POLITECNICO DI TORINO
Repository ISTITUZIONALE

Model-Checking Speculation-Dependent Security Properties: Abstracting and Reducing Processor Models for Sound and Complete Verification

Original
Model-Checking Speculation-Dependent Security Properties: Abstracting and Reducing Processor Models for Sound and Complete Verification / Cabodi, G., Camurati, P., Finocchiaro, F., Vendraminetto, D.. - In: ELECTRONICS. - ISSN 2079-9292. - 8:9(2019). [10.3390/electronics8091057]

Availability:
This version is available at: 11583/2753232 since: 2023-04-27T13:24:18Z

Publisher:
MDPI

Published
DOI:10.3390/electronics8091057

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

05 September 2026

electronics

MDPI

Article

Model-Checking Speculation-Dependent Security Properties: Abstracting and Reducing Processor Models for Sound and Complete Verification †

Gianpiero Cabodi, Paolo Camurati, Fabrizio Finocchiaro * and Danilo Vendraminetto

Dipartimento di Automatica e Informatica, Politecnico di Torino, 10129 Turin, Italy; gianpiero.cabodi@polito.it (G.C.); paolo.camurati@polito.it (P.C.); danilo.vendraminetto@polito.it (D.V.)

* Correspondence: fabrizio.finocchiaro@polito.it

† This paper is an extended version of our paper published in the Proceedings of the International Conference on Codes, Cryptology, and Information Security, Rabat, Morocco, 22–24 April 2019; pp. 462–479.

Received: 24 July 2019; Accepted: 16 September 2019; Published: 19 September 2019

Abstract: Spectre and Meltdown attacks in modern microprocessors represent a new class of attacks that have been difficult to deal with. They underline vulnerabilities in hardware design that have been going unnoticed for years. This shows the weakness of the state-of-the-art verification process and design practices. These attacks are OS-independent, and they do not exploit any software vulnerabilities. Moreover, they violate all security assumptions ensured by standard security procedures, (e.g., address space isolation), and, as a result, every security mechanism built upon these guarantees. These vulnerabilities allow the attacker to retrieve leaked data without accessing the secret directly. Indeed, they make use of covert channels, which are mechanisms of hidden communication that convey sensitive information without any visible information flow between the malicious party and the victim. The root cause of this type of side-channel attacks lies within the speculative and out-of-order execution of modern high-performance microarchitectures. Since modern processors are hard to verify with standard formal verification techniques, we present a methodology that shows how to transform a realistic model of a speculative and out-of-order processor into an abstract one. Following related formal verification approaches, we simplify the model under consideration by abstraction and refinement steps. We also present an approach to formally verify the abstract model using a standard model checker. The theoretical flow, reliant on established formal verification results, is introduced and a sketch of proof is provided for soundness and correctness. Finally, we demonstrate the feasibility of our approach, by applying it on a pipelined DLX RISC-inspired processor architecture. We show preliminary experimental results to support our claim, performing Bounded Model-Checking with a state-of-the-art model checker.

Keywords: model-checking; secure CPU architecture; speculative execution; taint propagation; abstraction and reduction; pipeline flushing; confidentiality; reorder buffer; spectre; meltdown

1. Introduction

Information security has gained increasing attention, over the last years, not only from the technical community but also from the general public. This trend is highly related to the awareness that (the lack of) security affects business, privacy and even health information. Actors involved in information security, on both sides, are using increasingly sophisticated methods and tools. Software is becoming more complex, hardware more compounded, services are growing in number. All these trends induce the attack surface to develop bigger and wider. While the main focus of cyberattacks is on software vulnerabilities, it is in the hardware spectrum that the challenge is becoming hard, due to

the intrinsic complexity, the incompleteness of simulation techniques, and the scalability issues of formal verification.

In this paper, we address scalability of formal verification techniques, by focusing on a recently emerged class of hardware vulnerabilities, related to speculative execution of instructions in microprocessors, such as Spectre [1] and Meltdown [2], that pinpoint and exploit vulnerabilities in the processor design jointly with known side channel attacks.

Security weaknesses of speculative executions are of great concern because of the intrinsic complexity of hardware designs and the inherent difficulty of their verification.

Spectre and Meltdown are clear examples of violations to two classes of key security requirements: confidentiality and integrity. Confidentiality safeguards that an unauthorized party cannot obtain sensitive information [3], whereas integrity is the assurance that the information is trustworthy and accurate (i.e., not tampered) [4].

Belonging to the class of side-channel attacks, they exploit the physical environment of a system to extract secret data (or information dependent on them) from its (externally visible) state, rather than exploiting flaws in the implementation (i.e., software bugs and buffer overflows).

Side channels, though actively researched [5–8] in recent years, were never considered to be big threats, as usually demanding very peculiar expertise and knowledge about the target. In the case of Meltdown and Spectre, data leakage can occur using simple exploits, and without any particular prior knowledge/expertise.

In the case of speculative execution, information can flow through side channels, based on a well-known gap between the concrete realization of the Instruction Set Architecture (ISA) and processor-specific speculation-related behaviors of high-performance MicroArchitectures (MA). Although registers and memory show no visible distinguishable architectural effect, side effects might happen at the microarchitectural level. It is this subtle discrepancy between MA and ISA [9] that Spectre/Meltdown rely on. So, while the ISA behavior is proven completely correct by verification and validation steps, the actual MA can attain certain states which could lead to potential threats, e.g., cache prefetched data exploited by established techniques [10–12].

In other words, as authors in [13] brilliantly stated: *"Spectre variants are a form of side-channel attack in which microarchitectural state, formerly intended to be isolated from retired execution becomes observable at an architectural level by an attacker program sharing resources with the victim. This state can include secrets loaded into shared architectural state including data referenced via speculation prior to completing access, validity, or bounds checks."*

The great impact of Meltdown and Spectre has not entirely been recognized yet [14]. Some mitigations have been undertaken but they tend to impact very heavily on performance. The reported magnitude of impact varies depending on the industry sector and expected workload characteristics [15]. Despite all the adopted countermeasures, the paramount question is still there: how do we prevent such attacks?

In this paper, we propose a novel approach to verifying security properties in pipelined out-of-order processors that can be applied to check speculation-based vulnerabilities.

The main idea is to use abstraction and reduction techniques to create a feasible model of the original design. The correctness of our approach is proved by resorting to well known literature on formal processor verification, such as model checking techniques, taint propagation, refinement of abstract models.

Though the approach is not yet automated, we describe the key steps and we employ a use case to show that it is viable: we perform it on a processor inspired by Hennessy and Patterson's DLX RISC processor [16]. The design has been enriched by adding pipelining and reorder buffer to be Tomasulo's algorithm compliant.

After the application of model transformations, which preserve functionality and correctness, we feed the final abstract model to a model checker.

The transformations could be summarized as follows:

- data abstraction,
- pipeline reduction,
- taint encoding,
- refinement.

The present work is an extended version of [17]. The main differences are:

- Section 5 has been revisited and expanded to better illustrate the underlying details of the approach.
- Model abstraction, taint propagation and verification have been expanded to better illustrate the underlying details of the approach.
- Tables and Figures have been added.
- Comments and explanations have been added to figures and tables.
- A full detailed example has been provided to show the feasibility of our approach.
- The correctness of the approach has been revisited and expanded: references to related state-of-the-art publications have been added.

The organization of the paper is the following:

- Section 2 provides state-of-the-art related work and background notions.
- Section 3 presents the processor architecture model.
- Section 4 gives a detailed description of an implementation of Spectre and Meltdown.
- Section 5 shows the verification approach we adopt on our case-study processor model.
- Section 6 produces experimental results to support the viability of our methodology.
- Section 7 proposes remarks and future directions in this field.

