Rheinland-Pfälzische
Technische Universität
Kaiserslautern
Landau

Implementing and Analyzing Speculative Privacy Tracking in the RISC-V Out-of-Order Core BOOM

Implementierung und Analyse von *Speculative Privacy Tracking* im RISC-V Out-of-Order Prozessor BOOM

Masterthesis

Author: Simón Blanco Ortiz

Matr.-No.: 430128

Date: February 5, 2026

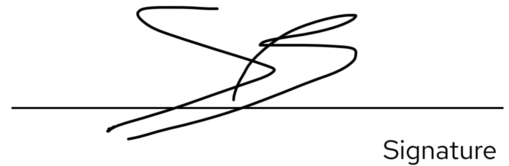
Chair of Electronic Design Automation

Examiner: Prof. Dr.-Ing. Wolfgang Kunz
Advisors: M. Sc. Tobias Jauch
M. Sc. Philipp Schmitz
M. Sc. Alex Wezel

Declaration of Authorship

I hereby declare that I have completed this thesis independently in accordance with the examination regulations without the assistance of third parties except for the support provided by my supervisor. I declare that I have cited all sources and materials used completely and accurately, and that I have marked everything that has been taken from other people's work unchanged, abbreviated or analogously.

Kaiserslautern, February 5, 2026



Signature

Abstract

Speculative execution is a key contributor to the performance of modern out-of-order processors, but it has been shown to enable transient execution attacks that leak sensitive data through microarchitectural side channels. While software-based mitigations exist, they incur significant performance overheads and provide incomplete protection. Hardware-based defenses offer a more transparent and robust approach, but remain challenging to implement and verify at the register-transfer level.

This thesis presents SPT-BOOM, the first register-transfer level (RTL) implementation of Speculative Privacy Tracking (SPT) in an industry-grade superscalar out-of-order processor. SPT is software-transparent protection mechanism against transient execution leakage, which is integrated into BOOMv3, an open-source RISC-V core, and introduces several design adaptations to accommodate architectural constraints and support formal verification.

SPT-BOOM is evaluated in terms of performance, hardware overhead, and security. FPGA-based evaluation using SPEC CPU2006 benchmarks shows an average performance slowdown of 72 % compared to baseline BOOM, exceeding the overhead reported in prior simulation-based studies and highlighting the cost of RTL implementation choices. Hardware synthesis results for the AMD Virtex UltraScale+ VCU118 FPGA indicate moderate area overhead of 7.5 % look-up-tables increase and 3.5 % flip-flop increase. Security analysis establishes a formal verification framework using the Unique Program Execution Checking (UPEC) methodology, providing a foundation for exhaustive verification of SPT against transient execution attacks.

Kurzfassung

Die spekulative Ausführung von Befehlen ist ein wesentlicher Faktor für die hohe Leistungsfähigkeit moderner Out-of-Order-Prozessoren. Es zeigte sich allerdings, dass diese Performance-Verbesserungen sogenannte "Transient Execution Attacken" ermöglichten, bei denen geschützte Daten über Seitenkanäle in der Mikroarchitektur selbst sichtbar werden. Zwar gibt es softwarebasierte Gegenmaßnahmen, diese sorgen jedoch für erhebliche Leistungseinbußen und bieten nicht unbedingt einen vollständigen Schutz. Hardwarebasierte Gegenmaßnahmen stellen einen transparenteren und robusteren Ansatz dar, sind jedoch auf der Registertransferebene nach wie vor schwierig zu implementieren und zu überprüfen.

Diese Masterarbeit präsentiert SPT-BOOM, die erste Implementierung von Speculative Privacy Tracking (SPT) auf der Registertransferebene in einem komplexen, superskalaren Out-of-Order-Prozessor. SPT ist ein Schutzmechanismus gegen Transient Execution Attacken, der im Rahmen der vorliegenden Arbeit in den Open-Source RISC-V Prozessor BOOMv3 integriert wird und mehrere Designanpassungen unternimmt, um die Sicherheitsmechanismen zu implementieren und eine formale Verifizierung zu unterstützen.

Anschließend wird SPT-BOOM hinsichtlich seiner Performanz, seines Hardwareoverheads und seiner Sicherheit bewertet. Die FPGA-basierte Auswertung unter Verwendung der SPEC CPU2006 Benchmarks zeigt eine durchschnittliche Leistungsminderung von 72 % im Vergleich zur unsicheren Ausgangsversion des Prozessors. Dies übersteigt die in früheren Studien gemeldeten, simulationsbasierten Overheads und verdeutlicht die Kosten der Entwurfsentscheidungen in der Implementierung auf Registertransferebene. Die Ergebnisse der Hardware-Synthese zeigen einen moderaten Area-Overhead von 7,5 % (*Look-up Tables*) und 3,5 % (*Flip-Flops*). Die Sicherheitsanalyse im Hinsicht auf Vertraulichkeit etabliert einen Rehmen zu formalen Verifikation auf Basis von Unique Program Execution Checking (UPEC) und bildet die Grundlage für eine umfassende Verifikation von SPT gegen Transient Execution Seitenkanäle.

Contents

1 Introduction **1**

2 Background **2**
2.1 CPU microarchitecture basics . . . 2
2.2 Transient Execution Attacks . . . 3
2.3 BOOM . . . 5
2.4 Speculative Privacy Tracking (SPT) . . . 5
2.5 Formal Verification . . . 6
2.5.1 Bounded Model Checking . . . 7
2.5.2 Interval Property Checking . . . 7
2.6 Unique Program Execution Check (UPEC) . . . 7

3 Related Work **10**
3.1 Speculative Taint Tracking (STT) . . . 10
3.2 SecureBOOM . . . 10

4 Design & Implementation **12**
4.1 Design Goals . . . 12
4.2 High-Level Architecture . . . 12
4.3 Protection Policy . . . 13
4.4 Tainting . . . 14
4.4.1 Special Case: Conditionally Deterministic Instructions . . . 15
4.5 Untainting . . . 15
4.5.1 Transmitter becomes non-speculative . . . 16
4.5.2 Internal Propagation . . . 16
4.5.3 External Propagation: Untaint Broadcast Bus . . . 18
4.5.4 Store-to-Load Forwarding Propagation . . . 19
4.6 Deviations from SPT . . . 20

5 Evaluation **22**
5.1 Experimental setup . . . 22
5.2 Performance Evaluation . . . 22
5.3 Hardware Overhead . . . 24

6 Security Analysis **26**
6.1 Threat Model . . . 26
6.2 Impact of Deviations . . . 26
6.3 UPEC for SPT-BOOM . . . 27

6.3.1 Setup 27
6.3.2 Property structure 27
6.3.3 SPT invariants 29

7 Future Work **31**

8 Conclusion **32**

A Tooling: *signals2dve* **33**
A.1 Example 33

1 Introduction

Modern superscalar processors rely heavily on speculative execution to achieve high performance [1]. Instructions are routinely executed before control-flow decisions, memory dependencies, or privilege checks are fully resolved, allowing the processor to keep its pipelines busy. While this behavior is architecturally invisible, it complicates reasoning about microarchitectural behavior and its security implications.

The discovery of transient execution attacks demonstrated that speculative execution can influence observable microarchitectural state even when architectural effects are eventually rolled back [2]. As a result, speculation can no longer be treated solely as a performance optimization, but must be constrained by explicit security considerations.

A wide range of mitigations has been proposed in response [3, 4, 5, 6]. Software-based defenses have seen broad deployment, but they are typically reactive, attack-specific, and costly in terms of performance. More fundamentally, they attempt to compensate for insecure microarchitectural behavior rather than preventing it, which motivates the development of hardware mechanisms that enforce security properties directly within the processor pipeline. Speculative Privacy Tracking (SPT) [7] is one such hardware-based approach. Prior work has evaluated SPT at the architectural level, but its feasibility as a register-transfer-level design in a realistic out-of-order processor, and its suitability to rigorous formal verification, have not previously been demonstrated.

This thesis addresses that gap by implementing Speculative Privacy Tracking in BOOM [8], an open-source superscalar out-of-order RISC-V processor. The resulting design, referred to as SPT-BOOM, integrates SPT into BOOM's microarchitecture and introduces several deviations from the original proposal to accommodate architectural constraints and support verification.

Beyond implementation, this work places particular emphasis on formally verifying SPT's security guarantees. A verification framework based on Unique Program Execution Checking [9] is developed to analyze transient execution side channel leakage at the register-transfer level, with the design structured to support interval property checking and systematic identification of potential vulnerabilities.

In summary, the contributions of this thesis are:

- An RTL implementation of Speculative Privacy Tracking integrated into a realistic superscalar out-of-order processor.
- A detailed analysis of the architectural adaptations and deviations required to realize SPT at the register-transfer level.
- An empirical evaluation of performance and hardware overhead using FPGA synthesis and standard benchmarks.
- A formal security analysis framework based on UPEC, enabling exhaustive detection of transient execution leakage.

2 Background

2.1 CPU microarchitecture basics

Modern processors employ advanced techniques to improve performance [1]. The most ubiquitous is pipelining. A pipelined processor divides the total execution of an instruction in steps or stages, such as fetching and decoding the instruction, execution in the respective functional unit, memory access and writeback of the results to the register file. This way, at any given cycle, each stage holds and processes a different instruction. Although a single instruction will require at least the same execution time from start to finish in a pipelined processor as compared to a purely sequential one, this changes as soon as we consider the execution of a sequence of instructions (i.e., a program). Once the pipeline is fully occupied, the processor can execute up to one instruction per clock cycle, assuming every main stage takes only one clock cycle to finish its task. In other words, a pipelined processor maintains the instruction latency while increasing the throughput (i.e., the number of instructions executed per clock cycle) of its sequential equivalent, assuming the same clock frequency.

Superscalar processors increase throughput by dispatching and executing multiple instructions in parallel. This is achieved by increasing the instruction width of each pipeline stage, so multiple instructions can flow in parallel through the pipeline.

However, these optimizations only improve the maximum throughput, and their benefits can be completely outweighed in the case of frequent pipeline stalls. The instructions in a program often depend on each other, and the dependencies between stages in the pipeline become increasingly complex. In some situations the pipeline may halt, waiting for a dependency to be solved. Events that can stall the pipeline are denominated hazards, and there are 3 types:

- Data hazards are caused when the execution of two depending instructions overlap. There are three main scenarios in which this can happen: RAW (read after write), WAW (write after write) and WAR (write after read).
- Structural hazards are caused by two (or more) instructions needing the same resource. One of them is forced to wait for the resource to become available.
- Control hazards are caused by instructions that are waiting for condition resolution of a previous control-flow instruction.

Register renaming can completely eliminate WAR and RAW data hazards, as they are simply a type of name dependence at the Instruction Set Architecture (ISA) level. Dynamically mapping the architectural registers with the physical registers in the microarchitecture removes this type of dependency. On the other hand, RAW hazards are a true data dependency. They can be mitigated with *forwarding*, which creates a shortcut path

to make results of instructions available to dependent ones without the latency of waiting for them to reach the actual register file at the writeback stage.

