# A Formal Approach for Detecting Vulnerabilities to Transient Execution Attacks in Out-of-Order Processors

Mohammad Rahmani Fadiheh*, Johannes Müller*, Raik Brinkmann†
Subhasish Mitra‡, Dominik Stoffel*, Wolfgang Kunz*

*TU Kaiserslautern, Germany, †OneSpin Solutions GmbH, Germany, ‡Stanford University, USA,

***Abstract*—Transient execution attacks, such as Spectre and Meltdown, create a new and serious attack surface in modern processors. In spite of all countermeasures taken during recent years, the cycles of alarm and patch are ongoing and call for a better formal understanding of the threat and possible preventions.**

**This paper introduces a formal definition of security with respect to transient execution attacks, formulated as a HW property. We present a formal method for security verification by HW property checking based on extending *Unique Program Execution Checking (UPEC)* to out-of-order processors. UPEC can be used to systematically detect all vulnerabilities to transient execution attacks, including vulnerabilities unknown so far. The feasibility of our approach is demonstrated at the example of the BOOM processor, which is a design with more than 650,000 state bits. In BOOM our approach detects a new, so far unknown vulnerability, called Spectre-STC, indicating that also single-threaded processors can be vulnerable to contention-based Spectre attacks.**

***Index Terms*—Formal verification, HW security, Spectre.**

## I. Introduction

THE emergence of transient execution attacks was a turning point for the security of hardware/software systems. The endless chain of new discoveries of such attacks, starting with Spectre [1] and Meltdown [2], followed by MDS attacks [3], and continuing to date, for example with SMoTherSpectre [4], creates a new and fundamental challenge in security verification for both hardware (HW) and software (SW). Transient execution attacks exploit side effects of *transient instructions*. While these instructions are not part of the correct flow of the program, the processor may still execute them but later discards their result, e.g., after executing instructions due to a mispredicted branch.

The main difference between previously known microarchitectural side channel attacks and transient execution attacks lies in their observability at the ISA level. Previous attacks consist of a program that contains a secret-dependent access pattern to some resource in the microarchitecture that is visible in an ISA-level analysis. In contrast, vulnerabilities to transient execution attacks are not visible based on ISA-level semantics since the fundamental part of the attack, the *transient instructions*, are not part of program execution at the ISA abstraction level.

For this reason, an approach may be considered where ISA semantics is enhanced based on the underlying hardware in order to reason about the security of a program at the presence of transient execution attacks [5], [6], [7]. However, this imposes new security requirements on the software [6], especially on kernel and OS software, which further complicates the SW development process. Instead, it is highly desirable to restore the trust in the executing hardware since this will make it possible for the SW developers to focus in a traditional way only on vulnerabilities which are visible through ISA semantics, relying on the mature family of techniques in symbolic execution, constant-time and data-oblivious programming. Therefore, in this paper, we address the problem at the HW level and formulate security w.r.t. transient execution attacks as a problem of *hardware verification*.

### A. Previous Work

Cheang et al. [6] formally defined *Secure Speculation* as an observational determinism property over four traces. A related approach was proposed by [7], who formalized *Speculative Non-interference* as the requirement for security against speculative execution attacks. The basic idea is to verify at the SW level whether there is any security violation (w.r.t. a defined security policy) that can only occur if the program is executed using the speculative semantics. Therefore, these formulations, which can be seen as a mathematical definition for Spectre attacks, are effective in verifying software. However, secure speculation and speculative non-interference do not provide a generic method for detecting transient execution attacks in hardware since they do not take the micro-architecture into account. Furthermore, they only model speculative attacks based on known channels. Although these methods can be theoretically extended to attacks based on out-of-order execution (e.g., Meltdown or MDS), modeling such attacks at the SW level may not be beneficial, since the attack is carried out using attacker-provided SW code rather than gadgets in the system SW, and thus the attack vector is not available a-priori for such analysis.

An efficient long-term solution to transient execution attacks will require certain modifications in hardware, including cache architectures which prevent speculative side effects [8] and microarchitectural modifications which prevent data propagations through forwarding between speculative instructions [9]. These techniques, however, either block only a certain category of known attacks (e.g., cache-based attacks) or, due to lack of formal guarantees, resort to overly conservative measures, e.g., by preventing any speculative data propagation. HyperFlow [10] is an example of an SoC completely designed using a special, security-driven HDL. However, the security guarantee comes at a high price of performance and memory overhead, and imposes drastic changes in the overall design flow, from HW design to OS development.

Cabodi et al. [11] present one of the first HW security verification approaches targeting speculative attacks which extends over their previous work [12] and the work of [13], pioneering the adoption of *taint analysis* in the HW domain. It is, however, in the nature of taint analysis that it requires assumptions on the paths to be analyzed. As a result, such an approach can be effective in checking a design for known variants of Spectre, but faces its limitations with respect to unknown variants. Furthermore, in contrast to the analysis of [11], which is conducted on an abstract model of the processor, the approach proposed here is applied to its RTL implementation.

