
# Design of Access Control Mechanisms in Systems-on-Chip with Formal Integrity Guarantees

Dino Mehmedagić, Mohammad Rahmani Fadiheh, Johannes Müller, Anna Lena Duque Antón, Dominik Stoffel, and Wolfgang Kunz, *Rheinland-Pfälzische Technische Universität (RPTU) Kaiserslautern-Landau, Germany*

https://www.usenix.org/conference/usenixsecurity23/presentation/mehmedagic

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Open access to the Proceedings of the 32nd USENIX Security Symposium is sponsored by USENIX.

# Design of Access Control Mechanisms in Systems-on-Chip with Formal Integrity Guarantees

Dino Mehmedagić Mohammad Rahmani Fadiheh Johannes Müller
Anna Lena Duque Antón Dominik Stoffel Wolfgang Kunz

*Department of Electrical and Computer Engineering,*
*Rheinland-Pfälzische Technische Universität (RPTU) Kaiserslautern-Landau, Germany*

## Abstract

Many SoCs employ system-level hardware access control mechanisms to ensure that security-critical operations cannot be tampered with by less trusted components of the circuit. While there are many design and verification techniques for developing an access control system, continuous discoveries of new vulnerabilities in such systems suggest a need for an exhaustive verification methodology to find and eliminate such weaknesses. This paper proposes UPEC-OI, a formal verification methodology that exhaustively covers integrity vulnerabilities of an SoC-level access control system. The approach is based on iteratively checking a 2-safety interval property whose formulation does not require any explicit specification of possible attack scenarios. The counterexamples returned by UPEC-OI can provide designers of access control hardware with valuable information on possible attack channels, allowing them to perform pinpoint fixes. We present a verification-driven development methodology which formally guarantees the developed SoC's access control mechanism to be secure with respect to integrity. We evaluate the proposed approach in a case study on OpenTitan's Earl Grey SoC where we add an SoC-level access control mechanism alongside malicious IPs to model the threat. UPEC-OI was found vital to guarantee the integrity of the mechanism and was proven to be tractable for SoCs of realistic size.

## 1 Introduction

Although the research field of hardware security is not new, it has received new impetus in the recent past, especially through the discovery of the Spectre [21] and Meltdown [24] attacks, which highlighted how different hardware design optimizations can easily be abused to usurp security of a computing system as a whole. Numerous hardware bug discoveries followed, reporting a large variety of security vulnerabilities, as documented by their ever growing number in the MITRE common weakness enumeration (CWE) database [1].

The problem gets worse if not only cores but entire systems-on-chip (SoCs) are considered [12, 17]. SoCs are complex integrated circuits that, besides central processing units (CPUs), contain various specialized hardware units, such as accelerators, improving the overall performance of the system. Due to increasing diversification in SoC functionality requirements, designers increasingly resort to integrating third-party intellectual property (IP) hardware units into their SoCs. This poses a security threat, since third-party vendors could insert malicious hardware or firmware into their IPs, which can, upon integration, be used to compromise functionality of the entire SoC. Such IPs could, for example, send illegal messages to other parts of the SoC to delay, disrupt or even alter results of security-critical operations. Security verification of SoCs with third-party IPs becomes even more difficult if these IPs are sold as a black box, which is often the case. This means that the IP implementation details remain hidden from the SoC integrators.

As an example, suppose an SoC integrator is designing a RISC-V SoC. The integrator is provided with a direct memory access (DMA) engine by a third-party vendor with malicious intentions. Suppose that, unbeknownst to the SoC integrator, the DMA engine contains a hardware Trojan that uses the DMA's controller privileges to change data of a protected region in the SoC's memory. Even though the system's CPU has Physical Memory Protection (PMP) implemented which restricts access to protected memory and memory-mapped IPs for non-privileged and untrusted programs [39], the DMA can still modify any part of the memory, as it is a hardware component completely separate from the CPU. This way, the DMA engine can sabotage various functions of the SoC, such as its ability to perform correct encryption, interrupt servicing, controlling privilege levels of software processes, and many more. Attacks of this kind can, therefore, cause serious violations of the system's operation integrity.

Such security concerns motivate SoC designers to include a chip-level access control system that will detect and prevent malicious or illegal messages between any IPs. For example, the RISC-V community currently works on complementing PMP with IOPMP, a set of hardware SoC extensions that shall monitor all IP communication and block it if necessary [22].

Meanwhile, most semiconductor companies already have a system-wide proprietary protection for their chips. ARM realizes this with its System Memory Management Units (SMMUs) [8] and its TrustZone® technology [40], while Intel provides a similar solution through its VT-d IOMMUs [20]. Even though these mechanisms greatly reduce the chip's attack surface, it has been demonstrated that they can be circumvented by addition of a malicious IP block to the system [6,17]. This calls for an exhaustive verification methodology that is able to detect all security vulnerabilities of the access control mechanisms before tapeout.

In this paper, we propose an exhaustive verification methodology which can guarantee *integrity of security-critical operations* with respect to an SoC's access control mechanism configuration. The approach builds upon UPEC [14, 15], a formal method to verify confidentiality in processor cores (cf. Sec. 2.3).

While many security approaches rely on the verification engineer to explicitly formulate attack patterns in order to catch specific vulnerabilities in a design, our new methodology, called *UPEC for Operation Integrity (UPEC-OI)*, requires no such a priori security knowledge. All hardware integrity bugs of an SoC's access control configuration are covered by UPEC-OI. In fact, counterexamples provided by UPEC-OI guide the SoC designers to the exact parts of the microarchitecture that permit integrity attacks, giving them valuable insight on how to conduct precise, local fixes of the security bugs. UPEC-OI can therefore be used as a tool for designing SoCs with integrity-proof access control hardware, which is another key aspect that is explored in this work. Contributions of this paper are as follows:

- We define the requirements for the SoC-level access control mechanism that must be fulfilled to guarantee the integrity of any security-critical operation running on the SoC. We first formalize these requirements using the notion of a 2-safety hyperproperty [10] and then present how to make this practical by formulating interval properties [27, 36] for unbounded property checking (Sec. 3.2).
- We present how the general characteristics of SoC access control architectures can be leveraged to decompose the global verification problem in the spatial and temporal domain. This leads to a verification methodology involving several optimizations which are key for the computational tractability of our formal approach (Sec. 3.3).
- We propose a scalable and exhaustive verification algorithm for the system's operation integrity. The algorithm decomposes the global computational problem in terms of possible scenarios for the propagation of influences from a malicious IP. This decomposition is not only based on circuit structure (paths, components) but takes into account the semantics of the logic circuit when analyzing these propagation scenarios. The global proof is obtained by iteratively performing inductions over the proof instances generated by this decomposition. Furthermore, the algorithm integrates the optimizations that are made possible by our general methodology (Sec. 3.4).
- A design flow is presented which proposes using UPEC-OI in a verification-driven development approach. An SoC design resulting from this flow is formally guaranteed to be free of integrity vulnerabilities in its access control system (Sec. 4).
- We demonstrate the effectiveness and feasibility of the approach on an OpenTitan SoC by upgrading its interconnect to provide access control against untrusted IPs. We present vulnerabilities detected by the approach, as well as its computer resource usage. We prove for the final system design that it is free of any hardware vulnerabilities that could be exploited by a third-party IP to violate the integrity of any security-critical operation (Sec. 5).