Structural hazards can be mitigated by adding more instances of the most commonly blocked resources, but obviously this does not scale well on hardware. Out-of-order execution helps mitigate this problem, by allowing new instructions to execute as long as they do not depend on older unexecuted instructions and their required resources are available. It is crucial that afterwards, instructions are retired (i.e., committed) in-order, as program integrity depends on it.

Control hazards cannot be eliminated completely either, but they can be mitigated by predicting the outcome of conditional branches. Executing the next instructions based on the prediction before resolving the condition is called speculation, because its execution is not yet guaranteed. In case of an incorrect prediction, also known as *misspeculation*, the pipeline has to be flushed and the operations undone, causing a great overhead and performance penalty, compared to other hazards.

2.2 Transient Execution Attacks

As mentioned in the previous section, speculative processors can mistakenly execute some instructions that would not have been executed when carrying out the program in-order, one instruction at a time. A common example is a branch that was predicted *taken* but it resolved as *not taken*. A functionally correct processor must not make any result of this misspeculated instructions visible to the outside (i.e., to the *architectural state*), since those instructions were never part of the real control flow of the program. At the architectural level, the user should not be able to tell that speculation happened at all.

However, this changes when we look at the microarchitectural level, where instructions that execute speculatively will necessarily produce some temporary effect. If these instructions are later squashed, either because the branch prediction was incorrect or because the instruction triggers a fault, then their execution is classified as transient. When certain instructions are executed transiently, they can create side channels and leak secret data that was not supposed to be visible in this context. Such instructions are referred to as *transmitters*, and they are exploited in transient execution attacks, such as Meltdown [10] and Spectre [3].

Meltdown and related variants exploit the transient execution that occurs after a faulting instruction, before the processor raises the exception, usually at commit time. During this transient window, the value obtained by the faulting instruction can influence subsequent transmitter instructions, which encode the secret into a microarchitectural side channel, most commonly the cache. An instruction that triggers an exception, such as an illegal or privileged memory access, may still be executed transiently before the fault is architecturally handled. Meltdown variants differ from each other based on the fault condition and the microarchitectural buffer that causes the leak [2], but all follow the same principle. These attacks are conducted by actively exploiting the processor's out-

of-order execution, structuring the instruction stream such that the younger unprivileged memory access and transmitter instructions are scheduled and executed transiently while a faulting instruction is pending retirement. The transient window can be extended by deliberately delaying retirement by, for example, introducing long-latency instruction chains ahead of the faulting instruction, thereby providing sufficient time for the secret to be transmitted before the exception is raised. Although the faulting load never commits, its execution leaves persistent microarchitectural footprints (e.g., in the cache state), which can be exploited via timing-based side channels. However, out-of-order execution is not necessarily a requirement for this type of vulnerability. The Orc Attack [11] can even affect in-order processors, exploiting the interface between the core and the cache.

On the other side, Spectre and its variants exploit transient execution caused by control or data flow mispredictions. By carefully training the predictor and preparing the speculative path, an attacker can trick the processor into transiently accessing sensitive data in memory. After the misspeculated instructions are squashed once the wrong prediction is resolved, the microarchitectural footprint is already in the cache, and can be easily extracted using time-based side channels. Modern processors, while functionally correct on the ISA level, are still vulnerable to these type of attacks. Some variants have been patched, but new transient execution vulnerabilities keep being discovered in 2025 [12, 13, 14, 15]. In fact, one could argue that this is the price of modern CPU performance, and that conservative countermeasures such as delaying the execution of transmitters until they are no longer speculative, as exemplified by NDA [16], impose unacceptably high overhead by current commercial standards.

Since the discovery of these vulnerabilities in early 2018, there have been several patches issued at the software level to mitigate known exploits, but these are designed for specific scenarios and systems, applying conservative measures while leaving other cases uncovered and resulting in suboptimal performance compared to its equivalent implementation on hardware. Examples include *retpoline* sequences and speculation barriers to restrict indirect branch speculation in *Spectre-V2* [17], Kernel Page Table Isolation (KPTI) [6] to separate user and kernel mappings against Meltdown, and compiler-inserted bounds masking or speculation-control hardening to reduce *Spectre-V1* exposure [2]. Another solution for new designs could be to modify the software and ISA so the program can explicitly define what data is sensitive and actively protected while allowing the leakage of non-sensitive data, but this approach would require rewriting legacy software [18, 19]. For the above reasons, hardware emerges as the appropriate domain to tackle these vulnerabilities in a transparent fashion for the software and ISA, maintaining backward compatibility and minimal performance penalty, while establishing itself as the root-of-trust for the rest of the system.

2.3 BOOM

The Berkeley Out-of-Order Machine (BOOM) is an open-source RISC-V core developed by the Berkeley Architecture Research Group at the University of California [20]. It is a high-performance, synthesizable, and parameterizable core, primarily intended for computer architecture research. BOOM is implemented in Chisel [21], a hardware construction language embedded in Scala that enables the use of modern software abstractions to describe and generate parameterized RTL designs. At the time of writing, the latest release is BOOMv3, also known as SonicBOOM [8]. This version is used in the present work and, for the remainder of this thesis, will be referred to simply as "BOOM" or "baseline BOOM".

BOOM's microarchitecture is implemented as an out-of-order, superscalar pipeline organized in 7 stages: Fetch, Decode/Rename, Rename/Dispatch, Issue/RegisterRead, Execute, Memory and Writeback. The frontend is responsible for instruction fetch, branch prediction and instruction decoding, and it supplies the decoded instructions to the backend through the fetch buffer. The backend consists of the rename stage, the issue units, multiple execution units and a reorder buffer (ROB) to enforce in-order retirement of instructions. BOOM is also fully capable of handling floating point instructions with a separate register file and pipeline that handles the issue, register read, execution and writeback of these instructions. Memory operations are handled through a dedicated load and store queues (LDQ and STQ, respectively) in the load-store unit (LSU), which supports speculative execution and enforces memory ordering constraints.

BOOM comes with multiple configurations, ranging a minimal single-issue design (*SmallBoomConfig*) to fully superscalar configuration capable of issuing up to 5 instruction per clock cycle (*GigaBoomConfig*). Because compilation, synthesis and simulation are computationally expensive, this thesis focuses exclusively on *MediumBoomConfig*, which provides superscalability at the lowest hardware cost among the available configurations (2-wide).

Baseline BOOM is known to be vulnerable to Spectre-like transient execution attacks [22], which has been demonstrated in [23]. Combined with its status as a well-documented and active open-source project, this makes it a suitable candidate for the work of this thesis.

2.4 Speculative Privacy Tracking (SPT)

Speculative Privacy Tracking (SPT) [7] is a hardware protection scheme that protects both data-at-rest (i.e., data stored in memory) and data-in-flight (i.e., data being processed by the CPU) from being leaked by transient execution attacks, in a software-transparent fashion.

SPT's threat model considers that an attacker has access to any speculative or non-speculative microarchitectural side channel, and is able to arbitrarily induce speculative execution from any part of the system.

It works by blocking transmitters (i.e., instructions that can create side channels) from executing speculatively only in the cases in which they would leak secret data. If certain transmitter's operands hold data that has already been leaked by non-speculative execution, it can safely execute speculatively, as there is no need to protect public data. Blocking only the strictly necessary speculative instructions supposes a great performance improvement over other more conservative security strategies, while being completely software-transparent.

To keep track of the public status of each piece of data, SPT introduces the concept of *taint*. Tainted data is defined as any program data in memory or registers that has not been leaked through a side channel yet, and it will stay tainted as long as it remains secret. As a consequence, the non-speculative execution of a transmitter that leaks an operand holding tainted data, forces this data to become *untainted*. In other words, untainted data is considered to be public to an attacker. The following observations can be extracted from these definitions: It becomes necessary that all data in memory and registers starts out as tainted. Tainted data may or may not become untainted during program execution, but untainted data can remain always untainted, as it is impossible to "unleak" it. And finally, new instructions that use tainted operands will yield tainted results.

SPT's efficiency stems from its untaint propagation mechanism. In general, transmitters use operands produced by other instructions, so once a transmitter untaints one of its operands, this untaint event gets propagated so all active instructions in the pipeline that use the same operand can update their taint status. Applying a set of defined untaint algebra rules, this propagation can reach not only instructions using the same exact operand, but also other operands that depend on it. If an instruction holds a register either as an operand or a destination, for which all its taint sources (i.e., destination and other operands) have become untainted, it will also become untainted. This untaint propagation will eventually converge, since in-flight data can not become tainted again, and with enough transmitters, all data could become untainted.

2.5 Formal Verification

Traditional verification is based on simulation of the design and requires explicitly specifying test cases for all the different scenarios of interest. While this can be enough to provide some degree of confidence in the functional correctness of a design, it cannot completely assure it, leaving gaps for bugs and vulnerabilities to slip in the design by construction.

In contrast, formal verification rigorously proves, using mathematical methods, that a system's behavior conforms to a precisely defined specification. Its exhaustive nature can provide strong guarantees that a design's implementation is fully consistent with its functionality. However, formal verification requires significantly more expertise from verification engineers and relies on complex, often proprietary, tools, and precisely defined specifications. Large designs, such as modern processors, are not easy to verify using

formal methods due to state explosion, and require the use of sophisticated modeling techniques to reduce the complexity of the proofs.

2.5.1 Bounded Model Checking

Bounded Model Checking (BMC) is a formal verification technique that analyzes the behavior of a system up to a finite execution depth by reasoning over a bounded unfolding of its state transitions [24]. The sequential model of the system is symbolically unrolled for a fixed number of steps, typically from an initial time t to $t+n$, thereby eliminating feedback loops present in the original cyclic model. This unrolling usually starts from a known initial state, such as reset.

The initial state and primary inputs of the model are constrained through assumptions, which define the set of executions considered during verification. The property under verification is then encoded together with the unrolled model into a boolean formula expressed as $Y_{property} \oplus Y_{model}$ [25]. This formula evaluates to true only in the property and the model are not equivalent, and it is fed to a Boolean Satisfiability (SAT) solver to search for a satisfying assignment. If a satisfying assignment exists, it corresponds to a concrete counterexample trace that violates the property. If no counterexample is found, correctness is guaranteed, but only within the specified bound. While increasing the bound can improve coverage and, in principle, establish correctness over all reachable states, the exponential growth of the state space quickly renders this approach infeasible for complex designs.

2.5.2 Interval Property Checking

Interval Property Checking (IPC) is similar to Bounded Model Checking but differs in its treatment of initial states and proof scope. Unlike BMC, IPC does not assume a specific starting state, so a property proven with IPC holds over all execution intervals satisfying the property's triggering conditions, providing an unbounded proof.