### B. Contribution

We propose a formal security verification approach for HW fully covering the class of transient execution attacks. We extend *Unique Program Execution Checking (UPEC)* [14], to scale for large processors with out-of-order and speculative execution features. The proposed approach does not rely on a designer's a-priori knowledge about possible attacks. Given a confidentiality requirement, our method either formally proves the security of the design with respect to transient execution attacks, or automatically produces counterexamples pointing to possible vulnerabilities. The key contributions of this paper are as follows:

- A formal definition for the class of transient execution attacks is proposed (Sec. III). The definition, unlike previous works, considers a concrete microarchitecture and defines a transient execution attack based on the timing discrepancies between architectural and microarchitectural execution. This definition covers not only speculative attacks but also all other attacks based on transient execution such as Meltdown, MDS and Orc [14]. Distinguishing between architectural and microarchitectural state variables in our definition and selecting them iteratively during the proving procedure provides the basis for handling the high complexity of the verification problem. To the best of our knowledge, this is the first formal definition of transient execution attacks meant for property checking of *hardware*.

978-1-7281-1085-1/20/$31.00 ©2020 IEEE

- This paper works towards a structured and systematic formal methodology for HW security verification targeting all transient execution attacks in processors with out-of-order (OOO) execution and speculative execution (Sec. IV). The proposed method provides an exhaustive verification, without demanding a radically different design methodology such as, e.g., by security-driven HDL languages. The proposed approach matches well with current design and verification flows.
- The effectiveness of our method is demonstrated using the *Berkeley Out-Of-Order Machine (BOOM)* [15]. The considered design has more than 650,0000 state bits which clearly demonstrates the scalability of our approach (Sec. V).
- The fact that our approach can detect so far unknown attacks is demonstrated at the example of BOOM. In Sec. II we show the possibility of Spectre-STC, a new variant of the Spectre attack, which has been computed using the proposed approach. We use Spectre-STC also to demonstrate how our method can be used in design iterations to close the detected security gaps (Sec. V).

## II. Spectre-STC: A New Variant of Spectre

The Spectre-STC attack scenario demonstrates that also single-threaded processors can be vulnerable to contention-based Spectre attacks. We describe the attack on the example of the BOOM processor. A hypothetical *gadget* (= exploitable instruction sequence triggered by the attacker and running in privileged mode) is presented to show the feasibility of the attack. (Analyzing other, commercial processors for this vulnerability or searching existing kernel software for relevant gadgets is out of the scope of this paper.)

The attack's transient instruction sequence consists of two main components:

1) **Secret-Dependent Branch:** Assuming a processor allows nested branch speculation, there can be branch instructions in a speculative sequence which depend on the secret value brought to the pipeline through a speculative load. This causes the CPU to perform *different operations* depending on the value of the secret. It should be noted that this is different from the secret-dependent control flow vulnerability in cryptographic software, since a speculative secret-dependent branch may not depend on any sensitive value in the ISA-level flow of the program and dependency on the secret can only be created through speculation.
2) **Port Contention:** The ports to access microarchitectural resources are usually shared between many operations, and port contention can alter the timing of operations. This has security implications in SMT processors, in which two threads run concurrently and share all the resources [4]. In addition, Spectre-STC shows that even single-threaded processors may be vulnerable to leaking information through contention.
   In BOOM, the ALU, the integer multiplier and the integer division unit use the same write port to access the register file. In case of contention, there is a fixed-priority arbitration, in which the ALU has the highest and the division unit the lowest priority, regardless of the program order. This means a division instruction can be delayed by a later issued multiply or add instruction due to port contention.

Fig. 1 shows a possible gadget for the Spectre-STC attack. The basic idea is to start a division before the transient execution, and then, in the transient time window, using a secret-dependent branch, perform multiplications and additions (and create port contention) if the secret is equal to a probing value. The attacker can probe the secret by measuring how long it takes to execute the gadget code. It should be noted that this is not the only possible gadget, but rather the most simple one, used to demonstrate the attack.

### A. Security implications of Spectre-STC

SmotherSpectre [4] has presented a port contention based Spectre attack which is feasible in SMT processors. While the exploitable gadget in SmotherSpectre is similar to Spectre-STC, the root cause

```
1: DIV x1,x2,x3                 // x1 ← x2 ÷ x3
2: BR x1 ≥ 42, Line_m           // if x1 ≥ 42 jump to line m
3: LB x4, addr_of_secret        // x4 ← mem[addr_of_secret]
4: BR x4 ≠ probe_val, Line_n    // if x4 ≠ probe_val jump to line n
5: MUL                          // a mult. instruction with arbitrary operands
6: ADD                          // an add. instruction with arbitrary operands
   ...                          // more such mult.'s and add.'s
n: ...                          // an arbitrary sequence of instructions...
   ...                          // ...other than MUL and ADD
m: ...
```

Fig. 1. Pseudo-code of a Spectre-STC gadget: *addr_of_secret* and *probe_val* are values (typically stored in registers) that can be controlled by attacker-provided inputs. The division instruction takes multiple clock cycles, causing lines 3 to $m$ to execute speculatively (based on wrong prediction). Lines 5 to $n$ will only execute if the secret value is equal to the probing value *probe_val*. Assuming that the execution of sequences between lines 5 to $n$ and $n$ to $m$, will take longer than the DIV in line 1, register file write port contention between DIV and MUL/ADD instructions (and an additional delay) may or may not happen based on the value of the secret (line 4).

and the affected processors are different and, accordingly, Spectre-STC has some new security implications:

1) Spectre-STC is feasible on a single-threaded processor, which further expands the threat of port contention-based attacks. Allocating only mutually trusting threads to a physical core, which was believed to be a solution for the port contention side channel, is shown to be ineffective in a scenario like ours.

