Relational Models of Microarchitectures for Formal Security Analyses

Nicholas Mosier∗, Hanna Lachnitt∗, Hamed Nemati†, Caroline Trippel∗

∗Stanford University †CISPA Helmholtz Center for Information Security

{nmosier, lachnitt, hnemati, trippel}@stanford.edu

Abstract—There is a growing need for hardware-software contracts which precisely define the implications of microarchitecture on software security—i.e., security contracts. It is our view that such contracts should explicitly account for microarchitecture-level implementation details that underpin hardware leakage, thereby establishing a direct correspondence between a contract and the microarchitecture it represents. At the same time, these contracts should remain as abstract as possible so as to support efficient formal analyses. With these goals in mind, 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 formally defining LCMs, derived from the established axiomatic vocabulary used to formalize 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. To illustrate the efficacy of LCMs, we present two case studies. First, we 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 realistic-sized codebases, like libsodium.

I. INTRODUCTION

Hardware which is under-specified or whose implementation deviates from its specification can introduce correctness bugs and/or security vulnerabilities into seemingly correct and secure programs [5, 8, 17, 26, 27, 40, 67, 70, 78]. Unfortunately, microarchitectural attacks [25] expose a notable deficiency in how hardware-software contracts have historically defined software-visible state. Microarchitectural attacks are side/covert channel attacks which enable leakage/communication as a direct result of hardware optimizations. Thus, rather than consisting solely of state that can be directly accessed with committed user-facing instructions (i.e., architectural state), software-visible state actually also includes any microarchitectural 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 well-established way to counter microarchitectural attacks that manifest as timing channels is with constant-time (CT) programming—a paradigm that disallows the processing of secrets by transmit instructions [36, 80] (i.e., transmitters) which can leak their results, operands, or even data at rest in architectural structures [70] 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 [9, 23, 62, 63]. However, these restrictions are insufficient for modern hardware where secrets can be steered towards transient transmitters [39, 45, 53, 65, 68]. Also, the scope of transmitters extends beyond branch and memory instructions [70].

To address the need for security contracts, various proposals have emerged [7, 17, 18, 21, 24, 27–29, 50, 69, 76, 79, 83]. Some require hardware enhancements to explicitly track/enforce contract-level security primitives [7, 76, 79, 83]. Other contracts restrict the scope of hardware features that they consider when summarizing a microarchitecture’s security implications, focusing on in-order [18, 27, 28] and single-core processor designs [17, 18, 24, 27–29, 50, 69] for example.

Most recently, security contracts have been proposed which solely expose transient leakage through microarchitecture to software [17, 18, 21, 24, 27, 28, 50, 69] or highly restrict the non-transient leakage they can capture [29]. Notably, some of these contracts require hard-coding pre-defined observations that a program can produce as it executes on a microarchitecture, given known side/covert-channels [17, 21, 27, 29, 69].

Existing security contracts share a couple of key limitations. First, their use of known observations does not capture the root cause of microarchitectural leakage. Second, they are not easily verifiable with respect to microarchitecture. The extreme degree to which they abstract away hardware details makes it difficult (if not impossible) to establish a connection between contract primitives and hardware features described with hardware description languages (HDLs).

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 even synthesized from, the
microarchitectures they represent [32]. Moreover, a generic 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 the confidentiality guarantees of programs when they run on particular microarchitectures. While there has been a particular emphasis in the literature on formalizing security contracts operationally, we take an axiomatic approach in this work. In doing so, we expect LCMs to benefit from recent work on automatically synthesizing axiomatic specifications of hardware from RTL directly [32].

LCMs are designed to capture the root cause of microarchitectural leakage, described as follows. Specifically, for each architecture-level execution of a program, 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 victim 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 victim 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. A software-based attacker can distinguish these two microarchitecture-level execution scenarios by timing the load’s execution latency [25], leaking some function of the address bits involved in the culprit microarchitectural information flow.

Given these ingredients for hardware leakage, 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-imples. If so, the program is susceptible to hardware leakage. LCMs also leverage a program’s speculative semantics to reason about transient leakage [29].

In designing LCMs, we leverage our third insight—that memory consistency models (MCMs) [4, 42] 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 established, 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, 15, 47, 49, 75].

Overall, this paper lays the foundation for formally evaluating program security from high-level language code down to hardware microarchitectures and 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.
- **Leakage detection:** First, we demonstrate that our leakage definition faithfully captures a sampling of (transient and non-transient) microarchitectural attacks from the literature [17, 31, 37, 39, 70, 80]. Second, we develop a static analysis tool, CLOU, based on LCMs which automatically identifies and optimally repairs (via fence insertion) Spectre V1 [39] and Spectre V4 [33] vulnerabilities in programs. We use CLOU to analyze 15 Spectre V1 [38] and 14 Spectre V4 [19] benchmark programs and the libodium crypto-library [23].
- **SUBROSA toolkit:** To support future research, we design the SUBROSA toolkit, built on top of Alloy [35], to mechanize LCMs and support their formalization.

## II. 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. In this section, we discuss how MCMs are specified axiomatically, focusing on the architectural semantics they define for programs which LCMs use out-of-the-box.
A. Defining Memory Consistency Models Axiomatically

MCMs define the value(s) that can be legally returned by shared-memory loads in a parallel program. Since MCMs are central to reasoning about parallel program correctness, a large body of work is devoted to formalizing them [4, 11, 12, 14, 48, 55–57, 59, 61, 64, 73, 74]. In particular, an axiomatically formalized MCM consists of a predicate on candidate executions of programs.