The absence of initial-state constraints, however, introduces additional challenges. Arbitrary initial states that do not correspond to valid or reachable system configurations may lead to spurious counterexamples. Such counterexamples formally violate the specified property but correspond to behaviors that cannot arise during normal program execution. To address this issue, IPC proofs are often augmented with invariants that constrain the system to a superset of the reachable state space while preserving the commitments of the verified property. Deriving sufficiently strong invariants is non-trivial and can be computationally expensive, limiting scalability in practice [26].

2.6 Unique Program Execution Check (UPEC)

Unique Program Execution Checking (UPEC) [9] is a formal verification methodology aimed at identifying vulnerabilities to transient execution attacks such as Meltdown or

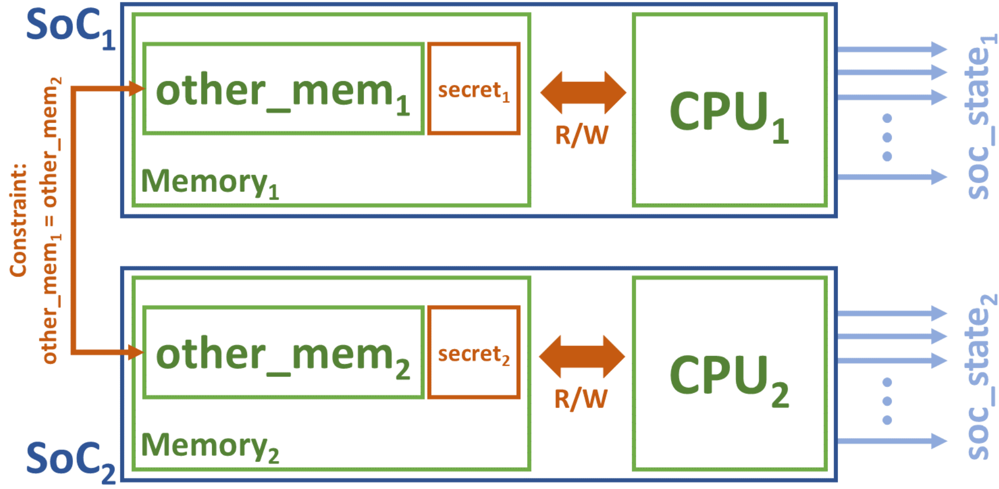


Figure 1: Computational model for UPEC [23]

Spectre, directly at the RTL. Unlike approaches that focus on specific, known exploits, UPEC does not rely on a priori knowledge on existing attacks. Instead, it defines a general security property that captures the absence of unintended information flow caused by transient execution.

Figure 1 depicts the computational model for UPEC, in which properties are formulated by comparing pairs of executions of the same program on two identical SoC instances (SoC_1 and SoC_2) that differ only in the secret data ($secret_1$ and $secret_2$), and checking whether transient execution can induce secret-dependent divergence in the system state (soc_state_1 and soc_state_2). Any divergence in the system state of both instances is considered a violation, but depending on which state variables they affect, these alerts are classified between leakage and propagation alerts as defined in the original UPEC work [9]:

L-alert A *leakage alert (L-alert)* is 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 *propagation alert (P-alert)* is 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.

In other words, L-alerts indicate an actual side channel leak, since the divergence has reached architectural state and is observable by an attacker. P-alerts, while confined to microarchitectural state and not directly exploitable, often precede L-alerts in later cycles, revealing potential propagation paths toward architectural leakage. Tracking P-alerts is therefore useful, because their absence is a sufficient, though not necessary, condition to show that no leakage paths exist, which is computationally easier than proving that no L-alerts ever occur.

UPEC has been evaluated on a range of processor designs, from simple in-order cores to complex out-of-order processors such as BOOM [23], demonstrating that the ap-

proach scales to realistic designs beyond minimal examples. Pure BMC or IPC approaches for such large designs exceed the capabilities of conventional computing resources because of their long sequential depth. This limitation is circumvented by applying UPEC with IPC from an initial symbolic state, which constrains the initial state with variables, instead of giving fixed values. However, IPC requires the use of invariants to mitigate the impact of spurious counterexamples in the proofs. Deriving an appropriate set of invariants is manual and significantly time-consuming, especially for large, complex designs like out-of-order processors, due to the extensive *bookkeeping* mechanisms that must be constrained to ensure correct instruction flow through the pipeline. This set of constraints is denominated *microequivalence* [11]. For all other aspects, applying UPEC is relatively automatic, enabling a secure-by-construction methodology which can be applied iteratively to identify and eliminate leakage paths during development, rather than retrofitting mitigations after vulnerabilities are discovered.

3 Related Work

3.1 Speculative Taint Tracking (STT)

Speculative Taint Tracking (STT) [27] is another hardware protection scheme created by Yu *et al* in 2019, the same authors of SPT. In this previous work, they consider a less strict threat model, by which they only protect speculatively accessed data.

STT uses a similar but conceptually different notion of taint, compared to SPT's. Only speculatively executed access instructions (i.e., loads) taint their output registers, and therefore any younger instruction that depends on them also becomes tainted. Tainted transmitters are stopped from being executed speculatively as in SPT, but the core difference is that once an instruction becomes non-speculative, or in other words, it passes the visibility point (VP) of the ROB, it becomes untainted and can execute. The tainting dependency of each instruction is maintained by tracking its youngest root of taint (YRoT), which is the youngest instruction that provided a tainted operand. Once the YRoT of an instruction becomes untainted, all depending instructions are also untainted, and the transmitter is safe to execute speculatively. What makes STT's untaint propagation simple and elegant is that it only requires to store the ROB index of the YRoT per register, and for every instruction, to compare it with the VP and head index of the ROB in order to untaint. Compared to STT, SPT's taint dependencies flow from one instruction to another making its implementation more complex in terms of hardware, but its protection scope is broader, protecting also non-speculative secrets from being leaked during transient execution.

3.2 SecureBOOM

SecureBOOM [28] is a formally-proven secure design of BOOMv3 [8] w.r.t. transient execution attacks. Jauch *et al.* [23] present a novel secure-by-construction RTL design methodology that uses formal verification to systematically detect potential leakage paths, which they use to create this variant. The design adapts the core principles of STT to BOOM, with some deviations from the proposed implementation. In SecureBOOM, tainting is moved from the rename stage to the register read stage, reducing the logic delay associated with resolving YRoT propagation during renaming [29]. Additionally, SecureBOOM introduces a new pipeline unit, the Information Flow Controller (IFC), responsible for checking unsafe transmitters at the start of execution. The IFC maintains a list of transmitters and receives the opcode, ROB index and taint information of the inflight instruction in every functional unit. When an unsafe transmitter is detected, the IFC sends a *kill* signal to the corresponding functional unit to prevent execution and simultaneously sends a *wait* signal to the issue unit, which will restart the instruction once it becomes safe under STT rules.

Overall, SecureBOOM demonstrates a formally verified, secure design of BOOM against transient execution attacks, applying STT's principles to control unsafe instruction execution and taint propagation in the pipeline. To the best of our knowledge, it remains the only other secure implementation of BOOM targeting these vulnerabilities. This work provides context and motivation for the design choices and methodology adopted in SPT-BOOM, which is described in the following sections.

4 Design & Implementation

In the original SPT work [7], Choudhary *et al.* evaluated this protection scheme using gem5's standard out-of-order CPU model, *O3CPU* [30]. In contrast, this thesis extends the core ideas of SPT by implementing them at the register-transfer level on a real superscalar out-of-order processor.

The chosen platform is BOOM, a superscalar out-of-order RISC-V processor [8] (cf. Section 2.3). The resulting implementation is referred to as SPT-BOOM and the version evaluated in this thesis is v1.0, available at [31].

In this chapter, we describe its implementation details and showcase deviations w.r.t the original design concept.

4.1 Design Goals

The main goal for this design is to implement SPT's core mechanisms taint and untaint propagation, and blocking of unsafe speculative transmitters, all while preserving base functionality and RISC-V compliance. We took some extra care trying to minimize performance costs in terms of area and latency, but optimizing these metrics was not part of the objectives. See Section 7 for some possible optimizations.

Due to the different architectures of both processors, SPT-BOOM necessarily introduces some deviations from the proposed implementation by its original authors. In an analogous way to SecureBOOM [28], we accept these differences for the sake of easier and more optimal implementation, as long as they do not jeopardize its security. These deviations are explained in more detail in Section 4.6.

4.2 High-Level Architecture

Figure 2 shows a simplified overview of SPT-BOOM architecture, showcasing what has been added (purple), what has changed (orange) and what remains (blue) from the basline implementation of BOOM.

Taint statuses are stored in a decentralized fashion, divided among the mapping table of the rename stage, each slot of the issue units, and each entry of the queues in load store unit. Then, the untaint broadcast bus (UBB) connects and synchronizes the taint status among the different stages of the pipeline.

Instructions are classified between transmitters and non-transmitters, based on whether their transient execution leaks at least one of their operands or destination register through a side channel. This distinction is implementedin the decode tables.

BOOM's ROB provides a Point of No Return (PNR) mechanism that matches SPT's Visibility Point concept, as it points to the first speculative instruction.

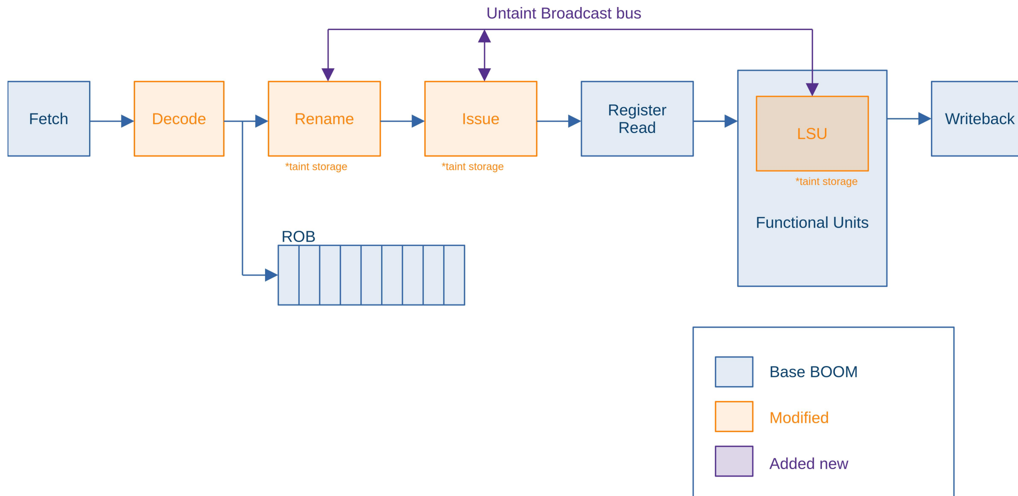


Figure 2: SPT-BOOM simplified architecture overview.

