Ordering Interventions for Hardware Security

Viansa Schmulbach

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2023-238
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2023/EECS-2023-238.html

November 24, 2023

Copyright © 2023, by the author(s).
All rights reserved.

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. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Acknowledgement

This project was advised by Professor Sanjit Seshia, Adwait Godbole, and Kevin Cheang, who have all been incredibly helpful.

Ordering Interventions for Hardware Security

Viansa Schmulbach
ansa@berkeley.edu

Abstract—**Hardware execution attacks exploit interactions in the processor microarchitecture. The goal of our research is to use formal verification tools to harden a processor implementation such that certain programs running on the processor are secure against transient execution attacks. In this paper, we formulate the task of hardening as a search problem for a minimal set of ordering constraints.**

I. Introduction and Example

Transient execution attacks, such as Spectre [1] and Meltdown [2], leak secret data from the victim to an adversary through a *side channel* [3]. These attacks exploit microarchitectural optimizations such as out-of-order (OoO) execution, speculation, caching. While relaxing in-order execution constraints improves performance, it also produces a vulnerable attack surface. Conversely, restricting certain microarchitectural behaviours can eliminate vulnerabilities. In this work, we aim to harden a given hardware design by synthesizing additional constraints over executions such that the resulting design securely executes a set of litmus test programs.

```
void victim_function(int x)
  { if (x < arr1.size()) tmp &= arr2[arr1[x]]; }
```

Fig. 1. Spectre v1: Bounds check bypass vulnerability

Spectre v1 (BCB) Vulnerability. We illustrate how microarchitectural (re)orderings are exploited through the Spectre v1 (bounds-check-bypass) [1] vulnerability (Fig. 1). An unprivileged attacker can call `victim_function` with a carefully chosen input `x` such that the first load speculatively (i.e. before the branch commits) accesses secret data from memory. Thus, the address of the second load depends on the secret. The (secret) address of the second load in the cache can then be observed by an attacker using a timing analysis technique, e.g., *Flush+Reload* [4].

This vulnerability leverages the fact that the second load was dispatched (cache interaction) before the branch committed (speculation resolution). A modification that enforces branch commit to happen before the second load is dispatched would mitigate this vulnerability. The scenarios before and after this mitigation are visualized as *microarchitectural happens-before graphs* (μhb-graph) [5], [6] in Fig. 2(a, b) respectively. A node in an hb-graph represents a single *microarchitectural event* and a directed edge between node n_1 and n_2 represents the fact that n_1 "happens-before" n_2 in the execution. Fig. 2(a) allows insecure executions (where the second load dispatch event occurs before the branch commit event). However, the executions from Fig. 2(b) are secure since branch commit is forced before the second load is dispatched.

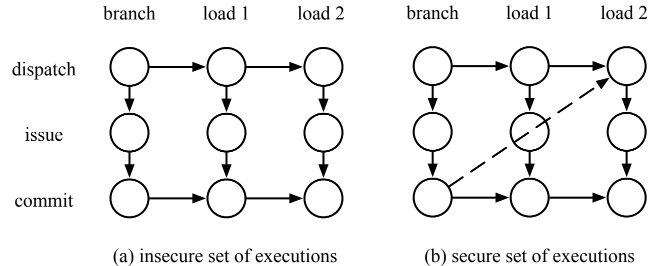

Fig. 2. Happens-before graph representative of the executions of `victim_function` from Fig. 1.

Contribution. We aim to develop a *synthesis-based repair technique* that generates additional ordering constraints such that a set of litmus test programs execute securely on the repaired hardware platform. Additionally, we aim to generate these constraints in a *minimal* way. This contrasts with existing work that repairs software (e.g., secure compilation [7]), manually develops mitigations specialized to certain hardware (e.g., [8], [9]), or performs verification/detection of vulnerabilities (e.g., [10], [11], [12]).

II. Problem Formulation

A. Processor Model

The processor model defines the set of executions that each program can produce. We begin by defining a generic processor model, followed by an example (Ex. 1).

1) Events and ordering constraints: The execution of a program takes place in a set of *stages* S as a set of microarchitectural events. The execution of each instruction takes place in a set of stages, denoted as S. A *microarchitectural event* is associated with each instruction-stage pair. Event-based executions can be viewed as the graph in Fig. 2, where columns are program instructions, rows are stages (from S) and nodes are events. This notion of event-based executions is adapted from the μspec specification language [5] and we refer the reader to prior work (ref. e.g., [6], [13], [14]) for details. The processor enforces a set of constraints, denoted as O, over the order in which the events for a program are executed. We assume that these constraints follow the μspec syntax and semantics [5], [6].

