Secure-by-Construction Design Methodology for CPUs: Implementing Secure Speculation on the RTL

Tobias Jauch*, Alex Wezel*, Mohammad R. Fadiheh†, Philipp Schmitz*, Sayak Ray‡, Jason M. Fung‡, Christopher W. Fletcher§, Dominik Stoffel*, Wolfgang Kunz*

*RPTU Kaiserslautern-Landau, Germany †Stanford University, USA
‡Intel Corporation, USA §University of Illinois Urbana-Champaign, USA

2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD) | 979-8-3503-2225-5/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICCAD57390.2023.10323843

***Abstract*—Spectre and Meltdown attacks proved Transient Execution Side Channels to be a notable challenge for designing secure microarchitectures. Various countermeasures against these threats were proposed on the electronic system level. However, addressing all possible attack scenarios requires the design and analysis of bit- and cycle-accurate implementations.**

We present a novel secure-by-construction RTL design methodology based on a new hardware protection framework underpinned by a generic control infrastructure that can be integrated into industry-grade microarchitectures. The methodology uses formal verification to systematically detect possible leakage paths and to customize the generic infrastructure accordingly for the design. We propose an iterative flow which semi-automatically leads to an RTL design that is guaranteed to be secure w.r.t. transient execution attacks. A case study for the methodology is conducted on BOOMv3, an open-source RISC-V processor with a deep out-of-order pipeline, and the resulting secure RTL design is benchmarked on an FPGA setup. Our design outperforms a design based on conservative countermeasures, improving the incurred overhead by 3× / 4× (depending on the threat model) while maintaining the same level of security.

I. Introduction

The emergence of Transient Execution Side Channel (TES) attacks, such as Spectre [1] and Meltdown [2], brought a whole new set of challenges to the design of secure hardware. These attacks exploit side effects of transient execution of instructions, i.e., the instructions are not part of the correct program flow, but are executed and later discarded due to mis-speculation or an earlier exception. These side effects, though not visible in the ISA-level view of a program execution, can form microarchitectural timing side channels. TES attacks mainly exploit high-end microarchitectural features such as speculative and out-of-order execution. These features significantly contribute to the performance of modern processors and mitigating TES attacks without degrading the performance is a major challenge.

Traditionally, the response to TES attacks was using microcode and software patches. Although being crucial for restoring trust in legacy systems, these patches could not provide full security for future variants in every case. The existence of TES attacks discovered after the patching of the original Spectre and Meltdown attacks (e.g., [3], [4]) is evidence to this.

Besides potential security gaps, given the evolution of and advances in exploit techniques, software patches primarily rely on synchronization barriers and fence instructions, which can deteriorate performance significantly, up to 200 % in some cases [5]. These issues underline the need for more holistic mitigations at the microarchitectural level.

Various microarchitectural schemes have been developed to address vulnerabilities to TES attacks (e.g., [6], [7], [8]). These schemes aim to prevent data leaks that are caused by speculative or out-of-order execution without completely disabling these features. Although there has been significant progress in this field regarding security guarantees as well as performance overhead of the hardware countermeasures, the evaluation of these techniques has so far mostly been done based on abstract models, such as gem5 [9]. These abstract models provide a good estimate of the impact of the designed architecture on performance. However, the lack of cycle-accurate behavior in these models results in a semantic gap leaving out specific details of the security, performance and design of these microarchitectural schemes at the register transfer level (RTL). Furthermore, these techniques rely on the designer's expertise to manually find timing side channels in a design. Such an ad-hoc approach could result in errors when considering highly complex microarchitectures at the RTL. Therefore, there is still an open challenge inbringing hardware protection mechanisms to the RTL in a systematic way.

A. Contribution

We address these challenges by proposing a new hardware protection framework that can be integrated into industry-grade microarchitectures. It is systematically customized in a secure-by-construction design flow to achieve the defined security targets. The proposed flow is an iterative procedure that interleaves design steps with formal verification to create security countermeasures against TES attacks in a semi-automated way. Our design and verification flow pinpoints security issues at the design phase and enables the designer to implement targeted security countermeasures rather than conservative and expensive blanket fixes.

The proposed approach utilizes a generic dynamic information flow tracking infrastructure to detect the flow of information from transiently accessed data, building upon secure speculation approaches such as Speculative Taint Tracking (STT) [6] and DOLMA [7]. The information flow policy, i.e., when to block propagation of tainted information, is determined based on a formal analysis of the microarchitecture using Unique Program Execution Checking (UPEC) [10].

The proposed systematic design flow provides an end product with a well-defined formal security guarantee. This formal guarantee enables the designer to adopt more aggressive optimizations without increasing the risk of compromising security because any security violation is guaranteed to be detected in the design flow.

The main contributions of the paper are as follows:

- A secure-by-construction RTL design flow based on a generic control infrastructure for security is proposed for designing microarchitectures that are secure against TES attacks (Sec. III). It replaces ad-hoc and error-prone patches by a holistic and systematic approach that requires no *a priori* knowledge about TES attacks and is backed by formal security guarantees.
- Implementing advanced security features, such as STT, without gaps normally exceeds the competences of the human design teams responsible for RTL design in state-of-the-art industrial flows. Therefore, we propose to decompose the design problem into standard RTL design tasks that do not require any extraordinary competences and additional security-specific measures that demand a sophisticated design analysis.

For the first part, we propose a generic and re-usable infrastructure including a centralized control unit (Sec. III-C). A designer can straightforwardly extend a given RTL design using this generic infrastructure. For the second part, we propose a tool-guided procedure (Sec. III-D) which incorporates the security objective into the generic infrastructure by an iterative and automated flow. The security-specific customization of the generic infrastructure is automated based on a formal analysis. This ensures that the design's sophisticated information flow policy, i.e., when to block propagation of tainted information, is determined automatically and implemented correctly – without any special expertise of the designer.

- A case study of the proposed design flow is conducted on the Berkeley Out-of-Order Machine (BOOM) [11], in which we developed a secure RTL implementation (Sec. IV). The secure design delivers an average performance overhead of 5.2 % and 36.0 % compared to the insecure baseline design, depending on the threat model. Our case study shows that RTL design details can create timing variations that are not visible in the more abstract models. Therefore, a systematic methodology based on cycle- and bit-accurate models is necessary.
- The implemented design, to the best of our knowledge, is the first formally verified RTL hardware implementation of a processor featuring secure speculation with competitive performance. The design is capable of running a Linux operating system and is comparable to medium-sized application processors used in industry.

II. Related Work

A. Mitigating TES Attacks

The emergence of transient execution attacks required immediate countermeasures. The first mitigations, some of them still in use, were based on software constructs augmenting vulnerable parts of the operating system, e.g., context-sensitive fencing [12].

Apart from software countermeasures, researchers developed new approaches to close the high-bandwidth side channels in the cache [13], [14], [15], [16]. These approaches, however, do not provide a holistic solution to TES vulnerabilities and do not mitigate attacks based on channels other than cache footprints.