2) SmotherSpectre is an inevitable by-product of SMT implementations. However, the Spectre-STC vulnerability can be introduced in case a speculative instruction is prioritized over an earlier non-speculative instruction when resolving resource contention. This can occur not only due to detailed microarchitectural decisions, such as the fixed priority arbitration for write port access in BOOM, but also due to high-level architectural features. In fact, any out-of-order processor featuring a non-pipelined functional unit with multi-clock-cycle operations may be vulnerable to Spectre-STC due to resource contention. This may happen, for example, if the execution order between a non-speculative division and a speculative division located in a secret-dependent branch is reversed.

3) The above scenario may be exploited by a malicious SW provider to create an invisible "backdoor" in the system. With the increasing role of shared HW and SW infrastructures involving components from numerous providers as well as from open-source domains, such a risk must be given appropriate attention.

## III. Transient Execution Attacks

In this section, we propose a mathematical definition for the class of transient execution attacks based on the concept of a "2-safety trace property" [16] and observational determinism [17].

### A. Auxiliary Definitions

**Definition 1** (2-safety trace property)**.** A *2-safety trace property* is a safety property relating two execution traces to each other. □

Witnesses and counterexamples to 2-safety trace properties are *pairs of traces* rather than a single trace.

A program $P$ receives a set of inputs which is considered public information, $X_p$, given as the contents of the register file and data memory (including cache) accessed by $P$. Any other contents of the register file and data memory are considered confidential information, $X_c$. In the microarchitectural implementation of our system, we consider a microarchitectural state $S_0$, which is defined for all state bits other than those holding $X_p$ and $X_c$. $S_0$ is the initial (starting) state for executing a program $P$. We distinguish the following notions:

**Definition 2** (Architectural Observation)**.** The *architectural observation* $O(P, X_p, X_c)$ of a program $P$ is the (time-abstract) sequence of valuations to the program-visible (architectural) registers as they are produced by *committing instructions* according to the ISA specification. □

The architectural observation is independent of any hardware implementation of the ISA.

**Definition 3** (Microarchitectural Execution)**.** The *microarchitectural execution* $\xi(S_0, P, X_p, X_c)$ of a program $P$ is the (clock-cycle-accurate) sequence of valuations to the program-visible (architectural) registers as they are produced during execution of $P$ on a specific hardware implementation/microarchitecture. □

**Definition 4** (Microarchitectural Observation)**.** The *microarchitectural observation* $\mu(S_0, P, X_p, X_c)$ of a program $P$ is the (time-abstract) sequence of valuations to the program-visible (architectural) registers as they are produced by *committing instructions* on a specific hardware implementation/microarchitecture. □

The microarchitectural observation can be obtained from the microarchitectural execution by keeping in the sequence all valuations to the architectural registers at the time points they are written (due to instruction commitment) and discarding all other "intermediate" valuations. For a functionally correct microarchitecture, for any tuple $< P, X_p, X_c >$, the microarchitectural and architectural observation are the same.

### B. Unique Program Execution

*Unique Program Execution* is proposed in [14] as a general security requirement for hardware to provide a root of trust for confidentiality. For the extensions of UPEC developed in this paper, we re-formulate UPEC as a trace property. A program $P$ executes uniquely w.r.t. a set of confidential information $X_c$, if and only if the sequence of valuations to the set of architectural registers is independent of $X_c$, in every clock cycle of program execution. This is equivalent to the following trace property:

$$\begin{aligned} &\forall P, S_0, X_p, X_c, X_c' : \\ &O(P, X_p, X_c) = O(P, X_p, X_c') \\ \Rightarrow\ &\xi(S_0, P, X_p, X_c) = \xi(S_0, P, X_p, X_c') \end{aligned} \tag{1}$$

This requires that if the results of a program do not depend on a secret at the ISA level of abstraction, its execution at the microarchitectural level must not depend on the secret either. This requirement will prevent any discrepancy between hardware and ISA that can violate confidentiality, for a given location of confidential information, e.g., a particular region in memory.