  To the best of our knowledge, this is the first time that this can be demonstrated for an RTL implementation of an SoC of realistic size.

## 2 Background

This section briefly explains some background concepts that our work builds upon. Section 2.1 covers basic architecture of modern SoCs, with emphasis on the system-wide communication between its IP units. Sec. 2.2 explains the principles behind interval property checking (IPC), a version of bounded model checking (BMC) that allows unbounded formal proofs of properties. Finally, Sec. 2.3 covers the UPEC verification methodology, upon which our approach is based.

### 2.1 SoC Architecture and Communication

A modern SoC comprises a number of components (IPs) that perform specific tasks and interact with each other based on a defined communication protocol, such as AMBA®AXI [7] or TileLink [34]. This communication is often asymmetrical, with some IPs assigned the role of controllers and others the role of peripherals. CPU cores, DMA engines and debug modules are typical examples of controller IPs. Controller IPs can initiate transactions and send requests addressed to specific peripheral IPs, such as memory modules, input/output (I/O) devices, hardware accelerators, etc. The peripherals are expected to fulfil the issued requests and respond accordingly.

Hardware that facilitates this communication and enforces these protocols within the SoC is usually referred to as the interconnect. This communication infrastructure can be implemented as a centralized bus system, a crossbar or a network-on-chip (NoC) [23]. Fig. 1 shows a basic SoC structure with a crossbar-type interconnect. Here, every controller (C) is connected to some or all peripherals (P) via dedicated channels. In this example, the communication protocol is enforced by a

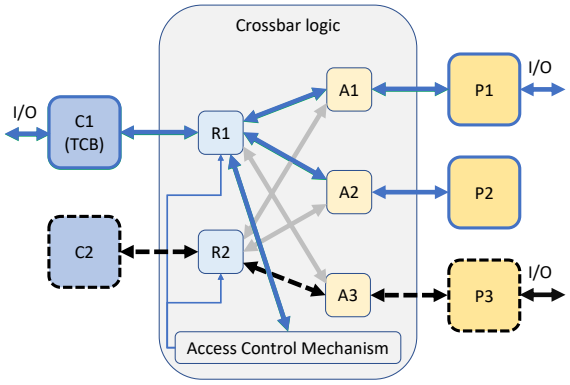



Figure 1: Basic SoC design with crossbar-type interconnect.

series of nodes. Every controller has a router node (R) responsible for correct routing of each request to the appropriate peripheral. In a similar manner, an arbiter node (A) exists for every peripheral. These nodes are responsible for correct arbitration if multiple requests are sent to the same peripheral simultaneously.

For the rest of the paper, our approach is described relating to SoCs with crossbar-type architectures. It should be noted, however, that the proposed approach can be applied to other on-chip communication structures in an analogous way.

## 2.2 Interval Property Checking

UPEC-OI uses a SAT-based formal verification technique called *Interval Property Checking (IPC)* [27, 36]. To verify that the design under verification (DUV) satisfies a property of temporal length $k$, an IPC solver "unrolls" the DUV into $k$ combinational copies of itself, such that each copy $j$ ($0 \leq j < k$) represents the circuit's state at the $j$-th clock cycle. It then assigns symbolic values to the DUV's input variables and its starting state. In principle, this means a SAT solver will search for any combination of input and state values in order to find a counterexample to the property. If the property holds for all possible input and starting state combinations, an unbounded formal proof of the property has been obtained. A failing property is documented by the solver with a *counterexample* that shows a starting state and inputs to the DUV leading to a state that violates the property. In IPC, the starting state may be unreachable from *reset*, in which case the counterexample is spurious and called a *false counterexample*. In this case, the property needs to be strengthened with *invariants* so that unreachable states are excluded in the IPC starting state.

## 2.3 Unique Program Execution Checking

The original *Unique Program Execution Checking (UPEC)* methodology [14, 15] is a formal verification methodology that exhaustively searches for transient execution side channels in RTL hardware designs. It is based on a 2-safety hyper-property [10] which verifies whether protected confidential information can affect program execution in terms of cycle-accurate timing or valuation of the architectural (program-visible) registers. Any violation of this statement indicates the existence of a side channel that could expose the secret.

assume:
  at $t$: $micro\_soc\_state_1 = micro\_soc\_state_2$;
  at $t$: $public\_mem_1 = public\_mem_2$;
prove:
  at $t+k$: $soc\_state_1 = soc\_state_2$;

Figure 2: UPEC property for confidentiality.

The UPEC property is specified as an interval property that is checked over a tailor-made *2-safety model*, which consists of two identical instances of the DUV. In the hardware domain, such 2-safety models are also referred to as *miters*. This allows for a formulation of UPEC using standard property specification languages (PSL), such as SystemVerilog Assertions (SVA). Fig. 2 shows the UPEC property in a pseudo-code PSL. The general structure of the property is that of an implication between an *assumption* and a *commitment*. In the assumption, initially, every microarchitectural state bit is assumed to be equal in value to its counterpart in the other instance. The only exceptions are the state variables that contain some secret information. Since they are the only state variables that differ at time point $t$, these signals are also the only possible cause of any difference appearing elsewhere in the system after $t$. This propagation of the difference is checked by the commitment term, $soc\_state_1 = soc\_state_2$, which states that all previously equal microarchitectural state variables remain equal. If the property fails, the difference has spread to other state variables of the core. If any of those affected state bits is program-visible, i.e., *architectural*, a side channel is detected. If the affected state bits are not program-visible, no side channel has yet been formed. In this case, however, the property's time $k$ needs to be extended in order to check whether the affected microarchitectural state can eventually cause a change in the architectural state of the core. This iterative approach eventually finds all transient execution side channels in the design. UPEC-OI adapts this approach for finding access control integrity vulnerabilities at the level of entire SoCs.

