.1) with respect to the same `xstate`. Their writes to the LSQ (which initially populate queue entries[4]) and cache will be ordered, and w_0's cache read will precede w_1's cache write. If w_0 immediately precedes w_1 in `co`, the two writes should also be ordered by `rfx` in the absence of interference (**`co`-non-interference**)—i.e., w_1's cache line read should be microarchitecturally-sourced by w_0's cache line write, a cache hit for w_1. As explained in §3.2.3, if a write and read are ordered by `rf`, $w \xrightarrow{\texttt{rf}} r$, they should be similarly ordered by `rfx` in the absence of interference (**`rf`-non-interference**). If a read and a write are ordered by `fr`, $r \xrightarrow{\texttt{fr}} w$, then they should be similarly ordered by `frx`—r will microarchitecturally read its cache line or LSQ entry before w microarchitecturally writes. If r writes `xstate` (a cache miss), then r will precede w in `cox`. Furthermore, consider the write w' that sourced r, $w' \xrightarrow{\texttt{rf}} r$. If w immediately follows w' in `co` and r writes `xstate`, r should microarchitecturally source w via `rfx` in the absence of interference (**`fr`-non-interference**), a cache hit for w.

A microarchitectural leak is detected when a consistent candidate execution violates a non-interference predicate.

4.2 LCMs by Example

We show that the LCM vocabulary faithfully detects leakage in a sampling of (transient and non-transient) microarchitectural attacks form the literature. In all examples, dashed edges in figures denote `com` relations with `comx` inconsistencies—they "point to" (via a directed edge) receivers, which are used to identify transmitters according to the rules in §4.1. All numerical instruction identifiers in our explanations refer to candidate execution graphs. Note that some edges are omitted in figures for clarity when they are not central to the exemplified leakage.

Spectre v1: Fig. 2b summarizes the candidate executions of vanilla Spectre v1 [37] (Fig. 1a). The program features a *speculation primitive* [48]—a conditional branch which induces a window of speculation in each event structure instantiated by the branch (i.e, each fork of the graph). Dashed `rf` edges point towards both receivers—both ⊥ nodes. Instruction 2 is an address transmitter; 5 and 5_S are data transmitters with access instruction 2; 6 and 6_S are candidate universal data transmitters with access instructions 5 and 5_S, respectively. Notably, some transmitters are transient (denoted with the subscript S) while others are non-transient. As mentioned in §3.2.4, the bounds check on index y restricts the leakage scope of 6. However, 6_S (a transient transmitter) is a *true* universal data transmitter which can be steered by an attacker to access arbitrary memory.

Fig. 3 shows another variant of Spectre v1 [27, 79] (code below) featuring the same speculation primitive—a conditional branch—and the same candidate universal data transmitters—6 and 6_S.

```
1   x = A[y];
2   if(y < size_A)
3     temp &= B[x];
```

However, this time the access instruction corresponding to both instructions 6 and 6_S, namely instruction 5, is non-transient. Alternately, in Fig 2b, the access instruction corresponding to instruction

[3] https://github.com/ctrippel/subrosa

[4] Note that although writes may create entries in the LSQ in co-order, the reality that writes produce data/addresses in execution order is modeled by relaxing rfx (§3.3).

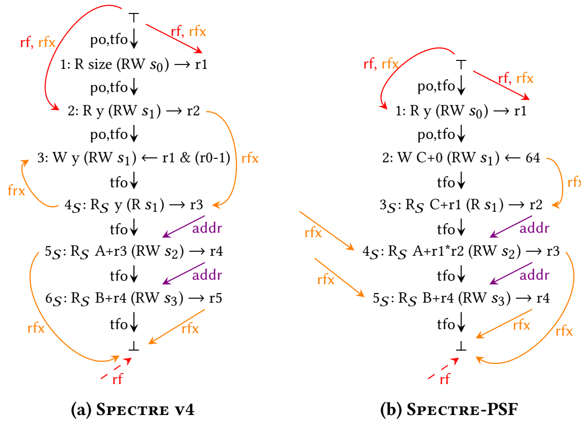


Figure 4: Spectre v4 [29, 31] and Spectre-PSF [15, 25]. LCMs detect a transient transmitter and transient access.

6 is non-transient (instruction 5) while the access instruction corresponding to instruction 6_S is transient (instruction 5_S). As in vanilla Spectre v1, the leakage scope of 6 is restricted by the bounds check on y. While the leakage scope of 6_S is significantly increased by comparison, it is still somewhat limited due to the fact that the access instruction (instruction 5) commits. Notably, STT [79] declared preventing the leakage of non-transiently accessed data as out of scope, although other related work captures leakage of this sort [15, 26].

Spectre v4: Fig. 4a is representative of Spectre v4, described by the code below.

```
1   y = y & (size_A - 1);
2   x = A[y];
3   temp &= B[x];
```

The speculation primitive is *store forwarding* (§3.3)—instruction 4_S reads from a stale same-address write. The frx relation between instructions 4_S and 3 illustrates this behavior; frx edges can also be understood as *reads-before*. In other words, 4_S reads from xstate element s_1 *before* s_1 is overwritten by 3. The figure also illustrates that instruction 4_S is microarchitecturally sourced from the first read of y, namely instruction 2, via an rfx relation. Ultimately, this behavior leads a true universal data transmitter (6_S) with a transient access instruction (5_S). Also, 5_S is a transient data transmitter, with transient access instruction 4_S.

We note that Spectre v4 exhibits particularly interesting microarchitectural behavior that is relevant for developing LCMs for Intel x86 microarchitectures (given that Spectre v4 has been observed on Intel processors [29, 31]). In particular, formally specifying an LCM for a particular ISA requires defining a *confidentiality predicate* (§3.2.2). Consider the consistency predicate for TSO from §2.1.3 which is the conjunction of the *sc_per_loc*, *rmw_atomicity*, and *causality* auxiliary predicates. Naively lifting *sc_per_loc* to constrain comx results in *sc_per_loc_x* = acyclic({rfx + cox + frx + tfo_loc}), where tfo_loc is defined as po_loc by substituting tfo for po. This straightforward predicate derivation would rule out the execution in Fig. 4a, which is in fact *possible* on x86 microarchitectures. An LCM, for Intel x86 processors (which permits Spectre v4) must clearly permit frx + tfo_loc cycles in its confidentiality predicate.

Spectre-PSF: Fig. 4b features a variant of Spectre v4 [15, 25], coined Spectre-PSF [19] (code listing below).

```
1   uint8_t A [16];
2   uint8_t C[2] = {0, 0};
3   if (y < size_C)
4     C [0] = 64;
5     temp &= B[A[C[y] * y]];
```

The speculation primitive is *alias prediction*. In particular, instruction 3_S reads from an incorrect memory location—illustrated by the rfx edge between instructions 2 and 3_S. This behavior leads to a true universal data transmitter (5_S) with a transient access instruction (4_S).