2. Preliminaries, Background, Related Works

We assume that the reader has basic knowledge of cybersecurity concepts, architecture of microprocessors, and formal verification. The notations used in this paper are defined whenever used for the first time.

In the following, we briefly overview some notions and related works that we deem important for the understanding of the subsequent sections.

2.1. Spectre and Meltdown Attacks

Speculative execution is a widely used performance improving feature in modern CPU designs. Within the context of speculative execution, the architectural state is intended to be largely unaffected by modifications at the microarchitectural level: if an instruction is not retired (not committed), e.g., due to branch misprediction, the processor is reverted to its previous state and execution restarts on the correct execution path. In a similar way, in the case of an exception handling, a pipeline flushing is enforced, and all the instructions occurred after the exception are cancelled, i.e., no architectural changes are made effective.

With out-of-order and speculative execution, an architecture allows many memory references to be issued but eventually aborted. However, incorrectly issued memory references may produce an indirect prefetching effect with consequent data transfers into the cache.

Although the attacker cannot directly access sensitive data, even when the cache holds it, the attacker can obtain indirect information depending on the secret, such as the memory addresses accessed by the victim.

Well-known side-channel techniques exist that, based on measuring cache access times, can identify previously accessed cache sets/pages. Meltdown and Spectre, for instance, use a special case of this kind of leakage. They build a covert channel, in order to transfer the microarchitectural state, which was modified by transient (i.e., not yet retired) instruction sequence, into the architectural state. At this point, specific cache timing attacks are employed, such as Flush+Reload [5]. Thus, the attacker

can recognize whether the monitored cache line was filled in with data, by simply measuring the access time.

Though side channels attacks have long been studied since the 1970s, only with Meltdown and Spectre it became clear that this class of attacks could be a potential threat for economic, privacy and security matters.

The full impact is not known at the time of writing. What we know is that there are two primary ways Meltdown and Spectre could impact business policies: increased risk of cyberattacks targeting sensitive data and a decrease in performance resulting from patches.

2.2. Formal Verification of Microprocessors with Out-of-Order Execution

Processors have always represented a serious challenge for design verification tools. Within this field, formal verification potentially offers a high degree of preciseness and automatic procedures. When complete, it can prove the correctness of a design.

Formal verification of processors is covered by a vast literature, ranging from more automated (yet poorly scalable) techniques, such as Model-Checking [18,19], to Theorem Proving systems that, though more powerful and complete, typically need much more manual (expert) work.

Most formal processor verification approaches tackle scalability (state explosion) by resorting to a couple of model transformations:

- model reduction: a form of case split, where only a properly selected subset of possible execution traces is considered;
- data abstraction: the model behavior is over-approximated by (partially) removing/transforming data (deemed) unnecessary to the proof; assumptions have to be made so that the soundness of the approach is guaranteed, e.g., arithmetic and logic functionalities are already verified. Possible refinement steps are needed, whenever the abstraction is unsound.

Theorem proving is the most general approach, virtually able to deal with more convoluted hardware designs and, in general, it is not limited to a specific configuration, and it can prove arbitrary processor configurations. Yet, as shown in the literature [20,21], developing proofs for real world designs tends to be very labour intensive.

Within the field of pipelined processor verification, theorem proving can be adapted and partially automated, thus leading to approaches that could even be considered to be generalizations of model checking.

Completion functions, used in [22], specify the desired effect of unfinished instructions on the pipeline. They can be viewed as a map between any out-of-order (OOO) processor state and a flushed state on the architectural side. The method leads to modularized and layered proofs that can be managed independently and used as hints for subsequent proofs. The authors in [23] have applied them to the verification of a Tomasulo compliant OOO processor.

Though partially automatic, the approach requires the user to manually define a set of completion functions, one per unfinished instruction and to manually define a way to compose them, in order to form the abstraction of the processor.

Burch and Dill [24] use a similar flushing technique in conjunction with the notion of uninterpreted functions. They make an abstraction of the behavior of a processor using uninterpreted function symbols, then verification is done by symbolic execution. To avoid state space explosion, they decompose the verification problem into three subproblems, and require the user to provide some extra control inputs. Informally, the first property states that the implementation correctly executes instructions from a flushed state; the second confirms that stalling does not affect the specification state; and the third checks that instructions are fetched correctly. One of the advantages of this approach is that it can be used to verify a hardware system without knowing the concrete implementation details. Another great benefit is that it is easily automated. Yet, special decision procedures for uninterpreted

function symbols are needed. Another limitation is the application to pipelined processors only, without any out-of-order feature.

Skakkebaek et al. [25] introduce a two part approach. First, the implementation is modified to derive an in-order abstraction; then, by exploiting domain-specific knowledge, they define a functional equivalence relation between the out-of-order implementation and the abstraction. Second, they prove that the abstraction is functionally equivalent to the ISA via a technique called incremental flushing, which is based on the Burch-Dill automatic flushing approach. Although the proposed method effectively applies to out-of-order processors, it heavily relies on human-guided theorem proving.

McMillan [26] verified Tomasulo's algorithm for out-of-order execution, tackling scalability by compositional model-checking. The main issue with his approach is that it is not fully automatic, and a good balance between number of invariants and state space explosion must be manually/heuristically reached.

Sajid et al. [27] extended and combined the BDD techniques with decision procedures for uninterpreted function symbols. Despite being a great improvement, their approach cannot be easily integrated with symbolic model checkers. Also, they have not considered the application of their work on the verification of OOO processors.

Berezin et al. [28] enhanced traditional model-checking by incorporating uninterpreted function symbols, and proposed effective and scalable propositional completion function for Tomasulo's algorithm. Their methodology enables the automatic verification of complex parametrized designs and allows them to verify Tomasulo's algorithm in any arbitrary configuration.

2.3. Verifying Cybersecurity by Tainting

None of the approaches described in the previous section was proposed for verification of cybersecurity properties. Though formal verification has been discussed and applied to cybersecurity, the main efforts have been devoted to expressing security properties, and/or to attain scalability by mixed dynamic/static techniques such as semi-formal and/or concolic (concrete symbolic) approaches [29,30].

Information Flow Tracking (IFT) is a method for ensuring confidentiality and integrity of systems that manipulate sensitive data by tracking how information moves throughout the system. IFT has been extensively employed in the security context for both hardware and software systems, both within formal and non formal approaches.

Basically, IFT models how labeled data moves through a system. The foundation of IFT is to label data by enhancing the hardware and/or the software to make information flows explicitly visible [31,32].

IFT techniques for hardware systems have been applied at different levels of abstraction, in order to ensure confidentiality and integrity of a system. For example, at the gate level, augmenting each logic primitive with additional IFT logic in the synthesized design netlist enables the tracking of all logical information flows.

In general, there are two parties (i.e., two variables in software systems, two modules in hardware). Information flows from one object A to the object B if and only if a change in A affects the value of B (i.e., interference).

Within this framework, one could define confidentiality and integrity properties as follows:

- Confidentiality: object A is confidential, while object B is public. An attacker could gain information about A by observing variations on B. In this case, the property would be that A must not flow to B.
- Integrity: object A is untrusted, while object B is trusted. An attacker could gain access to B through malicious modifications on A. Again, the property would be that A must not flow to B.

A natural way to implement this logic is by using the so called tainting. This approach works by marking sensitive data introduced at the source as tainted and monitoring taint propagation over the whole system until it reaches a sink (i.e., the target).