# 3 UPEC for Operation Integrity

This section presents the theoretical underpinnings of UPEC-OI. We begin by specifying the threat model for an SoC in Sec. 3.1. The formulation of the UPEC-OI property, first theoretically as a hyperproperty and then as an interval property, is given in Sec. 3.2. Optimizations necessary to make the approach feasible are discussed in Sec. 3.3. Finally, the full iterative induction procedure is presented in detail in Sec. 3.4.

## 3.1 Threat Model

Our threat model considers an SoC equipped with an access control mechanism capable of forming trust boundaries between different IPs. One controller IP is considered a *trusted computing base* (TCB). This is a trustworthy IP that has sole control over the access control policy. Prior to a security-critical operation, the trusted IPs required for its execution are to be isolated such that the secure operation cannot be tampered with by the rest of the SoC. The TCB elevates the required IPs to a higher security level by modifying the access control mechanism's configuration registers. This is done according to the security specification of the system, which defines the complete set of different security-critical modes that the system could be in and designates the trust boundaries between the IPs for each of those modes. An example of such an isolation mechanism is illustrated in Fig. 1. The high- and low-level security domains (i.e., trusted vs. untrusted IPs) are marked with continuous and dashed outlines, respectively. Inter-domain communication, marked with light gray channels, needs to be limited such that the security-critical operation cannot be tampered with. The access policy is enforced by the crossbar logic and trusted IPs.

Our threat model assumes that some security-critical operation is taking place. All IPs that are not in the higher security domain are assumed to have malicious intent to interfere with the secure operation. These IPs can work individually or jointly, sending multiple messages through the crossbar if necessary in an attempt to bypass the existing access control system. All high-security-level IPs and the outside environment are considered trustworthy and will not intentionally assist the low-security-level IPs in their malicious attempt.

## 3.2 UPEC-OI Property Formulation

We label the sets of our SoC's primary input and output variables as $X$ and $Y$, respectively, and its microarchitectural state variable set as $Z$. The state variables that configure the access control policy of the SoC are labeled as $L \subseteq Z$. The set of permissible access control configurations, i.e., the set of allowed valuations of $L$, is labeled as $\Lambda = \{\lambda_1, \lambda_2, ...\}$. This set is provided by the security specification of the system. We also define $m$ as the number of IP devices in the SoC. We group all of the SoC's hardware components in a set $D = \{D_0, D_1, ..., D_m\}$, such that $D_0$ is the SoC's crossbar logic, which includes $L$, while $D_1$ to $D_m$ are the SoC's IPs. As an example, our SoC in Fig. 1 would have $m = 5$, with $D_0$ encompassing all hardware in the "crossbar logic" box, while IPs C1, C2, P1, P2 and P3 would be assigned labels $D_1$ to $D_5$. Configuration registers of the "access control mechanism" box are labeled as $L$.

Each $D_i$ ($0 \leq i \leq m$) has its own sets of input, output and state variables, $X_i$, $Y_i$, and $Z_i \subseteq Z$. As illustrated in Fig. 1, an individual IP's output signals are either connected to the crossbar or belong to the SoC's primary outputs:

$$Y_i \subseteq X_0 \cup Y$$

We consider the crossbar itself not to have any primary I/O. If this is not the case in practice, then such primary I/O can be modeled as a separate IP. We can therefore say that

$$X_0 \cap X = Y_0 \cap Y = \emptyset$$

We also observe that each IP $D_i$ has a security level, $l_i$, as determined by the access control configuration $\lambda \in \Lambda$, where $\lambda$ is a valuation of the registers $L$. We consider a two-level security mechanism, such that high-security-level devices have their $l_i$ set to 1, while low-security-level devices have their $l_i$ set to 0. $D_0$ itself might not have a security level defined in $\lambda$, however UPEC-OI always considers $l_0 = 1$. We separate the SoC's state variables in $Z$ and output signals in $Y$ into high-security-level ($Z_h$, $Y_h$) and low-security-level ($Z_l$, $Y_l$) signals. We define the high-level state variables as all those that belong to the trusted IPs or the system's crossbar:

$$Z_h = \bigcup_{i=0}^{m} \begin{cases} Z_i, & l_i = 1; \\ \emptyset, & l_i = 0 \end{cases} \tag{1}$$

Similarly, we define the high-level output variables as all primary outputs belonging to trusted IPs:

$$Y_h = \bigcup_{i=0}^{m} \begin{cases} Y_i \cap Y, & l_i = 1; \\ \emptyset, & l_i = 0 \end{cases} \tag{2}$$

The state and primary output signals from untrusted IPs are low-security-level variables:

$$Z_l = Z \setminus Z_h \tag{3}$$

$$Y_l = Y \setminus Y_h \tag{4}$$

Note that the critical hardware infrastructure that regulates the access control configuration, i.e., the TCB, as well as all state variables $L$, is always a part of $Z_h$.

For the remainder of this paper, we model the SoC as a deterministic Mealy-type finite state machine. This allows us to formalize the notion of a security-critical operation, as well as the requirements for preserving its integrity:

**Definition 1** (*Security-Critical Operation*)**.** A security-critical operation of an SoC is a sequence of valuations of $Z_h$, during which the valuation of $L$ is an element of $\Lambda$ and is not changed by the TCB. □

**Definition 2** (*Operation Integrity*)**.** Integrity of a security-critical operation requires that the output behavior of the high-security-level domain, i.e., the values of SoC's high-security-level output variables $Y_h$, is determined only by the values of the system's input variables $X$ and its high-security-level state, as given by $Z_h$. □

We can express Def. 2 as a non-interference requirement for our SoC [16]. Since the SoC is a deterministic state machine, its outputs through $Y$ are always based on inputs provided through $X$ and states given by $Z$. We can then argue that the SoC's high-security-level domain outputs, $Y_h$, can only be non-deterministic w.r.t. $X$ and $Z_h$ (Def. 2) if $Y_h$ depends on $Z \setminus Z_h = Z_l$. By restricting information flow from $Z_l$ to $Y_h$, we preserve operation integrity. This is similar to the idea in [31], where a set of *observational determinism* properties is defined to ensure confidentiality of data in a software program. Observational determinism checks for unwanted information flows from high-security to low-security level domains. We adapt this concept to encompass SoC hardware and formulate restrictions to the information flow in the opposite direction. This allows us to reason about the non-interference with the SoC's secure state instead of the secure state's non-observability.