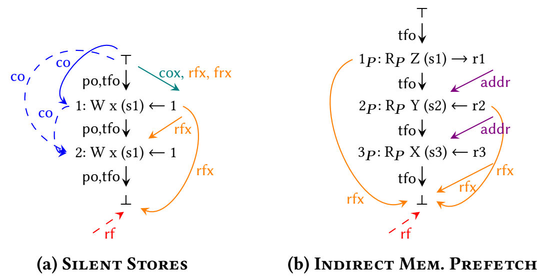
Spectre-PSF also features interesting execution behavior that can influence the placement of rfx edges in LCM candidate executions. Namely, read instructions can mis-predict the xstate they access such that they can microarchitecturally read data written by prior stores to different addresses.

Non-Spectre Attacks: Recent work shows that various microarchitectural optimizations can be leveraged to leak program data in a manner as severe as Spectre attacks [69]. Fig. 5 features programs that exercise two such optimizations.

Fig. 5a features leakage on hardware that implements *silent stores* [41], which avoid explicitly writing to memory when a store's data operand matches the current contents at its effective address. In a processor that implements silent stores, a store may behave microarchitecturally as a read (store is "silent") or a read-modify-write (store is not "silent")—more execution options than were possible on the microarchitecture of §4.1. In a processor that implements silent stores, two writes ordered by co, $w_0 \xrightarrow{\text{co}} w_1$, must be similarly ordered by cox in the absence of microarchitectural interference. If w_0 and w_1 are not related by cox, there was either an interfering write in between w_0's and w_1's cache accesses that wrote the same data as w_1, or w_0 and w_1 themselves wrote the same data. Here, instruction 2 is an xstate transmitter (due to a co/cox inconsistency). Unlike most xstate transmitters in this paper, instruction 2 transmits the data field of its accessed xstate s_0 rather than the address field. This is because the silent store optimization triggers based on the result of a *data comparison* while a cache hit/miss is triggered based on the result of an *address comparison*.

Fig. 5b features leakage on hardware implementing an *indirect memory prefetcher* (IMP) [80] (patented by Intel [81]). Such a prefetcher tries to detect programs of the form *for(i = 0...N) X[Y[Z[i]]]* and prefetch the cache line corresponding to *&X[Y[Z[i + Δ]]]*. The security implications of this optimization are discussed in recent work [69]. Notably, the authors point out that an IMP can construct a *universal read gadget* [47], and Fig. 5b indeed indicates that prefetch instruction 3_P is a universal data transmitter.

Note that Fig. 5b contains non-architectural prefetch instructions—similar in vein to transient instructions, they are not involved in architectural relations like com/po. To detect IMP-related leakage in programs, an enhanced version of LCMs could extend user-level programs with prefetch operations based on the presence of *prefetch primitives*—instruction sequences which can initiate hardware prefetches.



(a) SILENT STORES

(b) INDIRECT MEM. PREFETCH

Figure 5: NON-SPECTRE [69]. In (a), LCMs detect a non-transient transmitter of a non-transiently accessed `xstate`. In (b), IMPs can construct a universal data transmitter of prefetched data.

5 CLOU: DETECTING LEAKAGE WITH LCMS

We develop a static analysis tool, CLOU,[5] based on LCMs which automatically identifies and repairs Spectre vulnerabilities in programs. Our approach is inspired by a tool which restores sequential consistency via automated fence insertion for programs running on hardware implementing weak MCMs [1]. CLOU is implemented as a custom IR pass in LLVM [40]. It takes a ***C source file as input***, compiles it to LLVM IR using CLANG v12.0.0 `-O0`, and analyzes each defined public function one-by-one. Eventually, CLOU ***outputs a list of transmitters and a set of consistent candidate executions*** (in graph form) which give witness to detected software vulnerabilities. CLOU can also ***automatically insert mitigations*** (e.g., fences, like Intel's `lfence`) to repair vulnerable programs. Fig. 6 summarizes CLOU's architecture, which we detail in this section. CLOU is most optimized for detecting universal data and universal control transmitters, but it can identify the other transmitter variants as well (§3.2.4).

5.1 Constructing an Abstract CFG (A-CFG)

CLOU first transforms a function's LLVM IR control-flow graph (CFG) into a loop- and call-free *Abstract CFG (A-CFG)*—our name for a CFG that has undergone loop/function summarization and function inlining.

Loop Summarization: To eliminate a loop from a function's CFG, CLOU *summarizes* all of the ways in which it could be involved in hardware-induced leakage using a finite (and minimal) number of instructions as follows. First, recall that LCMs detect microarchitectural leakage by comparing architecture-level (via `com`) and microarchitecture-level (via `comx`) instruction interactions. This suggests a loop summarization approach which accounts for (1) how instructions in any loop instance can interact with instructions outside of the loop, and (2) how instructions in two arbitrary loop instances can interact with each other. Second, consider a memory alias analysis procedure (§5.2) that can summarize for all memory accesses in the loop the set of virtual memory locations they may access across all iterations. We conclude that with memory alias analysis, all relevant `com`/`comx` interactions involving loop instructions can be modeled with just two loop unrollings.

[5]https://github.com/nmosier/clou

Function Inlining: With loops summarized, CLOU inlines all function calls. Recursive calls are inlined twice via similar logic to that which enables loop summarization. For a call whose target function is not defined, CLOU interprets it as a load *or* store to one of its pointer operands—e.g., `memcmp(void *dst, const void *src, size_t n)` can behave as a load or store to `*dst` or `*src`. An SMT solver considers all possible options when searching for a way to construct a candidate execution featuring leakage.

5.2 Constructing a Symbolic Abstract Event Graph (S-AEG)

CLOU extends an A-CFG to produce a *Symbolic Abstract Event Graph (S-AEG)*—an over-approximation of all of the corresponding function's possible candidate executions.[6] An S-AEG features exactly the same set of nodes as the A-CFG from which it is derived. However, four categories of symbolic edges are added: control-flow (`po` and `tfo`), `dep`, `com`, and `comx`. Moreover, symbolic variables are associated with each S-AEG node and edge. Fig. 7 shows an example with node variables omitted for clarity. Legal assignments to these variables are constrained by a set of first-order logic formulas that describe what constitutes a consistent candidate execution, namely a consistency predicate (i.e., an MCM) and a confidentiality predicate (i.e., an LCM). Deriving a concrete candidate execution from an S-AEG may then be achieved by searching for a variable assignment which satisfies said formulas.

Specifying an MCM and LCM: Future versions of CLOU will be parameterizable, requiring an MCM and LCM to be provided as inputs alongside a C program. In its current form, CLOU hard-codes these inputs as follows. Other than the assumptions that the target hardware features write-allocate caches and does not implement silent stores [41] or alias prediction, we conservatively leave `comx` unconstrained; `com` is constrained by the TSO consistency predicate (§2.1.3).

