# Processor Hardware Security Vulnerabilities and their Detection by Unique Program Execution Checking

Mohammad Rahmani Fadiheh*, Dominik Stoffel*, Clark Barrett‡, Subhasish Mitra†‡, Wolfgang Kunz*

*Dept. of Electrical and Computer Engineering, Technische Universität Kaiserslautern, Germany
†Dept. of Electrical Engineering, Stanford University, Stanford, CA, USA
‡Dept. of Computer Science, Stanford University, Stanford, CA, USA

*Abstract*—**Recent discovery of security attacks in advanced processors, known as Spectre and Meltdown, has resulted in high public alertness about security of hardware. The root cause of these attacks is information leakage across *covert channels* that reveal secret data without any explicit information flow between the secret and the attacker. Many sources believe that such covert channels are intrinsic to highly advanced processor architectures based on speculation and out-of-order execution, suggesting that such security risks can be avoided by staying away from high-end processors. This paper, however, shows that the problem is of wider scope: we present new classes of covert channel attacks which are possible in average-complexity processors with in-order pipelining, as they are mainstream in applications ranging from Internet-of-Things to Autonomous Systems.**

**We present a new approach as a foundation for remedy against covert channels: while all previous attacks were found by clever thinking of human attackers, this paper presents a formal method called *Unique Program Execution Checking* which detects and locates vulnerabilities to covert channels systematically, including those to covert channels unknown so far.**

## I. Introduction

Subtle behaviors of microprocessors, at the level of their microarchitecture, are the root cause of security breaches demonstrated in the Spectre [1] and Meltdown [2] attacks. A microarchitectural side effect is an alteration of the clock cycle-accurate sequence of data in the program-visible registers of a processor during program execution, without affecting the program execution at the level of the Instruction Set Architecture (ISA). If these subtle alterations of program execution at the microarchitectural level can be caused by secret data, this may open a "side channel". An attacker may trigger and observe these alterations to infer secret information.

In microarchitectural side channel attacks, the possible leakage of secret information is based on some microarchitectural resource which creates an information channel between different software (SW) processes that share this resource. Various attacking schemes have been reported which can deduce critical information from the footprint of an encryption software on the cache [3], [4] or DRAM interface [5].

In these scenarios, the attacker process alone is not capable of controlling both ends of a side channel. In order to steal secret information, it must interact with another process initiated by the system, the "victim process", which manipulates the secret. This condition for an attack actually allows for remedies at the SW level which are typically applied to security-critical SW components like encryption algorithms. They prohibit the information flow at one end of the channel which is owned by the victim process.

This general picture was extended by the demonstration of the Spectre [1] and Meltdown [2] attacks. They constitute a new class of microarchitectural side channel attacks which are based on so called "covert channels" in hardware. These are special cases of microarchitectural side channels in which the attacker controls *both* ends of the channel, the part that triggers the side effect and the part that observes it. In this scenario, a single user-level attacker program, without the help of any other process, can establish a microarchitectural side channel that can leak the secret although it is not manipulated by any other program. Such HW covert channels not only can compromise the usefulness of encryption and secure authentication schemes, but can steal data from essentially anywhere in the system.

This paper presents new covert channels in average complexity processors that can have severe implications for a wide range of applications from Internet-of-Things (IoT) to Autonomous Systems where simple in-order processors are commonly used. Our results show that HW vulnerabilities by covert channels are not only a consequence of early architectural decisions on the features of a processor, such as out-of-order execution or speculative execution. In fact, they can also be introduced at a later design stage in the course of microarchitectural optimizations, targeting speed and energy, for example. Clearly, it cannot be expected from a designer to anticipate all "clever thinking" of potential attackers who attempt to create covert channels. Therefore, this paper is dedicated to presenting a new technique which *automatically* detects all microarchitectural side effects and points the designer to the HW components that may be involved in the possible creation of a covert channel.

The key contributions of this paper are as follows:

(i) This paper, for the first time, presents the experimental evidence that new kinds of covert channel attacks are also possible in simple in-order processors. We present the *Orc attack*, which uses a so far unknown type of covert channel.

(ii) We present a method for HW security analysis by Unique Program Execution Checking (UPEC). UPEC employs a formal analysis on the microarchitectural level (RTL). By employing the proposed UPEC methodology the designer can precisely assess during design time to what extent hardware security is affected by the detailed design decisions.

(iii) Based on UPEC, for the first time, covert channels can be detected by a systematic and largely automated analysis rather than only by anticipating the clever thinking of a possible attacker. UPEC can even detect previously unknown HW vulnerabilities, as demonstrated by the discovery of the Orc attack in our experiments.