We express this information flow restriction as a 2-safety hyperproperty [10]. We define the term *trace* to express a sequence $\tau = e_0 \cdot e_1 \cdot ... e_n$, with $e_t$ being a tuple $\langle i_t, s_t, o_t \rangle$, where $i_t$ is the valuation of our SoC's input variables $X$ at time point $t$, $s_t$ is its state, as represented by the value of $Z$ at $t$ and $o_t$ is the valuation of its output variables $Y$ at $t$. We can then define the following sequences:

- $state_h(\tau)$ is the sequence of states of the SoC's high-security-level domain in a trace $\tau$. The notation $state_h(\tau)[t]$ represents the valuation to the state variables in $Z_h$ at time point $t$.
- $in(\tau)$ is the sequence of valuations to the SoC's input variables $X$ in a trace $\tau$. Likewise, $in(\tau)[t]$ is the valuation to $X$ at time point $t$.
- $out_h(\tau)$ is the sequence of valuations to the SoC's high-security-level output variables $Y_h$ for a trace $\tau$. The term $out_h(\tau)[t]$ represents the value of $Y_h$ at a time point $t$.

We define a set $T$ that includes all traces $\tau$ of arbitrary length, for which $state_h(\tau)$ is a security-critical operation. For all the above defined sequences $s(\tau \in T)$, we define $s(\tau)[t..]$ to represent a suffix of $s$ that starts at and includes the time point $t$.

With these notations, we formalize Def. 2 as follows: For any two traces running security-critical operations on the SoC whose secure-domain state at time $t$ is identical, and whose input sequence at and after $t$ is the same, operation integrity guarantees that the secure domain's outputs after $t$ are also identical:

$$\begin{aligned} \textbf{\textit{OI}} \triangleq \{ (\forall \tau_1, \tau_2, t \qquad & \text{where } \tau_1, \tau_2 \in T, t \in \mathbb{N} : \\ state_h(\tau_1)[t] = state_h(\tau_2)[t] \ \wedge \ & in(\tau_1)[t..] = in(\tau_2)[t..] \\ \implies \quad out_h(\tau_1)[t..] = & \, out_h(\tau_2)[t..]) \} \end{aligned}$$

The formulated hyperproperty fails if any difference occurs in $Y_h$ at or after $t$. It can be noted that the only possible difference between the two traces at $t$ is the low-security-level domain's state, as given by $Z_l$. Therefore, any difference in the visible behavior of the SoC's secure domain after $t$ can only be caused by propagation from $Z_l$. We call $Z_l$ the *propagation source* in our hyperproperty.

It should be noted that *OI* defines the *hardware* property for operation integrity. Integrity of the overall hardware/firmware system also relies on properly developed firmware that activates and configures the underlying access control mechanism according to the security specification. For example, it is the responsibility of the TCB's firmware to make sure that all IPs elevated to the high-security-level domain are trusted, properly configured and available. Formulating such firmware-level verification targets is out of scope of this paper. However, precisely defining and verifying the hardware requirements from Def. 2 enables the separation of concerns in the security verification process and provides trust and the needed hardware guarantees for the firmware developers.

```
assume:
    at t:                 high_micro_state_1 = high_micro_state_2;
    during t..t+k:        primary_inputs_1 = primary_inputs_2;
    during t..t+k:        access_control_configured_1();
prove:
    at t+k:               secure_outputs_1 = secure_outputs_2;
    at t+k:               high_soc_state_1 = high_soc_state_2;
```

Figure 3: UPEC-OI property.

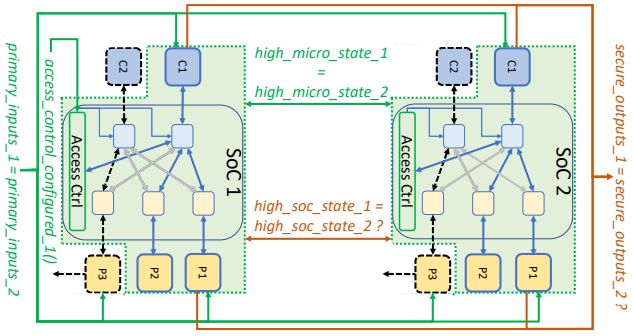



Figure 4: SoC miter for UPEC-OI

We now transform our *OI* hyperproperty into an interval property that is checked on a miter model. Fig. 3 shows the UPEC-OI property expressed with a pseudo-code PSL. An example setup of the miter model is represented graphically in Fig. 4. Assuming the same example for an access control policy as in Fig. 1, internal state bits of the low-security-level domain are labeled as the propagation source. The assumption $high\_micro\_state_1 = high\_micro\_state_2$ assumes all non-propagation-source state variables (area encircled by dotted lines) to be equal in value to their counterpart in the other instance at starting time $t$. The term $primary\_inputs_1 = primary\_inputs_2$, similarly, assumes the primary inputs $X$ of

the SoC to be equal in value for the two instances. This assumption has to hold for the whole duration $k$ of our property.

This setup models the two traces with identical input sequences and starting state, analogous to the *OI* hyperproperty. It also ensures $Z_l$ to be the only possible source of difference propagation.

The assumption macro $access\_control\_configured_1()$ restricts the property checker to only cover scenarios where a security-critical operation (Def. 1) is taking place on the SoC, i.e., it excludes from consideration traces which are not in $T$. It does this by constraining the valuation of $L$ to be one of the elements in $\Lambda$, as per the requirement of Def. 1. (In case there is more than one element in $\Lambda$, a separate property for each element can be formulated and verified. We assume that, in this process, each element of $\Lambda$ is correctly captured by $access\_control\_configured_1()$. In practice, the number of permissible configurations is typically small, since the trust boundary in an SoC is usually fixed based on the system architecture.)

The assumption $access\_control\_configured_1()$ is projected onto only SoC instance 1 of the miter model for the whole property duration $k$, essentially making $L$ constant in that SoC instance. The logical intersection of $access\_control\_configured_1()$ and $high\_micro\_state_1 = high\_micro\_state_2$ has two implications: Firstly, it implies that the valuation of $L$ in SoC instance 2 becomes the same element of $\Lambda$ at time $t$ as in instance 1, as $L$ is a subset of $Z_h$. Secondly, it implies that the TCB does not modify $L$ on its own in either instance: The TCB is also a part of $Z_h$ and therefore has identical behavior in both instances, unless influenced by the untrusted propagation source, $Z_l$. If the untrusted propagation source can tamper with the TCB behavior and can change the valuation of $L$, the resulting integrity violation will be detected since the valuation of $L$ is not restricted after $t$ in the second instance. Formulating this macro requires only architectural level knowledge of the access control mechanism's configuration registers and should thus be simple even for complex access control implementations. The interested reader can find an example formulation of this macro in Appendix A.