Unique Program Execution covers both *transient execution attacks* and *functional leakage*, i.e., information leakages due to bugs in security-critical functionalities such as memory isolation. A functional leakage is a counterexample not only to the property of Eq. 1 but also to the following, weaker property:

$$\begin{aligned} &\forall P, S_0, X_p, X_c, X_c' : \\ &O(P, X_p, X_c) = O(P, X_p, X_c') \\ \Rightarrow\ &\mu(S_0, P, X_p, X_c) = \mu(S_0, P, X_p, X_c') \end{aligned} \tag{2}$$

### C. Transient Execution Attacks

A *transient execution attack* is a counterexample to the following 2-safety trace property. It results from Eq. (1) by excluding functional leakages from consideration:

$$\begin{aligned} &\forall P, S_0, X_p, X_c, X_c' : \\ &[O(P, X_p, X_c) = O(P, X_p, X_c') \wedge \\ &O(P, X_p, X_c) = \mu(S_0, P, X_p, X_c) \wedge \\ &O(P, X_p, X_c') = \mu(S_0, P, X_p, X_c')] \\ \Rightarrow\ &\xi(S_0, P, X_p, X_c) = \xi(S_0, P, X_p, X_c') \end{aligned} \tag{3}$$

Consider a transient execution attack, given as a counterexample $(S_0, P, X_p)$ to the trace property of Eq. 3, executing an instruction sequence $A$ on the processor. The attack is called a Spectre attack if $\exists A_{victim} \subseteq A$ such that $A_{victim}$ has the proper privilege to access confidential input $X_c$, without an exception being raised. Otherwise, if no such $A_{victim}$ exists, the attack is called a Meltdown attack.

The counterexample to the above trace property is not necessarily the complete attack vector, but it can rather be the critical part of the attack which exfiltrates the secret and creates the covert channel. For example, the part of the original Spectre attack which poisons the branch prediction unit (by bad training) can be excluded from $P$ and be implicitly represented using $S_0$. By using the same $S_0$ in both traces, we exclude classical side channels from consideration, i.e., side channels which are not based on transient executions. Those rely on exploiting a specific victim SW whose footprint would need to be represented by a difference between initial states of the two traces. Note, however, that the trace property still includes the cases in which the attacker relies on a victim SW for bringing the secret to a particular location, e.g., data cache or register file. This is modeled using the same $S_0$ but different $X_c$ in the two traces. The above trace property can be easily adapted to SMT processors by considering the effect of any possible program as the second thread in computing $\mu$.

For the remainder of this paper, it is helpful to observe that any valid counterexample to the property in Eq. 3 fulfills the following condition:

$$\mu(S_0, P, X_p, X_c) = \mu(S_0, P, X_p, X_c') \tag{4}$$

This condition is referred to as *microequivalence* in the following sections.

```
assume:
    at t:              micro_soc_state1 = micro_soc_state2;
    during t..t+k:     secret_data_protected();
prove:
    at t+k:            soc_state1 = soc_state2;
```

Fig. 2. UPEC property formulated on a bounded model

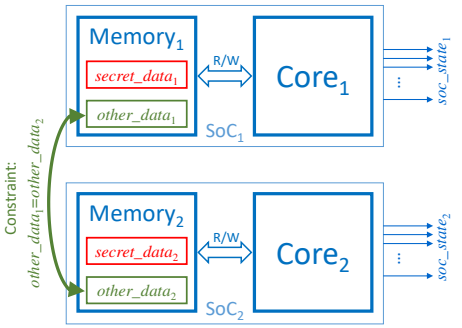


Fig. 3. Computational model for UPEC: The content of the memory is the same, except for the secret data.

## IV. Unique Program Execution Checking (UPEC)

The trace properties described in Sec. III reason about infinite traces, and consequently, they cannot be directly used to verify relevant HW designs of realistic size. In order to make UPEC tractable in practice, the trace properties are translated to a commercial property language and are proven on a bounded model to leverage SAT-based verification methods. As in [14], we obtain an *unbounded* proof by combining the computational model with a symbolic initial state (cf. Sec. IV-A). A key idea of this paper is to enable UPEC for OOO-designs by imposing *microequivalence* on the bounded model to avoid spurious counterexamples, as described in Sec. IV-C. For this purpose, we introduce an abstract model of OOO-processors in Sec. IV-B used to facilitate the specification of microequivalence in a practical methodology.

### A. Background: UPEC on a bounded model

Unique program execution is a 2-safety property, which is why [14] employed a "security miter", as shown in Fig. 3, consisting of two identical instances of the SoC. The miter circuit is built upon the assumption that $X_c$ resides in the data memory. Fig. 2 shows the pseudo-code of the UPEC property to be proven on the bounded miter model. In this property, the assumption *secret_data_protected*() specifies that any load instruction targeting the secret will either throw an exception or be discarded due to an earlier exception or misprediction. As a result, the property only considers programs that produce the same architectural observation independent of the value of the secret (as specified in the trace property in Eq. 1).

*micro_soc_state* is the set of all microarchitectural state variables, defined as all state variables outside memory and cache, and *soc_state* is some set of microarchitectural state variables which includes, as a subset, all architectural (program-visible) state variables. It is enough to only consider architectural state variables in *soc_state* to prove confidentiality if $k$ is large enough to cover the sequential depth of the circuit. However, including microarchitectural state variables in *soc_state* helps to provide a complete proof with shorter time windows and handle complexity in a sound way. A counterexample to the property is one of the following:

- L-alert: A counterexample leading to a state with $soc\_state_1 \neq soc\_state_2$ where the differing state bits are *architectural* state variables.
- P-alert: A counterexample leading to a state with $soc\_state_1 \neq soc\_state_2$ where the differing state bits are microarchitectural state variables that are not architectural state variables.