STT [6] and DOLMA [7] employ dynamic information flow tracking to prevent all possible TES vulnerabilities by selectively blocking speculative execution of certain instructions.

These approaches enforce the principle of *weak speculative non-interference* [17], i.e., no transient data access interferes with the timing of the committed instructions. Dynamic information flow tracking enables the processor to execute more instructions speculatively without risking information leakage. However, the security of the design depends on the ability of the designer to identify every instruction capable of forming a side channel, which is not a trivial task for a complex out-of-order microarchitecture. The mentioned strategies were developed and published based on an abstract processor model using the gem5 simulator [9] which simplifies certain behavioral aspects compared to an RTL design of a processor. Implementing a secure microarchitecture on RTL incurs several challenges that are addressed in this paper.

Another category of patches combines hardware and software measures to make systems more secure. Mitigations such as SpectreGuard [18], ConTeXT [19] or ProSpeCT [20] are based on annotating sensitive memory regions in software. Additional hardware modifications then ensure that the content of an annotated region cannot be leaked during transient execution. In addition to preventing TES attacks, these combined approaches are also able to ensure secure speculation for programs that comply with the constant-time programming paradigm. On the other hand, annotations at the software level require compiler and / or ISA support and incur additional manual effort.

B. Detecting TES Vulnerabilities in Hardware Designs

Simulation- and fuzzing-based techniques have been developed to detect vulnerabilities to TES attacks in hardware designs by generating directed and randomized tests [21], [22], [23]. Due to their non-exhaustive nature, these approaches cannot provide formal guarantees.

Other works employ hardware taint properties to verify various security objectives, including TES attacks [24], [25], [26]. Taint property verification pioneered the adoption of formal methods in hardware security. However, these techniques have limited detection capabilities for information leakage through previously unknown side channels or paths with large temporal length.

Unique Program Execution Checking (UPEC) [27], [10] is a formal security verification technique capable of verifying RTL hardware designs. UPEC aims to exhaustively verify the absence of vulnerabilities to TES attacks, even the ones based on previously unknown channels. It has been demonstrated to be scalable even to deep out-of-order processors with speculative execution [28]. UPEC has also been extended to other security targets, such as integrity [29], data-oblivious execution [30] and confidentiality in SoCs [31]. UPEC is employed at the core of the proposed design methodology and is further explained in Sec. III-B.

III. Secure-by-Construction Design Flow

This section presents the proposed secure-by-construction design methodology in detail. First, we discuss the underlying threat model (Sec. III-A) and the formal security analysis (Sec. III-B). Sec. III-C introduces a generic control infrastructure for security. Sec. III-D puts everything together and employs these concepts in our iterative design methodology.

A. Threat Model

Our approach is based on the commonly used threat model for TES. The security target is preventing any transient execution side channel from leaking the content of data memory.

This threat model assumes an attacker that can measure the timing of instruction execution with clock cycle accuracy, including the timing of the victim software execution. The victim process has sufficient privilege to access secret data and may transiently execute any instruction outside its correct program flow and later discard the results. The attacker can poison the predictors in order to trick the victim process into executing a specific gadget. This threat model includes any TES attack based on any microarchitectural side channel such as cache side channels [32], port contention [3], and data-dependent timing in functional units [33].

For this threat model we may assume that the victim does not leak its protected data in the correct program flow under sequential semantics. Hence, preventing attacks enabled by classical side channel analysis, e.g., monitoring the instruction cache footprint of square-and-multiply exponentiation in RSA [34], is not considered in this paper.

Our threat model protects data that resides in the data memory (*data-at-rest*), while any information in general-purpose or control status registers is considered non-confidential. Storing confidential information in the architectural registers must be handled responsibly by the software developer. Physical side channels, such as power or electromagnetic side channels, are also out of scope of this paper.

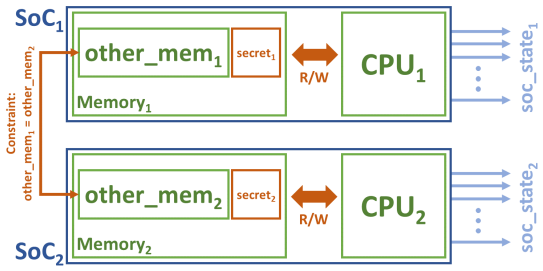


Fig. 1: Computational Model for UPEC

```
assume:
  at t:     micro_soc_state_1 = micro_soc_state_2;
  during [t, t+k]:     secret_data_protected();
  during [t, t+k]:     secret_load_transient();
prove:
  at t+k:     soc_state_1 = soc_state_2;
```

Fig. 2: UPEC property formulated as an interval property

To allow for a trade-off between performance and security, we distinguish between two different threat models, similar to the ones proposed in [13]: *spectre* and *futuristic*. The *spectre* model covers timing side channels caused by control-flow mispredictions, i.e., it only considers instructions after unresolved branch or jump instructions and neglects any other form of transient execution. The *futuristic* model extends this to *all kinds* of transient execution and includes prior exceptions, memory consistency model violations and load/store ordering failures. Such a distinction is helpful for the designer since the *spectre* model already covers the majority of known Spectre variants and prevents all control-flow-based universal read gadgets like Spectre-V1 and Spectre-V2. Additionally, it allows the microarchitecture to benefit more from out-of-order execution.

B. Formal Security Analysis with UPEC

The secure-by-construction design methodology proposed in this section employs a formal security analysis carried out by UPEC [10]. UPEC enables the methodology to deliver a design with formal security guarantees with respect to the threat models above (Sec. III-A).

UPEC exhaustively searches for TES vulnerabilities in an RTL design by checking whether or not transiently accessed data can interfere with program execution as observed by the cycle-accurate sequence of valuations to the architectural registers. In UPEC, any transiently accessed data is considered confidential and, in the following, referred to as *secret*. UPEC verifies a 2-safety hyperproperty on a self-compositional model, depicted in Fig. 1. The model consists of two structurally identical instances of the RTL design, and the only difference is the content of the memory locations holding secret data.

Fig. 2 shows the UPEC property. The property, specified in the form of an implication between an assumption and a commitment, is an interval property [35] starting from an arbitrary state. In the assumption part, *micro_soc_state* denotes the vector of all state variables in the design. The first assumption ensures that the two design instances in the model start from an arbitrary but equal microarchitectural state. The next two assumptions specify two requirements for the execution trace considered by the property: *secret_data_protected()* specifies that any user-level load instruction (i.e., a load issued by the attacker) targeting addresses outside the user's memory region is blocked by an exception. *secret_load_transient()* ensures that any privileged load instruction (issued by the victim) accessing the secret is transient, depending on the threat model. This means its result will be discarded and not committed to the architectural registers. By the use of these assumptions, we only consider executions where the secret is not accessed (read from memory) in the program's non-speculative semantics. In the commitment part, *soc_state* is a vector of state variables that includes all architectural registers of the design. Any discrepancy in the valuation of *soc_state* between the two instances must originate from the secret as it is the only difference between them. When verifying the UPEC property, the solver explores all possible scenarios under which the secret is read either illegally (blocked by an exception) or transiently, and checks how the secret can propagate or leak. In [10], a verification framework based on UPEC is presented. The verification framework yields an unbounded security guarantee using this bounded interval property.