Design decisions: We currently make a few key design decisions regarding S-AEG construction.

First, we assume LCMs where only memory instructions induce `xstate` accesses, under the assumptions outlined in §3.2.1.

Second, we assume a one-to-one correspondence between architectural addresses and modeled `xstate` locations, effectively modeling an infinitely-sized direct mapped cache. This assumption ensures that there will be no false negatives when searching an application for transmitters (although false positives are possible), regardless of the underlying cache implementation.

Third, given our empirical observations, CLOU specially considers a specific type of `addr` dependency, called `addr_gep` (*get-element-pointer address dependency*). `addr_gep` maps a `Read` to a `MemoryEvent`, where the `Read`'s return value (i.e., an index) is added to a base address to compute the `MemoryEvent`'s effective address.[7] Distinguishing `addr_gep` from other `addr` dependencies—which indicate the source instruction supplies a base address—enables CLOU to filter out benign SPECTRE v1 leaks (see 5.3).

[6]Compared to AEGs in prior work [1], S-AEGs are encoded more compactly as a set of first-order logic formulas, rather than as explicit graph data structures.
[7]LLVM IR features a pointer-only arithmetic instruction, called `getelementptr`, which signifies such behavior.

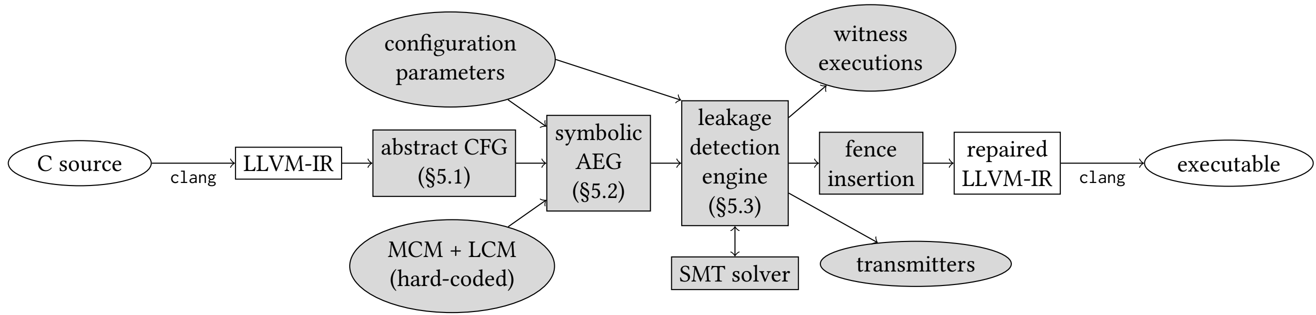


Figure 6: Clou accepts as input C source code and configuration parameters—ROB, LSQ, and window sizes (§5) for the S-AEG, and the desired transmitter class (§3.2.4) for the leakage detection engine. Clou currently considers hard-coded MCM and LCM inputs (§5.2). Clou produces a set of witness executions featuring leakage, a set of detected transmitters, and repaired LLVM-IR.

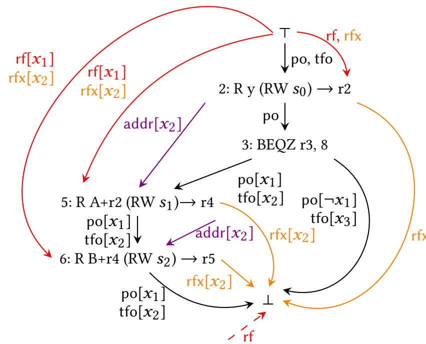


Figure 7: Clou produced this (simplified) S-AEG (§5.2) for the Spectre v1 program in Fig. 1a. Edges are labeled with SMT formulas that constrain their presence in a particular execution; edges without labels are present in all executions. Formulas x_1 and x_2 encode whether the if-body is non-speculatively or speculatively executed, respectively—the if-body is *mis-speculatively* executed if $\neg x_1 \wedge x_2$. Clou generates a set of constraints restricting S-AEG solutions, such as $x_1 \implies x_2$ and $\neg x_1 \implies x_3$ (i.e., `po` implies `tfo`).

Alias Analysis: Clou uses an alias analysis procedure to reduce the search space when looking for transmitters. First, Clou selectively applies LLVM's built-in alias analysis [40] to the S-AEG, only including constraints (that assert particular address pairs are unequal) when they are valid under Clou's CFG-to-A-CFG transformation. Next, Clou assumes that (1) all S-AEG stack allocations have distinct addresses and (2) alias analysis results do not hold during transient execution. Under these assumptions, Clou does not miss any true positive transmitters.

5.3 Leakage Detection Engines

Once an S-AEG has been constructed, Clou is ready to search the graph for potential transmitters. Clou initiates this procedure by adding all constraints encoded in the S-AEG to a Z3 SMT solver instance [20]. The next intuitive step is to directly encode as a constraint the expected behaviors of a leakage-free program (according to the non-interference predicates of §4.1), so that Z3 can search for violations of this constraint. Unsurprisingly, this approach produces a *large* number of transmitters, all of which are not equally interesting or dangerous. Thus, we develop specialized backends for Clou, called *leakage detection engines*, which are parameterized by transmitter type (Table 1) and which perform a directed search for a particular type of microarchitectural leakage.

We build two leakage detection engines for Clou, Clou-PHT and Clou-STL, which are designed to search for Spectre v1 and Spectre v4 style vulnerabilities, respectively. §4 shows that Spectre attacks violate the `rf`-non-interference predicate of our leakage definition in §4.1. Thus, Clou-PHT and Clou-STL both directly look for candidate executions which violate this condition—the result is a set of candidate transmitters. Clou iterates over the candidate transmitters to find those which are also data/control transmitters or universal transmitters (§3.2.4) according to the user's preference. In reality, an `addr` edge in the transmitter definitions of §3.2.4 can be realized as zero or more `data.rf` edges followed by an `addr` edge—i.e., $(\texttt{data.rf})^*.\texttt{addr}$. This means the value returned by a `Read` can be stored (`data`) and re-loaded (`rf`) any number of times before use in an `addr`-dependent access.

To filter benign leaks (§7), Clou-PHT requires the first `addr` dependency of a universal transmitter pattern to be of type `addr_gep`. In doing so, Clou-PHT assumes that all base pointers (architecturally) stored in memory are *not* attacker-controlled. `addr_gep` cannot be used to filter Spectre v4 leaks, however, since an attacker may also control a pointer's base address if mis-speculatively loading it from memory returns a stale attacker-controlled value.

Related to the above optimizations, both Clou-PHT and Clou-STL conduct taint-tracking of attacker-controlled data. The goal is to filter out candidate universal transmitters involving access instructions which cannot be steered towards arbitrary memory locations by an attacker. All top-level function inputs and non-pointer data in memory are initially assumed attacker-controlled.