The property is proven using a symbolic initial state on a bounded model, i.e., *micro_soc_state* at timepoint $t$ is symbolic (any state). As a result, with $k = 1$, it is guaranteed to find the first P-alert or prove its absence. The absence of P-alerts is a sufficient condition for confidentiality. In practice, designs may not fulfill this condition and thus, larger values are chosen for $k$ to find L-alerts or P-alerts with better diagnostics value. If no L-alert is found, P-alerts are evaluated through inductive proofs to verify whether the secret can propagate to an architectural register [14].

### B. General OOO-Processor Model

In this section, an abstract model for OOO-processors is presented, which has its focus on the bookkeeping mechanisms controlling the OOO-execution. This will be used to prune the state space explored in UPEC by removing unreachable sub-spaces violating correctness rules of out-of-order execution.

The abstract OOO-processor model consists of 1) different functional units (FU) and sets of buffers for in-flight instructions, 2) a reorder buffer (ROB) which ensures that the instructions commit in program order and 3) a prediction unit (PU).

Reorder Buffer: The ROB is a circular buffer and every instruction in the pipeline is assigned to a slot in the ROB with a unique ROB ID. The ROB IDs are assigned in program order and instructions commit their results in the order of their ROB IDs. The ROB has two pointers to maintain the order: ROB tail, which points to the next available slot for the newly dispatched instructions, and ROB head which points to the oldest uncommitted instruction. The instruction at ROB head commits when its result is available.

Prediction Unit: A subset of instructions, such as branch, are considered *speculation initiating instructions* (SPI), i.e., they can initiate speculation, marking the next instructions as *speculative*. Each in-flight instruction has a tag ($T$) to distinguish speculative and non-speculative execution.

In order to enable nested speculation, speculation tags must be defined in a hierarchical way which allows multiple valuations for instruction tags with hierarchical relations between them. For simplicity of the following discussion, we model $T$ as an integer.

A speculation tag $T_i$ is *above* tag $T_j$ in the nesting hierarchy, denoted as $T_i < T_j$, if any misprediction which will discard instructions with speculation tag $T_i$, will also discard all instructions with tag $T_j$. Since there can be different tags of in-flight instructions, each SPI, besides having its own tag $T$, generates an additional "spawn" tag $T'$ which becomes the speculation tag of the subsequent instructions. The prediction unit ($PU$) determines the correctness of each prediction. Its outputs include a single-bit flag ($PU.mispred$), which is set in case of a misprediction, the spawn tag ($PU.T'_{spi}$) and the speculation tag ($PU.T_{spi}$) of the corresponding SPI instruction. For each in-flight instruction with tag $T$, the speculation resolution works as follows:

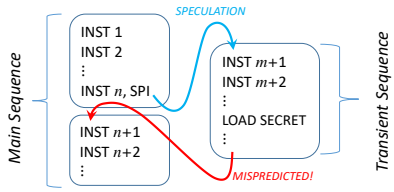


Fig. 4. General model for transient execution attack: microarchitectural flow

$$
\begin{aligned}
&if(PU.mispred)\\
&\quad if \quad (T \geq PU.T'_{spi}) \qquad\qquad discard\ instruction\\
&else\\
&\quad if \quad (T = PU.T'_{spi}) \qquad\qquad T \leftarrow PU.T_{spi}
\end{aligned}
$$

Functional Units: The data path of the OOO-processor consists of several functional units ($FU$), which receive for each instruction they process its ROB ID, $FU.inst\_ID$, and a set of operands, $FU.inst\_operands$, and return the result of the instruction, $FU.inst\_result$.

All other aspects of an OOO-processor such as operation scheduling, register renaming, fetch width, etc., are abstracted away.

### C. Preventing False Counterexamples by Microequivalence

For processors of medium complexity with *in-order* pipelining, the miter construction of Fig. 3 ensures that only little effort is required for manually creating invariants to eliminate spurious counterexamples [14]. This is because the miter enforces equivalence of all components in the two SoC instances except for the secret data and its propagations. However, this changes drastically when out-of-order processors are considered. The symbolic initial state can then include starting states, for example with inconsistent instruction tags or IDs, so that invalid execution orders are considered. As a result, secret-dependent microarchitectural observations can be generated which are spurious counterexamples to the UPEC property.

The key idea to tackle this problem is to constrain the UPEC proof by *microequivalence*. This means that the solver only considers microarchitectural observations that are independent of the secret. Note that this does not restrict the generality of the UPEC proof w.r.t. transient execution attacks, as follows immediately from Eq. 3.

Instead of engaging into a tedious process of modeling all relevant functional behaviors by assertions (invariants), we (conservatively) over-approximate microequivalence in terms of certain data structures used in the bookkeeping mechanisms of OOO-execution, as introduced by the processor model of the previous section.

Fig. 4 shows a general model of how instructions execute in a transient execution attack. Since the program must have a secret-independent architectural observation, there must be no instruction in the *main sequence* that depends on the secret. However, instructions in the *transient sequence* (in a Spectre-style attack these typically belong to a privileged process called by the attacker) can access the secret but cannot commit. In the following, we assume that the transient instructions are discarded due to a misprediction event. For the case that an exception discards the transient instructions the approach is analogous.