C. Generic Control Infrastructure for Security

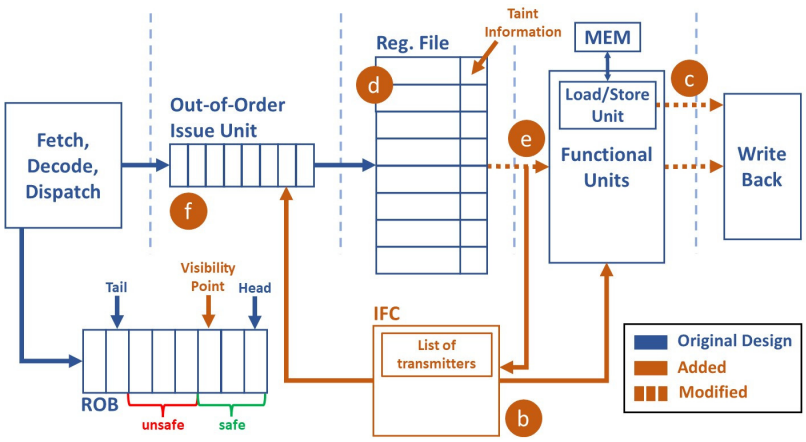
STT and DOLMA showed that dynamic information flow tracking has the potential for enabling secure speculation with low performance overhead. This is due to the fact that it allows the microarchitecture to speculate more freely as long as transiently accessed data is not able to leak via a side channel. However, achieving maximum performance with these approaches without compromising security is challenging, since the designer needs to manually inspect many scenarios to identify all possible side channels. This process can be susceptible to errors and may lead to security gaps.

Our design methodology borrows ideas from STT and DOLMA to build a generic control infrastructure for security. However, it replaces the manual inspection with a formal UPEC analysis, as described in Sec III-D. The infrastructure enables the designer to enforce different information flow policies in a systematic way and avoids ad-hoc local patches.

Fig. 3 shows how our methodology augments a standard out-of-order execution pipeline with a generic control infrastructure for security. It consists of a generic tainting infrastructure and an information flow controller (IFC).

1) Generic Tainting Infrastructure: Every register in the general-purpose register file is instrumented with an additional field for taint information. In-flight instructions in the execute stage have an additional attribute for the taint information regarding their destination register. This information includes a single bit denoting the taint which is set and cleared based on taint and untaint rules. These rules rely on the *visibility point* [6] in the Re-Order Buffer (ROB), which is defined as the youngest non-speculative instruction, depending on the threat model. An unsafe load instruction, i.e., a load instruction which lies between the visibility point and the ROB tail, taints its destination register in the register file (Fig. 3c). Subsequently, any instruction that has a tainted operand sets the taint bit for its destination register (Fig. 3e). Based on the taint rules, the effect of speculatively accessed data is tracked across the pipeline over the course of instruction execution.

For minimum performance penalty, the output of speculative load instructions must be untainted as soon as they are proven non-transient, i.e., when they pass the visibility point in the ROB. This can be challenging to implement, as retracing instruction dependencies through the ROB is difficult. Tracking the youngest root of taint (YRoT), i.e., the youngest instruction that initiated the taint bit, simplifies this problem by removing the chaining of dependencies [6]. Therefore, the taint information of the register file and of in-flight instructions also have a field that stores an identifier pointing to the YRoT. This identifier, for example, can be the index of the ROB entry for the corresponding instruction. In case of a load instruction, the YRoT of the destination register is set to the identifier of the load instruction itself (Fig. 3c). For any other instruction, the YRoT of the destination register is set as the youngest YRoT among the tainted operands (Fig. 3e). Using the YRoTs, the register file clears the taint



(a) Generic Tainting Infrastructure and IFC

```
interface io.FU(
  input   inst_opcode
  input   rd.taint
  input   ROB_idx
  output  kill_signal
)
interface io.issue_unit(
  output  wait
  output  wait_idx
  output  wait_YRoT
)

for each FU in funtional_units{
  if (is_transmitter(FU.inst_opcode)
      && FU.rd.taint){
    FU.kill_signal = 1
    issue_unit.wait = 1
    issue_unit.wait_idx = FU.ROB_idx
    issue_unit.wait_YRoT = FU.rd.
     YRoT
  }
}
```

(b) IFC Module

```
// load-store unit to write back
// interface
if (is_unsafe(load.ROB_idx)){
  rd.taint = 1
  rd.YRoT = load.ROB_idx
}
```

(c) Tainting

```
for each r in register_file{
  if (is_safe(r.YRoT)){
    r.taint = 0
  }
}
```

(d) Untainting

```
// example for a FU with two source
// registers
if (rs1.taint || rs2.taint){
  rd.taint = 1

  if (rs1.taint && !rs2.taint){
    rd.YRoT = rs1.YRoT
  }
  if (!rs1.taint && rs2.taint){
    rd.YRoT = rs2.YRoT
  }
  if (rs1.taint && rs2.taint){
    rd.YRoT = youngest(rs1.YRoT, rs2
     .YRoT)
  }
}
```

(e) Taint Propagation

```
for each slot in issue_unit{
  if(IFC.wait && IFC.wait_idx ==
     slot.ROB_idx){
    slot.state = WAIT
    slot.YRoT = IFC.wait_YRoT
  }
  if(slot.state == WAIT && is_safe(
     slot.YRoT){
    slot.state = READY
  }
}
```

(f) Issue Unit

Fig. 3: Generic Control Infrastructure for Security

bit of any register as soon as its YRoT points to a safe instruction (Fig. 3d).

It should be noted that in this paradigm, the data memory does not need to be tainted manually since any transient memory access is considered confidential information until the load instruction becomes non-transient.

2) Information Flow Controller: The proposed information flow controller (IFC) selectively inhibits execution of certain in-flight instructions without stalling the entire pipeline. Our iterative design methodology (Sec. III-D) utilizes the IFC to implement information flow policies in a semi-automated way.

The pseudo code for the IFC module is described in Fig. 3b. The IFC receives the opcode, ROB index and taint information of the in-flight instruction in every functional unit. It is equipped with a list of transmitters, i.e., instructions capable of forming a side channel. The function *is_transmitter()* checks the opcode of every in-flight instruction against the list of transmitters. During runtime, if there is any tainted in-flight transmit instruction, the IFC sends a *kill* signal to the corresponding functional unit to prevent the execution of the transmitter. The IFC also notifies the issue queue with a *wait* signal and sends the necessary information to re-issue the instruction if it becomes safe. The designer does not need to set up the list of transmitters manually, and only needs to implement the IFC with an empty list. The list will then be generated and refined through the iterative design methodology by formal analysis.