An extended version of the paper can be found in [6].

## II. Related Work

Information flow tracking (IFT) has been widely used in the field of security for HW/SW systems. Its core idea is to enhance the hardware and/or the software in a way that the flow of information is explicitly visible [7], [8]. Also the CC-hunter technique instruments a processor to uncover covert channel communication at run time [9]. These methods incur high overhead on the design and demand modifications at different levels of the system, such as in the instruction set architecture (ISA), the operating system (OS) and the HW implementation.

Quantitative analysis of timing side channels by SVF [10] is a design-time technique to secure a system against illegal timing information flows. The main goal is to identify a reasonable trade-off between security and performance overheads rather than making a design provably secure.

Software verification techniques have been developed to verify security requirements in software [11], [12], [13]. However, the security threats in HW/SW systems are not limited to the software alone. Vulnerabilities can emerge from HW/SW interaction or, like in the case of Spectre and Meltdown, from the hardware itself. This category of security threats may only be visible at a specific level of hardware abstraction. It poses new challenges in modeling the information flow through hardware as well as in defining and solving the relevant verification tasks.

Extending information flow analysis with instruction level abstraction (ILA) is proposed in [14] to create a HW-dependent model of the software and formally verify security requirements at the HW/SW interface. Vulnerabilities at the microarchitectural HW level are, however, not covered. Other previous work on applying *formal* methods to detect security vulnerabilities in hardware is based on the idea of adopting notions of *taint analysis* for software in the HW domain. This was pioneered in [15], [16] and represents the research which is the most closely related to our work. In those approaches a HW security requirement, such as a specific confidentiality requirement, is formulated in terms of a *taint property* [15] along a certain path in the design. In order to formulate the properties in CTL, certain assumptions about the attack are required which significantly restrict the coverage of the method. As an alternative, a miter-based equivalence checking technique, with some resemblance to our computational model in Sec. IV, has been used in previous approaches [16], [17]. Although this increases the generality of the proof, it still restricts the attack to a certain path. Moreover, since some of this work considers verification at the architectural level, vulnerabilities based on microarchitectural side channels are not detectable.

Taint analysis by these approaches has shown promise for formal verification of certain problems in hardware security, for example, for proving key secrecy in an SoC. However, these methods demand making assumptions on what paths are suspicious. They require some "clever thinking" along the lines of a possible attacker. As a result, non-obvious or unprecedented side channels may be missed.

## III. Orc: A New Kind of Covert Channel Attack

For reasons of performance, many cache designs employ a pipelined structure which allows the cache to receive new requests while still processing previous ones. However, this can create a *Read-After-Write (RAW) Hazard* in the cache pipeline, if a load instruction tries to access an address for which there is a pending write.

A RAW hazard needs to be properly handled in order to ensure that the correct values are read. A straightforward implementation uses a dedicated hardware unit called *hazard detection* that checks for every read request whether or not there is a pending write request to the same cache line. If so, all read requests are removed until the pending write has completed. The processor pipeline is stalled, repeating to send read requests until the cache interface accepts them.

In the following, we show an example how such a cache structure can create a security vulnerability allowing an attacker to open a covert channel. Let's assume we have a computing system with a cache with write-back/write-allocate policy and the RAW hazard resolution just described. In the system, some confidential data (*secret data*) is stored in a certain protected location (*protected address*).

For better understanding of the example, let us make some more simplifying assumptions that, however, do not compromise the generality of the described attack mechanism. We assume that the cache holds a valid copy of the secret data (from an earlier execution of privileged code). We also simplify by assuming that each cache line holds a single byte, and that a cache line is selected based on the lower 8 bits

```
1: li x1, #protected_addr     // x1 ← #protected_addr
2: li x2, #accessible_addr    // x2 ← #accessible_addr
3: addi x2, x2, #test_value   // x2 ← x2 + #test_value
4: sw x3, 0(x2)               // mem[x2+0] ← x3
5: lw x4, 0(x1)               // x4 ← mem[x1+0]
6: lw x5, 0(x4)               // x5 ← mem[x4+0]
```

Fig. 1. Example of an Orc attack (RISC-V code)

of the address of the cached location. Hence, in our example, there are a total of $2^8 = 256$ cache lines.