4.3 Protection Policy

SPT enforces a protection policy in which the execution of each transmitter is delayed until it is safe. Transmitter safety is defined by either of the following conditions:

1. All operands whose speculative use could leak secret data have become untainted, or
2. the transmitter itself has become non-speculative.

Consequently, each transmitter is guaranteed to be issued and executed non-speculatively. Performance benefits arise when operands become untainted while the instruction is still speculative, allowing earlier execution without compromising security.

For non-memory operations, this check is performed within the issue slots. A new state, *s_wait*, is added to the slot state machine to accommodate unsafe transmitters. Figure 3 illustrates the updated state diagram, with the new state and transitions highlighted in orange. When a valid transmitter enters the slot, it transitions to *s_wait* if neither safety condition is satisfied. Once one of the conditions is met, execution proceeds to *s_valid_1* or *s_valid_2*, the original issue-ready states. Safe instructions bypass *s_wait* entirely, incurring no additional latency. For clarity, *s_valid_1* and *s_valid_2* are represented as a single state, as their transitions remain unchanged.

For memory operations, the LSU enforces the same safety conditions on address operands, which represent the potentially leaked data when the instruction accesses memory.

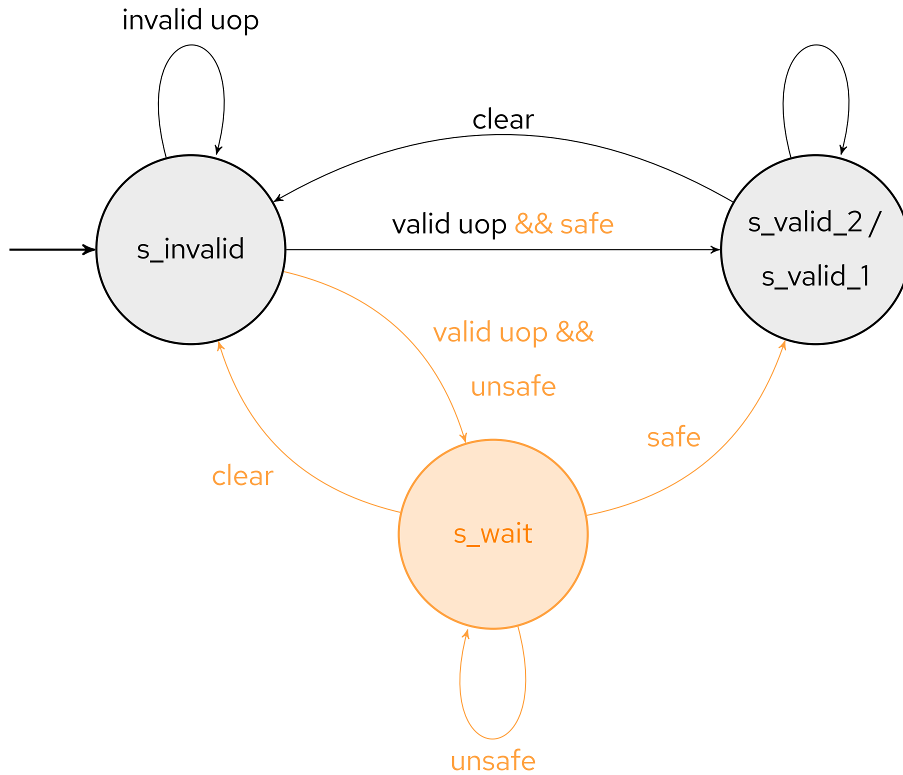


Figure 3: Issue slot machine state diagram, with SPT-BOOM modifications in orange.

4.4 Tainting

In SPT, access instructions (i.e., loads) are the only source of taint, which is applied exclusively at the rename stage. Initially all registers and memory addresses start off as tainted, since all non-program data is considered secret until it is leaked. The mappings from architectural/logical registers to physical registers are stored in the maptable in the rename stage. Each mapping entry in this table has been expanded with an additional bit to denote its taint status.

When a new instruction enters the rename stage, it performs a request to the maptable with its operands and destination registers. The maptable then returns the physical register IDs with the taint status of the operand registers, and creates a new mapping for the destination register , whose taint status is set under the following conditions:

1. For load instructions, the destination register is always tainted.
2. For other instructions that have a valid destination register, it is tainted if at least one of the operands is tainted (see Section 4.4.1 for exceptions).
3. For instructions without a valid destination register, it is never tainted, as these instructions cannot propagate taint to other ones. However, they could still be transmitters and create a side channel.

The last condition covers also instructions with x0 (hardwired 0 in RISC-V) as their destination register, since its value can never be a secret. Store instructions are an exception for this case, because they can pass their data to a load through memory or directly through *store-to-load forwarding* (Section 4.5.4), even though they lack a destination register.

4.4.1 Special Case: Conditionally Deterministic Instructions

There are special cases in which instructions with a valid destination register are able to safely skip taint propagation without compromising security. When the output of an instruction does not depend on the actual content of its operands but only on which operands are used, it means that its result can be known by an attacker before its execution and there is no need to taint the destination register or apply any blocking. An example would be a bitwise AND operation of any tainted register with x0. The result will be known to an attacker without knowing the content of the tainted operand, so tainting the result would result in a potential performance loss.

We call these instructions *conditionally deterministic*, and in order to detect them at rename stage for correct tainting, we added a new field to the decode tables that classifies every micro-operation (uop) code in four types:

1. **DET_X:** Non deterministic. Examples: LOAD, ADD
2. **DET_Z:** Deterministic if any operand is x0. Examples: AND, MUL
3. **DET_Z1:** Deterministic only if rs1 is x0. Example: shifts operations such as SLL, SRL and SRA.
4. **DET_E:** Deterministic if rs1 is equal to rs2. Example: XOR

These conditionally deterministic instructions represent edge cases that occur infrequently in typical program execution. How often and whether they occur at all, will depend on the programmer's decisions. For example, how the chosen compiler sets registers to 0 in assembly (i.e. `addi rd, x0, 0` vs. `xor rd, rd, rd`)

4.5 Untainting

In SPT, untaint events originate exclusively when transmitter instructions become non-speculative. When a transmitter reaches the Visibility Point (or PNR in BOOM terms), it is guaranteed that any potential side channel it creates will only leak data that is certain to become public upon commit, making its execution safe. This untainting can propagate to other instructions, leaving four primary sources of untaint for instructions:

1. Transmitter becoming non-speculative
2. Internal propagation

3. External propagation: Untaint Broadcast Bus
4. Store-to-Load forwarding

4.5.1 Transmitter becomes non-speculative

As discussed in Section 2.2, transmitters are instructions that can create side channels during transient execution. SecureBOOM [28] provides a formally verified classification of such transmitters for BOOMv3 based on existing instruction-level flags, which we adopt as an initial baseline.

SPT-BOOM extends this classification by explicitly determining, for each instruction, whether it acts as a transmitter and, if so, which of its operands may be transmitted. This information is encoded as a mask in the decode tables. The initial assignment of these masks is derived from SecureBOOM's classification and engineering analysis to the best of our knowledge. Thanks to UPEC's iterative workflow, this classification can be verified and refined to obtain formal guarantees later on.

All non-memory transmitters are enforced within the Issue Units, while load and store transmitters are handled by the Load–Store Unit (LSU). When an instruction enters an issue slot, it is blocked only if all the following conditions hold: (i) the instruction is classified as a transmitter; (ii) it holds tainted operands that may be transmitted, i.e., at least one bit overlaps between the taint mask and the transmittable-operand mask; and (iii) the instruction is still speculative.

If any of these conditions is not satisfied, the instruction is considered safe under the SPT policy and is allowed to execute without any latency penalty. Otherwise, execution is stalled until at least one blocking condition is cleared (c.f. Section 4.3). The same behavior applies to the address register of memory instructions in the LSU before issuing requests to memory.

Since transmitter classification is static and taint removal may depend on the execution of other instructions, the only original and guaranteed source of untaint is the instruction transitioning from speculative to non-speculative execution. In SPT-BOOM, this transition is detected by comparing the instruction's ROB index against the PNR. An instruction is considered non-speculative once it lies between the ROB head and the PNR, while instructions beyond the PNR remain speculative and are therefore subject to blocking. Figure 4 resembles this categorization in the ROB.

Once a transmitter becomes untainted by this method, its transmittable operands will be marked as untainted, ready to be propagated internally in the issue slot or LSU, and externally in the untaint broadcast bus.

4.5.2 Internal Propagation

Two types of internal propagation rules are applied to valid non-memory instructions in the slots of each Issue Unit: *forward propagation* and *backward propagation*. Internal

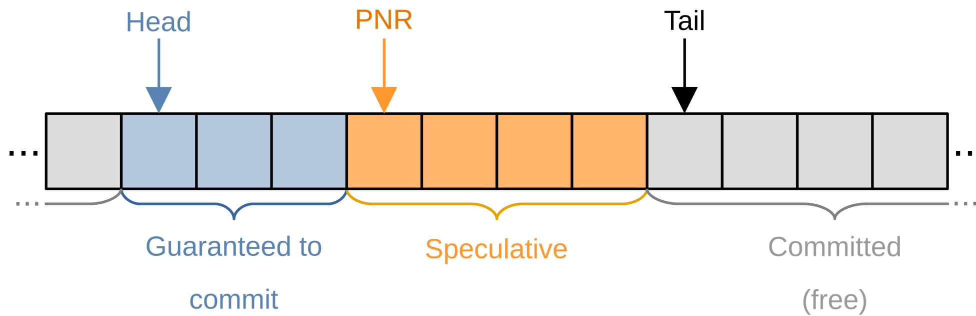


Figure 4: BOOM's ROB circular buffer.

propagation for memory-access instructions is performed directly in the LSU, where they are queued and handled (Section 4.5.4).

The forward propagation rule untaints the destination register whenever all the used operands have been already untainted. It applies to any instruction that only depends on their operands (i.e., not loads).

On the other hand, the backward propagation rule goes in the opposite direction, and untaints an operand if all the other operands and destination register have been untainted already. However, this rule is only valid for arithmetically invertible instructions. This means that knowing the value of the destination register and one of the operands (or two for 3-operand instructions) allows for the deduction of the other operand's value. In these situations, an attacker would be able to extract the tainted operand anyway, so it is safe to untaint it.