In case of functional units with multi-cycle execution, the IFC kills transmit instructions at the beginning of the execution phase. Therefore, there is no need to incorporate complex logic to abort multi-cycle operations in the middle of their execution. This significantly simplifies the integration of the IFC into the pipeline. In the unlikely event that transmitters exist that leak prior to the execution stage [36], we rely on the exhaustive analysis of UPEC to detect them. In this case, a local fix can be applied.

The issue queue processes the *wait* command from the IFC. Each issue slot compares the ROB index of its instruction with the ROB index of the instruction killed by the IFC. If they match, the slot saves the YRoT that corresponds to this instruction and goes to the WAIT state. As soon as the YRoT becomes safe, the slot goes back to the READY state and re-issues the instruction according to the existing scheduling logic (Fig. 3f).

It should be noted that the design flow is inherently robust against design mistakes in the infrastructure. The UPEC formal analysis can discover any bug in the introduced design instrumentation that violates the formalized security target.

Unlike mitigations proposed in the literature [6], [7], [37], [18], [19], [20] which require a security engineer integrating patches across a complex pipeline, our approach creates a generic and centralized security infrastructure. Designing this infrastructure does not require any in-depth security knowledge since only the implementation of the visibility point depends on the threat model. Therefore, it creates a separation of concerns by decoupling the tool-based security analysis from the manual design tasks which are straightforward.

D. Iterative Design Methodology

Fig. 4 shows the proposed iterative design flow for the secure-by-construction methodology. The basic idea is to interleave formal

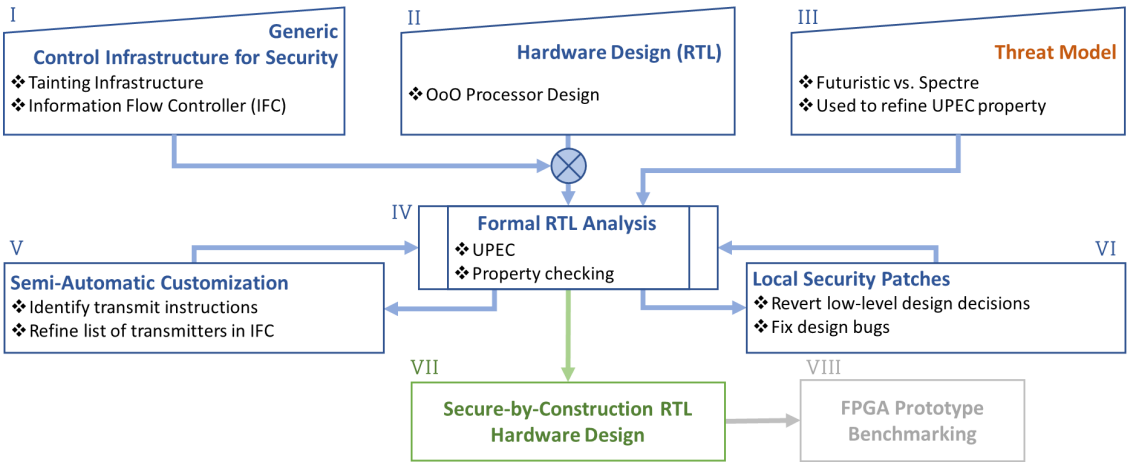


Fig. 4: Secure-by-Construction Design Flow

security verification by UPEC with the design steps and iteratively verify and patch the design. In each iteration, UPEC pinpoints a vulnerability by producing an execution trace that exposes a microarchitectural timing side channel. The trace points to a transmit instruction whose opcode is then added to the list of transmitters in the IFC. Through these iterations, the list of transmitters in the IFC is refined in a semi-automatic way such that the IFC blocks any instruction leaking tainted data. In the last iteration, UPEC certifies the design to be secure w.r.t. the threat model. This interleaving of incremental design and verification steps removes the need for targeting security in a separate verification flow. The starting point of the methodology is the RTL implementation of an out-of-order microarchitecture (Box II in Fig. 4). The designer needs to select the desired threat model w.r.t. TES attacks (Box III in Fig. 4), as described in Sec. III-A. The RTL implementation is then instrumented with the generic control infrastructure for security (Box I in Fig. 4), according to Fig. 3. The list of transmitters in the IFC is initially empty and the IFC does not stall any instructions. The threat model is also used to refine the macro *secret_load_transient()* in the UPEC property (cf. Fig. 2).

The iterative design flow begins with verifying the initial design with UPEC (Box IV in Fig. 4). The proof, most likely, points to a TES vulnerability and shows an execution trace describing the propagation and leakage of the secret from the memory to the architectural registers. The trace consists of a sequence of instructions that use transiently accessed data. The designer now needs to identify the instruction being the culprit for the side channel. This is easy because the SAT solver always finds the shortest possible trace as a counterexample [38] and therefore, the sequence does not contain any irrelevant instructions. Any instruction after the transient load is crucial for the side channel and blocking it breaks the propagation chain. In the majority of cases, selecting the first instruction right after the transient load is sufficient. Once the transmitter has been identified, its opcode is added to the list of transmitters in the IFC (Box V in Fig. 4).

In some cases, the designer may conclude from the counterexample that a local patch can achieve security without performance overhead (Box VI in Fig. 4). For example, if the side channel exists due to specific RTL design decisions or even design bugs, reverting these decisions may remove the side channel without impacting the performance. Furthermore, the formal analysis also points out any mistake in the implementation of the generic control infrastructure for security. It is easy to identify counterexamples pointing to such a bug, because the counterexample will show a trace with a transmit instruction that is already in the list of transmitters. Such an execution trace indicates that the taint propagation is not implemented properly or the IFC is not integrated correctly.

Once the detected gap is patched, the design is re-verified. This process is continued until the solver can no longer produce a counterexample and therefore certifies the design to be secure w.r.t. the threat model (Box VII in Fig. 4).

Thanks to the exhaustive proofs of UPEC, the final design is enhanced to not miss any side channel, even previously unknown ones. Furthermore, since the counterexamples only point to the instructions that are absolutely necessary for forming a side channel, the design avoids blanket fixes and unnecessary performance overheads. The process completely relieves the designer from any *a priori* knowledge about side channels, and prevents any mistakes due to misunderstanding the counterexamples. For example, if the designer selects the wrong instruction as a transmitter, this will be detected since the UPEC proof will return the same counterexample in the next iteration. More importantly, the UPEC property is agnostic to the underlying information flow infrastructure and therefore detects any bug in this part of the design that can lead to information leakage. This is crucial for designing a secure microarchitecture, since, as our case study in Sec. IV shows, implementing information flow tracking schemes can be prone to design errors.

The main advantage of the proposed flow lies in the fact that it enables designers to embark on more aggressive optimizations without risking the security of the design. The final product can be benchmarked using simulation / FPGA emulation (Box VIII in Fig. 4) and the designer can optimize different parts of the design to achieve the desired performance. Any such design update within or after the iterative flow is checked by UPEC to evaluate the security implications of the optimizations. This is particularly important for out-of-order microarchitectures, where designers need to employ a variety of techniques to increase throughput while minimizing the critical path of the design.