In summary, Clou-PHT and Clou-STL differ only with regard to the speculation primitives they consider—*control-flow speculation* versus *store forwarding*, respectively. This design choice is intended to help the user understand which speculation primitives can be exploited to construct transmitters, an approach common to all state-of-the-art static analysis tools that detect transient execution vulnerabilities in programs [15, 17].

6 RESULTS

We run Clou on 36 Spectre benchmarks and 4 crypto-libraries. The only other tools that support both PHT- and STL-style leakage detection are Pitchfork [15] and Binsec/Haunted (BH) [17]. Pitchfork scales poorly (see Table IV in [17]) for the workloads of interest, so we compare Clou to BH. However, unlike Clou, BH does not distinguish between the different classes of transmitters we define (Table 1). Unless otherwise stated, all Clou experiments assume realistic ROB/LSQ capacities of 250/50. For BH, we use ROB/LSQ capacities of 200/20 from the original paper [17]. All experiments are run on an Intel(R) Xeon(R) Gold 6226R CPU @ 2.90GHz server featuring 4 processors, 16 cores per processor, and 512 GB of RAM.

6.1 Analyzing Spectre Benchmarks with Clou

We run Clou on 15 Spectre v1 (**PHT**) [36], 5 Spectre v1.1 (**FWD**), and 14 Spectre v4 (**STL**) [18] benchmarks from prior work plus 2 Spectre v1.1-style benchmarks of our own (**NEW**). We run Clou on each program and record the type(s) of transmitters it *detects*—data transmitters (DTs), universal data transmitters (UDTs), control (CTs), and universal control transmitters (UCTs). We also manually inspect each program and record the type of transmitter it is *intended* to feature, using annotations from the benchmark authors.

Clou identifies all intended transmitters in the **PHT** programs. Clou also identifies a *new attack variant* in all PHT programs—a DT involving a transient instruction prefetching a cache line for a non-transient tfo-*prior* instruction. *Speculative interference attacks* [11] exhibit a similar phenomenon.

For the **STL** programs, Clou identifies *more transmitters* than intended by the authors that are in some cases *more severe*. For example, in STL01 (below), Clou identifies the intended transmitter—the access to pub_array in line 5. The benchmark authors' comments discuss its implications as a DT.

```
1  void case_1(uint32_t idx) {
2    uint32_t ridx=idx&(ary_size-1);
3    uint8_t **pp=&sec_ary; uint8_t ***ppp=&pp;
4    (**ppp)[ridx]=0;
5    tmp &= pub_ary[sec_ary[ridx]];} // transmitter
```

Specifically, line 5's access to sec_ary[ridx] may transiently read stale data before it is overwritten in line 4. However, Clou also finds a candidate execution where the same transmitter facilitates *universal* data leakage. Namely, if line 5's stack read of idx computes its address before line 2's stack write of idx does, it can access stale data left on the stack. The authors of the STL benchmarks consider this sort of speculative behavior valid—in fact it is central to STL02.

Clou also finds that STL13 is *incorrectly labeled as "secure"* in the benchmark *and* flagged as secure by the benchmark authors' static analysis tool, BH [17]. It features data leakage when a return instruction bypasses a store to the stack.

Clou detects false positive leakage for 7 **STL** programs for two reasons. First, Clou does not perform semantic analysis and thus cannot reason about the implications of index masking, a mitigation technique used by many STL programs. Also, Clou does not consider the impact of loops on speculation depth when summarizing them. Therefore, false positive leakage involving instructions which cannot exist in the processor simultaneously (e.g., due to ROB size) may be flagged.

Some STL benchmarks are intended to be *safe* due to their use of C's register keyword to prevent *storing* an array index (to the stack). We find that Clang -O0 disregards the register keyword and stores the index to memory anyway, enabling it to be bypassed. We manually repair the LLVM IR output to create the effect of register.

Clou finds all intended leakage in the **FWD** and **NEW** benchmarks. Notably, Pitchfork fails to detect leakage in both **NEW** tests, although BH succeeds. We crafted NEW01 (below) to showcase novel form of Spectre v1.1 leakage which involves an attacker-controlled speculative write to a pointer/index in memory with a secret returned by an attacker-controlled access instruction. The overwritten pointer/index is then used to access memory (in line 4), transmitting the secret returned by the access instruction.

```
1  void new_1(size_t idx1, size_t idx2) {
2    if (idx1 < sec_ary1_size && idx2 < sec_ary2_size)
3      sec_ary2[idx2] += sec_ary1[idx1] * 512;
4    *ptr = 0; }
```

A key feature of Clou is its ability to insert a minimal number of fences to repair Spectre v1 and Spectre v4 leaks. Clou can repair Spectre v1.1 leaks as well, but not minimally. We direct Clou to perform fence insertion in all benchmarks and confirm that all initially-detected leakage is mitigated—with 1 (resp. 2) fence per vulnerable program for **PHT** and **STL** (resp. **FWD** and **NEW**) benchmarks.

6.2 Analyzing Crypto-Libraries with Clou

We use Clou to analyze the crypto-libraries tea [73], donna [12], libsodium [21], and OpenSSL [56]. Table 2 summarizes our findings. Each row represents a distinct application, for which Clou analyzes each public function individually. Fig. 8 shows per public function runtimes for libsodium specifically. Table 2 denotes for each application the number of public functions (**PFun**), the number of unique functions after inlining (**Fun**), and the static lines of code (**LoC**). BH [17] specifically analyzed the secretbox function in libsodium and the ssl13-digest and mee-cbc functions in OpenSSL. Thus, Table 2 presents isolated performance results for these functions using the same source code versions that BH originally analyzed [17]. Other results use new libsodium v1.0.18 and OpenSSL v3.1.0-dev.

6.2.1 Completeness Guarantees. In analyzing large codebases, Clou sacrifices completeness for performance by: (1) leveraging a "sliding window" approach, in which for each candidate transmitter Clou only considers the set of instructions in the S-AEG that can reach the transmitter in W_{size} instructions, (2) allowing at most one speculative write (excluding the transmitter instruction itself) in any detected transmitter pattern (§5.3), and (3) disallowing non-speculative writes in any detected transmitter pattern. Furthermore, we direct Clou to ignore universal transmitter patterns involving non-transient access instructions when searching for UDTs/UCTs, instead classifying them as DTs/CTs. While greater than a pure DT/CT, the leakage scope of a universal transmitter involving a non-transient access instruction is still somewhat limited since the access instruction must commit.