The authors in [33] propose a new kind of security properties called taint-propagation properties. The core idea is that information flow analysis is extended with instruction level abstraction (ILA) to create a model of the interactions between hardware and software. The method, thus, can express security properties at the HW/SW boundary and it is shown to be successful in software/hardware co-verification. However, they do not cover vulnerabilities at the microarchitectural level.

Further notable research on adapting software taint analysis in the hardware domain, in order to detect security vulnerabilities, has been pioneered in [34–36].

As an alternative to standard model-checking approaches, the Secure Path Verification technique has been presented in [37]. Starting from the notion of taint-propagation properties, the authors develop a new class of properties: Path properties. As for its parent class, a path property specifies a source, a destination and environmental constraints for taint propagation. Path properties also introduce path constraints to enforce secure information flow properties. The work also describes a verification engine based on a variant of equivalence checking, where taint propagation is straightforward as directly supported by the model checker engine.

As shown before, IFT can be applied at different levels of precision and abstraction and this can lead to different verification results in terms of soundness. The repercussions of this choice are discussed in Sections 5.1 and 5.3.

3. Processor Model

This section describes a case study, the pipelined microprocessor that we adopt to introduce our methodology. The processor is based on the renowned Hennessy and Patterson's DLX architecture [16], a 32-bit generic RISC processor architecture that we enhanced by implementing pipelining and speculative execution.

The processor, whose high-level design is shown in Figure 1, is a load/store architecture and adopts a seven-stage pipeline. Speculative execution is supported by means of a reorder buffer and reservation stations, controlled by a Tomasulo's algorithm [38]. Tomasulo's algorithm supports out-of-order instruction execution based on data-flow order, rather than sequential order. In case of mispredicted instructions a misprediction exception is raised, the execution is stopped, and the pipeline is flushed. In a similar way, if there is an instruction cache miss, the pipeline is stalled until the instruction is available. In case of an exception, any microarchitectural step occurring after the exception is not committed to the architectural state.

Stages IF1, IF2 and ID handle instruction fetch and decode. After stage ID, instructions wait for their operands to be available. They are then issued to the appropriate functional unit. This is done in the reservation stations (RS), where instructions are treated as in a FIFO queue until they are dispatched to the corresponding execution unit.

An instruction is ready for issue when operands, destinations, and execution units are available, Reservation stations are hardware data structures that hold instructions waiting for execution. Operation results available in reservation stations are possibly forwarded to reduce RAW hazards (i.e., result values go back to the RS, therefore also in the ROB, so that dependent instructions have their operands ready), and they support out-of-order execution.

For the sake of simplicity and without losing correctness, we consider the execution units to be fully pipelined, i.e., their throughput is one instruction every cycle. Likewise, regarding our model all instructions take the same number of clock cycles to complete their execution, though in reality they do differ for timing span.

The reorder buffer is used to maintain the program order of the instructions so that they can be committed in that order to respect the ISA semantics.

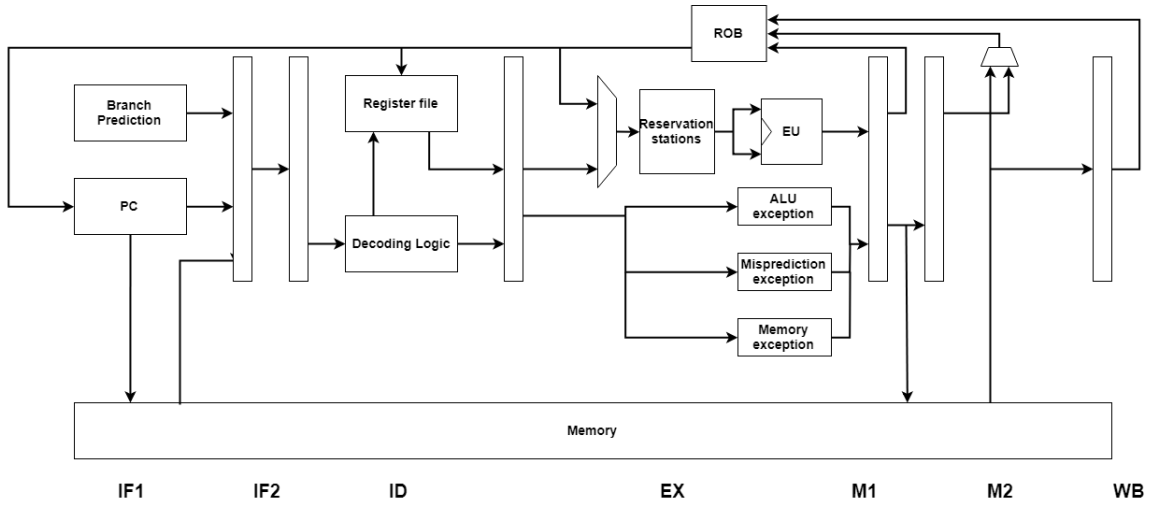

Figure 1. The pipelined processor model with 7 stages: **[IF1]** A 32-bit instruction is fetched from memory at the address given by the program counter PC (branch prediction logic is included); **[IF2]** This is meant to be a virtual address translation unit, possibly coupled with a TLB, but in this case study it is just a combinational delay. We retain this stage for future work; **[ID]** Instruction Decoding, fetching of values from registers, evaluation of branch conditionals and target addresses, pipeline hazard detection and control flow of program execution. Opcodes and operands are fed to the Reorder Buffer (ROB); **[EX]** The EX stage executes both integer and floating-point instructions and generates exceptions as needed. This stage includes reservations stations, providing operands to and acting as a scheduler for execution units; **[M1]** access to data memory for load/store instructions. This enables a bypass towards the ROB; **[M2]** access to data memory for load/store instructions; **[WB]** instruction commit, exception handling and storing values to the register file. This includes the ROB that retires instructions to the register file.

4. Attack Description

To show the weakness of CPU architectures, we decided to replicate the well-known Spectre [1] and Meltdown [2] attacks, which both abuse the out-of-order execution to disclose internal microarchitectural state information.

Both cases are based on side-channel attacks. A side-channel attack is any attack based on information gained from the implementation of a computer system, rather than the implemented algorithm itself or the Instruction Set Architecture. Side-channel attacks typically exploit timing, power, electromagnetic leaks, or other sources of information.

In the case of Spectre and Meltdown, the attack is based on a covert side-channel that exploits the timing of cache-based memory accesses. A malicious attacker intentionally induces a speculated execution of mispredicted instructions. The effects of these instructions (not yet retired, thus called transient) are not intended to be committed into the architectural state, as the instructions will be cancelled and their results discarded. However, they change the internal processor microarchitectural state, due to a memory read affecting the cache state, observable by a (timing-based) side-channel: retrieving data from (un-cached) memory takes longer than retrieving it from cached addresses.

A Meltdown attack provides a way for a user level attacker to read the entire kernel space memory, including all physical memory mapped in the kernel, being able to bypass the privileged-mode isolation. A Spectre attack has a more limited scope, the memory within the address space of another (victim) process.

Although different, both attacks rely on common "building blocks". The first consists of executing one or more instructions which would never be computed in the proper execution path. Applying Meltdown's naming convention, we call transient instruction an out-of-order executed instruction, which will leave measurable side effects. We also call transient instruction sequence any sequence of instructions which contains at least one transient instruction. To be able to exploit transient

instructions to perform an attack, the sequence needs to hold a secret value which will be leaked by the attacker.

The second building block consists of transferring the architectural state affected by the transient instruction sequence to propagate and extract the leaked information.