IV. Case Study on BOOM

To show the feasibility of our design methodology, we present a proof-of-concept mitigation of all TES vulnerabilities according to the threat models defined in Sec. III-A in BOOMv3 [1].

A. Overview on BOOM

The Berkeley Out-of-Order Machine (BOOM) [11] is an open-source superscalar out-of-order core, designed and maintained by

[1] https://github.com/RPTU-EIS/SecureBOOM

the UC Berkeley Architecture Research Group. The core's third major release, BOOMv3, implements the RV64GC variant of the RISC-V ISA. Since the medium configuration of BOOM used in our work delivers performance comparable to the ARM Cortex A9 core, the case study shows that the proposed approach is feasible for industrial-grade processor designs. Spectre-style attacks are possible in BOOMv3 and no mitigation has been implemented yet [39]. With respect to Meltdown, measures have been taken in BOOMv2 to prevent this kind of attack.

B. Implementing the Generic Control Infrastructure for Security

Following the design flow described in Sec. III, we begin the case study with implementing the generic control infrastructure for security (Fig. 3) in the existing BOOMv3 design. We created two different versions of BOOMv3, one for the *futuristic* and one for the *spectre* threat model. For the visibility point in the *futuristic* model, we need to identify the youngest non-transient instruction in the ROB, i.e., the youngest instruction that is bound to commit. The straightforward solution would be to simply consider the ROB head as the visibility point. However, this is overly conservative and incurs high performance overhead. The other possibility is to utilize the *Point of No Return (PNR)* pointer existing in the ROB of BOOMv3. The PNR runs ahead of the ROB head and indicates the oldest instruction w.r.t. program order that can be squashed due to misprediction or an exception. All the instructions between PNR and ROB head will definitely commit and can therefore be considered as non-transient (safe). In our design for the futuristic model, we use the PNR for defining safe and unsafe instructions and setting and clearing taint bits. To this end, the *is_safe()* and *is_unsafe()* functions of Fig. 3 are refined by comparing the given ROB index to the PNR. As the PNR moves forward, the untainting logic clears the taint bit of every register whose YRoT becomes safe (Fig. 3d).

For the *spectre* model, the design of BOOMv3 allows for a different approach for defining safe and unsafe instructions. Each in-flight instruction in BOOMv3 has a *branch mask* which represents the number of unresolved older branches or jumps. Each bit of the branch mask corresponds to a specific in-flight branch or jump. The BOOMv3 design utilizes these masks to resolve the speculations as soon as a branch is executed. If a branch is predicted correctly, the corresponding bit in all branch mask registers is cleared. The branch mask allows us to implicitly define the visibility point for BOOMv3 under the *spectre* model. Any instruction with a branch mask equal to zero is safe under the *spectre* threat model and therefore the *is_safe()* and *is_unsafe()* functions are refined by checking the corresponding branch mask of the given ROB index. If a load instruction is on a correctly speculated path, the core clears its branch mask bit by bit as the older jump and branch instructions resolve. Consequently, the untainting logic clears all the taint bits that are initiated by this load.

Re-using the branch mask and the PNR for defining safe and unsafe instructions significantly simplifies the implementation of the generic control infrastructure for security, since the designer does not need to implement a separate pointer in the ROB for the visibility point.

The rest of the generic control infrastructure for security is implemented in BOOM similar to Fig. 3.

C. Formal Analysis Setup

The formal security analysis by UPEC is at the core of our presented design methodology. The first step in setting up the formal UPEC proof is to generate the UPEC computational model for the BOOMv3 design, similar to Fig. 2. In this model, we soundly blackbox [10] modules that have a large number of state bits, such as memory banks inside the data cache, to improve the scalability of the proofs without introducing verification gaps. Furthermore, a copy of the secret also resides in the data cache, which enables the solver to detect secret leakage to architectural registers with counterexamples of shorter temporal length. The UPEC property and its constraints and invariants are created and refined based on the verification methodology described in [10].

Once the property and the model are set up, the formal analysis can be started. UPEC allows to decompose the security verification into a set of smaller proofs such that each of them analyzes secret propagation to different microarchitectural state variables. The solver exhaustively considers every resulting scenario and detects all state variables to which the secret can propagate. If a propagation to an architectural register is detected, it is marked as a leakage alert (L-alert).

In our case study on BOOMv3 multiple leakage alerts are detected, and they can be attributed to one of the following factors:

1) A new transmit instruction is detected that can form a transient execution side channel. The IFC must be refined to block the execution of this instruction when it is tainted. The designer may also opt for a local patch, if the timing side channel is due to a design mistake rather than a feature.
2) There is a bug in the taint logic, e.g., the taint is not set, propagated or cleared correctly. This is the case when the counterexample points to a transmit instruction that is already added to the list of transmitters in the IFC. The designer has to manually fix the information flow tracking logic.

D. Iterative Design Methodology for BOOMv3

The experiments described in this subsection evaluate the transmit instructions for both the *futuristic* and the *spectre* threat model. The first transmit instructions identified by our formal analysis are the *branch*, *jump and link register*, and *load* instructions. These instructions are added to the list of transmitters so that the IFC blocks their execution unless the corresponding taint bit is zero. The side channels through these instructions have also been previously identified in BOOM [10] as well as in x86 processors [6], [7]. Therefore, designers may opt to consider them as transmit instructions before the first iteration of formal analysis.

The next iteration of formal analysis on the updated design produced a counterexample that pointed to a bug in the tainting infrastructure. A subtle bug in the taint propagation logic caused tainted operands not to be propagated to the destination register for a specific type of instruction. This bug and a few other scenarios that were pointed out by the formal analysis show that implementing dynamic information flow tracking is highly prone to error.

The next round of formal analysis showed that address translation of the store instructions leaks the address through the translation lookaside buffer (TLB) and can form a transient execution side channel. A similar attack pattern has also been reported in [7]. It should be noted that in BOOMv3 store instructions are decomposed into two operations, namely store address (STA) and store data (STD). The (micro-)opcode associated with the store address operation translating the address of the store is added to the list of transmitters in the IFC.