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 (with all branches resolved). Precisely, an event structure \( E \) consists of:

- **MemoryEvent**: a set of all program instructions that read or write memory. MemoryEvent is a subset Event—a set which contains all program instructions.
- **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 Location it accesses.
- **po**: a binary relation that maps each Event to all committed Events that follow it in program order—po is a per-thread total order on committed instructions.

2) Candidate Executions: An event structure for a program can be extended to a set of candidate executions, each of which differs with respect to the shared memory interactions between instructions that are realized. Concretely, we can complete an event structure to produce a candidate execution by adding an execution witness \( X \) to \( E \), which is comprised of three new relations involving same-address MemoryEvents:

- **rf (reads-from)**: a binary relation that maps each Write to all same-address Reads that read from it.
- **co (coherence-order)**: a binary relation that maps each Write to all same-address Writes that follow it in coherence order.
- **fr (from-reads)**: a binary relation that maps each Read to all co-successors of the Write that it read from.

Collectively, rf, co, and fr comprise the \( \text{com} \) relation—\( \text{com} = rf + co + fr \), where + is set union.

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 (allowed) or inconsistent (disallowed) with respect to it. In constructing this predicate, axiomatic MCM specifications often consider a wider range of events (e.g., fences) 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.
- **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).

For MCMs which do not order Reads with po-later MemoryEvents by default (i.e., via po), a dep (dependency) relation is used to selectively enforce these orders. dep encodes syntactic dependencies 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.
- **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.
- **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.

An example consistency predicate defines the Total Store Order (TSO) MCM used by Intel x86 processors [34]. It is composed of the conjunction of three auxiliary predicates—\( \text{sc-per_loc, rmw-atomicity, and causality} \) [4]. Below we define the most relevant two for the ideas presented in this paper:

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

B. An Axiomatic 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 \( \text{com} \) relation (§II-A2) 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.

III. Leakage Containment Models

A. 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 [21, 24, 66]. However, MCMs themselves do not offer a complete security contract solution. To demonstrate why, consider the classic SPECTRE V1 [39] 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 where nodes are MemoryEvents labeled with the Location they access—per the address relation—and edges denote types of “happens-before” [41] (i.e., sequencing) relationships—per relations such as 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—also by convention, no rf edges are explicitly drawn since initialization writes are not explicitly modeled. Second, 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 (§II-A2).

Figs. 1c and 1d thus also constitute candidate executions. Moreover, they constitute consistent candidate executions according to TSO (§II-A3) 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 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 which cannot be discerned from Figs. 1c and 1d. In summary, MCMs cannot directly capture microarchitectural leakage out-of-the-box.

B. An Axiomatic Microarchitectural Semantics

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, ⊤ represents explicitly the set of architectural/microarchitectural writes that initialize relevant architectural/microarchitectural state. ⊥ 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 (⊥) does not share memory with the executing program, and thus it cannot interact with the program architecturally (i.e. via com). ⊥ may only be involved in a com relation with ⊤. However, ⊤ can interact with the program microarchitecturally, such as by probing cache state. Thus, ⊤ can be involved in comx (§III-B2) relations with program instructions. Each straight-line path through po edges from ⊤ to ⊥, together with the com relation, denotes a distinct candidate execution. In Fig. 2a, the com relation has been explicitly drawn—i.e., note the presence of rf edges in contrast to Figs. 1c and 1d which do not model initialization writes. Edges missing a source node are implicitly related to ⊤.

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. Fig. 2a illustrates that xstate elements s₀, s₁, and s₂
are accessed on behalf of \textbf{Read} instructions 2, 5, and 6, respectively. Furthermore, all three \textit{xstate} accesses are \textit{microarchitectural} read-modify-write operations, denoted by “\textit{RW}” before the \textit{xstate} identifier in the figure. In other words, \textit{R} \textit{y} (\textit{RW} \textit{s}_0) \rightarrow \textit{r}_2 means that architectural \textit{Read} event \textit{R}, which accesses architectural \textit{Location} \textit{y}, \textit{induces} a microarchitectural read-modify-write of \textit{xstate} element \textit{s}_0.

The \textit{xstate} identifiers used by LCMs, such as those featured in Fig. 2a, may represent a \textit{set} of hardware state elements in a microarchitecture. Furthermore, an instruction can access a \textit{vector} of \textit{xstate} rather than a single \textit{xstate} element. Crucially, instructions which access common \textit{xstate} elements are capable of \textit{communicating} microarchitecturally—modeled by a new communication relation, \textit{com} (§III-B2). In fact, the sole reason for modeling \textit{xstate} is to establish \textit{com} for a given candidate execution. Instructions may also access different \textit{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 \textit{xstate} accessed on behalf of architectural \textit{memory instructions} only. In particular, the \textit{xstate} elements we model in this paper are intended to capture the ways in which \textit{same-core memory instructions} can communicate \textit{microarchitecturally}—said \textit{xstate} then effectively represents the core-private cache lines and store buffer entries that are accessed on behalf of architectural memory instructions.