Restriction (1) may result in Clou missing or mis-classifying some universal transmitters. As long as addr dependencies span

App. (PFun/Fun/LoC)	Tool	Time (s) (DT/CT/UDT/UCT)	Bugs (DT/CT/UDT/UCT)
`litmus-pht`	Clou-PHT	3.2/3.4/2.8/2.6	29/86/14/3
(15/15/200)	BH-PHT	20.9	22
`litmus-stl`	Clou-STL	7.3/2.4/4.3/2.4	33/4/17(14)/0
(14/16/312)	BH-STL	6.1	13
`litmus-fwd` (5/9/129)	Clou-PHT	2.4/1.5/1.8/1.4	9/17/2/0
	Clou-STL	4.2/2.0/2.3/1.4	11/24/2/0
	BH-PHT	0.4	3
	BH-STL	588.9	12
`litmus-new` (2/2/100)	Clou-PHT	0.5/0.4/0.5/0.4	6/4/2/0
	Clou-STL	0.9/0.5/0.5/0.4	7/4/2/0
	BH-PHT	0.5	2
	BH-STL	32.0	3
`tea` (2/4/102)	Clou-PHT	0.25/0.17	0/0
	Clou-STL	0.79/0.51	0/0
	BH-PHT	0.37	0
	BH-STL	18.4	4
`donna` (1/21/874)	Clou-PHT	3252.8/**3670**	0/0
	Clou-STL[1]	**27683**/**21853**	514(0)/0
	BH-PHT	**3600**	0
	BH-STL	**3600**	15
`secretbox` (1/12/142)	Clou-PHT	495.8/495.2	0/0
	Clou-STL	512.0/507.2	0/0
	BH-PHT	2611.4	17
	BH-STL	**21600**	26
`ssl13-digest` (1/23/1563)	Clou-PHT	80.7/90.8	0/0
	Clou-STL	1237.8/7989.8	98(0)/53(0)
	BH-PHT	4375	13
	BH-STL	**21600**	1
`mee-cbc` (1/6/1157)	Clou-PHT	**443735**/**595650**	7(0)/85(0)
	Clou-STL	**47606**/**646215**	17(0)/6(0)
	BH-PHT	**21600**	17
	BH-STL	**21600**	2
`libsodium`	Clou-PHT	995/1078	7(0)/20(0)
(646/733/7078)	Clou-STL[2]	49453.6/13046	1266(1)/275(89)
`OpenSSL`	Clou-PHT	**171997**/–	755(60)/–
(3307/5408/161552)	Clou-STL	**779209**/–	11531(3383)/–

Table 2: For each public function, Clou is run once per transmitter type of interest. For crypto-libraries, Clou looks for UDTs and UCTs only. The *serial* runtime of Clou and BH are presented in the "Time" column. Both Clou and BH are run with BH's original timeouts (runtimes in bold)—6 hours wall runtime for `libsodium`/`OpenSSL` functions and 1 hour for all others [17]. For the `OpenSSL` library (not analyzed by BH), Clou imposes a timeout of 1 hour per C source file. Clou is highly parallel, so total serial runtime greatly exceeds wall runtime. [1]$W_{size} = 350$; [2]$W_{size} = 350$ **(UDT) and** $ROB = 200, W_{size} = 250$ **(UCT).**

less than W_{size} instructions, Clou is only at risk of mis-classifing some universal transmitters as vanilla DTs/CTs; it will not miss them entirely. Clou is also guaranteed to correctly classify all universal transmitters which leak transiently accessed secrets, if `addr` dependencies with committed sources do not span more than $W_{size} - ROB_{size}$ instructions. One could devise a static analysis pass to confirm that programs adhere to these requirements.

Restriction (2) enables Clou to conduct a directed search for universal transmitters that are more likely to be a true positives. Multiple speculative writes imply multiple `rf` edges that compound the imprecision of alias analysis.

Finally, the implication of Restriction (3) is subsumed by our decision to have Clou ignore universal transmitter patterns involving non-transient access instructions when searching for UDTs/UCTs.

6.2.2 Crypto-Library Analysis Post-Processing. We inspect all universal transmitters flagged in Table 2 to filter out false positive (§6.1) and low priority transmitters. In evaluating larger code bases, we observe one class of data leakage (resembling our **NEW** benchmarks) that Clou misclassifies as universal. This special case involves an `addr.data.rf.addr` pattern where the transmitter leaks a pointer value which it reads (via `rf`) from a speculative write. In short, for such a transmitter to facilitate a *universal* data leakage, the source and destination of `data` must access *different* addresses. Furthermore, we consider transmitters to be *low priority* if they require more than one read to speculatively accesses stale data. While we manually filtered out these false-positive and low-priority cases, it is possible to build a post-processing mechanism that performs this step automatically. The results of post-processing filtering are *not* reflected in Table 2; they were used in our qualitative transmitter analysis only (§6.2.3).

In general, most false positives flagged by Clou are due to imprecise alias analysis. Thus, we include a second transmitter count in parentheses in Table 2 that gives the number of universal transmitters under worst-case alias analysis, where all `data.rf` edges are assumed to be erroneous. In other words, this statistic only counts UDTs/UCTs of the form `addr_gep.(addr/ctrl)` (for Clou-PHT) and `addr.(addr/ctrl)` (for Clou-STL). There are comparatively few of these restricted patterns, but they are much more likely to represent true-positive leakage. Note, however, that many of the true-positive crypto-library vulnerabilities Clou uncovered *do* involve speculative writes of intermediate values, e.g., Listing 1.

6.2.3 Crypto-Library Analysis Results. In `tea`, BH flags four instances of leakage; Clou flags none for two reasons. First, all four leaks involve DTs (not UDTs). Second, they involve stores of return addresses to the stack which are not present at the LLVM-IR level. In `secretbox`, BH flags 43 instances of leakage that Clou misses since it analyzes a x86 binary compiled with stack protector checks. Clou analyzes the LLVM-IR before any stack protectors are added. All vulnerabilities reported by BH are within the function `__libc_message`, which is called on stack protector check failure. In `donna`, `ssl13-digest`, and `mee-cbc`, Clou finds considerably more transmitters than BH. BH does report 13 and 17 distinct PHT leaks in `ssl3-digest` and `mee-cbc`, respectively. However, these are all (non-universal) DTs/CTs, and BH times out before finding any of the universal leakage discovered by Clou.

Clou is the first tool to search all public `libsodium` functions for PHT and STL leakage. Large codebases, like `OpenSSL`, are generally considered outside the scope of formal program analysis tools. However, we successfully run Clou-PHT (resp. Clou-STL) in UDT detection mode on 90%/58% (resp. 81%/60%) of `OpenSSL`'s

functions[8]/LoC. Clou evaluates more of OpenSSL than any prior work.