The basic mechanism for the *Orc attack* is the following. Every address in the computing system's address space is mapped to some cache line. If we use the secret data as an address, then the secret data also points to some cache line. The attacker program "guesses" which cache line the secret data points to. It sets the conditions for a RAW hazard in the pipelined cache interface by writing to the guessed cache line. If the guess was correct then the RAW hazard occurs, leading to slightly longer execution time of the instruction sequence than if the guess was not correct. Instead of guessing, of course, the attacker program iteratively tries all 256 possible cache locations until successful.

Fig. 1 shows the instruction sequence for one such iteration. The shown `#test_value` represents the current guess of the attacker and sets the RAW hazard conditions for the guessed cache line. The sequence attempts an illegal memory access in instruction #5 by trying to load the secret data from the protected address into register `x4`. The processor correctly intercepts this attempt and raises an exception. Neither is the secret data loaded into `x4` nor is instruction #6 executed because the exception transfers control to the operating system with the architectural state of instruction #5. However, before control is actually transferred, instruction #6 has already entered the pipeline and has initiated a cache transaction. The cache transaction has no effect on the architectural state of the processor. But the execution time of the instruction sequence depends on the state of the cache. When probing all values of `#test_value`, the case will occur where the read request affects the same cache line as the pending write request, thus creating a RAW hazard and a stall in the pipeline. It is this difference in timing that can be exploited as a side channel.

Assuming the attacker knows how many clock cycles it takes for the kernel to handle the exception and to yield the control back to the parent process, the attacker can measure the difference in execution time and determine whether the lower 8 bits of the secret are equal to `#test_value` or not. By repeating the sequence for up to 256 times (in the worst case), the attacker can determine the lower 8 bits of the secret. If the location of the secret data is byte-accessible, the attacker can reveal the complete secret by repeating the attack for each byte of the secret. Hardware performance counters can further ease the attack since they make it possible to explicitly count the number of stalls.

This new covert channel can be illustrated at the example of the RISC-V Rocketchip [18]. The original Rocketchip design is not vulnerable to the Orc attack. However, with only a slight modification (17 lines of code (LoC) in an RTL design of $\sim$250,000 LoC) and without corrupting the functionality, it was possible to insert the vulnerability. The modifications actually optimized the performance of the design by bypassing a buffer in the cache, by which an additional stall between consecutive load instructions with data dependency was removed. There was no need to introduce any new state bits or to change the interface between the core and the cache. The attack does not require the processor to start from a specific state: any program can precede the code snippet of Fig. 1. The only requirement is that `protected_addr` and `accessible_addr` reside in the cache before executing the code in Fig. 1.

The described vulnerability is a very subtle one, compared to Meltdown and Spectre. It is caused by a RAW hazard not in the processor pipeline itself but in its interface to the cache. It is very hard for a designer to anticipate an attack scenario based on this hazard. The timing differences between the scenarios where the RAW hazard is effective and those where it isn't are small. Nevertheless, they are measurable and can be used to open a covert channel.

This new type of covert channel discovered by UPEC gives some important messages:

(i) Subtle design changes in standard RTL processor designs, such as adding or removing a buffer, can open or close a covert channel. Although specific to a particular design, such vulnerabilities may inflict serious damage, once such a covert channel becomes known in a specific product.

(ii) The *Orc attack* presented above is based on the interface between the core (a simple in-order core in this case) and the cache. This provides the insight that the *orchestration* of component communication in an SoC, such as RAW hazard handling in the core-to-cache interface, may also open or close covert/side channels. Considering the complex details of interface protocols and their implementation in modern SoCs, this can further complicate verifying security of the design against covert channel attacks.

(iii) The new insight that the existence of covert channels does not rely on certain types of processors but on decisions in the RTL design phase underlines the challenge in capturing such vulnerabilities and calls for methods which can deal with the high complexity of RTL models and do not rely on *a priori* knowledge about the possible attacks.

These challenges motivate the UPEC approach proposed in the following sections.

## IV. Unique Program Execution Checking (UPEC)

*Confidentiality* in HW/SW systems requires that an untrusted user process must not be able to read protected secret data. In case of a microarchitectural covert channel attack, the attacker cannot read the secret data directly. Nevertheless, confidentiality is violated because the execution timing of the attacker process depends on the secret data, and the timing information is measurable, e.g., through user-accessible counters. These timing differences may stem from various sources that need to be exhaustively evaluated when verifying confidentiality.

In the following, we refer to the computing system to be analyzed for security as System-on-Chip (SoC) and divide its state variables into two sets: state variables associated with the content of its memory (main memory and memory-mapped periphery) and state variables associated with all other parts of the hardware, the *logic parts*.