The commitment term $secure\_outputs_1 = secure\_outputs_2$ monitors the secure domain's output variables $Y_h$. As per our definition, a counterexample to this statement represents an integrity violation. However, proving a property with this commitment term alone would only provide a guarantee bounded to time $k$. Proving operation integrity *exhaustively* requires $k$ to equal the SoC's sequential depth. In practice this is not feasible, as unrolling an SoC of realistic size for more than a few clock cycles is prohibitively expensive.

In order to solve this problem, UPEC-OI pursues an inductive reasoning approach, inspired by the UPEC methodology for detecting confidentiality violations [15]. The key element of UPEC-OI is monitoring the internal microarchitectural state of the high security domain, as represented by $Z_h$. This is covered by the second commitment term, $high\_soc\_state_1 = high\_soc\_state_2$. Including microarchitectural state variables in the proof is a key factor for the scalability and exhaustiveness of our technique, and forms the foundation of the proposed verification methodology for obtaining unbounded proofs, as described in Sec. 3.4.

We adapt the terminology from [15] to classify the counterexamples obtained from our property depending on which of the two commitments are violated:

**Definition 3** (*T-alert*)**.** If a counterexample to the UPEC-OI property demonstrates a trace in which the cycle-accurate sequence of valuations to $Y_h$ is a function of the propagation source, i.e., the valuations to $Y_h$ become different between the SoC instances in the miter model, then integrity of the operation is violated and the counterexample constitutes a *trespass alert*. □

**Definition 4** (*P-alert*)**.** If a counterexample to the UPEC-OI property demonstrates a trace in which the cycle-accurate sequence of valuations to the secure domain's microarchitectural state $Z_h$ is a function of the propagation source, i.e., the valuations to $Z_h$ become different between the SoC instances in the miter model, then the counterexample is not a violation of the operation integrity and constitutes a *propagation alert*. □

Our overall methodology relies on finding P-alerts to decompose the verification problem. By checking and modifying the UPEC-OI property iteratively for incrementally increasing values of $k$, we can track the difference propagation from $Z_{ps}$ across the SoC's microarchitecture to see whether it can eventually reach a secure output variable and cause a T-alert.

## 3.3 Possible Optimizations

This section presents optimizations to the UPEC-OI property and verification model that help make the methodology scalable as well as more intuitive and comprehensible to the verification engineer.

### 3.3.1 Adding Architectural State Variables as T-alert triggers

Even though observing the output variables $Y_h$ is enough to detect all operation integrity violations, it is worth noting that propagation of the difference to certain bits in $Z_h$ can obviously lead to changes in the secure domain's behavior at a later time. These are state variables that are directly used by the secure software running on the SoC, i.e., the secure domain's *architectural* state variables $Z_a \subseteq Z_h$. By acknowledging that changes to these bits already constitute a T-alert, we do not need to further increase the property time window $k$ to track the inevitable propagation of this difference to $Y_h$. This optimization greatly improves the scalability of

our methodology. Even though these architectural signals are specific to each design, identifying them is simple. For CPUs, obvious candidates are the register file, control and status registers, the program counter, etc. For peripherals, these could be any memory elements that can easily be accessed by secure controllers, e.g., through a *load* request via the interconnect.

### 3.3.2 Spatial Decomposition of the Propagation Source

In order to fully verify the secure operation's integrity, all unsecure variables in $Z_l$ need to be modeled as the propagation source. Although it is possible to do this with all signals at once, it can be beneficial to decompose the threat surface spatially into several propagation sources $Z_{ps} \subseteq Z_l$ to be checked individually. Modeling one low-security-level IP as $Z_{ps}$ at a time is a good rule of thumb. In our example SoC of Fig. 4, this means modeling P3 as a propagation source and constraining C2 as equal between the two instances, in the same way as constraining all of $Z_h$. After the verification is complete, C2 is modeled as the next $Z_{ps}$. This decomposition makes the debugging process easier by avoiding difficult counterexamples where multiple unrelated propagations from different low-security-level IPs occur simultaneously.

It should be noted that this optimization does not compromise the generality of the proof. An attack requiring multiple IPs to work together is still guaranteed to be detected by UPEC-OI thanks to IPC's symbolic-state initialization of all other involved IPs and/or the secure domain's state. A counterexample then demonstrates a scenario where the differing influence from $Z_{ps}$ in the two instances determines whether the joint attack is successful.

### 3.3.3 Sound Blackboxing

The computational complexity of the IPC proof can be further reduced by exploiting two key observations: (1) All communication (and therefore interference) between IPs in an SoC is realized through their interfaces with the crossbar. These IP interface signals, thus, serve as information bottlenecks which can be monitored to infer changes of the IP's internal state. (2) IPs whose states were constrained to be identical in the two instances at $t$ will go through identical states also after $t$ for as long as the influence from $Z_{ps}$ has not yet reached them.

An individual IP can be *blackboxed* by hiding its internal state bits and primary I/O and approximating its interface with the crossbar as new primary I/O. This reduces the size of the model and speeds up computation time. Since two instances of an IP with the same state will produce identical outputs to the rest of the system, we can blackbox all IP devices $D_i$ whose $Z_i \cap Z_{ps} = \emptyset$ and simply constrain their output variables to the crossbar as equal in value in the two instances. In the commitment of our property, instead of monitoring the internal state of the IP, it is sufficient to monitor its input variables from the crossbar. If a counterexample shows a propagation of the difference to the IP's interface, the IP needs to be whiteboxed and the property check repeated.

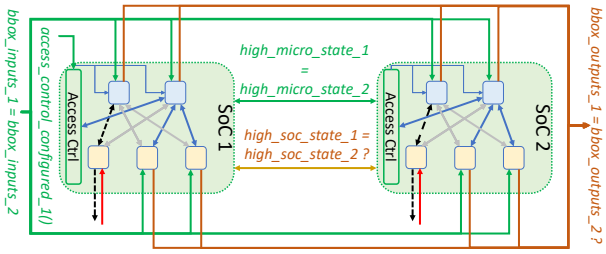



Figure 5: SoC miter with sound blackboxing of all IP devices

If a part or all of an IP's internal state variables are modeled as $Z_{ps}$, the opposite can be done: instead of leaving its state variables unconstrained, it is enough to blackbox the component and leave its output variables to the remaining part of the system unconstrained. However, this comes with a trade-off: complete blackboxing of the propagation source IP allows the IPC solver to assume any output combinations to its interface with the crossbar. Therefore, it can present counterexamples that the IP in question could not create in reality, possibly causing unnecessary overhead in the verification process. However, if the goal is to create an access control mechanism that is resilient to any sorts of inputs from a connected IP, e.g., to protect against third-party IPs with unknown RTL, the IPC's free input representation of the component becomes a useful and exhaustive verification tool.