Clou uncovers many novel Spectre gadgets in* libsodium *(3 UDTs and 4 UCTs) and* OpenSSL *(6 UDTs and 2 UCTs).[9] A less severe type of UDT, which combines Spectre v1.1 and Spectre v4, is also found in 116 libsodium functions. Note that many transmitters uncovered by Clou can be grouped into equivalence classes, where each class of transmitters can be mitigated by preventing a single culprit speculative access. We report one gadget per equivalence class. *A novel PHT gadget in OpenSSL's* SSL_get_shared_sigalgs *function [57, 65], shown below, appears to be most severe vulnerability uncovered by Clou.*

Listing 1: OpenSSL SSL_get_shared_sigalgs Vulnerability

```
1   int SSL_get_shared_sigalgs(SSL *s, int idx, int *psign,
2                              int *phash, int *psignhash,
3                              uint8_t *rsig, uint8_t *rhash) {
4     const SIGALG_LOOKUP *shsigalgs;
5     if (s→ shared_sigalgs == NULL
6         || idx < 0 || idx >= (int)s->shared_sigalgslen
7         || s→ shared_sigalgslen > INT_MAX)
8       return 0;
9     shsigalgs = s->shared_sigalgs[idx];
10    if (phash != NULL) *phash = shsigalgs->hash;
11    if (psign != NULL) *psign = shsigalgs→ sig;
12    if (psignhash != NULL) *psignhash = shsigalgs→ sigandhash;
13    if (rsig != NULL) *rsig = (uint8_t)(shsigalgs→ sigalg & 0xff);
14    if (rhash != NULL)
15      *rhash = (uint8_t)((shsigalgs→ sigalg >> 8) & 0xff);
16    return (int)s→ shared_sigalgslen; }
```

Line 6 performs a bounds check on an attacker-controlled index parameter idx. If the load of s->shared_sigalgslen experiences a cache miss, the bounds check will take a long time to resolve. Thus, a processor with branch prediction may guess that idx is in bounds and speculatively execute the lines following the if-statement. Such mis-speculation can enable a speculative out-of-bounds load of an arbitrary secret from s->shared_sigalgs[idx], speculatively writing it into pointer variable shsigalgs on line 9. The struct member access shsigalgs->hash on line 10 dereferences the secret as a pointer, directly leaking the secret's value into the cache.

6.2.4 Clou Performance. We observed that for most functions, Clou spends nearly all its runtime solving SMT queries in Z3 [20], and the total solving time is dominated by a small subset of challenging queries.[10] The abnormally long libsodium function runtimes in Fig. 8 and the timeouts in Table 2 generally occur when Z3 has gotten "stuck" while solving such a query. We leave it to future work to optimize Clou's usage of the SMT solver to reduce the number of difficult-to-solve queries.

7 RELATED WORK

Detecting Transient Leakage: Researchers have proposed tools to detect Spectre-style vulnerabilities at the binary [15, 17, 26, 55, 71] and LLVM-IR levels [28, 70, 77]. However, all existing tools either scale poorly or face qualitative limitations. Spectector [26] detects

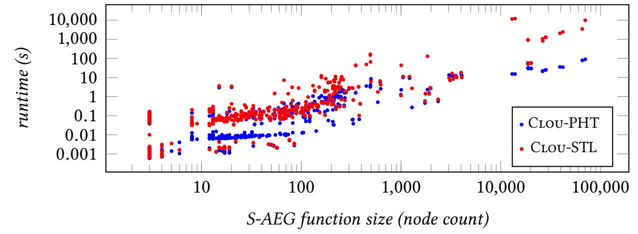


Figure 8: Serial CPU runtime vs. function size for Clou's libsodium analysis. No functions time out.

Spectre v1 gadgets in code using symbolic execution. Besides being limited to Spectre v1, Spectector [26] does not scale well to large codebases and is also based on the program-counter security model [50] which disallows branching on secrets. Pitchfork [15] uses symbolic execution to detect Spectre v1/v1.1/v4 violations; but, its implementation is unsound [8], and its Spectre v4 detection scales poorly [17]. BH [17] also uses symbolic execution and supports detection of Spectre v1/v1.1/v4 violations; it scales better than Pitchfork by symbolically reasoning about transient/non-transient program behaviors simultaneously.

Clou is not based on symbolic execution; it uses axiomatic LCM definitions to holistically represent legal program executions as directed graphs which can be systematically checked for leakage. Clou is currently restricted to detecting Spectre v1/v1.1/v4 leakage but scales better than all prior approaches finding new vulnerabilities in real-world crypto-libraries. LCMs also support branching on secrets and are not limited to reasoning about vulnerabilities involving transient execution.

To our knowledge, Clou is the only tool for identifying Spectre-style vulnerabilities in programs that distinguishes between DTs/CTs and UDTs/UCTs. This enables Clou to efficiently focus on and detect the most severe leakage in any application component without requiring secrecy labels. However, adding support for secrecy labels to Clou can help filter benign DTs/CTs, as is done in prior work.

Formalizing Transient Leakage: Recent research applies formal rigor to reasoning about the impact of transient execution attacks on software [8, 15, 25–27, 55, 59, 68]. Cauligi et al. [15] defines *speculative constant-time* using an adversarial semantics for speculative execution. Similar to LCMs, their modeling approach captures a variety of transient execution attacks. InSpectre [25] features an operational model to support reasoning about transient execution attacks and countermeasures. Guarnieri et al. [27] proposed operational hardware-software contracts to explicitly expose to software which aspects of microarchitectural state are observable to an adversary as a program executes. Concurrent work [19] proposes using axiomatic MCM definitions to formally model and automatically detect *access instructions* (§3.2.4) in programs—namely memory read events which are capable of accessing secrets. In contrast to LCMs, this approach cannot determine if the data read by a particular access instruction can be leaked via a transmitter nor can it support transmitter classification. Modeling microarchitectural leakage which does not involve architectural read events accessing

[8]Specifically, all functions which are callable from public functions.
[9]Clou's libsodium/OpenSSL vulnerability repository [51]. We have responsibly disclosed these gadgets to the library developers.
[10]In many cases, the reasons why this particular query is challenging is not apparent at the source level of the function being analyzed.

secrets (as for silent stores) is also not supported. Further, analyses based on this technique take up to 90 minutes to inspect the benchmarks which Clou evaluates in under a second.

Finally, Blade [68] uses a static type system to eliminate transient leakage from CT cryptographic code. Blade prohibits `Spectre v1` leakage by breaking flows from transiently-typed expressions to sinks with a hypothetical fence called *protect*. Similar to Blade, LCMs can synthesize a minimum number of fences; they can also effectively use the *protect* fence. In contrast to Blade's conservative type system, LCMs are more accurate and lead to fewer false-positives.

8 CONCLUDING REMARKS

We propose LCMs—new security contracts that enable programmers, compiler writers, and runtime designers to reason about the security implications of hardware on software. LCMs support precisely pinpointing hardware-related vulnerabilities in programs. In turn, they support the design and development of (1) formal analysis frameworks, like subrosa and (2) tools which can detect and repair vulnerable programs, like Clou. One limitation of LCMs is the type of side-channels they capture—LCMs capture leakage that results from inter-instruction *interactions* through hardware state rather than from operand-dependent variable time execution of individual instructions (e.g., due to subnormal floating point optimizations [5]). Such an enhancement to the formalism is left for future work.