To understand what the above \textit{xstate} modeling choice means for \textit{how} memory instructions access these abstract \textit{xstate} elements, consider the following. In general, (cacheable) architectural \textit{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 store buffer, architectural reads \textit{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 store buffer state, stores always behave as microarchitectural writes. Given \textit{xstate} elements which collectively represent the core-private cache line and store buffer entries accessed on behalf of an architectural memory instruction: \textit{read hits} \textit{xstate} (from the cache or from a pending store in the store buffer), \textit{read misses} read-modify-write \textit{xstate} (namely a cache line), and \textit{writes} read-modify-write \textit{xstate} (namely a cache line which subsumes the store buffer write).

2) \textit{Modeling Microarchitectural Information Flow:} Fig. 2a shows that LCMs define a \textit{com} relation which lifts \textit{com} [4] to \textit{xstate} accesses; \textit{com} relates same-address operations, while \textit{com} relates same-\textit{xstate} operations. Recall that LCMs use the \textit{com} relation to encode the architectural information flows that distinguish program executions according to their architectural semantics. Likewise, LCMs use the \textit{com} 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 \textit{com}, a similar \textit{confidentiality predicate} must be defined to rule out illegal instantiations of \textit{com} according to a specific hardware implementation. §V-B discusses features of confidentiality predicates that are required to capture different sorts of known hardware optimizations.

3) \textit{Modeling Microarchitectural Leakage:} LCMs formalize \textit{microarchitectural leakage} by (1) determining which microarchitectural semantics (\textit{com} edges) are implied by a given architectural semantics (\textit{com} edges), and (2) detecting when a program’s microarchitectural semantics deviates from architectural expectation.

For example, consider an \textit{rf} edge which relates a \textit{write} to a \textit{same-core read} that it sources—called \textit{rf}-internal or \textit{rfi} [4]. Consider also our \textit{xstate} of interest which corresponds to core-private processor cache lines or store buffer entries. In the absence of interference, an \textit{rfi} edge which relates some \textit{Write} \textit{w} to some \textit{Read} \textit{r}, implies a consistent \textit{rfx} edge—an \textit{rfx} edge which relates \textit{w} to \textit{r}. In other words, if non-interference holds, a read \textit{r} which \textit{architecturally} reads from a same-core write \textit{w} will further \textit{microarchitecturally} read from the core-private cache line or buffer entry populated by \textit{w}. If \textit{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 \textit{r}'s and \textit{w}'s cache accesses. If \textit{r} forwards data from an interfering store residing in the store buffer rather than reading from \textit{w}, then it exhibits memory address mis-speculation (§III-C) 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 \textit{rf} edges are lacking consistent \textit{rfx} edges. The endpoints of these culprit \textit{com} edges—\textit{com} for both—constitute \textit{receivers} of microarchitectural leakage. In this paper, we define three classes of \textit{transmitters} which can microarchitecturally convey information to a receiver, described as follows.

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

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

Third, \textit{universal data transmitters} (resp. \textit{universal control-}
speculative semantics (b) The microarchitectural semantics indicates that the order on all instructions that are

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

flow transmitters) are data transmitters (resp. control-flow transmitters) whose access instruction is the target of an addr dependency, originating at some Read r'. For example, a chain of the form r' → access → transmit → receiver indicates that the memory location supplied to the access instruction is controlled by the data returned by r'. If an adversary can control the contents of the memory location referenced by r', it can read arbitrary memory [50]. In Fig. 2a, instructions 2, 5, and 6 are address transmitters, 5 and 6 are data transmitters, and 6 is a universal data transmitter.

C. An Axiomatic Speculative Semantics

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 [17, 28, 58], 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 [28, 29]. 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 (⊥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 (5S and 6S) 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 rfx edges, in legal candidate executions. Figs. 4a and 4b in §V give examples of data leakage which results from store forwarding and alias prediction, respectively.

Store forwarding permits reads to forward values from older stores in the reorder buffer (ROB) whose addresses have resolved and whose data is ready. Such a store must be 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 from a store with a potentially mismatching address—i.e., a load may forward data from a store even if its address has not resolved.

IV. THE SUBROSA TOOLKIT

LCM event structures match those of MCMs (§II-A1), while LCM candidate executions (§II-A2) additionally include tfo and comx relations. Moreover, po-derived relations in MCMs, such as dep, are derived from tfo in LCMs.

We mechanize the LCM vocabulary in a toolkit built in Alloy [35], called SUBROSA, which we plan to open-source. SUBROSA is akin to similar MCM frameworks—e.g., the hered simulator that takes as input an axiomatic MCM specification defined in the .cat domain specific language (DSL) [4]. SUBROSA supports the design and formal analysis of custom LCM specifications using our axiomatic vocabulary. This section highlights some features of SUBROSA.

Beyond Memory, Control-Flow, and Fence Events: In this paper, we focus on hardware leakage that results from memory systems optimizations. Thus, our case studies (§V and §VI) consider the same set of Events as MCMs—memory, control flow, and fences. However, since microarchitectural leakage can result from xstate interactions between arbitrary program instructions, SUBROSA supports defining LCMs which feature any Events (i.e., instructions) of the designer’s choosing.
Beyond Architectural State: In both MCMs and LCMs, locations are represented symbolically, as is xstate in LCMs. Namely, it is not relevant for MCM or security analyses what concrete hardware state is accessed on behalf of a given instruction. Instead, it is important that we can identify when a pair of instructions may access the same hardware state—i.e., when a pair of instructions may be related via \text{com} or comx.

