MCU-Wide Timing Side Channels and Their Detection

Johannes Müller[1], Anna Lena Duque Antón[1], Lucas Deutschmann[1], Dino Mehmedagić[1], Cristiano Rodrigues[2], Daniel Oliveira[2], Keerthikumara Devarajegowda[3], Mohammad Rahmani Fadiheh[4], Sandro Pinto[2], Dominik Stoffel[1], and Wolfgang Kunz[1]
[1]RPTU Kaiserslautern-Landau, [2]Universidade do Minho, [3]Siemens EDA, [4]Stanford University

ABSTRACT

Microarchitectural timing side channels have been thoroughly investigated as a security threat in hardware designs featuring shared buffers (e.g., caches) and/or parallelism between attacker and victim task execution. However, contradicting common intuitions, recent activities demonstrate that this threat is real even in microcontroller SoCs without such features. In this paper, we describe SoC-wide timing side channels previously neglected by security analysis and present a new formal method to close this gap. In a case study on the RISC-V Pulpissimo SoC, our method detected a vulnerability to a previously unknown attack variant that allows an attacker to obtain information about a victim's memory access behavior. After implementing a conservative fix, we were able to verify that the SoC is now secure w.r.t. the considered class of timing side channels.

KEYWORDS

Timing Side Channels, Formal Verification, Hardware Security

1 INTRODUCTION

For the last three decades, microarchitectural timing side channels have been a subject of interest in academia and industry. Timing side channel attacks extract confidential data by measuring the time it takes to process data or by examining footprints that confidential data leaves in the system. While such attacks were demonstrated to exploit a wide range of microarchitectural features, they mostly target systems that allow parallelism between attacker and victim execution or systems featuring buffers such as caches that can hold a footprint of the confidential data.

By contrast, microcontroller units (MCUs) are constructed with relatively simple processor cores and usually offer neither attacker concurrency nor footprint buffers. This may explain why they are neglected in the timing side channel literature.

Recently, however, it has been shown that even simple MCUs can be vulnerable to timing side channel attacks [1, 11, 15]. The reported attacks exploit interrupts [15] and memory bus arbitration [1, 11] to bypass security infrastructure and steal confidential data. The BUSted attack [11], in particular, demonstrates a new class of timing side channels that exploit the plethora of complex IPs and on-chip communication structures (buses, interconnect) that MCUs come with: An attacker task executing on the processor core accesses peripheral IPs and instructs them to actively spy on a future execution of confidential programs by a victim. Instead of analyzing unintentional footprints left by the victim, the attacker makes the spying IPs collect confidential information about the victim's memory accesses and save it in their own state.

In practice, the threat of vulnerable MCUs is of particular relevance since such systems are found in virtually every industrial application domain. Apparently, a better understanding of SoC-wide timing side channels is needed, along with a systematic approach to detect them. New formal methods must be developed capable of providing formal security guarantees in MCUs. In case a side channel is detected, effective countermeasures are required to reliably remove the vulnerability. This paper addresses these needs:

- We present, to the best of our knowledge, the first formal verification method to detect timing side channels in MCU-style SoCs. The method is exhaustive w.r.t. the considered threat model. We exploit critical characteristics of MCUs to formulate bounded properties spanning as little as two clock cycles and still obtain unbounded proof results, i.e., covering attacks that span potentially thousands of clock cycles. (Sec. 3)
- With the proposed method, we discovered a vulnerability to a previously unknown variant of the BUSted [11] attack. In this scenario, an accelerator IP and a memory device can be leveraged to record and retrieve the memory access behavior of a victim task. Compared to previous attack variants, no timer IP is required. (Sec. 4.1)
- We conducted a case study on the Pulpissimo SoC [12]. We propose and implemented a countermeasure against timing side channels in SoCs and could prove that the system is secure. (Sec. 4.2)

2 TIMING SIDE CHANNELS IN SOCS

This paper focuses on timing side channels in MCU SoCs. In this section, we describe the threat model (Sec. 2.1) and illustrate how timing side channels in SoCs can be exploited (Sec. 2.2).