For information leakage to happen, the transient sequence must affect the behavior of the main sequence. Consequently, a spurious behavior (due to invalid execution order) can only lead to a false counterexample to the UPEC proof if it creates a false interrelation between the transient and the main sequence (e.g., secret value being forwarded from the transient sequence to the main sequence). Due to the UPEC miter structure, any spurious behavior within each block is irrelevant for the proof. This means that the bookkeeping mechanisms (ROB, PU, FU instruction IDs) must be constrained to ensure the program order only between the three code blocks in Fig. 4, but not necessarily within each block. This observation is key and allows us to approximate microequivalence effectively in terms of tag and ID consistency, and by a partitioning of the ROB into a committable and uncommittable part, as will be elaborated in the following.

In the following, the ROB ID of the highest mispredicted SPI in the hierarchy, $\mathsf{SPI}\ \mathsf{inst}_n$ in Fig. 4, is denoted by *root_ID*. The ROB is partitioned into two sets:

- *Committable set*: ROB slots with an ID that is before the *root_ID*, i.e., ROB IDs which are between ROB head and *root_ID*. Instructions in these slots can commit their results.
- *Uncommittable set*: ROB slots with an ID that is after the *root_ID*, i.e., ROB IDs which are not between ROB head and *root_ID*. Instructions in these slots cannot commit their results. They are invalidated when the misprediction signal is asserted.

Based on the partitioning of the ROB, we describe three sets of assumptions which together create an over-approximation of the microequivalence requirement which does not restrict the generality of the proof. They are denoted in the following by ME-1, ME-2,..., ME-6.

*1) ROB Consistency:* The bookkeeping mechanism must be constrained to prevent spurious behaviors in speculative execution scenarios, in which instructions commit before speculation resolution. The following set of assumptions prevents occurrence of spurious counterexamples in which instructions in the uncommittable ROB slots commit their results.

**ME-1 (Root Instruction Pending)**: (i) The ROB slot with *root_ID* contains an SPI instruction. (ii) The SPI is mispredicted and (iii) it remains valid (pending) until its misprediction is signaled by the prediction unit.

Note that this condition restricts the search of the solvers to instruction sequences containing transient executions which are the root cause of the attacks targeted by UPEC. This bound on the search space contributes significantly to the tractability of UPEC. A condition similar to ME-1 is formulated for the case that transient executions are terminated by an exception, and is omitted here for reasons of space.

**ME-2 (Uncommittable Slots Invalidated)**: In the clock cycle following the misprediction of the SPI with *root_ID*, every valid ROB slot in the uncommittable set is invalidated.

Also ROB tail is critical for maintaining relevant information on program order. A newly fetched instruction is assigned to the ROB tail slot and its speculation tag is determined by the latest branch instruction. In a functionally correct design, if the latest instruction belongs to the uncommittable set, then ROB tail must also point to a slot in the uncommittable set. If this condition is violated in the symbolic initial state, this can create a spurious scenario in which a branch in the transient sequence can control an instruction in the main sequence.

**ME-3 (ROB tail Consistency)**: Until misprediction, ROB tail points to an uncommittable ROB slot.

Note that, for our purposes, it does not matter which of the uncommittable ROB slots exactly ROB tail points to. This strongly simplifies the specification of ME-3.

*2) Functional Unit Consistency:* Due to our assumption that the program must have a secret-independent architectural observation (cf. Sec. III), the two SoC instances in the UPEC miter are not allowed to commit different values. (The only difference can be the time point of commit.) Consequently, the instructions in the committable ROB slots must always produce the same results. We must prevent spurious counterexamples that violate this condition. Instead of specifying the details of a correct data hazard handling and operand forwarding as an invariant, it is sufficient to formulate the following assumption for every functional unit.

**ME-4 (FU Consistency)**:

$$(FU_1.inst\_ID = FU_2.inst\_ID \wedge$$
$$(FU_1.inst\_operands \neq FU_2.inst\_operands \vee$$
$$FU_1.inst\_results \neq FU_2.inst\_results)$$
$$\rightarrow FU_1.inst\_ID \text{ is assigned to uncommittable ROB slot}$$

This is based on the observation that the same instruction on the same FU can only have different operands and/or results if the secret has propagated to this FU. Since we are not targeting functional leakage at the SW level, this can only occur in transient executions and the instruction must be assigned to the uncommittable set.

assume:
  at $t$: $micro\_soc\_state_1 = micro\_soc\_state_2$;
  during $t..t+k$: *secret_data_protected*();
  during $t..t+k$: *microequivalence*();
prove:
  at $t+k$: $soc\_state_1 = soc\_state_2$;

Fig. 5. UPEC property for OOO-processors.

*3) Speculation Consistency:* Secret-dependent SPIs can occur within the transient sequence. Their misprediction must not lead to discarding instructions of the main sequence. If the initial state violates this condition due to tag inconsistency this produces a false counterexample in which one CPU instance commits an instruction while the other one does not.