**Definition 1** (Microarchitectural State Variables)**.** The *microarchitectural state variables* of an SoC are the set of all state variables (registers, buffers, flip-flops) belonging to the *logic part* of the computing system's microarchitecture. □

A subset of these microarchitectural state variables are *program-visible*:

**Definition 2** (Architectural State Variables)**.** The *architectural state variables* of an SoC are the subset of microarchitectural state variables that define the state of program execution at the ISA level (excluding the program state that is represented in the program's memory). □

**Definition 3** (Secret Data, Protected Location)**.** A set of *secret data* $D$ is the content of memory at a *protected location* $A$, i.e., there exists a protection mechanism such that a user-level program cannot access $A$ to read or write $D$. □

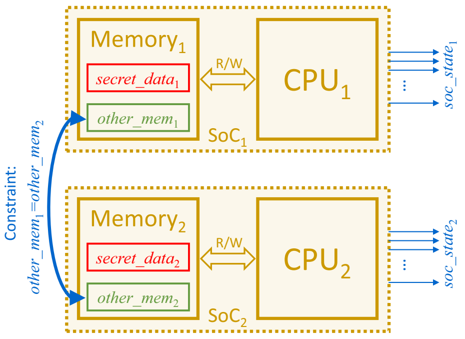

Fig. 2. Computational model for UPEC

The protected location may be in the main memory space, in peripherals or in other type of storage in the *non-logic part* of the computing system. In addition, there may exist temporary copies of the secret data in the cache system.

**Definition 4** (Unique Program Execution)**.** A program *executes uniquely w.r.t. a secret* $D$ if and only if the sequence of valuations to the set of architectural state variables is independent of the values of $D$, in every clock cycle of program execution. □

In other words, a user-level program executes uniquely if different secrets in the protected location do not lead to different values of the architectural states or to different time points when these values are assigned.

**Definition 5** (Confidentiality/Observability)**.** A set of secret data $D$ in a protected location $A$ is *confidential* if and only if any user-level program executes uniquely w.r.t. $D$. Otherwise $D$ is *observable*. □

Based on this definition, confidentiality is established by proving unique program execution at the RTL. In order to make property checking applicable to UPEC, we present a tailor-made computational model and formulate a specific property to be proven on this model.

Fig. 2 shows the model that is used in our UPEC approach. It can be derived automatically from the RTL description of the design and only requires the user to provide the protected memory region. In this model, $SoC_1$ and $SoC_2$ are two identical instances of the logic part of the SoC under verification. *Memory*$_1$ and *Memory*$_2$, as indicated in the figure, hold the same set of values except for the memory location of a defined secret data.

Based on this model, we propose the *UPEC property*: For a system to be secure w.r.t. covert channel attacks, the computational model derived from the design's RTL description has to fulfill the following CTL property:

$$\begin{aligned}\mathsf{AG}\,(&\mathit{secret\_data\_protected}\\ &\wedge \mathit{micro\_soc\_state}_1 = \mathit{micro\_soc\_state}_2\\ &\rightarrow \mathsf{AG}\,\mathit{soc\_state}_1 = \mathit{soc\_state}_2)\end{aligned} \tag{1}$$

In this formulation, *micro_soc_state* is a vector of all microarchitectural state variables, as defined in Def. 1, *soc_state* is some vector of state variables which includes, as a subset, all architectural state variables as defined in Def. 2 but not necessarily all other microarchitectural state variables. The constraint *secret_data_protected* specifies that a protection mechanism in the hardware is enabled for the secret data memory location.

The property in Eq. 1 fails if and only if, in the system under verification, there exists a state, *soc_state*, such that the transition to the next state, *soc_state*', depends on the secret data. This covers all situations in which program execution is not unique.

For reasons of computational complexity standard solutions of CTL model checking will fail so that a method specialized to this problem has been developed, as described in Sec. V.

Importantly, in our methodology we will consider situations where *soc_state*, besides the architectural state variables of the SoC, includes some or all microarchitectural state variables, such as the pipeline buffers. Producing a unique sequence for a superset of the architectural state variables represents a sufficient but not a necessary condition for unique program execution. This is because secret data may flow to microarchitectural registers which are not observable by the user program, i.e., they do not change program execution at any time, and, hence, no secret data is *leaked*.

We therefore distinguish the following kinds of counterexamples to the UPEC property:

**Definition 6** (L-alert)**.**
A *leakage alert (L-alert)* is a counterexample leading to a state with $soc\_state_1 \neq soc\_state_2$ where the differing state bits are *architectural* state variables. □

L-alerts indicate that secret data can affect the sequence of architectural states. This reveals a direct propagation of secret data into an architectural register (that would be considered a *functional* design bug), or a more subtle case of changing the timing and/or the values of the sequence without violating the functional design correctness and without leaking the secret directly. UPEC will detect the HW vulnerability in both cases. While the former case can be covered also by standard methods of functionally verifying security requirements, this is not possible in the latter case. Here, the opportunity for a covert channel attack may be created, as elaborated for the Orc attack in Sec. III.