2.1 Threat model

We assume a system featuring a single-threaded single-core processor (simply called *CPU* in this paper), and several peripheral IPs and on-chip communication structures, as is commonplace in MCUs. The system supports multitasking by time multiplexing, i.e., individual tasks do not execute simultaneously but are separated by a context switch. The security threat considered in this work

This work was supported partly by Bundesministerium für Bildung und Forschung Scale4Edge, grant no. 16ME0122K-16ME0140+16ME0465, by Intel Corp., Scalable Assurance Program and by Siemens EDA.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
DAC '24, June 23–27, 2024, San Francisco, CA, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 979-8-4007-0601-1/24/06…$15.00
https://doi.org/10.1145/3649329.3656541

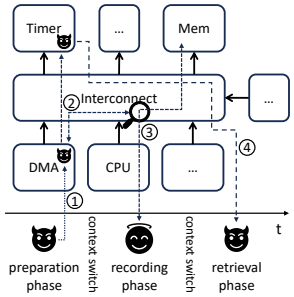


Figure 1: Example of timing side channel attack on an MCU

consists of an *attacker task* and a *victim task* executing on the CPU, with the attacker trying to obtain confidential information about the victim execution. Our threat model addresses SoC-wide microarchitectural side channels. Therefore, we can assume that the attacker has no direct access to the memory regions allocated to the victim task and that the victim does not store confidential data in attacker-accessible memory regions or IP registers. Since the focus of this work is on MCU-style SoCs without caches or special-purpose microarchitectural buffers, such as a branch history table, we can further assume in our threat model that the confidential information does not leave footprints in the CPU.

2.2 MCU timing side channels in a nutshell

A timing side channel attack on an MCU SoC usually consists of many steps [11]. For the purpose of this work, we concentrate on the key part of such attacks, i.e., we describe how confidential information can be obtained by an attacker. We refer the most interested readers to related work [11] for a complete understanding of the attack methodology. In this work, we divide the exploitation of timing side channels in MCUs into 3 phases which are separated by context switches:

- *Preparation:* The attacker task executes instructions on the core. In the preparation phase, IPs in the SoC are configured to "spy" on activities in the system.
- *Recording:* The second phase of the attack window begins with a context switch to the victim task. The victim performs security-critical operations while, simultaneously, the spying IPs collect information about the execution and store it in the system state.
- *Retrieval:* In the third phase, control of the core is switched back to the attacker task. The attacker retrieves the confidential information stored in the state of the system IPs.

As a simple example, consider an MCU SoC featuring, amongst other IPs, a simple CPU, memory, a DMA and a timer, as illustrated in Fig. 1. In the preparation phase, the attacker instructs the DMA to perform some memory accesses and afterwards start the timer ①. After a context switch to the victim task, the recording phase of the attack begins. As instructed by the attacker, the DMA performs the memory accesses and then starts the timer ②. If the victim task tries to access the memory concurrently with the DMA, contention is created ③. As a result, the start of the timer is delayed. In the retrieval phase of the attack, the attacker task reads the timer state or waits for a timer overflow event ④. Based on the timer state, the attacker can deduce information about the victim task's memory accesses. This information can then be used to break the isolation provided by the security infrastructure, as previous attacks have demonstrated [11].

3 FORMAL DETECTION METHOD

3.1 Basic Idea

This work presents a novel formal verification method for detecting vulnerabilities to SoC-wide timing side channel attacks, as described in Sec. 2. The method builds upon an existing security verification technique called Unique Program Execution Checking (UPEC) [3]. One key ingredient of UPEC is the underlying computational model which is a *bounded* one, i.e., its properties describe finite time windows. However, it employs a symbolic starting state and thus generates proof results with *unbounded validity*. Nonetheless, while it is, in principle, possible to describe the entire time window in which an attack occurs in a finite UPEC property, this entails hundreds, if not thousands of clock cycles of behavior. Verifying such properties for realistic MCU designs is computationally infeasible.

Fortunately, it is not necessary to capture the entire three-phase attack sequence described above in a UPEC property, as we now explain.

Observation 1. All victim behavior, which is considered confidential per our threat model, can be spied upon in the recording phase only after the victim task accesses memory or peripheral devices. In the HW implementation these accesses are sent and received at the interface between the CPU and the rest of the system.