Modeling Microarchitectural Data-Flow: To define an LCM for a microarchitecture, rules for constructing legal instantiations of \text{com} must be established. To do this, one must first identify the conflict sets of the microarchitecture in question—sets of instructions that are capable of interacting via a common set of xstate elements. At the finest granularity, conflict sets may be defined by identifying data-flow registers within a microarchitecture—hardware state elements that may be written to by some instructions and read from by others, thereby facilitating microarchitectural data-flow.

For a given data-flow register, the sorts of instructions that may access it during their execution comprise its corresponding conflict set. Recent work shows that it is possible to identify such data-flow registers with the help of formal RTL property verifiers [16]. Further, as shown in this paper, it is possible to merge conflict sets which facilitate the same sort of data-flow.

Modeling Transient Execution: SUBROSA supports custom user-defined speculation primitives \cite{51, 52} which constrain the legal placement of tfo edges in a candidate execution. Instructions defined as speculation primitives are capable of inducing transient execution. Note that if an instruction is transiently executed, SUBROSA assumes that it executes completely with respect to its updates on xstate.

V. DETECTING LEAKAGE IN REAL-WORLD EXAMPLES

In this section, we use §III’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.

A. Formalizing Microarchitectural Leakage

To construct our leakage definition, we define mappings from the building blocks of an LCM's architectural semantics—the rf, co, and fr—to the building blocks of its microarchitectural semantics—the 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 store buffer entry. While we could model these state elements as a pair of xstate locations, we choose to merge them since they both facilitate microarchitectural data-flow between architectural memory instructions. We also limit our mappings below to a single-core execution setting where caches are direct-mapped feature a write-allocate cache policy.

If two writes are ordered by \text{co}, w_0 \xrightarrow{co} w_1, they should be similarly ordered by \text{cox} and frx. This is because two same-address writes behave as microarchitectural read-modify-writes (§III-B1) with respect to the same xstate. Their writes to the store buffer 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 \text{rfx} in the absence of interference—i.e., \text{w}_1 should be microarchitecturally-sourced by \text{w}_0’s cache line (a cache hit). As explained in §III-B3, if a read and write are ordered by \text{rf}, \text{w} \xrightarrow{rf} \text{r}, they should be similarly ordered by \text{rfx} in the absence of interference. If a read and a write are ordered by \text{fr}, \text{r} \xrightarrow{fr} \text{w}, then they should be similarly ordered by \text{frx} \xrightarrow{fr} \text{r} will microarchitecturally read its cache line or store buffer entry before \text{w} microarchitecturally writes. A microarchitectural leak is detected when an architecturally-implied microarchitectural relation is missing from a consistent candidate execution.

B. 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 denote \text{com} relations with \text{comx} inconsistencies—they “point to” (via a directed edge) receivers, which are used to identify transmitters according to the rules in §III-B3. All numerical instruction identifiers refer to candidate execution graphs. Moreover, 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 \cite{39} (Fig. 1a). The program features a speculation primitive—a conditional branch which induces a window of speculation in each event structure instantiated by the branch (i.e., each fork of the graph). Two inconsistent rf edges point towards receivers—both $\perp$ nodes. Instruction 2 is an address transmitter, 5 and 5_s are data transmitters with access instruction 2; 6 and 6_s are universal data transmitters with access instructions 5 and 5_s, respectively. Notably, some transmitters are transient while others are non-transient.

Fig. 3 shows another variant of SPECTRE V1 \cite{29, 80} (code below) featuring the same speculation primitive—a conditional branch—and the same universal data transmitters—6 and 6_s. However, this time the access instruction corresponding to both universal data transmitters (instruction 5) is non-transient.
In Fig 2b, the access instruction corresponding to instruction 6 is non-transient (instruction 5) while the access instruction corresponding to instruction 6\_S is transient (instruction 5\_S).

Notably, STT [80] declared preventing the leakage of non-transiently accessed data as out of scope, although other related work captures leakage of this sort [17, 28].

**SPECTRE V4:** Fig. 4a is representative of SPECTRE V4, described by the code below.

\[
\begin{align*}
1 & \text{ } y = y \& \text{(size}_A - 1); \\
2 & \text{ } x = A[y]; \\
3 & \text{ } \text{temp } &= B[x];
\end{align*}
\]

The speculation primitive is *store forwarding* (§III-C)—instruction 4 reads from a stale same-address write. The \texttt{frx} relation between instructions 4\_S and 3 illustrates this behavior. \texttt{frx} edges can also be understood as *reads-before*. In other words, 4\_S reads from \texttt{xstate} element \texttt{s1} before \texttt{s1} is overwritten by 3. The figure also illustrates that instruction 4 is microarchitecturally sourced from the first read of \texttt{y}, namely instruction 2, via an \texttt{frx} relation. Ultimately, this behavior leads to 6\_S manifesting as a transient universal data transmitter of data transiently accessed by 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 [31, 33]). In particular, formally specifying an LCM for a particular ISA requires defining a *confidentiality predicate* (§III-B2). Consider the consistency predicate for TSO from §II-A3 which is the conjunction of the \texttt{sc_per_loc}, \texttt{rmw_atomicity}, and \texttt{causality} auxiliary predicates. Naively lifting \texttt{sc_per_loc} to constrain \texttt{comx} results in \texttt{sc_per_loc} \texttt{x} = \texttt{acyclic\{frx + comx + frx + tfo_loc\}}, where \texttt{tfo_loc} is defined as \texttt{po_loc} by substituting \texttt{tfo} for \texttt{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 cycles in \texttt{frx + tfo_loc} in its confidentiality predicate.