2) Event semantics: While the constraints O define the *order* in which events from the program are executed, we also assume that each stage has certain *functional semantics* associated with it. We have a transition relation T such that $T(s)$ defines the semantics of executing the stage s. A platform has two components, (a) a set of orderings constraints over events O and (b) a set of transition semantics for each stage:

T_s. Consequently, the behaviour of a program executing on the platform is defined by the ordering constraints between stages O, and the semantics for each node T: $M(O,T)$. For a platform M, and a program P executed on this platform, we get a set of executions $Ex(P,M)$. We now provide an example of a processor model that we use in our experiments.

Example 1. *The processor allows the following groups of instructions - ALU instructions, memory instructions, and branch instructions. Each instruction executes in four stages: fetch, dispatch, issue, and commit. The implicit ordering constraints of this model are similar to other OoO processors and are precisely the constraints shown in Fig. 1(a). That is, both dispatch and commit are done in-order in our example design. ALU and branch instructions are marked as complete immediately following dispatch, while load-store instructions are placed in an out-of-order load-store queue. A branch may take arbitrary latency to commit.*

B. Security Property

Our threat model allows an attacker to execute a victim program P and subsequently, observe specific signals from the platform. This threat model is parameterized by the location of victim secrets and the attacker-observable state, and is specified as a non-interference-based security property (e.g., [15], [16]). For simplicity of discussion, we keep the threat model parameters implicit. We denote $NI(P,M)$ to mean that the executions $Ex(P,M)$ satisfy non-interference. This allows us to define what it means for a program to be secure.

Definition 1 (Program security)**.** Let $M(O,S)$ be the platform model. A program P is secure if $NI(P,M(O,S))$ holds.

Problem Statement. Given the input: (a) a processor model $M(O,S)$ (b) a grammar for ordering constraints (c) a set of litmus test programs **P** that are required to be safe (d) a threat-model (parameterized by V_{sec}, V_{obs}) and represented as a NI property we generate minimum set of ordering constraints O' such that all programs $P \in \mathbf{P}$ satisfy $NI(P,M(O \cup O',S))$.

III. Approach

In our problem statement, we refer to a grammar for ordering constraints as an input. Now, we would like to give a concrete example of a grammar defining an ordering constraint OC:

$\langle inst \rangle$ ::= $i_1 \mid i_2 \mid \cdots \mid i_k$ $\qquad$ $\langle OC \rangle$ ::= $\langle CP \rangle \Rightarrow \langle CE \rangle$

$\langle CP \rangle$ ::= $\langle CP \rangle \wedge \langle CP \rangle$ | IsStore($\langle inst \rangle$) | IsLoad($\langle inst \rangle$) | IsBranch($\langle inst \rangle$) | po($\langle inst \rangle$,$\langle inst \rangle$)

$\langle CE \rangle$ ::= $\langle CE \rangle \wedge \langle CE \rangle$ | hb($\langle inst \rangle.\langle S \rangle$, $\langle inst \rangle.\langle S \rangle$)

Each ordering constraint contains a precondition (CP) that implies some execution constraint (CE). The precondition may place restrictions on certain instruction types or enforce program order (po) between two instructions. The execution constraint enforces the "μ happens-before" (hb) relationships between instructions. The instruction nonterminals ($inst$) are the stream of instructions in the order that they are executed in the program, and the nonterminal stages (S) are precisely stages in S as given in the problem statement.

In order to encode hb into a SMT [17] problem, we define timestamps for each microarchitectural event. A timestamp contains the following fields: { `ts: timestamp, done: boolean` }. The processor model maintains a global timestamp that increments at every step. When a microarchitectural event completes, the processor model updates the corresponding timestamp entries with the current time.

Constraint Example. We might write the dotted edge in Fig. 2 as the following constraint:

$$\big(\text{IsBranch}(i_1) \wedge \text{IsLoad}(i_3) \wedge \text{po}(i_1,i_2) \wedge \text{po}(i_2,i_3)\big) \implies \text{hb}(i_1.commit,\, i_3.dispatch)$$

where the clause $hb(i.commit, j.dispatch)$ can be written with our timestamp implementation as follows: $i_1.commit.\texttt{ts} \leq i_3.commit.\texttt{ts}$

The language of OC describes all possible ordering constraints. Now, let us define the minimality of constraints as the following:

Definition 2. (Minimality). Let C_1, C_2 be two sets of ordering constraints. C_1 is at least as minimal as C_2 if $C_2 \implies C_1$.