Similarly to conditionally deterministic instructions, there is a special case for conditionally invertible instructions. These are instructions that, depending on which operands they use (regardless of their arbitrary content), become invertible, allowing to apply backward propagation rules. Although there are more combinations, we distinguish the following cases:

1. **INV_X:** Non invertible. Examples: loads and stores
2. **INV_ZR1:** Invertible only if rs1 is x0. No examples were found.
3. **INV_ZR2:** Invertible only if rs2 is x0. Examples: REM and shifts
4. **INV_ZRX:** Invertible if either rs1 or rs2 is x0. Examples: OR
5. **INV_ER1:** Invertible if rs1 and rs2 are the same register. Examples: AND, OR
6. **INV_ZIM:** Invertible if the immediate is 0. Examples: SLLI and other immediate shifts
7. **INV_FUL:** Always invertible. Examples: ADD, SUB

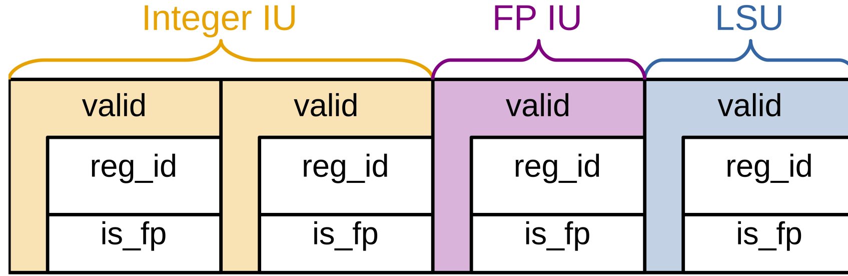


Figure 5: Untaint Broadcast Bus (UBB) structure.

And again, these are infrequent edge cases. How often they really occur in practice depends on the programmer or compiler used. For this reason, operations like multiplication and division, which are more prone to data loss when inverted (e.g., by truncation or losing the remainder), were considered not invertible to take a more conservative approach.

4.5.3 External Propagation: Untaint Broadcast Bus

The Untaint Broadcast Bus (UBB) facilitates the sharing of new untaint events across the whole pipeline, in order to synchronize the decentralized taint storage. As Figure 2 shows, it connects the Rename Stage, the Issue Units and the LSU. The Issue Units broadcast untaint events globally from internal (forward and backward) propagation and transmitters becoming safe. The LSU does the same from store-to-load forwarding data untaints and address untaints once a tainted load or store becomes safe to execute. The Rename Stage cannot untaint on its own, therefore it only listens to the bus.

Figure 5 shows the structure of the UBB. The bus is divided in lanes, which consist of a valid signal to indicate an untaint broadcast is happening, the physical register index that has been untainted, and a single bit flag indicating if it is an integer or floating point register. Each writer is assigned a number of lanes equal to its instruction width, preventing excessive queuing of untaint events.

Figure 6 illustrates an example of both external and internal untaint propagation within the pipeline. For clarity, the example assumes a single bus lane, with the valid signal and the floating-point flag asserted as appropriate. The sequence begins with the broadcast of physical register index 22 on the UBB (node u). This register is used as the first operand by *slot 1* in the issue unit and is therefore marked untainted locally (transition $u \rightarrow f$), along with the corresponding entry in the rename map table (transition $u \rightarrow m$). Since both operands of the instruction in *slot 1* are now untainted, the forward propagation rule applies, allowing the destination register to be untainted internally in the next clock cycle (transition $f \rightarrow d$). At this point, however, the UBB is occupied, preventing the slot from immediately broadcasting the untainting of destination register 43. To handle this condition, the pending untaint event is recorded in the *broadcast_queue*, which is set to `0001` (transition $f \rightarrow b$). This bitmask tracks untaint events awaiting broadcast, where the least

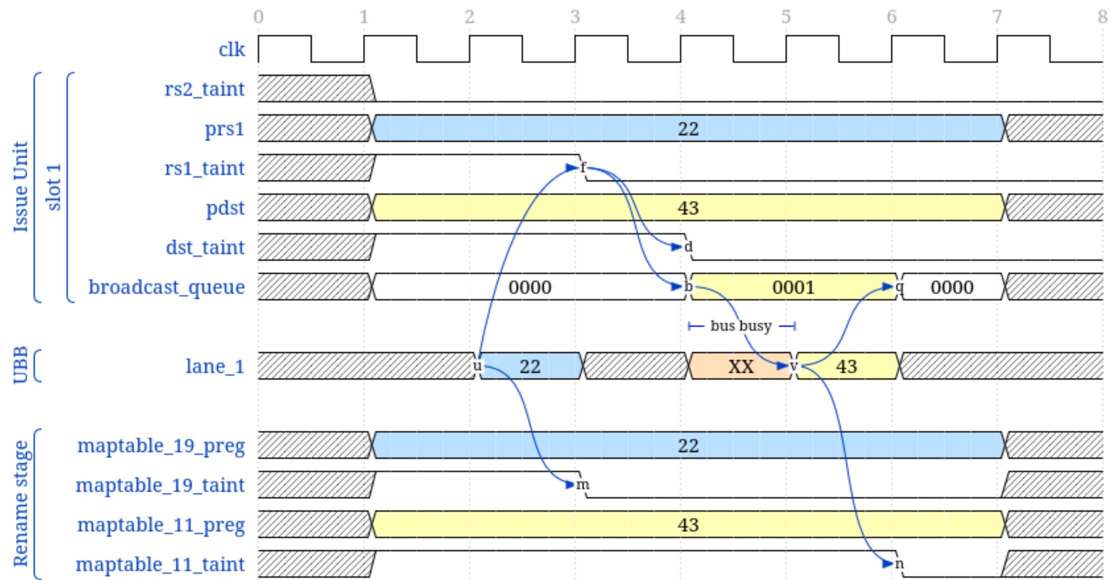


Figure 6: Example of external and internal propagation of untaint events

significant bit corresponds to *pdst*. The entry remains set until the corresponding register index is successfully broadcast via the UBB (transition $v \to q$). Once it has been broadcast, the untaint event is applied globally: the rename map table is updated accordingly (transition $v \to n$), and all other issue slots and LSU entries holding a matching physical register observe and process the untaint event (omitted in Fig. 6 for simplicity).

4.5.4 Store-to-Load Forwarding Propagation

When a load accesses the same address as a previous store that is still present in the LSU, it gets its data forwarded directly without passing through memory, in what is called *Store-to-Load forwarding*. This mechanism must satisfy three conditions to be secure, as defined by SPT [7]:

1. The store must forward to the load (i.e., be a correct match).
2. Load's address must be untainted so it can be fired.
3. The addresses of all stores older than the load and younger than the forwarding store (included) are untainted.

While conditions 1 and 2 may seem relatively obvious, condition 3 arises from the fact that forwarding and non-forwarding cases are indistinguishable to the attacker as long as the load's output remains tainted. Allowing untaint propagation for the data operand in this case would expose the forwarding decision and leak information about tainted data, since the stores' addresses are still tainted (i.e., they are unsafe to be translated).

To keep track of the last condition, Choudhary *et al.* [7] propose to add an internal broadcast in the LSU, so every cycle a store untaints its address, it broadcasts its Load

and Store Queue (LSQ) index, so that each load entry can check if it is between the index of the forwarding store and its own index. While straightforward to implement in gem5, this work adopts an alternative approach that reuses existing LSU logic to reduce RTL complexity. The "ForwardingAgeLogic" module in the LSU returns the youngest store that matches the address of the load, so it will forward from it, as it must not forward from an older store that would be overwritten. We modified this component to also count the stores with tainted addresses between the match (included) and the load. If the count is zero, the forwarding can be carried out.

This type of untaint is also bidirectional. If the store's data is untainted, it will untaint the load's data, and vice versa. However, we block all forwarding in our implementation, as keeping track of already executed loads in the LDQ presents implementation complexity that outweighs the potential benefits. Given this constraint, in SPT-BOOM backward propagation of data from the load to the store is impossible to happen, since the destination register of the load cannot be untainted before executing.

4.6 Deviations from SPT

Some deviations described in this section have already been introduced individually in earlier sections. Here, they are all collected and discussed systematically, together with the underlying reasons for each deviation. These differences arise either from mismatches between the ISA and microarchitectural assumptions of the original SPT proposal and those of the BOOM processor, or from explicit design choices made during the SPT-BOOM implementation.

Deviation 1: **No register partial access modes**. For registers that support partial access, SPT implements a taint status bit for every access mode allowed. However, RISC-V lacks partial access modes, so every instruction reads and writes the full register width. As a result, SPT-BOOM maintains a single taint bit per architectural register.

Deviation 2: **Register-to-register moves**. SPT allows backward propagation to occur in register-to-register moves, such as `MOV r1, r2`, but this class of instructions is not present in the RISC-V ISA. Instead, the same functionality is expressed using ALU instructions such as: `ADDI r1, r2, 0` or `ADD r1, r2, x0`. In these cases, backward propagation applies identically, since additions are arithmetically invertible and one of the operands is always untainted (`x0` or the immediate value).

Deviation 3: **Floating point registers are also tracked**. SPT-BOOM extends taint tracking to floating-point registers. As described in Section 4.5.3, the broadcast bus is shared, and a flag distinguishes between integer and floating point register indices.

Deviation 4: LSU's internal broadcast bus for pending tainted stores. SPT proposes an internal bus where stores broadcast their LSQ index once they become untainted, allowing load entries to track pending tainted stores as part of the conditions for store-to-load untaint propagation. In SPT-BOOM, this functionality is implemented by computing the number of pending matching stores in parallel with the lookup of the youngest store with a matching address (see Section 4.5.4).

Deviation 5: Conditionally deterministic and invertible instructions corner cases. SPT-BOOM introduces additional handling for conditionally deterministic and invertible instructions to cover corner cases in which taint can be removed more aggressively while preserving confidentiality. As described in Sections 4.5.2 and 4.4.1, the untainted data in these cases remains inferable by an attacker and therefore does not introduce additional leakage.

Deviation 6: LSU can write to the UBB. Choudhary *et al.* do not describe how untaint events originating inside the LSU (i.e., loads and stores becoming non-speculative, as well as load-to-store forwarding events) are propagated to the rest of the pipeline. To address this aspect, SPT-BOOM allows the LSU to write its untainted registers to the global bus by adding a dedicated lane (Section 4.5.3).

Deviation 7: Store-to-Load forwarding is blocked until untaint conditions are met In SPT, the store-to-load forwarding conditions (cf. Section 4.5.4) delay only the propagation of untaint between the store and load, while the forwarding itself proceeds normally. However, tracking all store–load pairs after execution to handle untaint propagation is expensive in hardware. In SPT-BOOM, these conditions are integrated directly into the forwarding logic, blocking store-to-load forwarding until they are satisfied. This more conservative approach simplifies the RTL implementation by eliminating the need to track previously forwarded data for untaint propagation.

Deviation 8: No shadow cache. Although SPT initializes all loads as tainted, Choudhary *et al.* propose a *shadow cache* to track the taint status of recently accessed data. SPT-BOOM V1.0 does not implement this optimization, as it is not required to preserve functional correctness or the security guarantees of SPT. Omitting the shadow cache results in a more conservative memory model, simplifies the verification effort, and avoids the added complexity that its inclusion would impose on formal verification.

Overall, these deviations maintain the security guarantees of SPT while adapting the methodology to the constraints and design choices of the BOOM microarchitecture. Section 6.2 analyzes their security implications in detail.