Understanding the implications of this observation requires a closer look at the system. We inspect the design under verification at the Register Transfer Level (RTL). This allows us to reason in terms of the state variables, i.e., all state-holding elements (incl. memory), of the design. We denote the set of all state variables as S_{all} and define the following subset:

Definition 1. The set $S_{\neg victim} \subset S_{all}$ contains all state variables in the system except for the ones belonging

(1) to the CPU or
(2) to the memory locations allocated to the victim task.

Specifying $S_{\neg victim}$ is usually straightforward and is achieved by simple structural analysis of the RTL model and by symbolic modeling of memory address ranges. We provide more details on this process in Sec. 3.4. Following Obs. 1, it becomes clear that spying on the victim is possible only if at least one state variable of $S_{\neg victim}$, such as a control register of the interconnect, is affected during the recording phase by a dependency on the victim's confidential data. This helps us to reduce the property time window by excluding the preparation phase, as illustrated in Fig. 2 (a). Using a symbolic starting state, the time window starts when the victim execution affects the state variables in $S_{\neg victim}$ for the first time, during the recording phase.

We thereby decrease the number of clock cycles the property has to cover while still capturing any spying on security-critical victim behavior. However, describing the retrieval phase in the property, i.e., the information retrieval from the spying IPs, may require the property to span some additional hundreds of clock

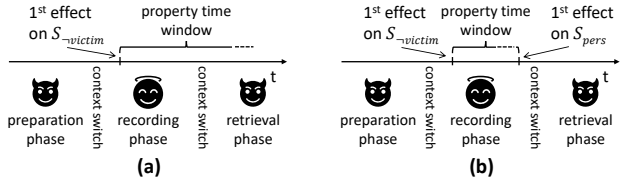


Figure 2: Reduced property time window, (a) after applying Obs. 1, and (b) after applying Obs. 1 and Obs. 2

cycles, especially because this phase includes the context switch from victim to attacker. A second observation helps us to cope with this issue:

Observation 2. For confidential information to be retrievable by an attacker task, it must persist in state variables in the system across a context switch.

Definition 2. We define $S_{pers} \subset S_{\neg victim}$ as the set of state variables

(1) that are accessible to the attacker task, and
(2) that are *persistent* across a context switch from the attacker to the victim task.

Any attacker exploitation requires confidential information to end up in S_{pers} at the end of the recording phase. With this observation the property time window can be further reduced because we only need to prove that the victim execution cannot affect any persistent state variables S_{pers} in the design. As illustrated in Fig. 2 (b), this gives us a definitive end point of the property time window. We provide details on how to obtain S_{pers} in Sec. 3.4 and how to prove by induction that the end point is valid.

In the remaining parts of this section, we sketch the UPEC computational model (Sec. 3.2) and then concretize the above observations to formulate and prove bounded UPEC properties that span only a small number of clock cycles while achieving global, unbounded proof validity (Sec. 3.3, 3.4 and 3.5).

3.2 Background: UPEC computational model

UPEC was originally introduced as an exhaustive detection method for transient execution attacks in CPUs. However, recent research has demonstrated its adaptability and its successful application to other security verification problems [2, 7, 8]. UPEC is based on Interval Property Checking (IPC) [14] at the RTL. In IPC, properties are formulated over a finite number of clock cycles using the RTL design's state variables and signals. For checking a property, the IPC solver employs a symbolic starting state which models all possible histories of inputs to the design. This is different from bounded model checking, where a concrete starting state is used.

The goal of a UPEC proof is to guarantee that an attacker task cannot extract confidential data located in the memory. This is achieved by a special 2-safety computational model: The RTL design is instantiated twice and all state variables of the design are assumed to be equal between the two instances, except for the confidential data. The UPEC property assumes an attacker task to execute on the processor core and proves *uniqueness* of the task execution, i.e., independence of the execution on the confidential data. This is checked by verifying that any difference between the two instances, which must originate from confidential data, cannot propagate to the attacker task and influence its execution.