**SPECTRE-PSF:** Fig. 4b features a variant of SPECTRE V4 [17, 27], coined SPECTRE-PSF [21] (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 reads from an incorrect memory location—illustrated by the \texttt{frx} edge between instructions 2 and 3. This behavior leads to a transient 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 \texttt{frx} edges in LCM candidate executions. Namely, read instructions can mis-predict the \texttt{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 [70]. Fig. 5 features programs that exercise two such optimizations. Fig. 5a features leakage on hardware that implements *silent stores* [44], which avoid explicitly writing to memory when a store’s data operand matches the current contents at its effective address. Here, instruction 2 is an \texttt{xstate} transmitter. Unlike most \texttt{xstate} transmitters in this paper, instruction 2 transmits the data field of its accessed \texttt{xstate} \texttt{s0} 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) [82] (recently patented by Intel [81]). Hardware featuring 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] + \Delta]]). The security implications of this optimization are discussed in recent work [70]. Notably, the authors point out an IMP.
can construct a universal read gadget [50], and Fig. 5b indeed indicates that prefetch instruction 3_p is a universal data transmitter.

VI. CLOU: DETECTING LEAKAGE WITH LCMs

We develop a static analysis tool, CLOU, based on LCMs which automatically identifies and repairs Spectre vulnerabilities in programs. Our approach is inspired by a tool which restores sequentially consistency, via automated fence insertion, for programs running on hardware implementing weak MCMs [1]. CLOU is implemented as a custom IR pass in LLVM [43]. It takes a C source file as input, compiles it to LLVM IR using CLANG v12.0.0, and analyzes each defined 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.

A. Constructing an Abstract CFG (A-CFG)

CLOU first transforms a function’s LLVM IR control-flow graph (CFG) into a loop/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 (§VI-B) that can summarize for all memory accesses in the loop the set of virtual memory locations it 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.

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.

B. 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. 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. Legal assignments to these variables are constrained by a set of first-order logic formulas that describe what constitutes a consistent candidate execution—including consistency and confidentiality predicates. Deriving a concrete candidate execution from an S-AEG may then be achieved by searching for a variable assignment which satisfies said formulas. Other than the assumption the target hardware features write-allocate caches and does not implement silent stores [44], we conservatively leave comx unconstrained; com is constrained by the TSO consistency predicate (§II-A3).

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 §III-B1. Second, we assume a one-to-one correspondence between architectural addresses and modeled xstate locations. This assumption may result in CLOU uncovering leakage that would be impossible due to cache collisions which are realized for a particular cache architecture. Third, given our empirical observations, CLOU also considers a special type of addr, called addr_gep (get-element-pointer address dependency). addr_gep maps a Read to a MemoryEvent, where the Read’s return value is added to a base address to compute MemoryEvent’s effective address. Distinguishing addr_gep from other addr dependencies—which indicate the source instruction supplies a base address—enables CLOU to filter out benign leaks (§VI-E).

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 [43] 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 hold during transient execution. These assumptions do not restrict the set of discoverable transmitters.

C. 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 all adding constraints encoded in the S-AEG to a Z3 SMT solver instance [22]. These constraints define the space of consistent candidate executions of the function under evaluation. The next intuitive step is to directly encode as a constraint the expected behaviors of a leakage-free program (according to §V-A), so that Z3 can search

3Compared 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.

4LLVM IR features a pointer-only arithmetic instruction, called `getelementptr`, which signifies such behavior.
STL01 intends to show that the access to sec_ary[ridx] in line 5 can transiently read stale data before it is overwritten in line 4, rendering the access to sec_ary[ridx] a data transmitter. However, CLOU finds a candidate execution where the same transmitter facilitates universal data leakage—line 5’s access to idx can read stale data before it is overwritten in line 2. CLOU also finds that STL13 is erroneously labeled as “secure” in the benchmark and flagged as secure by the benchmark authors’ formal tool [20]—it features data leakage when a return instruction bypasses a store to the stack.

CLOU detects some false positive leakage when analyzing STL programs due to two sources of imprecision. First, CLOU does not perform semantic analysis of instructions. Thus, it 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. Thus, false positive leakage involving instructions which cannot exist in the processor simultaneously (due to ROB size) may be flagged (e.g., in STL09).

STL03 and STL12 are intended to be safe due to their use of C’s register keyword to prevent storing an array index. We find that CLANG -00 disregards the register keyword and stores the index in memory anyway, enabling it to be bypassed. Thus, we manually repair the LLVM IR output to create the effect of register. Table I’s results incorporate this fix. One other instance of false positive leakage is flagged, but the use of register repairs the intended leakage.