5 Evaluation

5.1 Experimental setup

As recommended in the BOOM official documentation [20], we developed SPT-BOOM under the Chipyard Framework [32]. Chipyard's environment facilitates the process of generating the RTL design in Verilog from BOOM's source code written in Chisel, and allows simulation with several industry-standard tools. We used Synopsys' simulation tool VCS, version R-2020.12 [33], which comes with the bundled-in waveform viewer DVE, used for debugging and waveform inspection during development.

Organizing and maintaining consistent signal groupings across pipeline stages in DVE requires significant manual work. As the BOOM microarchitecture evolves during development (e.g., when new fields are added to the *uop* class or additional pipeline signals are introduced), these groupings must be updated repeatedly and consistently across multiple stages. This process is time-consuming, error-prone, and does not scale well with iterative design changes. We considered using other visualization tools, but their lack of support for VPD files, and the infeasibility of using a converter made us discard this option. For illustration, the dumpfile of a program had a size of 42.3 MiB in VPD format, but after converting it to VCD with *vpd2vcd* (bundled with VCS), it increased to 3.3 GiB, almost 80 times larger. As a result, working directly with compressed VPD files and DVE was the most practical option.

To address these workflow limitations, we developed *signals2dve* [34], a Python-based utility that generates DVE signal groups automatically from a static YAML configuration. This approach enables reproducible and maintainable waveform views across design iterations while preserving compatibility with the VPD format generated by VCS. A detailed description is provided in Appendix A.

For the security analysis we employed the formal verification tool Onespin 360 [35] and the UPEC [9] framework. This setup is described in more detail in section 6.3.1.

5.2 Performance Evaluation

During the implementation of SPT, we conducted functional debugging primarily using the *towers* benchmark from the *riscv-tests* suite [36]. This benchmark was chosen for its short execution time (completing in under three minutes in debug mode for the medium configuration) and for its frequent use of memory accesses and control-flow instructions, which makes it well suited for iterative validation of functional correctness.

For an initial performance evaluation, the complete RISC-V benchmark suite was simulated on both BOOMv3 and SPT-BOOM v1.0 using the *MediumBoomConfig*. Because each benchmark execution initializes a fresh pseudo-random seed, runtime measurements can vary. To mitigate this variability, each benchmark configuration was executed five

Table 1: Execution time comparison between BOOMv3 and SPT-BOOM v1.0 on RISC-V benchmarks

Benchmark	BOOMv3 time (ns)	SPT-BOOM time (ns)	Slowdown (%)
`dhrystone`	215,400	510,658	137
`median`	64,934	82,129	26
`mm`	330,376	469,620	42
`mt-matmul`	74,355	82,239	11
`mt-vvadd`	219,497	254,304	16
`multiply`	82,370	134,352	63
`pmp`	1,866,505	2,231,786	20
`qsort`	400,173	859,610	115
`rsort`	374,571	730,869	95
`spmv`	265,962	374,857	41
`towers`	47,366	57,430	21
`vvadd`	55,469	58,962	6
`whetstone`	425,674	433,854	2
Average	340,204	489,115	43

times, and the arithmetic mean of the runs was recorded. Table 1 summarizes these results and compares them to the baseline BOOM configuration.

For a more representative performance assessment, we synthesized SPT-BOOM V1.0 (*MediumBoomConfig*) for the FPGA (AMD Virtex UltraScale+ VCU118), and executed the SPEC CPU 2006 benchmarks [37] at 75MHz. SPT-BOOM demonstrates a 72 % performance overhead relative to baseline BOOM. This overhead significantly exceeds the 45 % reported in the original SPT work using gem5 simulation, highlighting the additional costs introduced by RTL implementation constraints and conservative design choices. Furthermore, this discrepancy is consistent with the difference between the results obtained by Yu *et al.* for STT [27], and its RTL implementation in SecureBOOM [23], with 18 % vs. 36 % performance overhead, respectively.

For direct comparison, Figure 7 shows the benchmark results for SPT-BOOM V1.0 alongside the results of other implementations:

- **SecureBOOM:** STT implementation for BOOMv3 (cf. Section 3.2)
- **Naïve Delay:** executes loads once they reach the head of the ROB. Effectively disables the execution of all speculative loads.

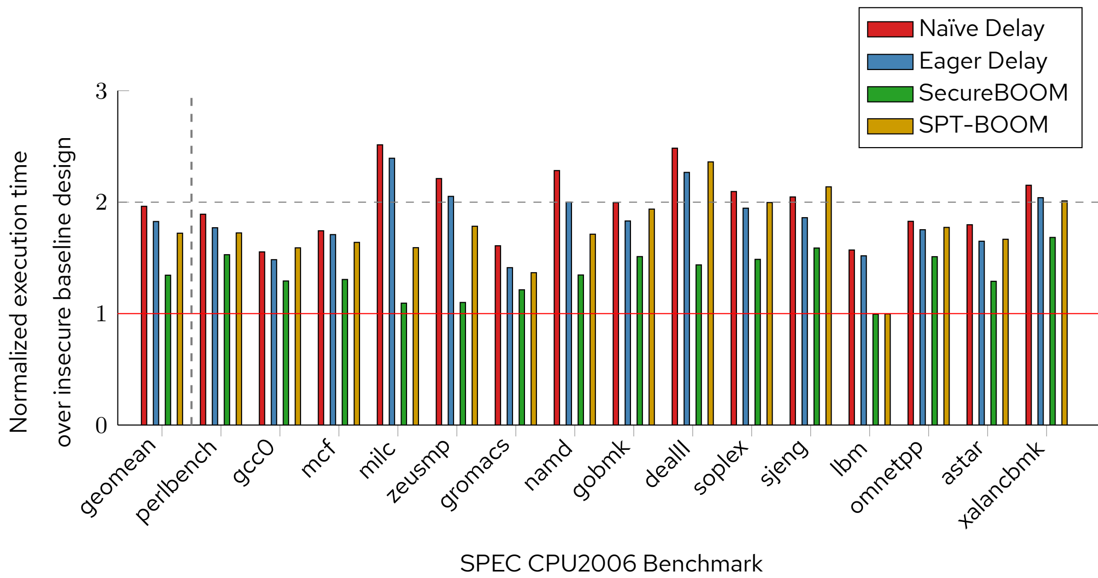


Figure 7: Normalized SPEC CPU2006 performance results

- **Eager Delay:** executes loads as soon as the reach the visibility point, so they are guaranteed to commit. Similar to SPT, but without propagation rules or blocking other transmitters.

5.3 Hardware Overhead

Hardware resource utilization for SPT-BOOM was extracted from the synthesized FPGA bitstream used to execute the SPEC CPU2006 benchmark suite (Section 5.2). Resource usage is reported in terms of look-up tables (LUTs), which implement combinational logic, and flip-flops (FFs), which implement sequential storage elements on the FPGA. Compared to the baseline BOOMv3 design, SPT-BOOM increases LUT utilization by roughly 7.5 % and FF count by 3.5 %. Table 2 summarizes the FPGA resource utilization of BOOMv3, SecureBOOM, and SPT-BOOM.

These results are derived from the synthesized FPGA implementation and primarily reflect the relative hardware overhead introduced by SPT-BOOM compared to baseline BOOMv3 and SecureBOOM. While these figures provide practical insight into the cost of adding SPT functionality, they do not directly correspond to silicon area or manufacturing cost. Additionally, the FPGA setup operates at a lower core frequency (75 MHz vs. ∼2 GHz in typical taped-out designs) and uses relatively fast DRAM, making memory accesses and cache misses less costly. As a result, timing-sensitive mechanisms, such as speculative load handling, may appear less expensive in this FPGA context than they would in a full silicon implementation. Accurate estimation of absolute hardware overhead would require synthesis targeting an ASIC technology.

Table 2: Relative FPGA hardware overhead compared to BOOMv3

Resource	SecureBOOM (%)	SPT-BOOM (%)
LUTs	8.24	7.51
FFs	3.30	3.51

6 Security Analysis

6.1 Threat Model

SPT-BOOM adopts the same threat model introduced by SPT, in which the adversary is assumed to be capable of intentionally triggering speculative execution and observing microarchitectural side channel effects across the system, including both speculative and architecturally committed behavior.

Both speculatively-accessed data and non-speculative secrets are protected under this model, while data transmitted over a non-speculative side channel explicitly falls out of the protection scope and can be used to improve performance.

6.2 Impact of Deviations

Section 4.6 presented the deviations introduced in SPT-BOOM with respect to SPT. Here, we analyze their impact on security.

Deviations 1, 2, 3 and 4 result solely from architectural differences between BOOM and the target of the original SPT design [30]. These deviations replicate or preserve the intended taint-tracking behavior of SPT within the BOOM microarchitecture.

Deviations 5 and 6 introduce new mechanisms that were not explored in the original SPT design. Deviation 5 addresses conditionally deterministic and invertible instructions. For the former instructions, the destination register is never tainted initially, preventing unnecessary propagation to subsequent instructions that depend on it. The latter instructions expand the set of invertible instructions for which backward propagation is applied. This behavior is safe provided that the relevant instructions are correctly identified, which can be formally verified using UPEC in the same manner as the list of transmitters. Deviation 6 allows the LSU to propagate untaint events to the rest of the pipeline. These events correspond to loads or store addresses that become non-speculative, as well as the data of store-to-load forwarding once its conditions are satisfied (Section 4.5.4). Since these untaint events only affect non-speculative data and follow SPT's general propagation rules, this mechanism does not compromise the security assumptions.

The last two deviations reduce the complexity of the RTL at the expense of additional overhead. Deviation 7 modifies the handling of store-to-load forwarding by postponing forwarding until all untaint propagation conditions are satisfied. Since this change only delays the availability of forwarded data and does not introduce additional information flow, it does not weaken the security guarantees of SPT-BOOM. Deviation 8 concerns the omission of the shadow cache, which likewise does not affect the functional correctness or security properties of the design. The shadow cache is an optional performance optimization. Excluding it simplifies verification and yields a more conservative memory model, at the cost of additional execution overhead.

6.3 UPEC for SPT-BOOM

UPEC can be applied to SPT-BOOM in an analogous way to its application to BOOM in [11]. While a complete end-to-end verification of SPT-BOOM V1.0 using UPEC has not been finalized within the scope of this thesis, the verification methodology and framework are fully established. This section documents the adopted verification setup, including the property structure and the definition of invariants, and identifies the remaining steps required to complete the formal verification. This ensures that the verification process is reproducible and can be extended systematically beyond the scope of this work.

6.3.1 Setup

A general template for UPEC applied to BOOM is available at [38]. It includes a hardware *miter*, which connects the two SoC instances (*BoomTile* and *TLMem*) required for UPEC. This piece of hardware, alongside the macro *state equivalence*, defines all invariants used to prevent spurious counterexamples. In particular, the invariants for microequivalence ensure the correct function of the core in terms of bookkeeping structures for in-flight instructions flowing through the pipeline.