UPEC-SSC(S):
assume:
during $t..t+1$: *Victim_Task_Executing()*;
during $t..t+1$: *Primary_Input_Constraints()*;
at t: *State_Equivalence(S)*;
prove:
at $t+1$: *State_Equivalence(S)*;

Figure 3: 2-cycle UPEC-SCC property

3.3 The UPEC-SSC method

We exploit Obs. 1 and 2, as described above, and formulate a UPEC property spanning only two clock cycles, as depicted in Fig. 3. The property asserts the macro *Primary_Input_Constraints()* which constrains the primary inputs to be equal between the two instances. With the macro *Victim_Task_Executing()* the property assumes that all address ranges representing memory regions or device registers allocated to the victim task are not directly accessible by potentially spying IPs in the SoC. This restricts the property to our threat model. In addition, the macro constrains victim accesses to protected addresses to be different between the two system instances while accesses to all other addresses are equal. Thus, only protected accesses are modeled as confidential information. The property makes no restrictions regarding the actual program executed as victim task; any possible software is considered by the IPC solver. Furthermore, the address ranges allocated to the victim task are modeled symbolically to represent any possible allocation. The macro *State_Equivalence(S)* is parameterized with a set of state variables S. It requires all state variables in S to be equal to their counterparts in the other SoC instance. We use S to model the part of the system that has not been affected by the victim behavior and thus cannot hold confidential information at clock cycle t. The property proves that also in clock cycle $t+1$ the unaffected part of the system remains unaffected by the confidential victim information.

This property is the centerpiece of the proposed *UPEC for System Side Channels (UPEC-SSC)* method. The verification procedure is sketched in Alg. 1. It starts with initializing S with $S_{\neg victim}$, i.e., the system state that is unaffected by the victim task execution before the victim accesses any component outside the core. Then, the UPEC-SSC property is checked for the state set S using an IPC solver and a counterexample is produced. The counterexample shows a violation of the property, i.e., a set of state variables S_{cex} that are different from their counterparts in the other design instance. We distinguish three cases for S_{cex}: In the first case, the property holds and $S_{cex} = \emptyset$. This means the victim cannot influence additional state variables, the design is secure, and the procedure terminates. In the second case, the property fails and S_{cex} includes elements from S_{pers}, i.e., data about the victim execution can propagate to a persistent IP, where it can be retrieved later by the attacker task. This case constitutes a vulnerability to an attack. In the third case, the property fails, but S_{cex} does not include elements from S_{pers}. In this case, the victim execution influences new state variables, but they cannot hold secret data across a context switch to the attacker task. As a result, the state variables in S_{cex} are removed from the non-influenced set of states S for the next check of the property. In every iteration of the *while* loop, the set of states variables S to which secret information from the victim can possibly propagate,

is reduced. This is repeated until, eventually, the design is either proven secure or vulnerable.

For a secure design, the procedure in Alg. 1 computes a set of state variables, S, for which property UPEC-SSC(S) becomes inductive. In the final iteration that leads to return *secure* (line 6), S has reached a fixed point in which property UPEC-SSC(S) (called in line 4) becomes the *step* of the induction. It proves, for the final set S, that if the victim has not influenced any state variable in S at a time t, it will also not do so at any time in the future. The induction *base* is given trivially by the fact that initially, before the victim becomes active on the CPU/system interface, no state variable has been influenced at all. For the final set S it holds that $S_{pers} \subset S \subset S_{\neg victim}$. Hence, it is proven that the victim has not caused any change of the system state that could be visible to the attacker after a context switch.

Algorithm 1 UPEC-SSC Procedure

```
1:  procedure UPEC-SSC
2:      S ← S_¬victim
3:      while true do
4:          S_cex ← check(UPEC-SSC(S))
5:          if S_cex = ∅ then
6:              return secure
7:          else if S_cex ∩ S_pers ≠ ∅ then
8:              return (vulnerable, S_cex)
9:          else if S_cex ∩ S_pers = ∅ then
10:             S ← S \ S_cex
```

3.4 UPEC-SCC state variables

The UPEC-SSC procedure proves non-interference of the victim task with the persistent states in the SoC, under the given constraints and for an arbitrary behavior of the system IPs.