A notable feature of CLOU is its ability to insert a minimal number of fences to repair SPECTRE V1 and SPECTRE V4 leaks. We direct CLOU to perform fence insertion in the PHT and STL benchmarks and confirm that all initially-detected leakage is mitigated with one fence per vulnerable program.

---

### Table I: CLOU’s evaluation of SPECTRE V1 [38] and SPECTRE V4 [19] benchmarks. **Bold** = newly discovered leakage; (*)¹ = false positive due to semantic analysis imprecision; (*)² = false positive due to imprecise loop summarization.

| SPECTRE V1 Benchmarks | Intended Leakage | Detected Leakage | SPECTRE V4 Benchmarks | Intended Leakage | Detected Leakage |
|-----------------------|-----------------|-----------------|-----------------------|-----------------|-----------------|
| PHT01                 | Uₚ                 | Uₚ               | STL01                 | D              | D, Uₚ            |
| PHT02                 | Uₚ                 | Uₚ               | STL02                 | Uₚ             | Uₚ              |
| PHT03                 | Uₚ                 | Uₚ               | STL03 (none)          | (Uₚ)¹          |
| PHT04                 | Uₚ                 | Uₚ               | STL04                 | D              | D, (Uₚ)¹         |
| PHT05                 | Uₚ                 | Uₚ               | STL05                 | Uₚ             | Uₚ              |
| PHT06                 | Uₚ                 | Uₚ               | STL06                 | Uₚ             | Uₚ              |
| PHT07                 | Uₚ                 | Uₚ               | STL07                 | Uₚ             | Uₚ              |
| PHT08                 | Uₚ                 | Uₚ               | STL08                 | Uₚ             | Uₚ              |
| PHT09                 | Uₚ                 | Uₚ               | STL09 (none)          | (Uₚ)¹, (Uₚ)¹² |
| PHT10                 | Uₚ                 | Uₚ               | STL09-obs            | D              | D, (Uₚ)¹         |
| PHT11                 | Uₚ                 | Uₚ               | STL10                 | Uₚ             | Uₚ              |
| PHT12                 | Uₚ                 | Uₚ               | STL11                 | D              | D, (Uₚ)¹         |
| PHT13                 | Uₚ                 | Uₚ               | STL12 (none)          | (Uₚ)¹          |
| PHT14                 | Uₚ                 | Uₚ               | STL13                 | D              | D, (Uₚ)¹         |
| PHT15                 | Uₚ                 | Uₚ               | —                     | —              | —               |

---

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

---
E. Analyzing libsodium with CLOU

To show scalability of CLOU, we use it to search for transient universal data transmitters in the libsodium crypto-library [23]. 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. Performance is summarized in Fig. 6. Furthermore, we sacrifice completeness for performance by limiting CLOU’s search space in two key ways. First, we allow at most one intermediate rf edge between subsequent addr dependencies (§VI-C). Second, we leverage 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 \( W_{\text{size}} \) instructions. In practice, we believe these limitations do not significantly impact CLOU’s ability to detect leakage. In particular, our sliding window approach excludes the discovery of universal data leakage in two key scenarios: (1) a live value sits in a register for a long time (greater than \( W_{\text{size}} \) instructions), a condition that compiler register allocation tries to avoid; or (2) a committed instruction stores an unchecked value (e.g. an array pointer constructed using an attacker-supplied index) to memory and does not read it for a long time.

We first analyze libsodium using CLOU’s SPECTRE v1 engine to search for §III-B3’s universal data transmitter signature. CLOU assumes \( d_{\text{spec}} = 250 \) and \( W_{\text{size}} = 500 \). Here, CLOU reports many benign candidate executions featuring the pattern `uint32_t *idxp; ... = array[*idxp];` where a pointer is loaded from memory and subsequently dereferenced, forming the first addr dependency. While valid, this leakage is low-risk for two reasons. First, in bug-free code, a pointer (e.g. idxp) will rarely be solely controlled by an attacker. Second, if the pointer is indeed not attacker-controlled, the same data will leak every time, so it acts as a non-universal data transmitter. To filter out these low-risk examples, we re-run CLOU’s SPECTRE v1 engine using a modified addr dependency chain (`addr_gep.addr`). Except for one false positive due to imprecise alias analysis, no universal data transmitters are found. Recent work does not find any SPECTRE v1 violations [17] in libsodium. However, they restrict their search using taint annotations which likely confirms our hypothesis that the flagged data transmitters are benign.

Next, we repeat the experiment above using CLOU’s SPECTRE v4 engine, except that results are not filtered according to `addr_gep`. Since programs feature many more store forwarding speculation primitives (loads) than control-flow speculation primitives (branches), there is a larger search space with more complex constraints. Thus, we set CLOU’s \( d_{\text{spec}} = 25 \) with \( W_{\text{size}} = 50 \) to ensure analysis terminates. No universal data transmitters are found, corroborating recent work [17] which analyzes libsodium with \( d_{\text{spec}} = 20 \).

CLOU employs various optimizations to scale to analyzing realistic-sized codebases. We omit a detailed description in our submission due to space constraints. Fig. 6 shows that 93%/92% of 861 functions are analyzed in less than one minute of serial execution time for SPECTRE v1/v4.

libodium defines 943 functions, but CLOU avoids re-analyzing functions which it has already inlined and checked elsewhere.

