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.
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 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 µhb-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.

![Happens-before graph](image)

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 visualized 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:
the stream of instructions in the order that they are executed between instructions. The instruction nonterminals (inst) are each 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 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:

\( \langle \text{inst} \rangle ::= i_1 \mid i_2 \mid \cdots \mid i_k \)  
\( \langle OC \rangle ::= \langle CP \rangle \Rightarrow \langle CE \rangle \)
\( \langle CP \rangle ::= \langle CP \rangle \land \langle CP \rangle \land \text{IsBranch} (\langle \text{inst} \rangle) \land \text{IsLoad} (\langle \text{inst} \rangle) \mid \text{IsStore} (\langle \text{inst} \rangle) \mid \text{po} (\langle \text{inst} \rangle, \langle \text{inst} \rangle) \)
\( \langle CE \rangle ::= \langle CE \rangle \land \langle CE \rangle \land \text{hb} (\langle \text{inst} \rangle, \langle \text{inst} \rangle, \langle \text{inst} \rangle, \langle \text{inst} \rangle) \)

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:

\( \text{IsBranch}(i_1) \land \text{IsLoad}(i_3) \land \text{po}(i_1, i_2) \land \text{po}(i_2, i_3) \)

\( \Rightarrow \text{hb}(i_1, \text{commit}, i_3, \text{dispatch}) \)

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

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.

![Fig. 3. Litmus Test 1](image)

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