Independent of our tainting infrastructure, our case study showed that the design updates implemented with BOOMv3 re-introduced a Meltdown vulnerability which had been securely patched in BOOMv2 [39]. Page faults and address misalignment errors raise exceptions (signal (1) in Fig. 5), however the corresponding load executes and sends the data back to the core, forming a transient execution side channel. After discovering this issue, we informed the development team who confirmed the gap. To mitigate the detected Meltdown vulnerability, we implemented a local patch depicted as

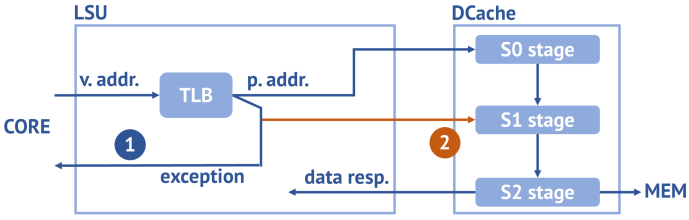


Fig. 5: Load-Store Unit in BOOM and implemented Meltdown mitigation

TABLE I: BOOMv3 Design Refinements

List of Transmitters	Local Security Patches
LB, LBU, LH, LHU, LW, BEQ, BNE, BGE, BGEU, BLT, BLTU, JALR, STA (for all store instructions), DIV, DIVU, DIVW, DIVUW, REM, REMU, REMW, REMUW, FDIV.S, FDIV.D, FSQRT.S, FSQRT.D	Meltdown Patch, bug fixes in the control infrastructure for security

signal (2) in Fig. 5. We used the existing *kill* signal for the S1 stage in the data cache to abort faulty loads from reaching the cache array in the S2 stage. With this mechanism, given that the cache interface is fully pipelined, killing faulty loads cannot have any effect on the scheduling of memory requests to the data cache. Although a faulty load occupies the data cache port for one clock cycle, the port is available again in the following cycle regardless of whether or not the load was aborted in the S1 stage.

Further iterations of the design methodology discovered multiple new transmit instructions which could only be detected in the cycle-accurate RTL implementation of the system. In BOOMv3, the *Integer Division Unit* and the *Floating Point Divide and Square Root Unit* have operand-dependent timing behavior. Instructions running on these functional units may, therefore, leak their operands through resource contention and act as transmit instructions. The opcodes of these instructions are added to the list of transmitters in the IFC.

In addition to detecting transmit instructions and other security vulnerabilities, our proposed design methodology also provides the opportunity for performance optimizations without the risk of creating new side channels or weakening the implemented security features. Through the course of this case study, we implemented two optimizations to speed up the performance of our security infrastructure. The first measure was to bypass a buffer in the issue queues to untaint safe instructions one clock cycle earlier. Furthermore, the PNR implemented in the baseline BOOMv3 design acted overly conservative regarding load instructions. A load operation in the ROB was only marked as safe after it received its data from the cache. Consequently, in case of a cache miss, the PNR was stuck for a relatively long time until this operation completed. We changed the PNR logic to advance as soon as the address of a load has been successfully translated and all older loads and stores in the ROB have already been acknowledged by the data cache. At this stage the load is no longer squashable and the PNR can be advanced safely.

Tab. I summarizes the refinements done by the proposed design methodology in our case study on BOOMv3. It includes the list of transmitters in the IFC that is created by the iterative flow in a semi-automatic manner, as well as the local patches that are guided by the counterexamples of the formal analysis. It should be noted that the transmitters are the same in both threat models. The difference between the *spectre* and *futuristic* threat model lies in the visibility point that defines when to taint and untaint information.

TABLE II: BOOMv3 Configurations

	Small	Medium	Large	Mega
Decode Width	1	2	3	4
ROB Entries	32	64	96	128
LD/ST Queue Entries	8/8	16/16	24/24	32/32
Total Issue Slots	24	48	72	96
CoreMark/MHz	2.28	3.66	4.45	4.96

In our case study the iterative design methodology rendered a secure design after 12 iterations. If an iteration of the design flow identified an instruction as a transmitter, other variants of the same instruction, e.g., signed/unsigned or single-/double-precision, were also added to the list of transmitters in the same iteration. It should be noted that UPEC decomposed the formal analysis into several proofs that can run in parallel. For BOOMv3 with the medium configuration 329 properties were generated by UPEC. Certifying the security of the core requires proving all properties. However, a leakage alert is usually detected after checking only a small subset of the properties. Each property check in the final iteration took 2 hours on average. The overall runtime of the security analysis depends heavily on the available computational resources. The memory usage of a single proof was 25 GB on average with a maximum of 33 GB. Each property was proven using the commercial property checker OneSpin 360 DV, on an Intel Xeon Gold 6234 CPU at 3.3 GHz running Ubuntu 18.04.

E. Performance Evaluation

To evaluate the impact of the implemented security features, we synthesized our verified design for an FPGA target and ran emulations of the secure core's behavior. In addition to the BOOMv3 design variants created by our design methodology for the two threat models, *SecureBOOM Futuristic* and *SecureBOOM Spectre*, we also implemented and emulated more straightforward fixes targeting transient execution of all load instructions. The simplest fix against TES is to block any speculative access to the data memory. Hence, we implement *naïve delay* [15] delaying all load instructions until they reach the head of the ROB. A less prohibitive version of the same principle is referred to as *eager delay* [15]. In this design variant, load instructions are only delayed until they reach the visibility point, depending on the threat model (cf. Sec. III-A).

We evaluate CPU performance by running the SPEC CPU 2006 [40] benchmark suite with the reference inputs. Since running all workloads on the original BOOMv3 design on the FPGA takes more than two weeks of wall-clock time, we selected the 15 benchmarks that pose the lightest workload in terms of the runtime on the original BOOMv3 design. These benchmarks consist of eight integer and seven floating-point workloads. We benchmark the standard medium configuration of BOOMv3 (cf. Tab. II) on an AMD Virtex UltraScale+ VCU118 at 75MHz.

The execution times normalized to the runtime on the insecure baseline design are depicted in Fig. 6. As expected, *naïve delay* is the worst-performing secure design with an overhead of roughly 2× on average for the tested benchmarks. In the futuristic threat model, *eager delay* comes with an overhead of 84.6 % whereas *SecureBOOM Futuristic* reduces it to 36.0 %. *SecureBOOM Spectre* is only 5.2 % slower than the insecure baseline design. The overhead for *eager delay* in this threat model is 20.9%.

Tab. III shows the hardware overhead of the implemented designs in terms of FPGA utilization compared to the insecure baseline design. *SecureBOOM Spectre* has a higher look-up table and flip-flop utilization than *SecureBOOM Futuristic* due to the fact that in

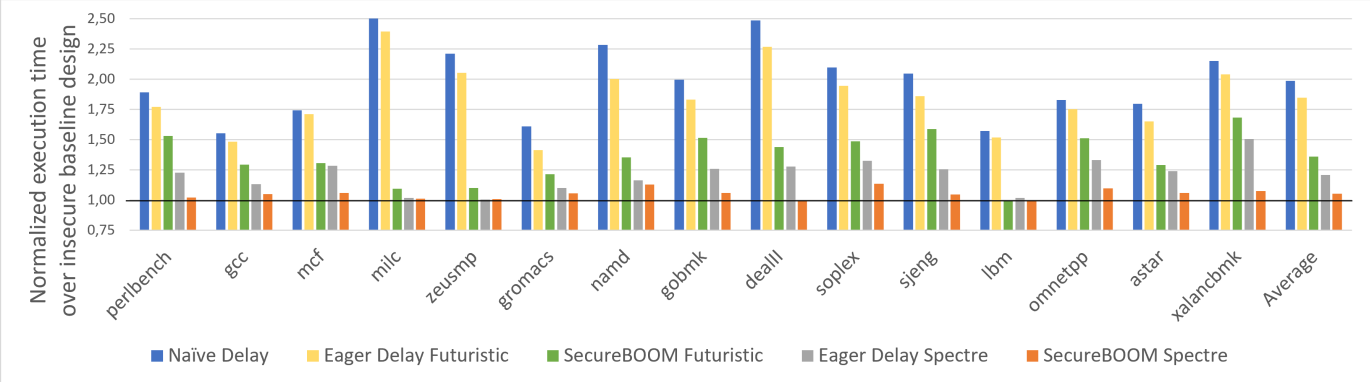


Fig. 6: SPEC CPU 2006 results for secure BOOMv3 variants in the medium configuration