The procedure requires the specification of the two state variable sets $S_{\neg victim}$ and S_{pers}. The set $S_{\neg victim} \subset S_{all}$ is compiled by excluding all state variables in the core and the victim memory space, from the set of all state variables in the SoC. Obtaining the state variables in the core is straightforward and requires simple structural analysis with minimal manual input from a verification engineer. The victim memory space, i.e., the memory regions containing the victim program and data, are determined by address ranges in the memory devices of the SoCs. We model the address ranges symbolically. This includes all possible victim memory layouts into the properties and allows the proofs to be independent of any particular victim software. The set of all state variables that are persistent, S_{pers}, is more challenging to compute. Determining this set for the entire SoC might be infeasible. This is, however, not needed. The procedure only requires us to specify whether state variables are part of S_{pers} if they appear in a counterexample. In practice, for the considered property, most of these state variables are buffers in the interconnect which are overwritten with every communication transaction and cannot be used to hold persistent information. Therefore, they are not part of S_{pers}. The remaining state variables included in counterexamples are usually registers or memory arrays in IPs and must be considered persistent. Some rare counterexamples may involve state variables that are neither buffers in the interconnect nor obviously persistent registers in IPs. These cases require closer inspection and additional proofs to qualify or disqualify them as part of S_{pers}. However, in all our experiments we have not observed any such cases. In fact, conservative fixes can block propagation to state variables beyond the interconnect, as discussed in Sec. 4.2.

UPEC-SSC-unrolled($k, S[\,]$):

```
assume:
    during t..t+1:      Victim_Task_Executing();
    during t..t+k:      Primary_Input_Constraints();
    at t:               State_Equivalence(S[0]);
prove:
    at t+1:             State_Equivalence(S[1]);
    ...
    at t+k:             State_Equivalence(S[k]);
```

Figure 4: Multi-cycle UPEC-SCC property

IPC properties, such as the employed UPEC-SSC property from Fig. 3, can produce false counterexamples. This happens when the solver assumes an unreachable state as its starting state. The standard solution in IPC is to employ invariants that exclude the unreachable behavior from the symbolic starting state. In some applications of IPC, formulating and proving invariants may require significant manual effort. The false counterexamples encountered in our experiments, however, involve only few state variables and the associated invariants are straightforward to formulate.

3.5 Generating longer counterexamples

The UPEC-SSC property in Fig. 3 proves a design under verification to be either secure or vulnerable within two clock cycles. Counterexamples to the property also span two clock cycles. Behavior affecting multiple state variables at multiple time points is included *implicitly* in the computed starting state of the counterexamples. In many cases, such implicit representations of behavior are cryptic and the corresponding counterexamples are of limited use to a verification engineer. Therefore, it is desirable to obtain longer counterexamples containing all signal valuations *explicitly*. Such counterexamples can be generated by unrolling the property over more than two clock cycles. Compared to the two-clock-cycle version from Fig. 3, the unrolled property, described in Fig. 4, has an additional parameter k that controls the number of unrollings. Secondly, the property uses a vector of state sets, $S[\,]$, to specify for each clock cycle a set of states which are not influenced by the victim task at this particular clock cycle.

The unrolled property leads us to the unrolled UPEC-SSC procedure Alg. 2. It starts by initializing the two sets $S[0]$ and $S[1]$ with $S_{\neg victim}$ and the parameter k with 1. The property is then checked with the initialized parameters. Note that this configuration of the property is identical to the first iteration in the original procedure in Alg. 1. If the property fails due to a difference in a persistent state, a vulnerability is detected. For all other counterexamples, the diverging state variables are removed from the set of affected state variables $S[1]$, for the next run of the property check. However, compared to the original procedure, $S[0]$, i.e., the victim task's influence at the beginning of the time window t, remains the same in all iterations. Checking the property and adjusting $S[1]$ is repeated until no new counterexamples are detected and the property holds.

Then, the property is unrolled an additional clock cycle by incrementing k and the set S is initialized with the set of the previous clock cycle. In this way, the property is unrolled clock cycle by clock cycle, until the victim task cannot influence any new state variables, i.e., the property holds for a new unrolling k without removing state variables from $S[k]$, or a vulnerability is detected.