From a high-level perspective, our attack, based on Meltdown, consists of 3 steps:

1. the attacker chooses an inaccessible memory location, then the contents of that memory location is loaded into a register;
2. a cache line is accessed by a transient instruction based on the secret contents of the register;
3. the attacker exploits a side-channel to probe the previously accessed cache line and leaks information depending on the sensitive data saved at the chosen memory location.

Listing 1, written in DLX assembly code, presents the basic implementation of the transient instruction sequence of our attack that is then used to leak information with a side-channel. We describe now in detail the actions performed and the effect of every single step of our attack, considering the presented assembly code but also the processor model under analysis.

Listing 1. Attack transient instruction sequence in DLX assembly code.

```
1 ; R1 = invalid address
2 ; R3 = probe array
3 LW   R2, 0(R1)
4 ADD  R4, R2, R3
5 LW   R1, 0(R4)
```

4.1. Step One

Referring to line 3 of Listing 1, `R1` register holds the invalid address, i.e., an attacker chosen target kernel address that is loaded into register `R2`.

The `LW` instruction is fetched, decoded into μOPs (Although this is true in general, in our case-study model there is no generation of micro-operations, i.e., every instruction is one μOP, due to the simplicity of the model itself.), allocated, and sent to the reorder buffer, waiting to be executed.

At this point, in order to enable out-of-order execution, register renaming maps the architectural registers (e.g., `R1`, `R2`, `R3` and `R4` in Listing 1) to underlying physical registers.

As for out-of-order execution, that aims at increasing CPU throughput, also the following instructions (lines 4 and 5) have already been fetched, decoded and allocated as μOPs, they are then sent on hold to the reservation stations, waiting to be issued to an execution unit.

In general, a μOP is delayed if operands are not available or if all execution units are already holding and executing other μOPs. In our example, the `ADD` instruction must wait for the result of the first `LW` instruction.

When the contents at the kernel address is loaded (line 3), subsequent instructions have already been issued in the reservation stations as speculated instructions, waiting for the kernel address to be available.

Once the needed data are fetched and available on the Common Data Bus (being accessible to the execution unit and being stored into registers), μOPs on hold can be executed. After the execution stage, completed μOPs are committed in-order (write-back stage), with their results affecting the architectural state of the CPU.

If any interrupt or exception, e.g., illegal access, arise during the execution stage, the corresponding exception unit handles it.

Therefore, when the `LW` instruction is retired, if an exception is thrown, then the pipeline is flushed to discard all the computations of the following speculated instructions.

At this point, there exists a race condition between raising an exception and the speculative execution of our attack, which is described in detail in Section 4.2 below.

4.2. Step Two

The set of instructions executed during Step One must be carefully crafted to make a transient instruction sequence.

To transmit the secret outside the CPU microarchitecture, first we define and allocate in memory a probe array, ensuring that no part of it will be cached. The original Meltdown attack prevents the hardware prefetcher from loading adjacent memory locations into the cache. For our purposes, in order to make our attack model as generic as possible, we avoid this operation.

Then, an address is determined, based on the (inaccessible) secret value, and will be used by the transient instruction sequence to perform an indirect memory access.

The secret value is added to the probe array base address, in order to compose the target address of the covert channel. The target address is then read and used to store the corresponding cache line.

Therefore, the goal of our crafted instruction sequence is to alter the cache state, based on the secret value read during Step 1.

4.3. Step Three

In Step 3, the attacker leaks and obtains the secret value (from Step One) exploiting a microarchitectural side-channel attack which propagates the cache state (from Step 2) into an architectural state.

This is due to the race condition mentioned in Step One: line 3 gets speculatively executed, depending on the value of the secret, so a part of the probe array will be accessed and, most importantly, cached.

To sum up, despite the program never executing lines 4 and 5, the cache state has changed.

As already stated, Meltdown and Spectre exploit well-known cache covert channel attacks but their specifics are considered out of scope.

5. Proof/Verification

To simplify the verification process, in our approach we propose several model transformations, oriented to scalability. The performed transformations can be aggregated into two different typologies:

- data abstraction: register values, along with all the units that handle these values, as well as the reorder buffer, reservation stations and execution units, are adequately abstracted. Tainting information and evaluation/propagation circuitry are then added to or replace them.
- model reduction: applying established processor verification approaches such as pipeline flushing and reduction by refinement map, we reduce speculation and parallel execution logic.

Considering that we strive to make our approach as general as possible, our CPU model does not explicitly feature any main memory, therefore confidential/secret information is implicitly linked to a given *protected/invalid* memory address. Consequently, we verify arbitrary sequences of instructions (i.e., no explicit program is provided).

The completeness of our approach, even though this is a pessimistic choice in the framework of our analysis, is guaranteed because all possible instructions sequences are covered, as well as being consistent with other state-of-the-art processor verification approaches.

In the following subsections we provide a detailed description of the main notions of the two typologies of transformations. Albeit a formal proof of correctness is not provided, we contribute with the basic intuitions that support their applicability.

5.1. Data Abstraction and Tainting

All related works cited in Section 2 follow the same verification approach: they over-approximate data (and consequently model behavior), provided that verification is sound. That is, this transformation must guarantee that an abstract counterexample always implies a concrete one.

In our case, for every register R_i we call V_i its contents. Then, without altering security properties, V_i is replaced by an abstract value V_i^+. As already stated, RAM memory is removed, adopting common

strategies used for data abstraction, which means drastically reducing its impact on the state of the model. Partial abstraction of data, not relevant to the property under verification, is often performed. Despite having greatly reduced the size of the model, we can still perform abstraction: for example, processor functionality (e.g., ALU data computation/evaluation) can be considered correct (already verified). In all abstraction processes, properly choosing a correct/adequate abstraction, either by automated or manual selection, is a challenging task.

Following known methods in hardware verification, we augment each value V_i^+ with its corresponding taint value T_i, as depicted in Table 1. The ALU, the data evaluation logic, is enhanced with taint-propagation logic. Taint values are injected, propagated and observed through memory and data transfer components. Propagation or combination of multiple taints is responsibility of the ALU.

Table 1. Concrete versus Abstract model transformation.

Register	Concrete	Abstract
$\mathbf{R_1}$	V_1	(V_1^+, T_1)
$\mathbf{R_2}$	V_2	(V_2^+, T_2)
⋮	⋮	⋮
$\mathbf{R_n}$	V_n	(V_n^+, T_n)

Table 2 shows that while the original processor model executes an arithmetic/logic operation $V_k = OP(V_i, V_j)$, the abstract model replaces it with $V_k^+ = OP(V_i^+, V_j^+)$ and provides that the transformation is sound. The first part is data evaluation and it is independent from taint values, while the second is taint propagation, which in its broad form involves both data and taint values.

The tainting precision, i.e., the degree of over-approximation, swings between two corner cases:

- full data dependence: data values are fully involved in taint computation; whenever computing T_k, actual operand data values are considered; for instance, a bitwise OR operation with all $V_i^+ = 1$, or a multiplication with $V_i^+ = 0$, could mask (block) a taint on the other operand (T_j);
- full abstraction from data values: for instance, taint propagation through a binary ALU operation propagates a taint on any of the operand terms ($T_k = OP^T(T_i, T_j) = T_i \vee T_j$).

The level of abstraction certainly affects the soundness of the overall approach: for detailed comments and proof of correctness of the approach, see Section 5.3.

Table 2. Comparison of data evaluation and taint propagation between concrete and abstract models.