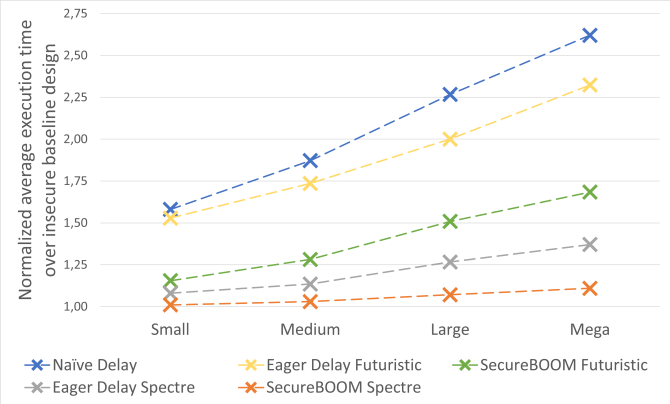


Fig. 7: SPEC CPU 2006 average normalized overheads for different BOOMv3 configurations

TABLE III: Hardware Overhead Compared to the Baseline Design

	Look-Up Tables	Flip-Flops
SecureBOOM Spectre	15.7 %	6.2 %
SecureBOOM Futuristic	8.2 %	3.3 %

the medium configuration of BOOMv3, the YRoT based on the 12-bit branch mask occupies more logic elements than the YRoT based on the 6-bit ROB index.

Fig. 7 shows the average performance overhead of the secure design variants for different configurations of BOOMv3 as listed in Tab. II. In this experiment, we used the test workload of the SPEC CPU 2006 benchmark suite. The goal of this experiment is to evaluate the performance degradation of different secure microarchitecture schemes w.r.t. the complexity of the pipeline. Our results suggest that all examined schemes cause more performance overhead on more complex pipelines. However, within each threat model the gap between the simple fixes and the *SecureBOOM* designs is widening with increasing pipeline complexity. Wider and deeper pipeline designs provide higher instruction-level parallelism (as indicated by the CoreMark [41] scores in Tab. II). This is partly thanks to larger time windows for speculation and out-of-order execution. As a result, conservative measures like *naïve* and *eager delay* stall more instructions for a longer time as the complexity of the underlying pipeline grows. On the other hand, *SecureBOOM*, as produced by our methodology, allows the pipeline to benefit from the speculative execution of certain instructions that are otherwise blocked by the *naïve* and *eager delay* designs.

It should be noted that the numbers reported in this case study are provided by FPGA emulation. In this set-up, the processor frequency is lower compared to taped-out designs (75 MHz vs. 2 GHz), while the DRAM runs at a much higher frequency of 1,666 MHz. Due to this large frequency gap between core and memory, memory access becomes less of a bottleneck and a cache miss becomes less expensive compared to a taped-out processor. Less penalty for a cache miss means that delaying the loads and subsequent cache refills in *naïve* and *eager delay* incur less overhead because the memory is relatively faster in the FPGA setup. Therefore, these protection schemes may cause an even higher overhead in a taped-out processor.

V. Discussion and Conclusion

This paper proposes a secure-by-construction RTL design methodology for out-of-order execution processors with guaranteed security against TES attacks. The proposed design flow employs formal analysis by UPEC to detect all instructions that can form transient execution side channels and then blocks the information leakage with the help of a new hardware protection framework based on a generic control infrastructure. This approach replaces the error-prone manual inspection of the design for identifying side channels with a systematic formal analysis. Our methodology does not require any drastic change to the industrial practice and works with standard and commercially available languages and verification tools.

The extensive case study on BOOMv3 provides novel insights for the computer architecture community. Our design flow identified multiple new transmit instructions which were introduced by the specific RTL implementation of the design and cannot be identified on any more abstract model. In addition, the case study shows that implementing secure speculation schemes tends to be susceptible to design mistakes. These findings attest to the fact that there is a need for a secure-by-construction design methodology. The presented methodology provides the opportunity for aggressive and fine-grained design optimization, which is demonstrated in Sec. IV. Furthermore, this paper discusses the impact of various factors, such as the threat model and the underlying pipeline complexity, on the performance overhead.

The proposed design flow is compatible with more advanced design optimizations, such as speculative data-oblivious execution [42] and value prediction [15]. Future work will explore the implementation and verification challenges of these techniques at the RTL.

Acknowledgements

The reported research was partly supported by DFG SPP *Nano Security* and by the Intel Corp. *Scalable Assurance Program*.

REFERENCES

[1] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019.

[2] 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*, 2018, pp. 973–990.

[3] M. Behnia, P. Sahu, R. Paccagnella, J. Yu, Z. N. Zhao, X. Zou, T. Unterluggauer, J. Torrellas, C. Rozas, A. Morrison, F. Mckeen, F. Liu, R. Gabor, C. W. Fletcher, A. Basak, and A. Alameldeen, "Speculative Interference Attacks: Breaking Invisible Speculation Schemes," in *Proc. of the 26th ACM Intl. Conf. on Arch. Support for Programming Languages and Operating Systems*, 2021, pp. 1046–1060.

[4] J. S. Vicarte, M. Flanders, R. Paccagnella, G. Garrett-Grossman, A. Morrison, C. W. Fletcher, and D. Kohlbrenner, "Augury: Using data memory-dependent prefetchers to leak data at rest," in *2022 IEEE Symp. on Security and Privacy*. IEEE Computer Society, 2022, pp. 1518–1518.

[5] L. Bowen and C. Lupo, "The performance cost of software-based security mitigations," in *Proc. of the ACM/SPEC International Conference on Performance Engineering*, 2020, pp. 210–217.

[6] 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 *Proc. of the 52nd Annual IEEE/ACM Intl. Symposium on Microarchitecture*, 2019, pp. 954–968.

[7] K. Loughlin, I. Neal, J. Ma, E. Tsai, O. Weisse, S. Narayanasamy, and B. Kasikci, "DOLMA: Securing speculation with the principle of transient non-observability," in *30th USENIX Security Symposium*, 2021.

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

[9] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH Comp. Arch. News*, vol. 39, pp. 1–7, 2011.

[10] 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, 2023.

[11] 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*, May 2020.