**Definition 7** (P-alert)**.** A *propagation alert (P-alert)* is a counterexample leading to a state with $soc\_state_1 \neq soc\_state_2$ where the differing state bits are microarchitectural state variables that are not architectural state variables. □

P-alerts show possible propagation paths of secret data from the cache or memory to program-invisible, internal state variables of the system. A P-alert, very often, is a precursor to an L-alert, because the secret often traverses internal, program-invisible buffers in the design before it is propagated to an architectural state variable like a register in the register file.

The reason why *soc_state* in our methodology may also include program-invisible state variables will be further elaborated in the following sections. In principle, our method could be restricted to architectural state variables and L-alerts. P-alerts, however, can be used in our proof method as early indicators for a possible creation of a covert channel. This contributes to mitigating the computational complexity when proving the UPEC property.

## V. UPEC on a bounded model

Proving the property of Eq. 1 by classical unbounded CTL model checking is usually infeasible for SoCs of realistic size. Therefore, we pursue a SAT-based approach based on "*any-state proofs*" in a bounded circuit model. This variant of Bounded Model Checking (BMC) [19] is called Interval Property Checking (IPC) [20] and is applied to the UPEC problem in a similar way as in [21] for functional processor verification.

For proving the absence of L-alerts by our bounded approach, in the worst case, we need to consider a time window as large as the sequential depth, $d_{\mathrm{SOC}}$, of the logic part of the examined SoC. However, employing a symbolic initial state

```
assume:
    at t:              secret_data_protected();
    at t:              micro_soc_state_1 = micro_soc_state_2;
    at t:              no_ongoing_protected_access();
    during t..t + k:   cache_monitor_valid_IO();
prove:
    at t + k:          soc_state_1 = soc_state_2;
```

Fig. 3. UPEC property (Eq. 1) formulated as interval property

enables the solver to often capture hard-to-detect vulnerabilities within much smaller time windows. A violation of the UPEC property is actually guaranteed to be indicated by a P-alert in only a single clock cycle needed to propagate secret data into some microarchitectural state variable of the logic part of the SoC. In practice, however, it is advisable to choose a time window for the bounded model which is as long as the length, $d_{\mathrm{MEM}}$, of the longest memory transaction. When the secret is in the cache, $d_{\mathrm{MEM}}$ is usually the number of clock cycles for a cache read. When the secret is not in the cache, $d_{\mathrm{MEM}}$ is the number of clock cycles the memory response takes until the data has entered the cache system, e.g., in a read buffer. This produces P-alerts of higher diagnostic quality and provides a stronger basis for inductive proofs that may be conducted subsequently, as discussed in Sec. VI.

Proving a property by IPC is usually more challenging than by BMC due to spurious counterexamples and proof complexity. In UPEC, we address this challenge in a structured way by adding well-defined constraints to the proof without restricting its generality. Since both SoC instances of our computational model start with the same initial state, all of the unreachable initial states and spurious behaviors have the same manifestation in both SoC instances and therefore do not violate the uniqueness property, except for the states related to the memory elements holding the secret value.

*Constraint 1, "no on-going protected accesses"*. A privileged process can freely access protected memory regions and copy its content to user-accessible registers. In order to exclude such an explicit leakage scenario, the proof must be constrained to exclude such initial states that implicitly represent ongoing memory transactions in which protected memory regions are accessed.

*Constraint 2, "cache I/O is valid"*. In order to exclude spurious behaviors of the cache controller, we ensure the protocol compliance and valid I/O behavior of the cache by instrumenting the RTL verification model with a special cache monitor.

It should be noted that these constraints do not restrict the generality of our proof. They are, actually, invariants of the global system. Their validity follows from the functional correctness of the operating system (OS) and the SoC. More details on constraints can be found in [6].

The interval property for UPEC is shown in Fig. 3. The macro *secret_data_protected*() denotes that in both SoC instances, a memory protection scheme is enabled in the hardware for the memory region holding the secret data. The macro *no_ongoing_protected_access*() implements constraint 1, and macro *cache_monitor_valid_IO*() implements constraint 2.