A blackboxing module specifies components whose outputs are constrained to be equal across both SoC instances whenever the inputs are equal as well. This "sound blackboxing" mechanism allows violations to be detected through P-alerts entering blackboxed modules, preventing the accidental masking of side channel leakage through overly aggressive abstraction. The cache is the primary candidate for blackboxing due to the high computational cost associated with its full symbolic modeling. An important exception is required to enable transient execution leakage detection: The content of the data cache is permitted to differ when the accessed address tag corresponds to the secret memory location. For the purposes of this setup, the secret address is fixed to *0x8abcd*, a symbolic address can be used for a final sign-off later.

The UPEC verification suite also provides the tooling to automate the generation of properties based on design connectivity. The first step is to generate the fanin and fanout of the state signals, corresponding to the number of drivers of each signal and the number of elements it drives, respectively. This is performed with the *gen_fanout.tcl* script in the formal tool Onespin. The resulting connectivity lists are later refined into the sequential fanin and fanout, which after providing a source for the secret (*dcache/data/array_0_0/array_0_0_ext/mem_0_0/ram* in our case), it reads all the possible propagation paths to other state signals per clock cycle, and generates a property for each of them. The structure of these properties is described in the following subsection.

6.3.2 Property structure

Every UPEC property is generated automatically, following the structure in Figure 8. At each cycle, the secret may propagate to a different set of signals based on the com-

Figure 8: IPC property structure for UPEC

```
1  property UPEC_path/to_signal_cycleN;
2    dependencies:
3      no_reset, br_tag_in_bounds, functional_invariants, st_dep_masks,
           ptw_secret, spt_invariants;
4    assume:
5      during[t, t+1]: secret_load_tainted;
6      during[t, t+1]: taint_tracking_setup;
7      during[t, t+(N+1)]: blackboxing;
8      during[t, t+(N+1)]: secret_data_protected;
9  prove:
10     at t+(N+1): soc1/path/to_signal == soc2/core/path/to_signal;
11 end property;
```

puted sequential fanout. For each such signal in each specific cycle *N*, the property verifies that the signal holds the same value in both instances *soc1* and *soc2*, as shown after the *prove:* clause. Assumptions, also known as constrains, are specified at the beginning through the *dependecies:* and *assume:* sections. The difference between them is that dependencies apply over the full interval of the property, whereas assumptions can be scoped to specific time ranges. An example for the latter are the invariants constrained only for the first two clock cycles $(t, t+1)$, which serve to set up the initial conditions of the property. The set of constraints used for every property are described below:

- **no_reset:** Ensures no reset happens during the property.
- **br_tag_in_bounds:** Ensures the branch tags don't take any illegal values.
- **functional_invariants:** Extensive list of functional invariants that ensure the bookkeeping structures work correctly as expected.
- **st_dep_masks:** For all committable LDQ entries, it ensures its st_dep_mask (i.e. the list of stores older than itself) does not overlap with the set of all uncommittable ones.
- **ptw_secret:** Ensures that the Page Table Walker does not access the secret address, preventing in-flight accesses to the secret memory location in the symbolic initial state that never underwent the security checks of SPT.
- **spt_invariants:** Combination of all SPT-BOOM-specific invariants that apply for the whole duration of the property. More details in Section 6.3.3.
- **secret_load_tainted:** Enforces the taint of a load accessing the secret address following SPT scheme. More details in Section 6.3.3.

- **taint_tracking_invariants:** Combination of all *_tracked_taints* invariants for SPT-BOOM. More details in Section 6.3.3.
- **blackboxing:** Applies the blackboxing constraints to the specific modules.
- **secret_data_protected:** Ensures that user-mode accesses to the secret address trigger a page fault, consistent with architectural memory protection.

These properties are generated iteratively on a cycle-by-cycle basis. If a property holds for a given signal, that signal is proven incapable of propagating the secret, allowing all signals driven by it to be pruned from subsequent analysis. In this manner, the results for a given cycle are used to significantly reduce the number of properties required for the next one, which is especially useful for higher cycle counts.

6.3.3 SPT invariants

Defining a sufficient set of invariants is the most time-consuming aspect of applying UPEC to a new design. A substantial portion of the invariant set from [11] can be reused. However, SPT-BOOM introduces additional logic and structures that require new constraints. At the time of writing, the invariant set is not yet complete. Nevertheless, this section documents the invariants that have already been implemented, identifies gaps that must be addressed, and describes the framework used to integrate them into UPEC.

Some of the implemented invariants include:

- ***_preg_inbounds and *_lreg_inbounds:** Constrain the physical and logical register index fields in the *uop* structure to remain within architectural bounds, based on the configuration parameters `maxPregSz` and `lregSz`. These invariants are essential for a correct taint status tracking based on register indices. The *preg* constraint has been applied to the rename stage, issue units, LSU, and untaint broadcast bus, while the *lreg* only to the rename stage, since it is used as the mapping lookup index.
- ***_non_spec_soundness:** Constrains the *nonspec* flag in the *uop* structure to remain consistent with the ROB state, preventing incorrect classification of speculative and non-speculative instructions.
- **secret_load_tainted:** Ensures that any load accessing the secret access is tainted, as untainted data stops being considered a secret, and therefore, protected.
- ***_tracked_taints:** Constrains all taint-tracking storage elements in the pipeline to remain consistent with a symbolic taint model that accounts for untaint broadcasts. Issue units are treated as a special case, as they may internally resolve untaint conditions prior to broadcasting.

These represent only an initial subset of the required invariants. Future work must constrain all newly introduced control fields, such as transmitter flags and instruction classification metadata, to ensure that they cannot be set inconsistently by the formal engine.

For example, transmitter instructions should always have at least some bit of the mask *tx_reg* set, so they can be identified.

This thesis contributes a modular approach for integrating SPT-BOOM invariants into the UPEC framework. Invariants are defined in a dedicated *inv* module, with a companion *grp* module organizing signals into iterable structures, and both are bound to each *BoomTile* instance in the miter. Separating the invariant logic from the miter allows the same definitions to be bound to the synthesized RTL for simulation on VCS without modifying the Chisel-generated hardware, providing reusability and practical modularity. Benchmark execution can assert the negation of each invariant to detect violations, serving as a lightweight consistency check prior to full formal verification.

7 Future Work

Several options remain open for extending and strengthening the results presented in this thesis.

First, completing the full UPEC verification of SPT-BOOM is a natural next step. While the verification framework is established, obtaining sufficient invariants to block spurious counter examples and proving all properties until no more P-alerts or L-alerts are found would provide strong formal guarantees and increase confidence in the design.

Second, performance optimizations can be explored. The current implementation prioritizes correctness and verifiability over efficiency. Potential improvements include introducing a shadow cache as proposed in the original SPT work, refining transmitter classification, and reducing the width of the Untaint Broadcast Bus. Each optimization would require careful security validation.

Third, extending SPT-BOOM to additional BOOM configurations, such as wider superscalar designs or different branch predictor setups, would provide insight into scalability. Evaluating the design on ASIC technology rather than FPGA would also allow more accurate estimates of area, power, and timing overhead.

Finally, the methodology developed in this thesis could be applied beyond BOOM. Porting SPT to other open-source cores or integrating it into commercial-grade designs would help assess its general applicability and practicality as a long-term hardware defense against transient execution attacks.

8 Conclusion

This thesis presented SPT-BOOM, the first RTL implementation of Speculative Privacy Tracking on a realistic superscalar out-of-order processor. By integrating SPT into BOOMv3, this work demonstrates that fine-grained, software- and ISA-transparent protection against transient execution attacks can be realized in hardware without prohibitive complexity.

The implementation required adapting SPT's conceptual model to BOOM's microarchitecture, resulting in several deliberate deviations from the original proposal. These deviations were motivated by architectural constraints, verification considerations, and practical implementation trade-offs, and were shown to preserve the intended security guarantees.

Performance evaluation on FPGA using SPEC CPU2006 benchmarks revealed an average overhead of 72 %, highlighting the cost of RTL implementation and conservative design choices. Hardware synthesis shows moderate area impact, confirming that comprehensive protection can be integrated without excessive silicon cost.

From a security perspective, SPT-BOOM establishes a clear path toward exhaustive formal verification using UPEC. While full verification remains ongoing, the design is structured to support secure-by-construction development and systematic elimination of leakage paths.

Overall, this work bridges the gap between architectural proposals and concrete hardware implementations of speculative execution defenses. It provides a concrete design, empirical performance characterization, and a formal verification framework that together can contribute to the future development of secure, high-performance processor microarchitectures.

A Tooling: ***signals2dve***

signals2dve is a lightweight Python utility developed to automate the addition and grouping of signals in the DVE waveform viewer, part of the verification suite VCS [33]. The tool works by reading a YAML configuration file, which allows to programatically define signals, with attributes like their radix for display, and hierarchical groups. The script dynamically expands this configuration file and generates the TCL structures to be included in a DVE session file. This approach enables the generation of multiple reproducible, configurable and easy to update session files tailored to different needs.

Our motivation to create this script is that adding signals and grouping them manually is extremely time consuming. Especially during iterative designs, when a small change like adding a new field to a common signal, like the *uop* class, required manually adding it for every instance present in the pipeline. By automating this process, signals2dve facilitates iterative changes and is extensible to anyone working with BOOM out of the box, or any other project after configuring a YAML file.

The script is available publicly at [34], alongside with usage instructions and sample configurations.

A.1 Example

One of the main advantages of the script is that it can dynamically resolve signal names and groupings. A good example for this feature is BOOM's ROB. For the medium configuration, the ROB is composed of two banks, with 32 entries each, interleaved in such a way that each bank holds only the entries with odd and even numbers, respectively.

Figure 9 shows a sample configuration snippet to generate a group for every entry in the ROB, ordered by index, alternating from one bank to another. This is easily achieved with the use of expressions and iterators, which help cover corner cases like the one shown for the first bank (b=0), where there bank index is left empty instead of represented with a '0'. Figure 10 illustrates the expanded configuration in DVE, with some signals omitted for clarity.