	Concrete	Abstract
Data abstraction + Tainting	V_i	(V_i^+, T_i)
Data evaluation	$V_k = OP(V_i, V_j)$	$V_k^+ = OP(V_i^+, V_j^+)$
Taint propagation	-	$T_k = OP^T(V_i^+, T_i, V_j^+, T_j)$

Also, the branch misprediction logic is abstracted and replaced by a non-deterministic value (choice); this operation can be performed without altering the correctness of our approach, since it generalizes (abstracts) the model under analysis.

Similarly, arithmetic/logic manipulation can be considered already verified, e.g., as pass-through circuitry for taints. Since we are interested only in data leakages from/to memory, we can move our focus taint propagation to memory access logic.

Let us now define the taint source/sink pair. A taint is thus injected at the memory data input, whenever a protected/invalid address is used, that is a protected address is loaded into the CDB. In other words, the taint (T_{MDR}) associated with the Memory Data Register (MDR) is set whenever the

address stored in the Memory Address Register (MAR) is not in the range of valid addresses ($VA.start$ and $VA.end$).

$$T_{MDR} = MAR < VA.start \vee MAR > VA.end$$

The taint is propagated through the abstract reservation stations, arithmetic/logic execution units and reorder buffer. A taint can be cleared only by the branch misprediction circuitry, which means in case of an instruction not committed but aborted.

The target (taint sink) under observation is the memory address. From the information flow point of view, a taint in a memory address register corresponds to the property we are verifying.

$$Prop = \neg(T_{MAR} == TAINT)$$

Considering, from a tainting point of view, a generic instruction sequence common to both Meltdown and Spectre attacks, the following actions will be performed:

- inject a taint, exploiting a mispredicted instruction which performs an invalid/protected memory access;
- transfer the taint into the (abstracted) reorder buffer, as a data expected to be committed;
- use the taint, as a data, as part of the computation of the address of a successive mispredicted memory access: the taint hits the target.

The taint propagation path is described in Figure 2. In details, the taint is injected at the memory read (**1**), when the address bus is filled in with an invalid address. As the instructions in the pipeline move through the different steps, the taint reaches the reorder buffer (**2**), then it moves on to reservation stations (**3**) as new instructions are fed by the instruction cache. The taint enters reservation stations because of the propagation rules, as tainted data are fetched as operands of newly arrived instructions. As soon as all the operands are available, the taint propagates to the corresponding execution unit (**4**). Eventually it reaches the Common Data Bus (CDB) and consequently again the reorder buffer (**5**).

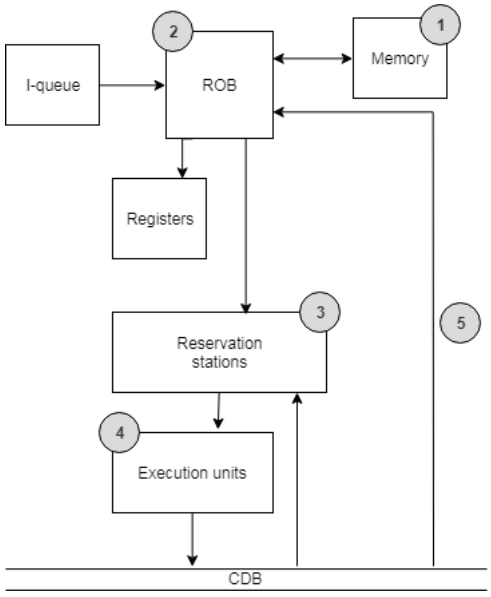

Figure 2. Taint propagation from source (memory read) to sink (reorder buffer) in our abstract model.

Table 3 shows how our model behaves with the instructions in Listing 1. The first instruction reads from memory (M) at an invalid address (IA) and stores an invalid data (ID) into the destination. This injects the taint into T_2. The second instruction computes the array displacement as it adds the probe array start address (PA) and the invalid data. The taint propagation logic, as previously stated, involves the OR operator on the taints associated with the operands, so in this case the taint propagates

to T_4. The last instruction reads in from memory at the address computed in the second instruction. The taint propagates to T_1 and reaches the target where the property fails.

Table 3. Our model applied to a real use case: a basic implementation of Spectre/Meltdown.

Instruction	Symbolic	Concrete
LW $R2, 0(R1)$	$V_2^+ \leftarrow M[V_1^+]$ $T_2 \leftarrow T_1$	$ID \leftarrow M[IA]$ $1 \xleftarrow{inject} 0$
ADD $R4, R2, R3$	$V_4^+ \leftarrow ADD(V_2^+, V_3^+)$ $T_4 \leftarrow T_2 \vee T_3$	$PA + ID \leftarrow ADD(ID, PA)$ $1 \leftarrow 1 \vee 0$
LW $R1, 0(R4)$	$V_1^+ \leftarrow M[V_4^+]$ $T_1 \leftarrow T_4$	$Y \leftarrow M[PA + ID]$ $1 \xleftarrow{target} 1$

Table 4 shows, in the rightmost table, the same instruction sequence as in Listing 1, while the leftmost table displays the pipeline evolution. The sequence implements a generic and simplified version of Meltdown/Spectre attack (as described in Section 4), in which:

(A) performs a memory access to an invalid address;
(B) executes an arithmetic operation using the secret data;
(C) performs a read from an array with a displacement related to the secret.

Table 4. Pipeline evolution of the proposed attack.

Clock ↓	IF	ID	EX	MEM	WB	Taint Status
1	A					
2	B	A				
3	C	B	A			
4		B/C	S	A		Taint source
5		C	B	S	A	Taint in ROB
6		C	S	B	S	Taint in EX
7			C	S	B	Taint sink/property asserted
8				C	S	
9					C	

A	**LW**	$R2, 0(R1)$
B	**ADD**	$R4, R2, R3$
C	**LW**	$R1, 0(R4)$

The vertical axis represents time in terms of clock cycles. Columns from IF to WB represent the different pipeline stages. The last column presents the taint status and its propagation. Cells filled with "S" are stalls.

Instructions enter the pipeline and proceed without stalls until clock 3. At clock 4, B and C are stalled to wait for their operands to be ready. When at stage MEM A reads from memory, the value is placed into the Common Data Bus and becomes available also to the stalled instructions. At clock 5, B goes on to the next stage, while C must wait for its operands to be prepared by B, thus a new stall is inserted. Finally, from clock 7 to 9 the pipeline proceeds without stalls.

The taint source is at clock 4 inside the MEM stage of instruction A, when the access to the invalid address is performed. At the WB stage of A, the taint propagates to the reorder buffer. Then the taint propagates to the execution unit at the EX stage of instruction B. Afterwards during clock 7 the taint, reaching the CDB, is collected by the taint sink and captured by the previously defined security property.

Compositional verification schemes are a very common way to exploit abstraction, where a given module is taken and verified on a local basis, while removing (i.e., abstracting away) the

remaining model components, considered to be the module's environment. A refinement of the environment, expressed as a set of constraints, is usually required to refine it to a sound abstraction, thus leading to assume-guarantee strategies where a given refinement is either a property (to be proved/guaranteed) when verifying a module, and an environment constraint (an assumption) for another module under verification.

Considering our case study, seeing the high simplification level reached by abstraction/refinement-based abstraction, we decided not to apply any compositional simplification [39].

5.2. Combining Model Reduction with Abstraction

The CPU model of our case study, at this point, is further simplified, applying state-of-the-art methodologies to formal verification of pipelines and speculation units [24,39,40].

Briefly, the processor model was already simplified by transforming data into taints. At this point instead, we are simplifying all intermediate states associated with the control logic for parallel execution, with a convenient reduction of the behavior and considering it as an abstraction, based on proper equivalence notions between the concrete model and the reduced one.