## VI. Methodology

Fig. 4 shows the general flow of UPEC-based HW security analysis. Checking the UPEC property (Eq. 1) is at the core of a systematic, iterative process by which the designer identifies and qualifies possible HW vulnerabilities in the design. The UPEC property is initialized on a bounded model of length $d_{\mathrm{MEM}}$ and with a proof assumption and obligation for the complete set of microarchitectural state variables.

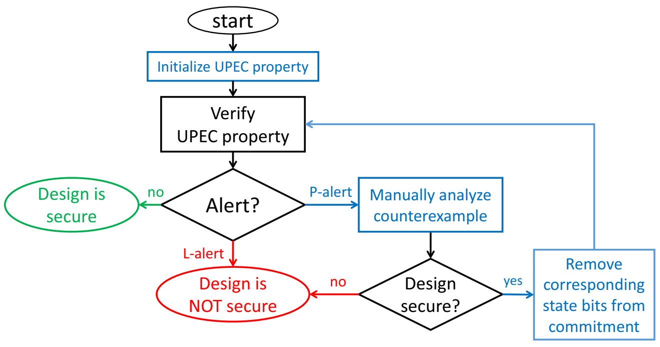

Fig. 4. UPEC Methodology

If the UPEC property can be successfully verified, then the design is proven to be free of side effects that can be exploited as covert channels. If the property fails it produces a counterexample which can be either an L-alert or a P-alert. An L-alert exposes a measurable side effect of the secret data on the architectural state variables, rendering the design insecure. A P-alert documents a side effect of the secret data on microarchitectural state variables that are not directly accessible by the attacker program. In principle, the designer can now remove the affected microarchitectural state variables from the proof obligation of the UPEC property (while keeping the complete set of microarchitectural state variables in the proof assumption), and then re-iterate the process to search for a different counterexample. The process is bound to terminate because, eventually, either an L-alert occurs or the design is secure.

In practice, however, the designer will not simply eliminate a P-alert but instead will analyze the counterexample. As mentioned before, an L-alert may have one or several shorter P-alerts as precursors. Since the P-alert is a shorter counterexample than the corresponding L-alert it can be computed with less computational effort, including shorter proof times. A P-alert belonging to an L-alert documents the earliest manifestation of a side effect and points the designer to the source of the vulnerability that causes the side effect. If the security compromise is already obvious from the P-alert the designer may abort the iterative process of Fig. 4 by deeming the P-alert as "insecure" and change the RTL in order to remove the vulnerability. This may be as simple as adding or removing a buffer. Note that if the designer wrongly deems a P-alert as "secure" then the security compromise is detected by another P-alert later, or, eventually, by an L-alert. The penalty for making such a mistake is the increase in run times for checking the later UPEC property instances.

If the procedure terminates without producing an L-alert this is not a complete proof that the design is secure, unless we increment the length of the model to $d_{\text{SOC}}$. The alternative is to take the P-alerts as starting point for proving security by an inductive proof of the property in Eq. 1 for the special case of an initial state derived from the considered P-alert. A P-alert can be deemed as secure if the designer knows that the values in the affected microarchitectural state variables will not propagate under the conditions under which the P-alert has occurred. In other words, in a secure P-alert, a certain condition holds which implicitly represents the fact that the propagation of secret data to architectural registers will be blocked. In order to conduct the inductive proof the designer must identify these blocking conditions for each P-alert. Based on the UPEC computational model (Fig. 2) the inductive proof checks whether or not the blocking condition always holds for the system once the corresponding P-alert has been reached.

Finally, there is always the conservative choice of making design changes until no P-alerts occur anymore, thus, establishing full security for the modified design w.r.t. covert channels.

TABLE I
UPEC METHODOLOGY EXPERIMENTS

| | $D$ cached | $D$ not cached |
|---|---|---|
| $d_{\text{MEM}}$ | 5 | 34 |
| Feasible $k$ | 9 | 34 |
| # of P-alerts | 20 | 0 |
| # of RTL registers causing P-alerts | 23 | N/A |
| Proof runtime | 3 hours | 35 min |
| Proof memory consumption | 4 GB | 8 GB |
| Inductive proof runtime | 5 min | N/A |
| Manual effort | 10 person days | 5 person hours |

## VII. EXPERIMENTS

We explored the effectiveness of UPEC by targeting different design variants of Rocketchip [18], an open-source RISC-V SoC generator. The considered Rocketchip design is a single-core SoC with in-order pipelined processors and separate data and instruction level-1 caches. All results were obtained using the commercial property checker OneSpin 360 DV-Verify on an Intel Core i7-6700 CPU with 32 GB of RAM, running at 3.4 GHz.