We address this issue by identifying the tag $T_{main}$ which is the highest tag (by its integer number) of the main sequence, i.e., the highest tag of instructions in the committable set. For $T_{main}$ to be useful in the following, we need to make sure that spurious initial states are excluded from consideration where instructions of the uncommittable set have tags smaller than or equal to $T_{main}$.

**ME-5 (Consistent Speculation Tag)**: Every in-flight instruction with ROB_ID in the uncommittable set must have a speculation tag greater than $T_{main}$.

Based on a $T_{main}$ fulfilling ME-5, we can ensure that counterexamples are generated such that SPIs in the transient sequence only spawn tags which are greater that $T_{main}$.

**ME-6 (Consistent Spawn Tag)**:

$$((PU_1.T_{spi} > T_{main}) \rightarrow PU_1.T'_{spi} > T_{main}) \wedge$$
$$(PU_2.T_{spi} > T_{main})) \rightarrow PU_2.T'_{spi} > T_{main})))$$

Implementing these conditions in a property language is straightforward and only involves identifying the buffers containing tags and IDs of in-flight instructions.

### D. UPEC Proof for OOO-Processor

ME-1 to ME-6 together form the assumption called *microequivalence*() in the UPEC property for OOO-processors, as shown in Fig. 5. The conditions formulated in *microequivalence*() constrain the search to programs with transient execution. The only general assumption we make for our approach is that transient executions are either terminated by misspeculation of any kind, or by exception. We are not aware of any OOO-architectures violating this assumption. In all other respects ME-1 to ME-6 are obviously fulfilled by any correct OOO-architecture. Their sufficiency for removing spurious counterexamples from the UPEC proof procedures has been made plausible in the previous section and is confirmed experimentally in the next section.

The developed formulation of microequivalence is a template that can be refined to any concrete implementation of an OOO-processor, as long as its architecture ensures in-order commit using a reorder buffer and complies with the general model of Sec. IV-B. This can be expected for the large majority of OOO-processors.

## V. Experiments

The feasibility of the proposed UPEC approach for OOO-processors is demonstrated by verifying a design version of the Berkeley Out-Of-Order Machine (BOOM) [15]. All results were obtained using the commercial property checker OneSpin DV 360 running on an Intel Core i7 with 32 GB of RAM at 3.4 GHz.

BOOM is a full-grown SoC with an OOO-core that features a branch prediction unit with support for nested branches, virtual memory translation with TLB, a non-blocking data cache with miss status handling registers (MSHR), page table walker, physical register file with dynamic mapping to logical registers, etc. The overall SoC (a single core and peripherals) consists of more than 650 k state bits.

As a demonstration that we can cover different classes of attacks, in our experiments, we decomposed the proofs into checks for Meltdown

TABLE I
COMPUTATIONAL EFFORT FOR THE PROOF PROCEDURES

| Vulnerability | Proof status | CPU | Mem. | $k$ |
|---|---|---|---|---|
| Spectre-STC v1 | detected (orig. design) | 2 min | 9 GB | 5 |
| Spectre-STC v2 | detected (on patch #1) | 16 min | 14 GB | 7 |
| Spectre (cache-based) | detected (on patch #2) | 14 min | 14 GB | 7 |
| Secure design variant | verified (on patch #3) | 2 min | 6 GB | 1 |
| Meltdown | verified (orig. design) | 1 min | 6 GB | 1 |

TABLE II
REQUIRED MANUAL EFFORT TO DEVELOP AND PROVE UPEC PROPERTY

| Task | Manual effort |
|---|---|
| Microequivalence | 4 person-weeks |
| Invariants | 4 person-days |
| Property and miter | 2 person-hours |

versus Spectre, i.e., we assume that the instruction accessing the secret either has the proper privilege to do so (Spectre class) or not (Meltdown class). Furthermore, the secret is assumed to reside in the main memory, with the possibility of a copy in the data cache. Tab. I shows the computational effort for the proof procedures in different patch cycles, as described below. Tab. II shows the amount of manual work required to develop and prove the UPEC property for BOOM.

For the class of Spectre attacks, our approach generated counterexamples to the UPEC property of Sec. III in terms of L-alerts (cf. Sec. IV-A) demonstrating that the considered BOOM design is vulnerable to a so far unknown variant of Spectre, which we describe in Sec. II. We employed an iterative design procedure where we iteratively patch the vulnerability detected in a selected UPEC counterexample, and then re-verify the design using UPEC. After fixing the Spectre-STC vulnerability through a minor fix in the first iteration, UPEC identified (by L-alert) a second version of the Spectre-STC vulnerability. This version is similar to the first one except that port contention happens on the TLB rather than on the write port to the register file. The patch-and-verify flow can be repeated until all the vulnerabilities have been removed. In the third iteration of our flow, we picked a UPEC counterexample pointing to the original Spectre attack, which uses the cache as side channel [1]. We applied a simple, conservative patch for this vulnerability. Our fix for Spectre prevents load instructions to execute before they have reached ROB head. This fix incurs performance penalties, however, serves well as a proof of concept for our verification methodology. The patched design coming out of this third iteration ("secure design variant" in Tab. I) was then formally verified by absence of UPEC P-alerts (cf. Sec. IV-A) to be secure with respect to transient execution attacks.