Fig. 5 shows our miter model example with all IPs blackboxed. P3, which is now modeled as a separate $Z_{ps}$, is approximated simply through its interface with the crossbar logic. All other input variables are constrained to be equal in value, and all output signals to blackboxed components are monitored in order to detect possible difference propagation.

### 3.3.4 Temporal Isolation of the Propagation Source

In addition to spatial segmentation of the threat surface into several propagation sources, we also introduce temporal isolation of the source's differential input. Assuming that the IP containing $Z_{ps}$ is blackboxed and that the influence of the IP is approximated by primary inputs to the rest of the SoC, it is sufficient to allow these primary inputs to be different in the two instances for only the property's initial clock cycle $t$. The generality of our proof is still not lost: if an attack requires a chain of multiple messages to be sent from the propagation source IP, the IPC solver can assume the miter model's state at $t$ as if some of those messages had already been sent. It can also continue delivering the remaining messages in the chain after $t$ through the malicious IP's interface. This optimization prevents the IPC solver from generating a chain of multiple unrelated attacks by the same $Z_{ps}$.

### 3.4 UPEC-OI Verification Methodology

As already mentioned, achieving an unbounded formal proof of operation integrity with the UPEC-OI property by only checking for T-alerts is infeasible in practice. Instead, our methodology decomposes the verification problem by checking not only for T-alerts but also for P-alerts in an iterative, induction-based approach. This splits the verification into two complementary procedures, a base and a step proof. The following section develops the details of our methodology, while also integrating the optimizations of Sec. 3.3.

**Algorithm 1** UPEC-OI Verification Approach

```
1:  procedure UPEC-OI(DUV, sec_config)
2:      miter ← Create_Miter_Model(DUV)
3:      PS ← Spatial_decomposition(DUV, sec_config)
4:      for each Z_ps in PS do
5:          miter ← BBox_IPs(miter)
6:          Create_UPEC-OI_Macros(miter, Z_ps)
7:          CEX ← IPC(UPEC-OI_Comb)
8:          if CEX ≠ ∅ then return CEX
9:          UPEC-OI_BASE_PROPERTY_CHECK // (Alg. 2)
10:         A = ⋃_{i=1}^{k} A_i
11:         UPEC-OI_STEP_PROPERTY_CHECK // (Alg. 3)
12:     return "SECURE"
```

Algorithm 1 shows the top-level view of the UPEC-OI verification approach. If an operation integrity vulnerability exists in a design, the algorithm finds it in form of a T-alert. Otherwise, it formally guarantees operation integrity for the given access control configuration. The algorithm requires two inputs: the DUV itself and its desired access control configuration, *sec_config*. As can be seen in lines 2–3, the approach begins by creating the miter model and decomposing the unsecure domain into several propagation sources, as explained in Sec. 3.3.2. The verification procedure is to be conducted for every propagation source separately. As discussed in Sec. 3.3.3, the miter model's IPs are blackboxed away in line 5. The only components that remain whiteboxed are the crossbar itself and any potential IPs whose state bits are in the immediate fanout of $Z_{ps}$. The necessary UPEC-OI property macros, such as *high_micro_state* or *access_control_configured()* are created according to the location of current $Z_{ps}$ (line 6).

In our algorithm, "IPC (*prop*)" marks running a check of property *prop* with a formal IPC verification tool. For every run, the tool returns either a counterexample where *prop* fails, or $\emptyset$ if it holds. In line 7, we run the solver for the first time with a special version of the UPEC-OI property, which we denote as UPEC-OI_Comb. This property checks for the existence of a combinational logic path between the propagation source and the secure domain's primary outputs. In this special case, the commitments are checked at the same clock cycle as the assumptions ($k = 0$). Since no state variables outside of $Z_{ps}$ can be different at time $t$, only the SoC's secure output variables $Y_h$ are checked. If this property detects a propagation through a combinational path to $Y_h$, an integrity violation is already discovered, the procedure is terminated and the counterexample is returned to the verification engineer (line 8). If this is, however, not the case, the procedure is ready to start the induction base process. (In Algorithm 1, UPEC-OI_BASE_PROPERTY_CHECK and UPEC-OI_STEP_PROPERTY_CHECK should be read as verbatim textual replacements of Algorithms 2 and 3, respectively, not as procedure calls. For example, Algorithm 2 generates the sets $A_i$ that are used in line 10. )

UPEC-OI_Base (*affected_P_var*, *k*, *alert_candidates*)
assume:
at $t$: $high\_micro\_state_1 = high\_micro\_state_2$;
during $t..t+k$: $primary\_inputs_1 = primary\_inputs_2$;
during $t..t+k$: $access\_control\_configured_1()$;
at $t+k-1$: $affected\_P\_var_1 \neq affected\_P\_var_2$;
during $t..t+k$: $bbox\_inputs_1 = bbox\_inputs_2$;
during $t+1..t+k$: $prop\_source\_inputs_1 = prop\_source\_inputs_2$;
prove:
at $t+k$: $alert\_candidates_1 = alert\_candidates_2$;

Figure 6: UPEC-OI property for induction base

Besides finding any possible T-alert, the purpose of the base procedure is to compute the set of *all* high-security-level microarchitectural state variables to which the influence from $Z_{ps}$ can propagate in a given time window $k$. Once all such variables are documented for a given $k$, $k$ is incremented, a new time frame is appended to the miter model and the process is repeated until, eventually, a terminal $k$ is found where no further variables are affected by difference propagation. Fig. 6 shows the UPEC-OI property modified for the base proof. In every base property check, a P-alert discovered in the previous iteration is added as an assumption to the property. This is done by passing the affected variable from the P-alert counterexample as an argument (*affected_P_var*) to the property, and then assuming it as unequal at time $k-1$, recreating the propagation scenario of the previously obtained counterexample. This decomposes the overall search into proof instances of reduced complexity. Blackboxed components of the SoC are constrained to produce inputs to the rest of the system that are equal between the two instances with the term $bbox\_inputs_1 = bbox\_inputs_2$. The term $prop\_source\_inputs_1 = prop\_source\_inputs_2$ ensures temporal isolation of the propagation source's differential input, as explained in Section 3.3.4.