Finding the minimal set of ordering constraints such that all programs satisfy non-interference when run on the model is a search problem over all sets of ordering constraints. A naive solution would be to iterate over all possible sets of constraints and return the most minimal set found. A more efficient solution is an open area being explored in our research.

IV. Conclusion

A. Preliminary Results

We have implemented and verified the processor in Ex. 1 in UCLID5 [18]. We created a UCLID5 program which ran our model on the litmus test shown in 3. For our threat model, we defined non-interference as the following: our attacker is able to observe, for any address, whether or not that address hits in the cache. Our program correctly synthesized a counterexample trace violating non-interference.

```
regs[x2] = addr    // addr is unconstrained
regs[x2] = mem[regs[x2]]
regs[x2] = mem[regs[x2]]
```

Fig. 3. Litmus Test 1

Additionally, we have implemented and verified our timestamp system, and have been able to enforce ordering constraints in simple programs.

B. Next Steps

Our primary next step is to improve the efficiency and scalability of our model and find an efficient algorithm for searching the space of ordering constraint sets. Ultimately, we would like to create a program which can correctly produce the additional constraint identified in Fig. 2 when given our model and Spectre v1 as inputs. However, the approach can work with numerous litmus tests, such as Spectre v1.1 [19].

REFERENCES

[1] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[2] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. C. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *USENIX Security Symposium*, 2018.

[3] J. Szefer, "Survey of microarchitectural side and covert channels, attacks, and defenses," *Journal of Hardware and Systems Security*, pp. 1–16, 2018.

[4] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 605–622.

[5] D. Lustig, G. Sethi, M. Martonosi, and A. Bhattacharjee, "Coatcheck: Verifying memory ordering at the hardware-os interface," *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.

[6] A. Godbole, Y. A. Manerkar, and S. A. Seshia, "Automated conversion of axiomatic to operational models: Theory and practice," *2022 Formal Methods in Computer-Aided Design (FMCAD)*, pp. 331–342, 2022.

[7] M. Patrignani and D. Garg, "Secure compilation and hyperproperty preservation," *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*, pp. 392–404, 2017.

[8] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher, "Speculative taint tracking (stt): A comprehensive protection for speculatively accessed data," *IEEE Micro*, vol. 40, pp. 81–90, 2019.

[9] O. Weisse, I. Neal, K. Loughlin, T. F. Wenisch, and B. Kasikci, "Nda: Preventing speculative execution attacks at their source," *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019.

[10] 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, pp. 288–28 815.

[11] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez, "Spectector: Principled detection of speculative information flows," in *2020 IEEE Symposium on Security and Privacy (SP)*, 2020, pp. 1–19.

[12] C. Trippel, D. Lustig, and M. Martonosi, "Checkmate: Automated synthesis of hardware exploits and security litmus tests," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 947–960.

[13] Y. A. Manerkar, D. Lustig, M. Martonosi, and M. Pellauer, "Rtlcheck: Verifying the memory consistency of rtl designs," *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 463–476, 2017.

[14] C. Norman, A. Godbole, and Y. A. Manerkar, "Pipesynth: Automated synthesis of microarchitectural axioms for memory consistency," *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2023.

[15] M. R. Clarkson and F. B. Schneider, "Hyperproperties," *2008 21st IEEE Computer Security Foundations Symposium*, pp. 51–65, 2008.

[16] J. A. Goguen and J. Meseguer, "Security policies and security models," *1982 IEEE Symposium on Security and Privacy*, pp. 11–11, 1982.

[17] C. Barrett, P. Fontaine, and C. Tinelli, "The Satisfiability Modulo Theories Library (SMT-LIB)," `www.SMT-LIB.org`, 2016.

[18] E. Polgreen, K. Cheang, P. Gaddamadugu, A. Godbole, K. Laeufer, S. Lin, Y. A. Manerkar, F. Mora, and S. A. Seshia, "Uclid5: Multi-modal formal modeling, verification, and synthesis," in *Computer Aided Verification: 34th International Conference, CAV 2022, Haifa, Israel, August 7–10, 2022, Proceedings, Part I.* Berlin, Heidelberg: Springer-Verlag, 2022, p. 538–551. [Online]. Available: https://doi.org/10.1007/978-3-031-13185-1_27

[19] V. Kiriansky and C. Waldspurger, "Speculative buffer overflows: Attacks and defenses," 2018.