VII. Related Work

Detecting Transient Leakage: Recent work simulates transient execution so that standard software analysis tools can detect classes of Spectre vulnerabilities in programs [17, 28, 30, 58, 71, 77]. LCMs take a similar approach via the tfo relation. SpecFuzz [58] and Spectector [28] detect SPECTRE v1 gadgets in code using fuzzers and symbolic execution engines, respectively; Pitchfork [17] uses symbolic execution to detect SPECTRE V1/V1.1/V4 violations. LCMs also capture SPECTRE v1.1 leakage, but we omit an example due to space limitations.

Researchers have proposed tools to detect Spectre-style vulnerabilities at the binary [17, 20, 28, 72] and LLVM-IR levels [30, 71, 77]. Nevertheless, all existing tools either scale poorly or face qualitative limitations. For example, Spectector [28] only detects SPECTRE v1 and does not scale well to large codebases. Spectector is also based on the program-counter security model [54] and thus disallows branching on secrets. LCMs support branching on secrets and are not limited to reasoning about vulnerabilities involving transient execution nor are they limited to capturing just a single Spectre variant. Pitchfork [17] detects SPECTRE v1/v4; however, its implementation is unsound [10], and its SPECTRE v4 detection scheme scales poorly.

Formalizing Transient Leakage: Recent research applies formal rigor to reasoning about the impact of transient execution attacks on software [10, 17, 27–29, 58, 60, 69]. Cauligi et al. [17] defines speculative constant-time using an adversarial semantics for speculative execution. Similar to LCMs, their modeling approach captures a variety of transient execution attacks including SPECTRE V1/V4. InSpectr [27] features an operational model to support reasoning about transient execution attacks and countermeasures. Guarneri et al. [29] proposed hardware-software contracts to explicitly expose to software which aspects of microarchitectural state are observable to an adversary as a program executes.

Concurrent work [21] also proposes to derive axiomatic security models from MCMs. Specifically, the authors use
bench/. Rev Gd.pdf US/home/tools/system-design-and-verification/10h

While Blade’s approach is limited to SPECTRE (software-visible) control/data via associated access instructions can be discussed above [§III-B3]. More generally, the architectural and microarchitectural semantics that LCMs define for programs enables the modeling and automatic detection of transmitters. Whether a transmitter leaks xstate or (universal) control/data via associated access instructions can be further deduced with the help of our transmitter taxonomy (§III-B3).

Just as CLOU searches programs for transmitters, the work discussed above [21] presents a tool KAIBYO which searches programs for access instructions capable of reading from a particular secret address. KAIBYO takes minutes (with a 90-minute timeout in some cases) to inspect the same benchmarks that CLOU analyzes in less than as second. Due to its focus on finding access instructions, KAIBO does not support optimal fence insertion.

Finally, Blade [69] uses a static type system to eliminate transient leakage from CT cryptographic code. Blade prohibits speculative 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. However, in contrast to Blade’s conservative type system, LCMs are more accurate and lead to fewer false-positives. Further, while Blade’s approach is limited to SPECTRE v1, LCMs capture different microarchitectural attacks and CLOU’s current implementation supports fence insertion for both SPECTRE v1/v4 without transient types.

VIII. CONCLUDING REMARKS

We propose LCMs as 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, as in §V-B. 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. Ultimately, we envision programmers using LCMs to specify security requirements by labeling data with trust domains. A compiler can then translate a high-level programs to secure assembly code that prevents leakage across domains.

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 [6]). Such an enhancement to the formalism is left for future work.

ACKNOWLEDGEMENTS

We would like to thank John Mitchell and Clark Barrett for their valuable discussions and feedback on this work.

REFERENCES

[1] J. Alglave, D. Kroening, V. Nimal, and D. Poetzl, “Don’t sit on the fence: A static analysis approach to automatic fence insertion,” vol. 39, no. 2, 2017.

[2] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell, “Fences in weak memory models,” 2010. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-14295-6_25

[3] J. Alglave, L. Maranget, S. Sarkar, and P. Sewell, “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] J. Alglave, L. Maranget, and M. Tautschnig, “Herding cats: Modelling, simulation, testing, and data mining for weak memory,” ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 36, no. 2, pp. 7:1–7:74, 2014.

[5] AMD, “Revision guide for AMD family 10h processors,” 2012. [Online]. Available: https://www.amd.com/system/files/techdocs/41322_10H_Rev_Gd.pdf

[6] M. Andryusco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, “On subnormal floating point and abnormal timing,” in 2015 IEEE Symposium on Security and Privacy, 2015.

[7] ARM, “Arm a64 instruction set architecture.” [Online]. Available: https://static.docs.arm.com

[8] ARM, “Cortex-A9 MPPCore, programmer advice notice, read-after-read hazards, ARM reference 761319,” 2011.

[9] ARM Mbed, “Mbed TLS,” https://github.com/armmbed/mbedtls.

[10] G. Barthe, S. Cauligi, B. Grégoire, A. Koutsos, K. Liao, T. Oliveira, S. Priya, T. Rezk, and P. Schwabe, “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.

[11] M. Batty, A. F. Donaldson, and J. Wickerson, “Overhauling SC atoms in C11 and OpenCL,” 43rd Symposium on Principles of Programming Languages (POPL), 2016.