In order to evaluate the effectiveness of UPEC for capturing vulnerabilities we targeted the original design of Rocketchip and two design variants made vulnerable to (a) a Meltdown-style attack and (b) an Orc attack, with only minimal design modifications. Functional correctness was not affected and the modified designs successfully passed all tests provided by the RISC-V framework. UPEC successfully captured all vulnerabilities. In addition, UPEC found an ISA incompliance in the *Physical Memory Protection* (PMP) unit of the original design.

For the *Meltdown-style attack* we modified the design such that a cache line refill is not canceled in case of an invalid access. While the illegal access itself is not successful but raises an exception, the cache content is modified and can be analyzed by an attacker. We call this a Meltdown-style attack since the instruction sequence for carrying out the attack is similar to the one reported by [2]. Note, however, that in contrast to previous reports we create the covert channel based on an in-order pipeline.

For the *Orc attack*, we conditionally bypassed one buffer, as described in Sec. III, thereby creating a vulnerability that allows an attacker to open a covert timing side channel.

In our experiments, the secret data is assumed to be in a protected location, $A$, in the main memory. Protection was implemented using the *Physical Memory Protection (PMP)* scheme of the RISC-V ISA by configuring the memory region holding the location $A$ of the secret data as inaccessible in user mode.

### A. Experiments on the original Rocketchip design

We conducted experiments on the *original* design for two cases: (1) $D$ initially resides in the data cache and main memory, and, (2) $D$ initially only resides in the main memory; cf. the columns labeled "$D$ in cache" and "$D$ not in cache" in Tab. I.

For the experiment with $D$ not in the cache, UPEC proves that there exists no P-alert. This means that the secret data cannot propagate to any part of the system and therefore, the user process cannot fetch the secret data into the cache or access it in any other way. As a result, the system is proven to be secure for the case that $D$ is not in the cache. Since the property proves already in the first iteration of the UPEC

TABLE II
DETECTING VULNERABILITIES IN MODIFIED DESIGNS

| Design variant/vulnerability | Orc | Meltdown-style |
|---|---|---|
| Window length for P-alert | 2 | 4 |
| Proof runtime for P-alert | 1 min | 1 min |
| Window length for L-alert | 4 | 9 |
| Proof runtime for L-alert | 3 min | 18 min |

methodology that there is no P-alert, the verification can be carried out within few minutes of CPU time and without any manual analysis.

For the case that $D$ is initially in the cache, we need to apply the iterative UPEC methodology (Fig. 4) in order to find all possible P-alerts. We also tried to capture an L-alert by increasing the length $k$ of the time window, until the solver aborted because of complexity. The second row in the table shows the maximum $k$ that was feasible. The following rows show the computational effort for this $k$.

Each P-alert means that the secret influences certain microarchitectural registers. As elaborated in Sec. VI, using standard procedures of commercially available property checking, we can establish proofs by mathematical *induction*, taking the P-alerts as the *base case* of the induction. In this way, we proved security from covert channels also for the case when the secret is in the cache. The manual effort for this lies within a few person days and is small compared to the total efforts for processor design. The complexity of an inductive proof for one selected case of the P-alerts is shown in Table I as an example.

### B. Experiments on the modified Rocketchip designs

Table II shows the proof complexity for finding the vulnerabilities in the *modified* designs. For each case, the UPEC methodology produced meaningful P-alerts and L-alerts. When incrementing the window length in search for an L-alert, new P-alerts occurred which were obvious indications of security violations. None of these violations exploits any branch prediction of the Rocketchip. For example, in the Meltdown-style vulnerability, within seconds UPEC produced a P-alert in which the non-uniqueness manifests itself in the valid bits and tags of certain cache lines. This is a well-known starting point for side channel attacks so that, in practice, no further examinations would be needed. However, if the designer does not have such knowledge the procedure may be continued without any manual analysis until an L-alert occurs. This took about 18 min of CPU time. For the design vulnerable to an Orc attack the behavior was similar, as detailed in Table II.

### C. Violation of memory protection in Rocketchip

UPEC also found a case of ISA incompliance in the implementation of the RISC-V PMP mechanism in Rocketchip. The PMP mechanism was not correctly implemented in Rocketchip. The PMP implementation allowed the modification of a locked memory configuration in privileged mode without requiring a reboot. This is clearly a vulnerability, and a bug with respect to the specification which was detected by applying UPEC. (In the current version of Rocketchip, the bug has already been fixed.)