ACKNOWLEDGMENTS

We would like to thank John Mitchell and Clark Barrett for their valuable discussions and feedback on this work. This work was supported in part by the National Science Foundation (NSF, Award Number 2017863) and the German Federal Ministry of Education and Research (BMBF) through funding for the CISPA-Stanford Center for Cybersecurity (FKZ: 13N1S0762). We also gratefully acknowledge a research gift from Intel Corporation.

REFERENCES

[1] Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. 2017. Don't sit on the fence: A static analysis approach to automatic fence insertion. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 39, 2 (2017), 1–38.

[2] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2010. Fences in Weak Memory Models. (2010). http://dx.doi.org/10.1007/978-3-642-14295-6_25

[3] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2011. Litmus: Running Tests Against Hardware. *17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS): Part of the Joint European Conferences on Theory and Practice of Software (ETAPS)* (2011).

[4] Jade Alglave, Luc Maranget, and Michael Tautschnig. 2014. Herding Cats: Modelling, Simulation, Testing, and Data Mining for Weak Memory. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 36, 2 (2014), 7:1–7:74.

[5] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham. 2015. On Subnormal Floating Point and Abnormal Timing. In *2015 IEEE Symposium on Security and Privacy*.

[6] ARM. 2013. ARM A64 instruction set architecture. https://developer.arm.com/documentation

[7] ARM Mbed. x. Mbed TLS. https://github.com/armmbed/mbedtls

[8] Gilles Barthe, Sunjay Cauligi, Benjamin Grégoire, Adrien Koutsos, Kevin Liao, Tiago Oliveira, Swarn Priya, Tamara Rezk, and Peter Schwabe. 2021. High-Assurance Cryptography in the Spectre Era. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*.

[9] Mark Batty, Alastair F. Donaldson, and John Wickerson. 2016. Overhauling SC Atomics in C11 and OpenCL. *43rd Symposium on Principles of Programming Languages (POPL)* (2016).

[10] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. 2011. Mathematizing C++ Concurrency. *38th Symposium on Principles of Programming Languages (POPL)* (2011).

[11] Mohammad Behnia, Prateek Sahu, Riccardo Paccagnella, Jiyong Yu, Zirui Neil Zhao, Xiang Zou, Thomas Unterluggauer, Josep Torrellas, Carlos Rozas, Adam Morrison, Frank Mckeen, Fangfei Liu, Ron Gabor, Christopher W. Fletcher, Abhishek Basak, and Alaa Alameldeen. 2021. Speculative Interference Attacks: Breaking Invisible Speculation Schemes. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.

[12] Dan Bernstein. 2008. curve25519-donna. https://code.google.com/archive/p/curve25519-donna/.

[13] Hans-J. Boehm and Sarita V. Adve. 2008. Foundations of the C++ Concurrency Memory Model. *29th Conference on Programming Language Design and Implementation (PLDI)* (2008).

[14] James Bornholt and Emina Torlak. 2017. Synthesizing Memory Models from Framework Sketches and Litmus Tests. *38th Conference on Programming Language Design and Implementation (PLDI)* (2017).

[15] 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*.

[16] 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)*.

[17] Lesly-Ann Daniel, Sébastien Bardin, and Tamara Rezk. 2021. Hunting the Haunter - Efficient Relational Symbolic Execution for Spectre with Haunted RelSE. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*.

[18] Lesly-Ann Daniel. 2021. Binsec/haunted benchmark. https://github.com/binsec/haunted_bench/.

[19] Hernán Ponce de León and Johannes Kinder. 2021. Cats vs. Spectre: An Axiomatic Approach to Modeling Speculative Execution Attacks. https://arxiv.org/abs/2108.13818

[20] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 337–340.

[21] Frank Denis. 2019. libsodium. https://github.com/jedisct1/libsodium.

[22] Craig Disselkoen, Radha Jagadeesan, Alan Jeffrey, and James Riely. 2019. The Code That Never Ran: Modeling Attacks on Speculative Evaluation. In *2019 IEEE Symposium on Security and Privacy (SP)*.

[23] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. 2016. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering* (2016).

[24] J. A. Goguen and J. Meseguer. 1982. Security Policies and Security Models. In *1982 IEEE Symposium on Security and Privacy*.

[25] 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*.

[26] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez. 2020. Spectector: Principled Detection of Speculative Information Flows. In *2020 IEEE Symposium on Security and Privacy (SP)*.

[27] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. 2021. Hardware-Software Contracts for Secure Speculation. In *2021 IEEE Symposium on Security and Privacy*.

[28] Shengjian Guo, Yueqi Chen, Peng Li, Yueqiang Cheng, Huibo Wang, Meng Wu, and Zhiqiang Zuo. 2020. SpecuSym: Speculative Symbolic Execution for Cache Timing Leak Detection. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*.

[29] Jann Horn. 2018. Speculative execution, variant 4: Speculative store bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528

[30] Yao Hsiao, Dominic P. Mulligan, Nikos Nikoleris, Gustavo Petri, and Caroline Trippel. 2021. Synthesizing Formal Models of Hardware from RTL for Efficient Verification of Memory Model Implementations. In *Proceedings of the Fifty-Fourth IEEE/ACM International Symposium on Microarchitecture (MICRO 54)*.

[31] Intel. 2018. Q2 2018 Speculative Execution Side Channel Update. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00115.html

[32] Intel. 2019. Intel® 64 and IA-32 Architectures Software Developer Manuals, Order Number: 325462-070US. https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf

[33] D. Jackson. 2012. Alloy Analyzer website. http://alloy.mit.edu/

[34] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. 2018. DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*.

[35] Vladimir Kiriansky and Carl Waldspurger. 2018. Speculative Buffer Overflows: Attacks and Defenses. *CoRR* abs/1807.03757 (2018). arXiv:1807.03757 https://dblp.org/rec/bib/journals/corr/abs-1807-03757 http://arxiv.org/abs/1807.03757.

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

[37] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2018. Spectre Attacks: Exploiting Speculative Execution. *CoRR* abs/1801.01203 (2018). https://arxiv.org/abs/1801.01203

[38] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (1978), 558–565.

[39] Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computing* 28, 9 (1979), 690–691.

[40] Chris Lattner and Vikram Adve. 2003. *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation.* Tech. Report UIUCDCS-R-2003-2380. Computer Science Dept., Univ. of Illinois at Urbana-Champaign.

[41] Kevin M. Lepak and Mikko H. Lipasti. 2000. Silent Stores for Free. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture.*

[42] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown. *CoRR* abs/1801.01207 (2018). https://arxiv.org/abs/1801.01207

[43] Jason Lowe-Power, Venkatesh Akella, Matthew K. Farrens, Samuel T. King, and Christopher J. Nitta. 2018. Position Paper: A Case for Exposing Extra-architectural State in the ISA. In *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy.*

[44] Daniel Lustig, Andrew Wright, Alexandros Papakonstantinou, and Olivier Giroux. 2017. Automated Synthesis of Comprehensive Memory Model Litmus Test Suites. *22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2017).