[12] M. Batty, S. Owens, S. Sarkar, P. Sewell, and T. Weber, “Mathematizing C++ concurrency,” 38th Symposium on Principles of Programming Languages (POPL), 2011.

[13] M. Behnia, P. Sahu, R. Paccagnella, J. Yu, Z. N. Zhao, X. Zou, T. Unterluggauer, J. Torrellas, C. Rozas, A. Morrison, F. McKeen, F. Liu, R. Gabor, C. W. Fletcher, A. Basak, and A. Alameldeen, “Speculative interference attacks: Breaking invisible speculation schemes,” in Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2021.

[14] H.-J. Boehm and S. V. Adve, “Foundations of the C++ concurrency memory model,” 29th Conference on Programming Language Design and Implementation (PLDI), 2008.

[15] J. Bornholt and E. Tørlak, “Synthesizing memory models from framework sketches and litmus tests,” 39th Conference on Programming Language Design and Implementation (PLDI), 2017.

[16] Cadence Design Systems, Inc., “Cadence JasperGold formal verification platform,” accessed 12th April 2021. [Online]. Available: https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform.html

[17] S. Cauligi, C. Disselkoen, K. V. Gleissenthal, D. Tüllsen, D. Stefan, T. Rezk, and G. Barthe, “Constant-time foundations for the new spectre era,” in Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, 2020.

[18] K. Cheang, C. Rasmussen, S. Seshia, and P. Subramanyan, “A formal approach to secure speculation,” in 2019 IEEE 32nd Computer Security Foundations Symposium (CSF), 2019.

[19] L.-A. Daniel, “Binsec/haunted benchmark,” 2021, https://github.com/binsec/haunted_bench/
C. Trippel, D. Lustig, and M. Martonosi, “CheckMate: Automated synthesis of hardware exploits and security litmus tests,” 51st International Symposium on Microarchitecture (MICRO), 2018.

C. Trippel, Y. A. Manerkar, D. Lustig, M. Pellauer, and M. Martonosi, “TriCheck: Memory model verification at the trisection of software, hardware, and ISA,” 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2017.

S. van Schaik, A. Milburn, S. Österhund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, “RIDL: Rogue in-flight data load,” S&P, May 2019.

M. Vassena, C. Disselkoen, K. v. Gleissenthall, S. Cauligi, R. G. Kici, R. Jhala, D. Tullsen, and D. Stefan, “Automatically eliminating speculative leaks from cryptographic code with blade,” Proc. ACM Program. Lang., 2021.

J. Vicarte, P. Shome, N. Nayak, C. Trippel, A. Morrison, D. Kohlbrenner, and C. W. Fletcher, “Opening Pandora’s Box: A Systematic Study of New Ways Microarchitecture Can Leak Private Data,” in ISCA ’21.

G. Wang, S. Chattopadhyay, A. K. Biswas, T. Mitra, and A. Roychoudhury, “Kleespectre: Detecting information leakage through speculative cache attacks via symbolic execution.”

G. Wang, S. Chattopadhyay, I. Gotovchits, T. Mitra, and A. Roychoudhury, “oo7: Low-overhead defense against spectre attacks via program analysis.”

A. Waterman and K. Asanović, “The RISC-V instruction set manual, volume I: Unprivileged ISA document, version 20190608-base-ratified,” SiFive Inc. and CS Division, EECS Department, University of California, Berkeley, Tech. Rep., June 2019. [Online]. Available: https://riscv.org/specifications/

J. Wickerson, M. Batty, B. M. Beckmann, and A. F. Donaldson, “Remote-scope promotion: Clarified, rectified, and verified,” 30th International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2015.

J. Wickerson, M. Batty, T. Sorensen, and G. A. Constantinides, “Automatically comparing memory consistency models,” 44th Symposium on Principles of Programming Languages (POPL), 2017.

J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Lauri, P. G. Neumann, R. Norton, and M. Roe, “The cheri capability model: Revisiting risk in an age of risk,” in Proceeding of the 41st Annual International Symposium on Computer Architecture, 2014.

M. Wu and C. Wang, “Abstract interpretation under speculative execution,” in Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation.

W. Xiong and J. Szefer, “Survey of transient execution attacks and their mitigations,” ACM Comput. Surv., vol. 54, no. 3, May 2021.

J. Yu, L. Hsiung, M. E. Hajj, and C. W. Fletcher, “Data oblivious ISA extensions for side channel-resistant and high performance computing,” in 26th Annual Network and Distributed System Security Symposium, NDSS, 2019.

J. Yu, M. Yan, A. Khzyha, A. Morrison, J. Torrellas, and C. W. Fletcher, “Speculative taint tracking (stt): A comprehensive protection for speculatively accessed data,” in Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, 2019.

X. Yu, C. J. Hughes, and N. R. Sathish, “Hardware prefetcher for indirect access patterns,” patentus US 2016/0188476 A1, 2016, US Patent 14/582,348. Filed December 24, 2014. Issued June 30, 2016.

X. Yu, C. J. Hughes, N. Sathish, and S. Devadas, “Imp: Indirect memory prefetcher,” in 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2015.

D. Zagieboylo, G. E. Suh, and A. C. Myers, “Using information flow to design an ISA that controls timing channels,” in 32nd IEEE Computer Security Foundations Symposium, CSF, 2019.