[12] M. Taram, A. Venkat, and D. Tullsen, "Context-Sensitive Fencing: Securing Speculative Execution via Microcode Customization," in *Proc. of the 24th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 395–410.

[13] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. W. Fletcher, and J. Torrellas, "InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy," in *51st Annual IEEE/ACM Intl. Symposium on Microarchitecture*, 2018, pp. 428–441.

[14] Y. Wu and X. Qian, "ReversiSpec: Reversible Coherence Protocol for Defending Transient Attacks," *arXiv:2006.16535*, 2020.

[15] C. Sakalis, S. Kaxiras, A. Ros, A. Jimborean, and M. Själander, "Efficient Invisible Speculative Execution through Selective Delay and Value Prediction," in *Proc. of the 46th Intl. Symp. on Computer Architecture*, 2019, pp. 723–735.

[16] P. Li, L. Zhao, R. Hou, L. Zhang, and D. Meng, "Conditional Speculation: An Effective Approach to Safeguard Out-of-Order Execution Against Spectre Attacks," in *IEEE Intl. Symp. on High Performance Computer Architecture*, 2019, pp. 264–276.

[17] M. Guarnieri, B. Köpf, J. Reineke, and P. Vila, "Hardware-Software Contracts for Secure Speculation," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 1868–1883.

[18] J. Fustos, F. Farshchi, and H. Yun, "SpectreGuard: An Efficient Data-Centric Defense Mechanism Against Spectre Attacks," in *Proc. of the 56th Annual Design Automation Conference*, 2019, pp. 1–6.

[19] M. Schwarz, M. Lipp, C. Canella, R. Schilling, F. Kargl, and D. Gruss, "ConTExT: A Generic Approach for Mitigating Spectre," in *Proc. Network and Distributed System Security Symposium.*, 2020.

[20] L.-A. Daniel, M. Bognar, J. Noorman, S. Bardin, T. Rezk, and F. Piessens, "ProSpeCT: Provably Secure Speculation for the Constant-Time Policy (Extended Version)," *arXiv preprint: 2302.12108*, 2023.

[21] O. Oleksenko, B. Trach, M. Silberstein, and C. Fetzer, "SpecFuzz: Bringing Spectre-Type Vulnerabilities to the Surface," in *29th USENIX Security Symposium*, 2020, pp. 1481–1498.

[22] M. Ghaniyoun, K. Barber, Y. Zhang, and R. Teodorescu, "INTROSPECTRE: A Pre-Silicon Framework for Discovery and Analysis of Transient Execution Vulnerabilities," in *ACM/IEEE 48th Annual Intl. Symp. on Computer Architecture*, 2021, pp. 874–887.

[23] S. K. Muduli, G. Takhar, and P. Subramanyan, "HyperFuzzing for SoC Security Validation," in *Proc. of the 39th Intl. Conference on Computer-Aided Design*, 2020, pp. 1–9.

[24] P. Subramanyan and D. Arora, "Formal Verification of Taint-Propagation Security Properties in a Commercial SoC Design," in *Design, Automation & Test in Europe Conf. (DATE)*, 2014, pp. 313–314.

[25] 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.

[26] 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 *Intl. Conf. on Codes, Cryptology, & Information Security*, 2019, pp. 462–479.

[27] 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 Conf. (DATE)*, 2019, pp. 994–999.

[28] M. R. Fadiheh, J. Müller, R. Brinkmann, S. Mitra, D. Stoffel, and W. Kunz, "A formal approach for detecting vulnerabilities to transient execution attacks in out-of-order processors," in *IEEE/ACM Design Automation Conference (DAC)*, 2020, pp. 1–6.

[29] D. Mehmedagić, M. R. Fadiheh, J. Müller, A. L. Duque Antón, D. Stoffel, and W. Kunz, "Design of Access Control Mechanisms in Systems-on-Chip with Formal Integrity Guarantees," in *2023 USENIX Security Conference*, 2023 (To Appear).

[30] L. Deutschmann, J. Müller, M. R. Fadiheh, D. Stoffel, and W. Kunz, "Towards a Formally Verified Hardware Root-of-Trust for Data-Oblivious Computing," in *Proc. of the 59th ACM/IEEE Design Automation Conference*, 2022, pp. 727–732.

[31] J. Müller, M. R. Fadiheh, A. L. Duque Antón, T. Eisenbarth, D. Stoffel, and W. Kunz, "A Formal Approach to Confidentiality Verification in SoCs at the Register Transfer Level," in *58th ACM/IEEE Design Automation Conference (DAC)*, 2021, pp. 991–996.

[32] 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:1801.01203*, 2018.

[33] M. Andrysco, D. Kohlbrenner, K. Mowery, R. Jhala, S. Lerner, and H. Shacham, "On Subnormal Floating Point and Abnormal Timing," in *IEEE Symposium on Security and Privacy*, 2015, pp. 623–639.

[34] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack," in *23rd USENIX Security Symposium*, 2014, pp. 22–25.

[35] J. Urdahl, D. Stoffel, and W. Kunz, "Path predicate abstraction for sound system-level models of RT-level circuit designs," *IEEE Trans. on Comp.-Aided Design of Integr. Circ. & Sys.*, vol. 33, pp. 291–304, 2014.

[36] J. R. S. Vicarte, P. Shome, N. Nayak, C. Trippel, A. Morrison, D. Kohlbrenner, and C. W. Fletcher, "Opening Pandora's Box: A systematic study of new ways microarchitecture can leak private data," in *ACM/IEEE 48th Intl. Symp. on Comp. Architecture*, 2021, pp. 347–360.

[37] M. Sabbagh, Y. Fei, and D. Kaeli, "Secure Speculation Execution via RISC-V Open Hardware Design," *Fifth Workshop on RISC-V for Computer Architecture Research*, 2021.

[38] E. Clarke, A. Biere, R. Raimi, and Y. Zhu, "Bounded Model Checking Using Satisfiability Solving," *Form. Methods Syst. Des.*, vol. 19, no. 1, pp. 7–34, 2001.

[39] C. Celio, J. Zhao, A. Gonzalez, B. Korpan, K. Asanovic, and D. Patterson, "BOOM - An open-source out-of-order processor," in *Chisel Community Conference*, 2018, URL: https://boom-core.org/docs/boom_processor_ccc18_celio.pdf, Acc. Date: 2021/07/13.

[40] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *SIGARCH Comput. Archit. News*, vol. 34, pp. 1–17, 2006.

[41] S. Gal-On and M. Levy, "Exploring CoreMark a Benchmark Maximizing Simplicity and Efficacy," *The Embd. Microproc. Benchmark Cons.*, 2012.

[42] J. Yu, N. Mantri, J. Torrellas, A. Morrison, and C. W. Fletcher, "Speculative Data-Oblivious Execution: Mobilizing safe prediction for safe and efficient speculative execution," in *ACM/IEEE 47th Intl. Symp. on Computer Architecture*, 2020, pp. 707–720.