In a separate experiment, we also examined the original design for the class of Meltdown attacks, as defined in Sec. III. The computational effort for this experiment, which proves security (by absence of P-alerts) of the original BOOM w.r.t. Meltdown, is also listed in Tab. I. The considered BOOM version implements a conservative measure against Meltdown, by performing address translation and cache access in sequence rather than in parallel. Thus, the core never accesses the data cache for an invalid address.

Based on these experiments the following observations can be made:

*Observation 1:* The required manual work is small compared to the effort required for design and verification of an OOO-processor. The UPEC property only relates to a small subset of functionalities in the design, i.e., the user does not need a detailed understanding of the implementation. The run times for the proof procedure show that restricting the property to microequivalence helps the solver to simplify the proof and makes the method feasible for large designs.

*Observation 2:* UPEC is capable of capturing vulnerabilities which are based on so far unknown exfiltration channels. This is a significant improvement over the state of the art in techniques for finding Spectre vulnerabilities. UPEC provides well-defined pointers to the security flaws in hardware, which promises to be a good starting point for a systematic approach to fix transient execution vulnerabilities and avoid conservative solutions such as the ones used in BOOM to prevent Meltdown or the full protection approach pursued in [9].

*Observation 3:* The proposed approach, due to its exhaustiveness, has promise for an efficient solution regarding legacy HW. By analyzing the RTL design of legacy systems, one can collect all possible attack scenarios feasible in the design. This information can be passed on to the SW domain, to enforce SW fixes only for the necessary cases and relevant gadgets.

## VI. CONCLUSION

In this paper we have shown how to extend UPEC for OOO-processors. We formulated the notion of microequivalence as an invariant for UPEC and showed how it can be approximated in the state space based on elements of OOO bookkeeping mechanisms commonly used in such processors. Based on these notions, UPEC has been shown tractable in BOOM.

Future work will address a more refined patch cycle and derive fine-tuned fixes directly from the generated counterexamples. This is expected to cause less performance degradation than currently practiced solutions incur due to excessive conservatism.

## REFERENCES


[1] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *arXiv preprint arXiv:1801.01203*, 2018.

[2] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *arXiv preprint arXiv:1801.01207*, 2018.

[3] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "RIDL: Rogue in-flight data load," in *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[4] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "SMoTherSpectre: exploiting speculative execution through port contention," *arXiv preprint arXiv:1903.01843*, 2019.

[5] S. Cauligi, C. Disselkoen, K. v. Gleissenthall, D. Stefan, T. Rezk, and G. Barthe, "Towards constant-time foundations for the new spectre era," *arXiv preprint arXiv:1910.01755*, 2019.

[6] K. Cheang, C. Rasmussen, S. Seshia, and P. Subramanyan, "A formal approach to secure speculation," in *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*. IEEE, 2019, pp. 288–304.

[7] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez, "SPECTECTOR: Principled detection of speculative information flows," *arXiv preprint arXiv:1812.08639*, 2018.

[8] G. Saileshwar and M. K. Qureshi, "Cleanupspec: An undo approach to safe speculation," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2019, pp. 73–86.

[9] O. Weisse, I. Neal, K. Loughlin, T. F. Wenisch, and B. Kasikci, "NDA: Preventing speculative execution attacks at their source," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2019, pp. 572–586.

[10] A. Ferraiuolo, M. Zhao, A. C. Myers, and G. E. Suh, "Hyperflow: A processor architecture for nonmalleable, timing-safe information flow security," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2018, pp. 1583–1600.

[11] G. Cabodi, P. Camurati, F. Finocchiaro, and D. Vendraminetto, "Model checking speculation-dependent security properties: Abstracting and reducing processor models for sound and complete verification," in *International Conference on Codes, Cryptology, and Information Security*. Springer, 2019, pp. 462–479.

[12] G. Cabodi, P. Camurati, S. F. Finocchiaro, F. Savarese, and D. Vendraminetto, "Embedded systems secure path verification at the HW/SW interface," *IEEE Design & Test*, vol. 34, no. 5, pp. 38–46, 2017.

[13] P. Subramanyan and D. Arora, "Formal verification of taint-propagation security properties in a commercial SoC design," in *Design, Automation & Test in Europe Conference (DATE)*. IEEE, 2014, pp. 313–314.

[14] M. R. Fadiheh, D. Stoffel, C. Barrett, S. Mitra, and W. Kunz, "Processor hardware security vulnerabilities and their detection by unique program execution checking," in *Design, Automation & Test in Europe Conference (DATE)*, 2019, pp. 994–999.

[15] C. Celio, P.-F. Chiu, B. Nikolic, D. A. Patterson, and K. Asanovic, "BOOMv2: an open-source out-of-order RISC-V core," in *First Workshop on Computer Architecture Research with RISC-V (CARRV)*, 2017.

[16] M. R. Clarkson and F. B. Schneider, "Hyperproperties," *Journal of Computer Security*, vol. 18, no. 6, pp. 1157–1210, 2010.

[17] A. W. Roscoe, "CSP and determinism in security modelling," in *Proc. IEEE Symposium on Security and Privacy*. IEEE, 1995, pp. 114–127.