Figure 9: *signals2dve* configuration file example

```yaml
env:
  tile: 'TestDriver.testHarness.chiptop.system.tile_prci_domain.tile_
      reset_domain.boom_tile'
  core: '$tile.core'
  width: 2
...
# ReOrder Buffer
  - name: 'ROB'
    base: '$core.rob'
    children:
      - path: '.rob_head_idx'
        radix: 'decimal'   # Change radix of the signal
        ...
    subgroups:
      - name: 'Entries'
        base: ''
        subgroups:
          - name: 'idx ${idx}'
            base: ''
            iterators:
              e: 32        # entries of ROB per bank
              b: 2         # banks of ROB
            # Expresions can be constructed with iterators for custom
                naming
            expr:
              # entry number since they are alternated between banks
              idx: '(e << 1) + b'
              # Bank display (since it is '' if b=0)
              bd: ' b if b != 0 else "" '
            children:
              - path: '.rob_val_${bd}_${e}'
              - path: '.rob_bsy_${bd}_${e}'
              - path: '.rob_unsafe_${bd}_${e}'
                ...
```

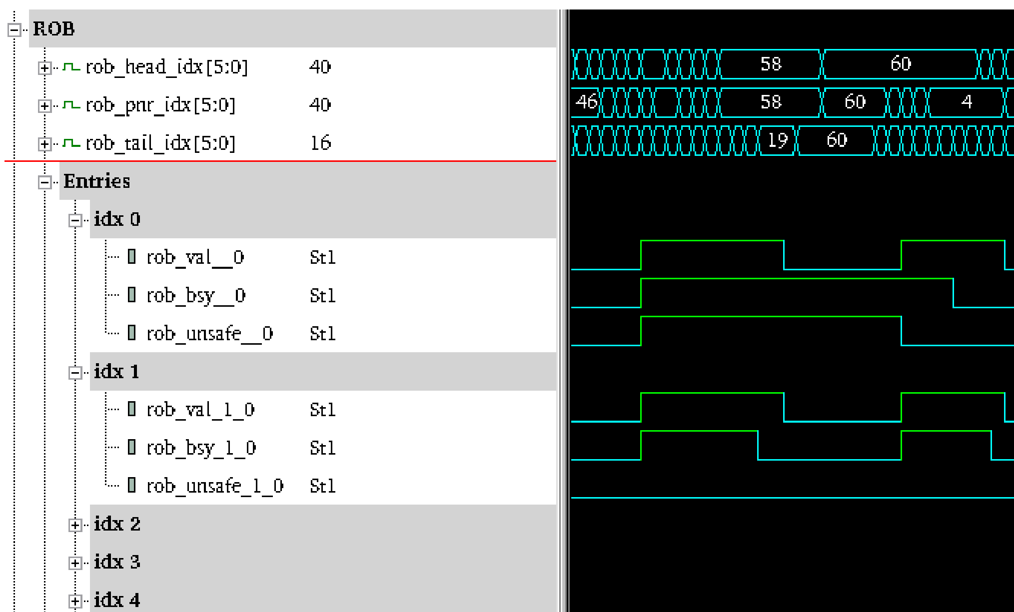


Figure 10: ROB entries grouping on DVE from example configuration

References

[1] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 5th ed. San Francisco, CA: Morgan Kaufmann, 2011.

[2] C. Canella, J. V. Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A Systematic Evaluation of Transient Execution Attacks and Defenses," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 249–266. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/canella

[3] 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 *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 1–19.

[4] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard, "KASLR is Dead: Long Live KASLR," in *Engineering Secure Software and Systems*, E. Bodden, M. Payer, and E. Athanasopoulos, Eds. Cham: Springer International Publishing, 2017, pp. 161–176.

[5] P. Turner, "Retpoline: A software construct for preventing branch-target injection," https://support.google.com/faqs/answer/7625886, Jan. 2018, google Security Blog.

[6] I. The Linux Kernel Organization, "Page Table Isolation (PTI)," https://www.kernel.org/doc/html/next/x86/pti.html, 2023, linux kernel documentation, version next-20230330, accessed 2026-01-14.

[7] R. Choudhary, J. Yu, C. Fletcher, and A. Morrison, "Speculative Privacy Tracking (SPT): Leaking Information From Speculative Execution Without Compromising Privacy," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 607–622. [Online]. Available: https://doi.org/10.1145/3466752.3480068

[8] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic, "SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine," in *Fourth Workshop on Computer Architecture Research with RISC-V*, vol. 5. International Symposium on Computer Architecture Valencia, Spain, May 2020, pp. 1–7.

[9] 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 *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, March 2019, pp. 994–999.

[10] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading Kernel Memory from User Space," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 973–990. [Online]. Available: https://www.usenix.org/conference/usenixsecurity18/presentation/lipp

[11] M. R. Fadiheh, A. Wezel, J. Müller, J. Bormann, S. Ray, J. M. Fung, S. Mitra, D. Stoffel, and W. Kunz, "An Exhaustive Approach to Detecting Transient Execution Side Channels in RTL Designs of Processors," *IEEE Transactions on Computers*, vol. 72, no. 1, pp. 222–235, Jan 2023.

[12] S. Wiebing and C. Giuffrida, "Training Solo: On the Limitations of Domain Isolation Against Spectre-v2 Attacks," in *IEEE S&P*, May 2025, cSAW Best Paper Award Runner-up, Intel Bounty Reward. [Online]. Available: Paper=https://download.vusec.net/papers/trainingsolo_sp25.pdfWeb=https://www.vusec.net/projects/training-soloCode=https://github.com/vusec/training-solo

[13] J. Kim, J. Chuang, D. Genkin, and Y. Yarom, "FLOP: Breaking the Apple M3 CPU via False Load Output Predictions," in *USENIX Security*, 2025.

[14] J. Kim, D. Genkin, and Y. Yarom, "SLAP: Data Speculation Attacks via Load Address Prediction on Apple Silicon," in *S&P*, 2025.

[15] H. Ragab, A. Mambretti, A. Kurmus, and C. Giuffrida, "GhostRace: Exploiting and Mitigating Speculative Race Conditions," in *USENIX Security*, Aug. 2024. [Online]. Available: Paper=https://download.vusec.net/papers/ghostrace_sec24.pdfWeb=https://www.vusec.net/projects/ghostraceCode=https://github.com/vusec/ghostraceVideo=https://www.youtube.com/watch?v=J4CvoGimuk8

[16] 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*, ser. MICRO-52. New York, NY, USA: Association for Computing Machinery, 2019, p. 572–586. [Online]. Available: https://doi.org/10.1145/3352460.3358306

[17] M. F. Abdul Kadir, J. K. Wong, F. Ab Wahab, A. F. A. Abidin Bharun, M. A. Mohamed, and A. H. Zakaria, "Retpoline Technique for Mitigating Spectre Attack," in *2019 6th International Conference on Electrical and Electronics Engineering (ICEEE)*, 2019, pp. 96–101.

[18] J. Yu, L. Hsiung, M. E. Hajj, and C. W. Fletcher, "Data Oblivious ISA Extensions for Side Channel-Resistant and High Performance Computing," Cryptology ePrint Archive, Paper 2018/808, 2018. [Online]. Available: https://eprint.iacr.org/2018/808

[19] M. Schwarz, M. Lipp, C. Canella, R. Schilling, F. Kargl, and D. Gruß, "ConTExT: A Generic Approach for Mitigating Spectre," in *Network and Distributed System Security Symposium 2020*, Feb. 2020, network and Distributed System Security Symposium 2020, NDSS ; Conference date: 23-02-2020 Through 26-02-2020.

[20] R.-V. B. Contributors, "BOOM Core Documentation (for BOOMv3)," https://docs.boom-core.org/en/latest, May 2025, accessed 2025-08-11.

[21] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: constructing hardware in a Scala embedded language," in *Proceedings of the 49th Annual Design Automation Conference*, ser. DAC '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 1216–1225. [Online]. Available: https://doi.org/10.1145/2228360.2228584

[22] C. Celio, J. Zhao, A. Gonzalez, B. Korpan, K. Asanovic, and D. Patterson, "BOOM: An Open-Source Out-of-Order Processor," Presentation slides, Chisel Community Conference, 2018, accessed 2026-01-14. [Online]. Available: https://boom-core.org/docs/boom_processor_ccc18_celio.pdf

[23] T. Jauch, A. Wezel, M. R. Fadiheh, P. Schmitz, S. Ray, J. M. Fung, C. W. Fletcher, D. Stoffel, and W. Kunz, "Secure-by-Construction Design Methodology for CPUs: Implementing Secure Speculation on the RTL," in *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, Oct 2023, pp. 1–9.

[24] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded Model Checking Using Satisfiability Solving," *Formal Methods in System Design*, vol. 19, no. 1, pp. 7–34, 2001. [Online]. Available: https://doi.org/10.1023/A:1011276507260

[25] A. Biere and W. Kunz, "SAT and ATPG: Boolean engines for formal hardware verification," in *Proceedings of the 2002 IEEE/ACM International Conference on Computer-Aided Design*, ser. ICCAD '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 782–785. [Online]. Available: https://doi.org/10.1145/774572.774687

[26] M. D. Nguyen, M. Thalmaier, M. Wedler, J. Bormann, D. Stoffel, and W. Kunz, "Unbounded Protocol Compliance Verification Using Interval Property Checking With Invariants," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 11, pp. 2068–2082, 2008.

[27] 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," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '52. New York, NY, USA: Association for Computing Machinery, 2019, p. 954–968. [Online]. Available: https://doi.org/10.1145/3352460.3358274

[28] "SecureBOOM," https://github.com/RPTU-EIS/SecureBOOM, May 2023, release v1.0, commit d0257a7, accessed 2025-08-12.

[29] B. Chen, R. Choudhary, K. Khulbe, A. Lee, A. Morrison, and C. W. Fletcher, "μstt: Microarchitecture design for speculative taint tracking," in *ICCD*, 2025. [Online]. Available: https://bluechen8.github.io/publications/uSTT/

[30] gem5 Simulation Framework, "O3CPU," 2025, accessed December 5, 2025. [Online]. Available: https://www.gem5.org/documentation/general_docs/cpu_models/O3CPU

[31] "SPT-BOOM," https://github.com/RPTU-EIS/SPT-BOOM, Nov. 2025, release v1.0, commit 6f925bc, accessed 2025-11-12.

[32] A. Amid, D. Biancolin, A. Gonzalez, D. Grubb, S. Karandikar, H. Liew, A. Magyar, H. Mao, A. Ou, N. Pemberton, P. Rigge, C. Schmidt, J. Wright, J. Zhao, Y. S. Shao, K. Asanović, and B. Nikolić, "Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs," *IEEE Micro*, vol. 40, no. 4, pp. 10–21, 2020.

[33] Synopsys, Inc., "VCS Functional Verification Solution," https://www.synopsys.com/verification/simulation/vcs.html, 2025, Version R-2020.12, accessed: 2025-12-30.

[34] "signals2dve," https://github.com/siblor/signals2dve, 2026, accessed: 2025-12-16.

[35] OneSpin Solutions GmbH, "OneSpin 360 DV-Verify," https://www.onespin.com, version 2025.3, accessed 2025-11-27.

[36] RISC-V Software Source Contributors, "riscv-tests: Risc-v architectural test suite," https://github.com/riscv-software-src/riscv-tests/, accessed 2025-12-12.

[37] Standard Performance Evaluation Corporation (SPEC), "SPEC CPU2006 Benchmark Suite," https://www.spec.org/cpu2006/, 2006, accessed: 2026-01-01.

[38] "UPEC BOOM verification suite," https://github.com/RPTU-EIS/upec-boom-verification-suite, 2023, accessed: 2025-11-11.