## VIII. CONCLUSION

This paper has shown that covert channel attacks are not limited to high-end processors but can affect a larger range of architectures. While all previous attacks were found by clever thinking of a human attacker, this paper presented UPEC, an automated method to systematically detect all vulnerabilities by covert channels, including those by covert channels unknown so far. Future work will explore measures to improve the scalability of UPEC to handle larger processors. By automating the construction of induction proofs described in Sec. VI, the UPEC computational model can be restricted to only two clock cycles, thus, drastically reducing computational complexity. In addition, a compositional approach to UPEC will be explored.

ACKNOWLEDGMENT

We thank Mark D. Hill (U. of Wisconsin), Quinn Jacobson (Achronix Semiconductor Corp.) and Simha Sethumadhavan (Columbia U.) for their valuable feedback. The reported research was partly supported by BMBF KMU-Innovativ 01IS17083C (Proforma) and by DARPA.

## REFERENCES

[1] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," *arXiv preprint arXiv:1801.01203*, 2018.
[2] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," *arXiv preprint arXiv:1801.01207*, 2018.
[3] C. Percival, "Cache missing for fun and profit," in *BSDCan*, 2005. [Online]. Available: http://www.daemonology.net/papers/htt.pdf
[4] Y. Yarom and K. Falkner, "FLUSH+ RELOAD: a high resolution, low noise, L3 cache side-channel attack." in *USENIX Security Symposium*, vol. 1, 2014, pp. 22–25.
[5] P. Pessl, D. Gruss, C. Maurice, and S. Mangard, "Reverse engineering intel DRAM addressing and exploitation," *ArXiv e-prints*, 2015.
[6] M. R. Fadiheh, D. Stoffel, C. Barrett, S. Mitra, and W. Kunz, "Processor hardware security vulnerabilities and their detection by unique program execution checking," *arXiv preprint arXiv:1812.04975*, 2018.
[7] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *ACM Sigplan Notices*, vol. 39, no. 11. ACM, 2004, pp. 85–96.
[8] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood, "Complete information flow tracking from the gates up," in *ACM Sigplan Notices*, vol. 44, no. 3. ACM, 2009, pp. 109–120.
[9] J. Chen and G. Venkataramani, "CC-Hunter: Uncovering covert timing channels on shared processor hardware," in *Annual IEEE/ACM Intl. Symp. on Microarchitecture*. IEEE, 2014, pp. 216–228.
[10] J. Demme, R. Martin, A. Waksman, and S. Sethumadhavan, "Side-channel vulnerability factor: A metric for measuring information leakage," in *Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2012, pp. 106–117.
[11] M. R. Clarkson and F. B. Schneider, "Hyperproperties," *Journal of Computer Security*, vol. 18, no. 6, pp. 1157–1210, 2010.
[12] D. Von Oheimb and S. Mödersheim, "ASLan++a formal security specification language for distributed spercystems," in *Intl. Symp. on Formal Methods for Components and Objects*. Springer, 2010, pp. 1–22.
[13] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *IEEE symposium on Security and privacy (SP)*. IEEE, 2010, pp. 317–331.
[14] P. Subramanyan, S. Malik, H. Khattri, A. Maiti, and J. Fung, "Verifying information flow properties of firmware using symbolic execution," in *Design, Automation & Test in Europe Conference (DATE)*. IEEE, 2016, pp. 337–342.
[15] P. Subramanyan and D. Arora, "Formal verification of taint-propagation security properties in a commercial SoC design," in *Design, Automation & Test in Europe Conference (DATE)*. IEEE, 2014, p. 313.
[16] 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.
[17] W. Hu, A. Ardeshiricham, and R. Kastner, "Identifying and measuring security critical path for uncovering circuit vulnerabilities," in *International Workshop on Microprocessor and SOC Test and Verification (MTV)*. IEEE, 2017.
[18] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz *et al.*, "The rocket chip generator," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17*, 2016.
[19] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, ser. TACAS '99. London, UK, UK: Springer-Verlag, 1999, pp. 193–207.
[20] M. D. Nguyen, M. Thalmaier, M. Wedler, J. Bormann, D. Stoffel, and W. Kunz, "Unbounded protocol compliance verification using interval property checking with invariants," *IEEE Transactions on Computer-Aided Design*, vol. 27, no. 11, pp. 2068–2082, November 2008.
[21] M. R. Fadiheh, J. Urdahl, S. S. Nuthakki, S. Mitra, C. Barrett, D. Stoffel, and W. Kunz, "Symbolic quick error detection using symbolic initial state for pre-silicon verification," in *Design, Automation & Test in Europe Conference (DATE)*. IEEE, 2018, pp. 55–60.