[45] Jeremy Manson, William Pugh, and Sarita Adve. 2005. The Java Memory Model. *32nd Symposium on Principles of Programming Languages (POPL)* (2005).

[46] Margaret Martonosi et al. 2017. Check: Research Tools and Papers. http://check.cs.princeton.edu

[47] Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L. Titzer, and Toon Verwaest. 2019. Spectre is here to stay: An analysis of side-channels and speculative execution. https://arxiv.org/abs/1902.05178

[48] Matt Miller. 2018. Mitigating speculative execution side channel hardware vulnerabilities. https://msrc-blog.microsoft.com/2018/03/15/mitigating-speculative-execution-side-channel-hardware-vulnerabilities/

[49] Marina Minkin, Daniel Moghimi, Moritz Lipp, Michael Schwarz, Jo Van Bulck, Daniel Genkin, Daniel Gruss, Berk Sunar, Frank Piessens, and Yuval Yarom. 2019. Fallout: Reading Kernel Writes From User Space. (2019).

[50] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. 2005. The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks. In *Proceedings of the 8th International Conference on Information Security and Cryptology.*

[51] Nicholas Mosier and Caroline Trippel. 2022. clou-bugs. https://github.com/nmosier/clou-bugs

[52] Vijay Nagarajan, Daniel Sorin, Mark Hill, and David Wood. 2020. *A Primer on Memory Consistency and Cache Coherence, Second Edition.* Morgan & Claypool Publishers. https://doi.org/10.2200/S00962ED2V01Y201910CAC049

[53] Kyndylan Nienhuis, Kayvan Memarian, and Peter Sewell. 2016. An Operational Semantics for C/C++11 Concurrency. *31st International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)* (2016).

[54] NVIDIA. 2017. Parallel Thread Execution ISA Version 6.0. http://docs.nvidia.com/cuda/parallel-thread-execution/index.html

[55] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. 2020. SpecFuzz: Bringing Spectre-type vulnerabilities to the surface. In *29th USENIX Security Symposium (USENIX Security 20).*

[56] OpenSSL 2021. OpenSSL: Cryptography and SSL/TLS Toolkit. https://www.openssl.org/.

[57] OpenSSL 2022. OpenSSL's implementation of `SSL_get_shared_sigalgs`. https://github.com/openssl/openssl/blob/d5530efada83825ef239a8458db541adc4b422ec/ssl/t1_lib.c#L2408

[58] Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A Better x86 Memory Model: x86-TSO. *22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs)* (2009).

[59] Marco Patrignani and Marco Guarnieri. 2021. Exorcising Spectres with Secure Compilers. arXiv:1910.08607 [cs.PL]

[60] Gustavo Petri, Jan Vitek, and Suresh Jagannathan. 2015. Cooking the books: Formalizing JMM implementation recipes. *29th European Conference on Object-Oriented Programming (ECOOP)* (2015).

[61] Thomas Pornin. 2016. Why Constant-Time. https://www.bearssl.org/constanttime.html

[62] Thomas Pornin. 2018. Constant-Time Toolkit. https://github.com/pornin/CTTK.

[63] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. 2017. Simplifying ARM Concurrency: Multicopy-atomic Axiomatic and Operational Models for ARMv8. *ACM Programming Languages* (2017).

[64] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. *CoRR* abs/1905.05726 (2019). arXiv:1905.05726 https://arxiv.org/abs/1905.05726

[65] The OpenSSL Project. 2021. *SSL_get_shared_sigalgs.* https://www.openssl.org/docs/man3.0/man3/SSL_get_shared_sigalgs.html

[66] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. 2018. CheckMate: Automated Synthesis of Hardware Exploits and Security Litmus Tests. *51st International Symposium on Microarchitecture (MICRO)* (2018).

[67] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue In-flight Data Load. *S&P* (May 2019).

[68] Marco Vassena, Craig Disselkoen, Klaus von Gleissenthall, Sunjay Cauligi, Rami Gökhan Kıcı, Ranjit Jhala, Dean Tullsen, and Deian Stefan. 2021. Automatically Eliminating Speculative Leaks from Cryptographic Code with Blade. *Proc. ACM Program. Lang.* (2021).

[69] Jose 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 *ISCA'21.*

[70] Guanhua Wang, Sudipta Chattopadhyay, Arnab Kumar Biswas, Tulika Mitra, and Abhik Roychoudhury. 2020. KLEESpectre: Detecting Information Leakage through Speculative Cache Attacks via Symbolic Execution. (2020).

[71] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. 2021. oo7: Low-Overhead Defense Against Spectre Attacks via Program Analysis. (2021).

[72] Andrew Waterman and Krste Asanović. 2019. *The RISC-V Instruction Set Manual, Volume I: Unprivileged ISA Document, Version 20190608-Base-Ratified.* Technical Report. SiFive Inc. and CS Division, EECS Department, University of California, Berkeley. https://riscv.org/specifications/

[73] David Wheeler and Roger Needham. 1994. The Tiny Encryption Algorithm. https://www.schneier.com/sccd/TEA.C.

[74] John Wickerson, Mark Batty, Bradford M. Beckmann, and Alastair F. Donaldson. 2015. Remote-Scope Promotion: Clarified, Rectified, and Verified. *30th International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (2015).

[75] John Wickerson, Mark Batty, Tyler Sorensen, and George A Constantinides. 2017. Automatically comparing memory consistency models. *44th Symposium on Principles of Programming Languages (POPL)* (2017).

[76] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. 2014. The CHERI Capability Model: Revisiting RISC in an Age of Risk. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture.*

[77] Meng Wu and Chao Wang. 2019. Abstract Interpretation under Speculative Execution. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation.*

[78] Jiyong Yu, Lucas Hsiung, Mohamad El Hajj, and Christopher W. Fletcher. 2019. Data Oblivious ISA Extensions for Side Channel-Resistant and High Performance Computing. In *26th Annual Network and Distributed System Security Symposium, NDSS.*

[79] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. 2019. Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture.*

[80] Xiangyao Yu, Christopher J. Hughes, Nadathur Satish, and Srinivas Devadas. 2015. IMP: Indirect memory prefetcher. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[81] X. Yu, C. J. Hughes, and N. R. Satish. 2016. Hardware prefetcher for indirect access patterns. US Patent 14/582,348. Filed December 24, 2014. Issued June 30, 2016..

[82] Drew Zagieboylo, G. Edward Suh, and Andrew C. Myers. 2019. Using Information Flow to Design an ISA that Controls Timing Channels. In *32nd IEEE Computer Security Foundations Symposium, CSF.*

[83] Tianwei Zhang and Ruby B. Lee. 2014. New Models of Cache Architectures Characterizing Information Leakage from Cache Side Channels. In *Proceedings of the 30th Annual Computer Security Applications Conference.*