Algorithm 2 Unrolled UPEC-SSC Procedure

```
1:  procedure UPEC-SSC-UNROLLED
2:      S[0], S[1] ← S_¬victim, S_¬victim
3:      k ← 1
4:      while true do
5:          S_cex ← check(UPEC-SSC-unrolled(k, S))
6:          if S_cex = ∅ then
7:              if S[k] = S[k − 1] then
8:                  return hold
9:              else
10:                 k ← k + 1
11:                 S[k] ← S[k − 1]
12:         else if S_cex ∩ S_pers ≠ ∅ then
13:             return (vulnerable, S_cex)
14:         else if S_cex ∩ S_pers = ∅ then
15:             S[k] ← S[k] \ S_cex
```

Note that, even if the procedure returns *hold*, an additional inductive proof is required to prove the system secure. This is because, even if the victim task's influence could not propagate to new state variables from clock cycle $k-1$ to k, there might be a future clock cycle $k+n$ where the propagation continues. Performing the inductive proof can be done by using the original algorithm (Alg. 1) with $S \leftarrow S[k]$ from the last unrolled clock cycle k.

4 CASE STUDY

We conducted a case study on Pulpissimo [12] which is an MCU-style SoC featuring a 2-stage pipelined RISC-V core, a DMA, an accelerator, and many peripheral IPs, including I/O interfaces. The entire SoC comprises more that 5M state variables. Applying UPEC-SCC to Pulpissimo yielded several counterexamples which point to potential vulnerabilities in the MCU. We started our analysis and debugging process by picking a counterexample that is of particular interest because it points to a so far unknown variant of the BUSted attack. In the following, we provide insights into this new attack variant and the detection of the corresponding vulnerability in Sec. 4.1. In Sec. 4.2, we elaborate on how to mitigate such vulnerabilities and how the UPEC-SSC method was used to prove the modified system secure w.r.t. the security objectives.

We conducted all experiments on a workstation PC featuring an Intel i9-13900k with 128 GB RAM running Linux and the commercial property checker OneSpin 360 DV by Siemens EDA.

4.1 New BUSted variant: accelerator + memory

UPEC-SCC generated a counterexample revealing a security vulnerability involving the Hardware Performance Engine (HWPE) accelerator IP and a memory device in the system. The HWPE can be configured to fetch its inputs directly from the memory, perform complex arithmetic operations on the data, and write the results back to a configured memory region.

The detected counterexample shows a scenario where the attacker sets up the HWPE to spy, i.e., to collect information about the victim task's memory accesses. Following the attack structure introduced in Sec. 2.2, the scenario can be divided into three phases. In the preparation phase, the attacker task chooses a memory region with write permissions and primes it with zeros. It then sets up the HWPE accelerator IP to progressively overwrite the memory region with non-zero values. The recording phase starts after a context switch to the victim task. The victim accesses a different memory location in the same memory device, concurrently with the HWPE's memory transactions. This behavior creates contention on the interconnect and the continuous accesses by the HWPE are delayed. In the retrieval phase of the attack, control in the core is once again transferred to the attacker task. The primed memory region contains information about the victim's memory accesses: By examining the progress made by the HWPE in overwriting the primed region, the attacker can deduce the number of memory accesses the victim has made to a particular memory device and launch an attack similarly as in the original BUSted attack [11].

We detected this scenario with the unrolled proof procedure of Alg. 2. The preparation phase of the attack is implicitly contained in the symbolic initial state of the proof. The proof window starts in the recording phase, when the victim task accesses the memory for the first time. We unrolled for 2 clock cycles to observe the delay of the HWPE memory access. The retrieval phase is implicitly modeled by classifying the state variables of the attacker-accessible memory region as persistent state variables and part of S_{pers}. The runtime of the proof iterations was below one minute.

The detected vulnerability has an important characteristic: It allows an attacker to open a timing side channel without the use of an actual timer. This undermines a cheap and popular countermeasure against timing attacks, where access to system timers is denied to untrusted tasks or artificial noise is added to timers to obstruct precise time measurement [6].