The base property's commitment part checks for difference propagation to any of the variables of interest (*alert_candidates*). Depending on the stage of our iterative procedure, one of three different variable groups are chosen as *alert_candidates*:

1. $TC$ – variables belonging to $Z_a$ or $Y_h$ that are not blackboxed *(T-alert candidates)*,

2. *PC* – all other so far unaffected high-security-level microarchitectural state variables that are not blackboxed *(P-alert candidates)*,
3. *BC* – interface signals with blackboxed components (treated similarly like outputs).

**Algorithm 2** UPEC-OI Base Property Check

```
1:  k ← 1
2:  A_0 ← {NIL}
3:  repeat
4:      A_k ← ∅
5:      for each v in A_{k-1} do
6:          CEX ← IPC(UPEC-OI_Base(v,k,TC))
7:          if CEX ≠ ∅ then return CEX
8:          repeat
9:              CEX ← IPC(UPEC-OI_Base(v,k,PC))
10:             a_k = Ploc(CEX)
11:             PC ← PC \ a_k
12:             A_k ← A_k ∪ a_k
13:         until CEX = ∅
14:         repeat
15:             CEX ← IPC(UPEC-OI_Base(v,k,BC))
16:             a_k = Ploc(CEX)
17:             Whitebox_affected_module(a_k)
18:             CEX ← IPC(UPEC-OI_Base(v,k,Y_h))
19:             if CEX ≠ ∅ then return CEX
20:         until CEX = ∅
21:     k ← k+1
22: until A_{k-1} ≠ ∅
```

Algorithm 2 shows the iterative base procedure. The first step initializes two control variables used in the approach. Variable $k$ indicates the base property time interval which increases as the procedure advances. $A_k$ is the set of all already affected P-alert variables at time $k$. The set $A_0$ is assigned a special symbol, *NIL*, to indicate that for the initial time point, $k = 1$, the *affected_P_var* argument to the UPEC-OI_Base property is ignored, i.e., the property omits the inequality assumption for *affected_P_var*.

The procedure first checks for T-alerts, as those immediately point to an integrity violation and require human attention (lines 6–7). If no T-alerts are found for a given $k$, the procedure continues by looking for P-alerts. If a P-alert is found, the operation $Ploc(CEX)$ extracts affected variables from the counterexample $CEX$ and groups them into a set $a_k$. These variables are now considered as affected, added to the global set $A_k$ and removed from the *PC* list. Once an affected microarchitectural state variable is removed from *PC*, it will no longer be a candidate variable in the future iterations of the property check. This removes the possibility of getting multiple P-alerts pointing to the same affected state variable and thus falling into an infinite loop. It also forces the IPC solver to exhaustively search for other state variables that can

```
UPEC-OI_Step (affected_P_var, P_alert_candidates,
              alert_candidates)
assume:
  at t:          P_alert_candidates_1 = P_alert_candidates_2;
  at t:          T_alert_candidates_1 = T_alert_candidates_2;
  during t..t+1: primary_inputs_1 = primary_inputs_2;
  during t..t+1: access_control_configured_1();
  at t:          affected_P_var_1 ≠ affected_P_var_2;
  during t..t+1: bbox_inputs_1 = bbox_inputs_2;
  at t+1:        prop_source_inputs_1 = prop_source_inputs_2;
prove:
  at t+1:        alert_candidates_1 = alert_candidates_2;
```

Figure 7: UPEC-OI property for induction step.

trigger a P-alert for the given time $k$. The process is repeated until no more P-alerts are found for the given $k$ (lines 8–13). It should be noted that $Ploc(\emptyset) = \emptyset$. Lines 14–20 check the blackboxed component interfaces. If a counterexample shows an input variable to a blackboxed device as affected, this IP is whiteboxed and all UPEC-OI macros are modified accordingly (line 17). If the newly whiteboxed module has any primary outputs, these have to be checked for possible T-alerts immediately after.

The base procedure is repeated, incrementing $k$, until a T-alert is detected or until $k$ reaches a time point where no P-alerts are found. In the latter case, we have obtained a bounded proof of our SoC's operation integrity for the time window $k$. Any T-alert that can occur at a time point later than $k$ can only be caused by a difference between the two instances in a microarchitectural state variable (i.e., P-alert) at a time point $k$ or earlier, which is guaranteed to be detected by our base proof. This observation forms the basis for our induction step to obtain an unbounded proof regarding T-alerts of arbitrary temporal length. We use the list of affected variables, denoted by $A$ in Algorithms 1 and 3, to decompose the step proof of the induction procedure.

Fig. 7 shows the UPEC-OI property modified for the step proof. Unlike its base counterpart, the step property does not assume complete microarchitectural state equivalence between the two instances of the SoC, but rather only equivalence of state variables that have not yet been affected in the base proof. This allows IPC's symbolic-state initialization to implicitly "fast-forward" from time $k$ to a time point when further propagation of the difference can occur. For that, the step property only needs to span two clock cycles (from $t$ to $t+1$). The variables that have been affected in the base proof are used in assumptions of the step property via the *affected_P_var* argument. The property is run iteratively similar to the base proof, as can be seen in Algorithm 3. It should be noted the main loop rooted in Line 1 iterates through all elements of $A$, including the elements that are appended within the loop rooted in Line 4.

If the step part also does not detect any T-alerts after completion, the complete UPEC-OI induction proof holds for the

**Algorithm 3** UPEC-OI Step Property Check

```
1:  for each v in A do
2:      CE ← IPC(UPEC-OI_Step(v, PC, TC))
3:      if CE ≠ ∅ then return CE
4:      repeat
5:          CE ← IPC(UPEC-OI_Step(v, PC, PC))
6:          a_1 = Ploc(CE)
7:          PC ← PC \ a_1
8:          A ← A ∪ a_1
9:      until CE = ∅
10:     repeat
11:         CE ← IPC(UPEC-OI_Step(v, PC, BC))
12:         a_1 = Ploc(CE)
13:         Whitebox_affected_module(a_1)
14:         CE ← IPC(UPEC-OI_Step(v, PC, Y_h))
15:         if CE ≠ ∅ then return CE
16:     until CE = ∅
```

given $Z_{ps}$. Finally, if the verification succeeds for every $Z_{ps}$ in the unsecure domain, operation integrity is guaranteed for the given access control configuration, as denoted by line 12 in Algorithm 1. A proof of completeness for the methodology is available in Appendix B.