Pipeline flushing is often applied, in order to highly simplify model behavior, by removing all possible interleavings of pipeline executions. As this could lead to an incomplete solution (missing possible wrong behaviors) the reduced model is considered to be an abstraction of the real model, where each concrete state (among the set of all possible pipeline interleavings) is considered to be a possible refinement of an abstract state in the reduced model: the correspondence is handled by refinement maps. Similar strategies are also applied whenever reducing the bit widths of registers, memory words and of memory addresses, the size of the register file, the number of reservation stations and execution units, etc. All such reductions are deemed as complete by considering the reduced model as an abstraction of the concrete (refined) model and/or by proving that the absence of a bug in the reduced model implies that no bugs are possible in the concrete (non-reduced) one.

While considerably simplifying the model, a reduction process can thus still guarantee verification completeness.

A detailed description of the reduction strategies is clearly out of our scope in this work, as any property preserving reduction (with proper refinement map or alternative theory) is applicable, provided that it guarantees completeness.

We here quickly introduce the simplifications applied on our model:

- pipeline flushing: all pipeline stages are flushed (collapsed), which simplifies the execution model of an instruction, since the next instruction is initiated only when the previous one reaches the reorder buffer;
- reorder buffer removal: ROB is reduced into a FIFO queue, which essentially delays instructions between execution and results availability into the register file; to be noted that the FIFO strategy preserves the original instruction order, assuring data dependency;
- reservation station replacement: as straightforward effect of previous pipeline flushing and ROB removal, reservation stations are bypassed (performing in fact a model reduction);
- execution units merge: considering the data abstraction performed on our model, computation parallelism is unnecessary, so just one instance of each execution unit is useful.

As a result, of the listed above simplifications, the complete original behavior is significantly reduced, moving from a pipelined architecture with speculation, to a fully sequential model with a FIFO-based delay between execution and obtainable results.

The performed reduction simplifications guarantee completeness, which means that taint propagation sets of instructions are not removed, given the following two conditions:

- consecutive instruction sequences comprise mispredicted instructions, simulating real instructions sequences made by an actual out-of-order CPU; this behavior is assured by a non-deterministic

tag associated with an instruction, which marks that instruction as mispredicted (this operation is performed during the abstraction transformation, as described in Section 5.1);
- the FIFO-based delay, replacing the ROB, simulates the (illegal) taint-propagation time from source to sink; this behavior is assured by a proper FIFO queue size and a proper non-deterministic queue control of *get* operations, which transfer data from ROB to the register file.

5.3. Correctness of the Approach

We now provide a concise proof of the correctness of our verification approach. Due to the descriptive nature of the paper, where we omit a rigorous formalism for models, properties and transformation steps, the proof is limited to a sketch, outlining the theoretical bases that support the proposed methodology.

As with all verification approaches based on model transformations, formal correctness means *soundness and completeness* of all model simplifications (abstractions and reductions) performed. In our case, we operate two classes of simplifications:

- Abstractions and reductions that do not affect secure information flow by taint propagation. This is a set of preliminary model transformations oriented to reduce the data width and the model behavior (pipeline and speculative execution): transformations guarantee the model functionality, and they have already been proved correct by related and state-of-the-art works on formal verification of processor designs. We do not claim any contribution in this field, and we assume them as correct
- Abstraction and reduction steps related to secure information flow by taint propagation. Though we resort to standard formalisms and transformations, we nevertheless need to show/prove that their combined application is complete and sound.

5.3.1. Model Abstraction and Reduction

As already written, our model abstraction and reduction (with refinement) steps are based on state-of-the-art formal verification approaches [24,40], that have already been proved to be sound and complete by their authors. We can assume them as correct in terms of processor design, but nothing is said on the processor state, potentially observed by a side-channel attack. In other words, we exploit state-of-the-art model simplifications that guarantee model functionality by pruning and simplifying complex intermediate states and behaviors.

Therefore, the remaining critical issue in our methodology is the correctness of taint encoding and manipulation all over the abstraction and refinement steps. This is what was missing in all previous processor verification works, and represents the main contribution of our work.

To this respect, we need to show that

- tainting does not affect the correctness of the model
- secure information flow by tainting is sound and complete.

The first item is straightforward. We can claim that since a taint is actually an enhanced data, added to the original one, it does not affect model functionality, provided that tainting logic is just *reading* data, while not affecting data evaluation. More formally, this is clearly shown in Table 2, where a result taint (in the more general case) is a function of operand data and taints ($T_k = OP^T(V_i^+, T_i, V_j^+, T_j)$), whereas result data do not depend on operand taints ($V_k^+ = OP(V_i^+, V_j^+)$). The second item (soundness and completeness of taint propagation) is analyzed in the next subsection.

5.3.2. Taint Encoding and Manipulation

In the strict sense, the tainting and transformation steps we perform are unsound as, due to data abstraction in the taint-propagation circuitry, we could obtain false negatives (taint-propagation traces not feasible on the concrete model).

In fact, we admit abstract execution traces that propagate the taint through the ALU logic, whereas no information leakage would characterize the actual model.

This issue can be imputed to the taint-propagation strategy we decided to adopt: the applied abstraction ignores the fact that a taint could be blocked/hidden by a real data. The taint computation rules need to take into account both the operation and the data involved. Precise rules would impose more complexity, so we had to reach a trade-off between precision and complexity:

- apply precise and narrower tainting rules, cutting off (adopting a more precise and detailed taint-propagation model) all false negatives;
- adopt an imprecise but efficient approach, thus accepting false abstract counterexamples. If this would be the case, counterexamples could be either:
  - post processed, leading to subsequent model refinements;
  - be converted into actual (equivalent) concrete counterexamples, by just exploiting them partially (e.g., by removing data and keeping control bits), as constraints for a further Bounded Model-Checking (BMC) run on the concrete model.

In our opinion the second approach is to a great extent coherent with the final aim of detecting data leakages and solving them.

Therefore, in the end soundness relies on a proper notion of (bi-simulation) equivalence between abstract and concrete counterexamples: any abstract counterexample is required to be mapped to at least one concrete counterexample by simple data/behavior refinement: Abstractions done in taint-propagation logic are sound if, for any tainting blocking based on data values, other non-blocking data exist.

The approach is complete as no reduction is done on taint computation and propagation (all reductions are done just based on equivalence/function preserving transformations, whose completeness has been proved for model-checking purposes).

6. Experimental Results

The approach presented in this paper was tested and verified on the case study described in Section 3.

The main purpose of our experimentation was not to provide detailed performance measures of different model-checking engines/tools, rather to show that resorting to proper abstractions and reductions allows tackling state explosion. This makes previously unfeasible problems now solvable in matter of seconds with a state-of-the-art model checker.

The processor was described in Verilog, then converted into the AIGER format [41] and verified using PdTRAV [42], a state-of-the-art academic model-checking tool we developed. Both Bounded and Unbounded Model-Checking (interpolation-based UMC) algorithms were used, with a peculiar focus on model reductions and transformations [43,44], multiple properties manipulations [45] and interpolants-based engines [46,47].

In detail, taints were encoded as binary data, branch prediction/misprediction circuitry was entirely abstracted and substituted by a non-deterministic (random) Boolean value. Moreover, parallel execution units were substituted by a taint propagation pass-through circuitry.

A taint not reaching the address output of the microprocessor model corresponds to the encoding of the confidentiality property we want to verify.