4.2 Countermeasure

For the purpose of evaluating UPEC-SCC in the case of a secure design, we implemented a conservative countermeasure that fixes the vulnerability described above. The fix could be proven to also eliminate all other counterexamples. We designated a memory region for the security-critical operation of the victim task and restricted the access to the corresponding memory device. We make use of Pulpissimo's architecture that features two types of memory, a public and a private memory, which are implemented as separate memory devices. The two memory devices are connected to the SoC by two separate crossbars. The private memory is only accessible to a few IPs (including the processor core). We mapped the security-critical memory region into the address space of the private memory device. Consequently, we only needed to restrict the access for very few IPs. The restrictions consist of a set of legal configurations for the corresponding IPs and can be compiled as a set of firmware constraints to be checked for compliance during firmware development and verification. This is similar to the process introduced in [8].

With this countermeasure in place, we ran the proof procedure of Alg. 1. After 3 iterations, the procedure proved the system to

be secure w.r.t. the considered threat model. The runtime of the iterations ranged between 58 seconds and 2 hours 52 minutes.

5 RELATED WORK

Only little research has been conducted on timing side channel attacks in MCU-style SoCs with single processors. BUSted [11] is the first comprehensive attack that uses MCU peripheral IPs to spy on a running victim task. The attack leverages contention on the MCU bus to breach Armv8-M TrustZone and retrieve confidential data. A similar timing side channel involving a DMA creating contention with a victim execution has been presented in [1]. This and the BUSted attacks are good examples for the type of vulnerabilities our presented method detects. In a Nemesis attack [15], instruction-dependent timing differences in the CPU interrupt logic are exploited to breach the isolation of the Sancus Trust Execution Environment (TEE). Even though Nemesis is demonstrated on an MCU, the exploited side channel (instruction execution time) is located inside the CPU and is therefore not within the class of vulnerabilities considered in this work.

There are several related types of timing attacks targeting systems that allow attacker concurrency or include caches. Contention-based attacks (e.g., [16, 18]) exploit simultaneous accesses to shared caches or memories in multi-core or multi-threaded CPUs. Timing attacks in NoCs (e.g., [10, 13]) deduce confidential information by accessing shared routers, while attacker and victim tasks are running on different network nodes. In cache-based timing side channels (e.g., [9, 17]) footprints left by a victim in caches produce differences in hit and miss times and allow an attacker to deduce confidential data. All these attacks are well understood and a plethora of effective countermeasures has been proposed. We therefore concentrate on the relatively new class of MCU-wide timing side channels, which pose a severe problem even without exploiting attacker concurrency or buffers like caches.

There are no dedicated formal methods for detecting vulnerabilities in MCUs. Formal verification methods targeting related security problems in similar systems are usually not well suited for exhaustive detection of timing side channels in MCU-style SoCs. Property checking approaches like [4] consider individual security properties targeting a limited set of security problems in SoCs. They do not focus on providing SoC-wide guarantees. Another popular verification approach applied to security problems in SoCs is Information Flow Tracking (IFT). As summarized in [5], IFT computes the information flow between a designated pair of source and sink in a design. We believe that, in principle, detecting timing side channels in SoCs can be formulated as an IFT problem. However, exhaustively covering *all* possible vulnerabilities seems unattainable. In addition, it is unlikely that contemporary solvers can handle the complexity of global IFT analysis in SoCs of realistic size. There are two works that apply exhaustive formal verification for detecting security vulnerabilities in SoCs. In [8] a 2-safety property is used to detect all functional design bugs causing confidentiality violations in SoCs. The work in [7] exhaustively detects integrity caused by untrusted third-party IPs in SoCs. Both approaches can guarantee the absence of vulnerabilities w.r.t. their respective security objectives. But, unlike this work, they do not target and do not scale to SoC-wide timing side channels.

6 CONCLUSION

In this paper, we present a novel formal verification methodology for detecting timing side channels in MCU-style SoCs. The method detected a so far unknown variant of a timing side channel attack which allows the attacker to record information about a victim task's memory accesses without requiring cache footprints or access to hardware timers. We demonstrate that the method is scalable for an SoC of realistic size and can prove the SoC to be secure w.r.t. the security objective after we implemented a conservative countermeasure. Our future work will explore a UPEC-SCC driven design methodology leading to new and less conservative countermeasures.

REFERENCES

[1] Marton Bognar, Jo Van Bulck, and Frank Piessens. 2022. Mind the Gap: Studying the Insecurity of Provably Secure Embedded Trusted Execution Architectures. In *2022 IEEE Symposium on Security and Privacy (SP)*.

[2] Lucas Deutschmann, Johannes Müller, Mohammad R. Fadiheh, Dominik Stoffel, and Wolfgang Kunz. 2022. Towards a Formally Verified Hardware Root-of-Trust for Data-Oblivious Computing. In *59th ACM/IEEE Design Automation Conference (DAC'22)*.

[3] Mohammad R. Fadiheh, Alex Wezel, Johannes Müller, Jörg Bormann, Sayak Ray, Jason M. Fung, Subhasish Mitra, Dominik Stoffel, and Wolfgang Kunz. 2023. An Exhaustive Approach to Detecting Transient Execution Side Channels in RTL Designs of Processors. *IEEE Trans. Comput.* 72, 1 (2023), 222–235.

[4] Nusrat Farzana, Fahim Rahman, Mark Tehranipoor, and Farimah Farahmandi. 2019. SoC Security Verification using Property Checking. In *2019 IEEE International Test Conference (ITC)*.

[5] Wei Hu, Armaiti Ardeshiricham, and Ryan Kastner. 2021. Hardware Information Flow Tracking. *ACM Comput. Surv.* 54, 4, Article 83 (may 2021), 39 pages.

[6] W.-M. Hu. 1991. Reducing timing channels with fuzzy time. In *Proceedings. 1991 IEEE Computer Society Symposium on Research in Security and Privacy*. 8–20.

[7] Dino Mehmedagic, Mohammad Rahmani Fadiheh, Johannes Müller, Anna Lena Duque Antón, Dominik Stoffel, and Wolfgang Kunz. 2023. Design of Access Control Mechanisms in Systems-on-Chip with Formal Integrity Guarantees. In *2023 USENIX Security Conference*.

[8] Johannes Müller, Mohammad R. Fadiheh, Anna Lena Duque Antón, Thomas Eisenbarth, Dominik Stoffel, and Wolfgang Kunz. 2021. A Formal Approach to Confidentiality Verification in SoCs at the Register Transfer Level. In *58th ACM/IEEE Design Automation Conference (DAC'21)*.

[9] Colin Percival. 2005. Cache missing for fun and profit. In *BSDCan*. http://www.daemonology.net/papers/htt.pdf

[10] Cezar Reinbrecht, Altamiro Susin, Lilian Bossuet, Georg Sigl, and Johanna Sepúlveda. 2016. Side channel attack on NoC-based MPSoCs are practical: NoC Prime+Probe attack. In *2016 29th Symposium on Integrated Circuits and Systems Design (SBCCI)*.

[11] Cristiano Rodrigues, Daniel Oliveira, and Sandro Pinto. 2024. BUSted!!! Microarchitectural Side-Channel Attacks on the MCU Bus Interconnect. In *2024 IEEE Symposium on Security and Privacy (SP)*.

[12] Pasquale Davide Schiavone, Davide Rossi, Antonio Pullini, Alfio Di Mauro, Francesco Conti, and Luca Benini. 2018. Quentin: an Ultra-Low-Power PULPissimo SoC in 22nm FDX. In *2018 IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S)*.

[13] Martha Johanna Sepúlveda, Jean-Philippe Diguet, Marius Strum, and Guy Gogniat. 2015. NoC-Based Protection for SoC Time-Driven Attacks. *IEEE Embedded Systems Letters* 7 (2015).

[14] Joakim Urdahl, Dominik Stoffel, and Wolfgang Kunz. 2014. Path Predicate Abstraction for Sound System-Level Models of RT-Level Circuit Designs. *IEEE Trans. on Comp.-Aided Design of Integrated Circuits & Systems* 33, 2 (Feb. 2014), 291–304.

[15] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2018. Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic. In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*.

[16] Yao Wang, Andrew Ferraiuolo, and G Edward Suh. 2014. Timing channel protection for a shared memory controller. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*.

[17] Yuval Yarom and Katrina Falkner. 2014. FLUSH+ RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack.. In *USENIX Security Symposium*.

[18] Zirui Neil Zhao, Adam Morrison, Christopher W Fletcher, and Josep Torrellas. 2022. Binoculars: Contention-Based Side-Channel Attacks Exploiting the Page Walker. In *31st USENIX Security Symposium*.