As already stated in [39], the original full microprocessor model (inclusive of speculation logic, Tomasulo's module, pipeline, multiple execution units and data paths) would be very difficult to verify: the model consisted of more than 120 K gates and more than 3 K latches. The resulting model after all the simplifications (abstraction, reduction and tainting), was converted into an AIGER file consisting of 2724 AND gates and 106 latches, resulting by reducing the register file to 8 registers of 5 bits, and the rest of the control and data logic accordingly. We also tested larger versions, by expanding the number

of data registers up to 16 and their size up to 32 bits. The largest AIGER file (after Cone-Of-Influence reduction) included 7245 AND gates and 395 latches. We could verify it by the BMC verification engine, finding a counterexample of 11 clock steps in times ranging from less than 1 s to 9 s.

The counterexample retrieved after the verification process presented a data leakage issue: this bug was made out of an instruction sequence starting with an invalid memory read (operation which injected the taint), followed by an arithmetic computation of a memory address (operation which propagated the taint), and finally a further memory read to the tainted address. This counterexample can be considered to be an abstracted/reduced version of the example presented in Table 3.

Then the cybersecurity flaw was eliminated (and formally verified by model-checking, again in less than 1 s and 15 s, with both a Bounded Model-Checking and an IC3 engine) by patching the model in this way: all speculated instructions with data dependencies are prevented by an instruction with invalid memory access. More in detail, designing a non-buggy (efficient) version of the processor was out of our scope, we rather wanted to have a working non-buggy model. We thus exploited the exception generation logic, making it active before committing the exception generation instruction: whenever an invalid read is done by a speculated instruction, an exception is not generated until confirming (committing) the instruction (due to branch prediction logic). The exception detection logic immediately catches the invalid access, then it blocks the read data (thus not available for subsequent operations) until the instruction is committed or aborted: if committed, the exception is raised, if aborted, execution resumes on the correct path.

7. Conclusions and Future Work

In this paper, our goal is to describe a formal verification procedure able to find confidentiality security leaks in contemporary CPU microarchitectures featuring out-of-order/speculative execution, in order to prevent future cybersecurity speculation-based attacks (as Meltdown or Spectre).

The proposed technique is based on state-of-the-art formal verification concepts, as model abstraction and model refinement, as well as exploiting novel ideas from the information flow tracking field, as taint injection and taint propagation, merging schemes taken from this two different areas of research to reach our goals.

In future works we will automate the simplifications performed to the model under analysis (abstraction/reduction), since in this paper the described transformations were manually driven. Our final goal will be an automated (eventually in part) transformation and simplification mechanism.

Author Contributions: All authors have contributed equally to the work reported.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

ALU	Arithmetic Logic Unit
BDD	Binary Decision Diagram
BMC	Bounded Model-Checking
CDB	Common Data Bus
EX	Execution
FIFO	First-in First-Out
HW/SW	Hardware/Software
IC3	Incremental Construction of Inductive Clauses for Indubitable Correctness
ID	Instruction Decode
IF	Instruction Fetch

IFT	Information Flow Tracking
ILA	Instruction Level Abstraction
ISA	Instruction Set Architecture
MA	MicroArchitecture
MAR	Memory Address Register
MDR	Memory Data Register
MEM	Memory
μOP	Micro-operation
OOO	Out-of-order
PC	Program Counter
PdTRAV	Politecnico di Torino Reachability Analysis and Verification
RAW	Read After Write
RISC	Reduced Instruction Set Computer
ROB	Reorder Buffer
RS	Reservation Station
TLB	Translation Lookaside Buffer
UMC	Unbounded Model-Checking
WB	Write-Back

References

1. Kocher, P.; Genkin, D.; Gruss, D.; Haas, W.; Hamburg, M.; Lipp, M.; Mangard, S.; Prescher, T.; Schwarz, M.; Yarom, Y. Spectre attacks: Exploiting speculative execution. *arXiv* **2018**, arXiv:1801.01203.
2. Lipp, M.; Schwarz, M.; Gruss, D.; Prescher, T.; Haas, W.; Mangard, S.; Kocher, P.; Genkin, D.; Yarom, Y.; Hamburg, M. Meltdown. *arXiv* **2018**, arXiv:1801.01207.
3. Beckers, K.; Heisel, M.; Hatebur, D. *Pattern and Security Requirements*; Springer: Berlin, Germany, 2015.
4. Boritz, J.E. IS practitioners' views on core concepts of information integrity. *Int. J. Account. Inf. Syst.* **2005**, *6*, 260–279. [CrossRef]
5. Yarom, Y.; Falkner, K. FLUSH+ RELOAD: A high resolution, low noise, L3 cache side-channel attack. In Proceedings of the 23rd USENIX Security Symposium (USENIX Security 14), San Diego, CA, USA, 20–22 August 2014; pp. 719–732.
6. Yang, B.; Wu, K.; Karri, R. Scan based side channel attack on dedicated hardware implementations of data encryption standard. In Proceedings of the 2004 International Conference on Test, Charlotte, NC, USA, 26–28 October 2004; pp. 339–344.
7. Lin, L.; Kasper, M.; Güneysu, T.; Paar, C.; Burleson, W. Trojan side-channels: Lightweight hardware trojans through side-channel engineering. In *International Workshop on Cryptographic Hardware and Embedded Systems*; Springer: Berlin/Heidelberg, Germany, 2009; pp. 382–395.
8. Tehranipoor, M.; Wang, C. *Introduction to Hardware Security and Trust*; Springer Science & Business Media: Berlin, Germany, 2011.
9. Lowe-Power, J.; Akella, V.; Farrens, M.K.; King, S.T.; Nitta, C.J. A case for exposing extra-architectural state in the ISA: Position paper. In Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy, Los Angeles, CA, USA, 2 June 2018; p. 8.
10. Joy Persial, G.; Prabhu, M.; Shanmugalakshmi, R. Side channel attack-survey. *Int. J. Adv. Sci. Res. Rev.* **2011**, *1*, 54–57.
11. Fan, J.; Guo, X.; De Mulder, E.; Schaumont, P.; Preneel, B.; Verbauwhede, I. State-of-the-art of secure ECC implementations: A survey on known side-channel attacks and countermeasures. In Proceedings of the 2010 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST), Anaheim, CA, USA, 13–14 June 2010; pp. 76–87.
12. Zhou, Y.; Feng, D. Side-Channel Attacks: Ten Years after Its Publication and the Impacts on Cryptographic Module Security Testing. *IACR Cryptol. EPrint Arch.* **2005**, *2005*, 388.
13. Hill, M.D.; Masters, J.; Ranganathan, P.; Turner, P.; Hennessy, J.L. On the Spectre and Meltdown Processor Security Vulnerabilities. *IEEE Micro* **2019**, *39*, 9–19. [CrossRef]
14. Bennett, R.; Callahan, C.; Jones, S.; Levine, M.; Miller, M.; Ozment, A. How to live in a post-meltdown and-spectre world. *Commun. ACM* **2018**, *61*, 40–44. [CrossRef]

15. Prout, A.; Arcand, W.; Bestor, D.; Bergeron, B.; Byun, C.; Gadepally, V.; Houle, M.; Hubbell, M.; Jones, M.; Klein, A.; et al. Measuring the Impact of Spectre and Meltdown. In Proceedings of the 2018 IEEE High Performance extreme Computing Conference (HPEC), Waltham, MA, USA, 25–27 September 2018; pp. 1–5.
16. Patterson, D.A.; Hennessy, J.L.; Goldberg, D. *Computer Architecture: A Quantitative Approach*; Morgan Kaufmann: San Mateo, CA, USA, 1990; Volume 2.
17. Cabodi, G.; Camurati, P.; Finocchiaro, F.; Vendraminetto, D. Model Checking Speculation-Dependent Security Properties: Abstracting and Reducing Processor Models for Sound and Complete Verification. In Proceedings of the International Conference on Codes, Cryptology, and Information Security, Rabat, Morocco, 22–24 April 2019; pp. 462–479.
18. Clarke, E.M.; Emerson, E.A. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs*; Springer: Berlin/Heidelberg, Germany, 1981; pp. 52–71.
19. Clarke, E.M.; Emerson, E.A.; Sistla, A.P. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst. (TOPLAS)* **1986**, *8*, 244–263. [CrossRef]
20. Damm, W.; Pnueli, A. Verifying out-of-order executions. In *Advances in Hardware Design and Verification*; Springer: Berlin, Germany, 1997; pp. 23–47.
21. Sawada, J.; Hunt, W.A. Processor verification with precise exceptions and speculative execution. In Proceedings of the International Conference on Computer Aided Verification, Vancouver, BC, Canada, 28 June–2 July 1998; pp. 135–146.
22. Hosabettu, R.; Srivas, M.; Gopalakrishnan, G. Decomposing the proof of correctness of pipelined microprocessors. In Proceedings of the International Conference on Computer Aided Verification, Vancouver, BC, Canada, 28 June–2 July 1998; pp. 122–134.
23. Hosabettu, R.; Srivas, M.; Gopalakrishnan, G. Proof of correctness of a processor with reorder buffer using the completion functions approach. In Proceedings of the International Conference on Computer Aided Verification, Trento, Italy, 6–10 July 1999; pp. 47–59.
24. Burch, J.R.; Dill, D.L. Automatic verification of pipelined microprocessor control. In Proceedings of the International Conference on Computer Aided Verification, Stanford, CA, USA, 21–23 June 1994; pp. 68–80.
25. Skakkebæk, J.U.; Jones, R.B.; Dill, D.L. Formal verification of out-of-order execution using incremental flushing. In Proceedings of the International Conference on Computer Aided Verification, Vancouver, BC, Canada, 28 June–2 July 1998; pp. 98–109.
26. McMillan, K.L. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In Proceedings of the International Conference on Computer Aided Verification, Vancouver, BC, Canada, 28 June–2 July 1998; pp. 110–121.
27. Goel, A.; Sajid, K.; Zhou, H.; Aziz, A.; Singhal, V. BDD based procedures for a theory of equality with uninterpreted functions. In Proceedings of the International Conference on Computer Aided Verification, Vancouver, BC, Canada, 28 June–2 July 1998; pp. 244–255.
28. Berezin, S.; Clarke, E.; Biere, A.; Zhu, Y. Verification of out-of-order processor designs using model checking and a light-weight completion function. *Form. Methods Syst. Des.* **2002**, *20*, 159–186. [CrossRef]
29. Cadar, C.; Dunbar, D.; Engler, D.R. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08), San Diego, CA, USA, 8–10 December 2008; Volume 8, pp. 209–224.
30. Godefroid, P.; Levin, M.Y.; Molnar, D. SAGE: Whitebox fuzzing for security testing. *Commun. ACM* **2012**, *55*, 40–44. [CrossRef]
31. Suh, G.E.; Lee, J.W.; Zhang, D.; Devadas, S. Secure program execution via dynamic information flow tracking. In Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems, Boston, MA, USA, 7–13 October 2004; Volume 39, pp. 85–96.
32. Tiwari, M.; Wassel, H.M.; Mazloom, B.; Mysore, S.; Chong, F.T.; Sherwood, T. Complete information flow tracking from the gates up. In Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, Washington, DC, USA, 7–11 March 2009; Volume 44, pp. 109–120.
33. Subramanyan, P.; Malik, S.; Khattri, H.; Maiti, A.; Fung, J. Verifying information flow properties of firmware using symbolic execution. In Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE), Dresden, Germany, 14–18 March 2016; pp. 337–342.

34. Subramanyan, P.; Arora, D. Formal verification of taint-propagation security properties in a commercial SoC design. In Proceedings of the conference on Design, Automation & Test in Europe, Dresden, Germany, 24–28 March 2014; p. 313.
35. Cabodi, G.; Camurati, P.; Finocchiaro, S.; Loiacono, C.; Savarese, F.; Vendraminetto, D. Secure embedded architectures: Taint properties verification. In Proceedings of the 2016 International Conference on Development and Application Systems (DAS), Suceava, Romania, 19–21 May 2016; pp. 150–157.
36. Ardeshiricham, A.; Hu, W.; Marxen, J.; Kastner, R. Register transfer level information flow tracking for provably secure hardware design. In Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE), Lausanne, Switzerland, 27–31 March 2017; pp. 1691–1696.
37. Cabodi, G.; Camurati, P.; Finocchiaro, S.F.; Savarese, F.; Vendraminetto, D. Embedded systems secure path verification at the hardware/software interface. *IEEE Des. Test* **2017**, *34*, 38–46. [CrossRef]
38. Tomasulo, R.M. An efficient algorithm for exploiting multiple arithmetic units. *IBM J. Res. Dev.* **1967**, *11*, 25–33. [CrossRef]
39. Jhala, R.; McMillan, K.L. Microarchitecture verification by compositional model checking. In Proceedings of the International Conference on Computer Aided Verification, Paris, France, 18–22 July 2001; pp. 396–410.
40. Manolios, P.; Srinivasan, S.K. A complete compositional reasoning framework for the efficient verification of pipelined machines. In Proceedings of the 2005 IEEE/ACM International Conference on Computer-Aided Design, San Jose, CA, USA, 6–10 November 2005; pp. 863–870.
41. Biere, A.; Heljanko, K.; Wieringa, S. AIGER 1.9 and Beyond. 2011. Available online: fmv.jku.at/hwmcc11/beyond1.pdf (accessed on 18 September 2019).
42. Cabodi, G.; Nocco, S.; Quer, S. Benchmarking a model checker for algorithmic improvements and tuning for performance. *Form. Methods Syst. Des.* **2011**, *39*, 205–227. [CrossRef]
43. Cabodi, G.; Camurati, P.; Garcia, L.; Murciano, M.; Nocco, S.; Quer, S. Speeding up Model Checking by Exploiting Explicit and Hidden Verification Constraints. In Proceedings of the Conference on Design, Automation and Test in Europe (DATE 2009), Nice, France, 20–24 April 2009; pp. 1686–1691.
44. Cabodi, G.; Nocco, S.; Quer, S. Strengthening Model Checking Techniques With Inductive Invariants. *IEEE Trans. CAD Integr. Circuits Syst.* **2009**, *28*, 154–158. [CrossRef]
45. Cabodi, G.; Nocco, S. Optimized Model Checking of Multiple Properties. In Proceedings of the Design, Automation and Test in Europe (DATE 2011), Grenoble, France, 14–18 March 2011; pp. 543–546.
46. Cabodi, G.; Palena, M.; Pasini, P. Interpolation with Guided Refinement: Revisiting Incrementality in SAT-based Unbounded Model Checking. In Proceedings of the 2014 Formal Methods in Computer-Aided Design (FMCAD), Lausanne, Switzerland, 21–24 October 2014; pp. 43–50.
47. Cabodi, G.; Loiacono, C.; Vendraminetto, D. Optimization techniques for Craig Interpolant compaction in Unbounded Model Checking. In Proceedings of the 2013 Design, Automation & Test in Europe Conference & Exhibition (DATE), Grenoble, France, 18–22 March 2013; pp. 1417–1422.

© 2019 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).