## 4 UPEC-OI-Driven Design Flow

This section proposes a UPEC-OI-driven design flow for access control mechanisms in SoCs. Conventional verification approaches require that detailed functional specifications of the considered security mechanisms are created. This, however, is quite often prone to errors, since mapping abstract security requirements into a detailed functional specification is a difficult task. In UPEC-OI we can completely avoid this problem. Instead of creating functional specifications we start from the given threat model and only formulate the global UPEC-OI property of Sec. 3. Counterexamples provided by UPEC-OI directly point to the source of an integrity vulnerability and provide the designer with information about possible solutions on how to mitigate or eliminate such security flaws.

Fig. 8 shows the basic steps of our design flow. The access control security specifications (box I) are high-level descriptions of the access control mechanism. They encompass details such as the number of security levels the system should have, granularity of the domain separation (IP-level, address-level, etc.) and, optionally, more advanced features, such as, e.g., separation of read/write permissions. UPEC-OI properties are then formulated according to this information (cf. Sec 6.2). The initial RTL design of the SoC (box II) is fed to the UPEC-OI verification procedure of Sec. 3.4 (box III).

If UPEC-OI finds a T-alert, it returns a counterexample which shows a scenario where operation integrity is breached.

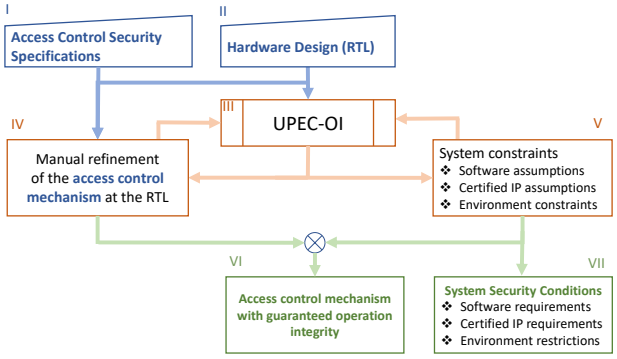



Figure 8: UPEC-OI-driven design flow.

The manual inspection of the counterexample provides the designer with valuable information on exact RTL signals involved and pathways that need to be closed. The designer can then manually refine the RTL design (box IV) until the vulnerability is eliminated, i.e., it is no longer detected by UPEC-OI.

Alternatively, counterexamples provided by UPEC-OI can also show T-alerts which are facilitated by undesired behavior of the trusted part of the system. These scenarios might not pose an actual security threat if they correspond to behavior that is spurious for the final design running secure firmware. In such a case, additional constraints can be added to the UPEC-OI properties. These restrict the proof from exploring scenarios where the trusted domain exhibits behavior that is precluded or forbidden in the deployed system. We present several classes of counterexamples which might require such constraints (box V):

*Faulty behavior of a modifiable IP*: A trusted component that can be reliably refined upon request of the system integrator is facilitating the integrity violation. Rather than extending the access control hardware, the system integrator can add a formal specification for the IP in question that forbids such unwanted behavior. These constraints can usually be specified at the interface of the IP with the rest of the system. The IP design can then be enhanced and verified accordingly.

*Wrong configuration*: The counterexample shows behavior violating operation integrity due to the IPC solver assuming a spurious configuration of the hardware (e.g., the CPU's memory protection is set up incorrectly). In such a case, constraints are added to the UPEC-OI properties in form of ISA-level configuration restrictions for programs performing these security-critical operations. The compliance of the programs with these requirements can then be verified on the firmware or software level.

*Malicious inputs*: The counterexample shows a T-alert that is caused jointly by the propagation source and input patterns from the outside. If these input combinations are not realistic to occur during a security-critical operation, such as, e.g., a debug request being sent through a JTAG interface, the

UPEC-OI properties can be constrained from these cases, and such input pattern restrictions can be provided to the end user as formal conditions for operation integrity guarantees.

*Unreachable state space*: The counterexample shows faulty behavior that is primarily due to IPC's initialization of the miter model in a state that is not reachable by the SoC from reset. One or more *hardware invariants* have to be written [27] to restrict the solver from entering that state space.

After a T-alert has been handled accordingly, the verification is restarted. This approach is repeated until no further operation integrity vulnerabilities exist. The end product is a secure-by-construction access control mechanism hardware (Box VI). Furthermore, all additional constraints used in the flow are now available as formal conditions that must be enforced by software, other parts of hardware and/or the environment to ensure operation integrity (Box VII). Appendix C elaborates on this in more detail.

## 5 Approach Evaluation

We evaluated the proposed methodology by conducting a case study on OpenTitan's Earl Grey SoC [4]. This section elaborates on how the chip was modified to include an SoC-level access control mechanism, which was then enhanced through the UPEC-OI-driven design flow. We present some of the bugs discovered by the verification methodology and the feasibility analysis of the approach. All experiments were conducted on a commercial formal verification tool OneSpin 360 DV running on an Intel i7-6700 @ 3.40 GHz machine with 32 GB of RAM and a Linux operating system. The base model for our modified design was the OpenTitan silver release edition v4 [3]. All our modifications and the verification code are openly available on GitHub [5].

### 5.1 Modifications on OpenTitan

OpenTitan is an open-source project aimed at developing a reliable Root-of-Trust (RoT) SoC which can be used to execute security-critical operations within a larger computing environment [4]. Fig. 9 shows the basic structure of the Earl Grey SoC. At the heart of this chip is a RISC-V Ibex CPU core [2] which acts as the SoC controller and communicates with various memory-mapped peripherals through *load* and *store* requests. The SoC contains a few dozen peripherals, including memories, I/O devices, resource managers, encryption and hashing IPs, etc. Depending on their required clock rate, the IP devices are either connected to the faster *xbar_main* crossbar or the slower *xbar_peri* crossbar. The IP devices and crossbars use the Ultra-Light TileLink protocol (TL-UL) [34] for communication.

All OpenTitan IPs are developed in-house and verified to comply with the overall SoC security requirements. Therefore,

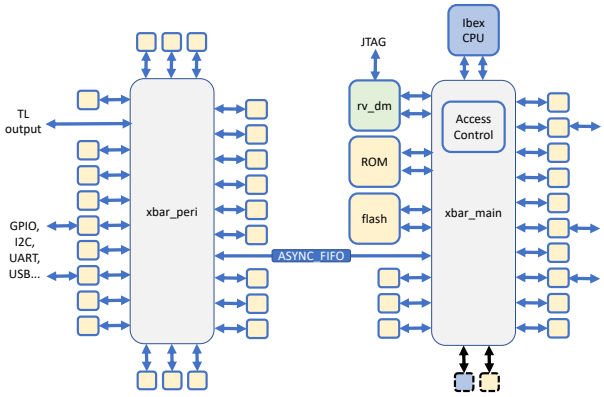



Figure 9: OpenTitan Earl Grey SoC with added components.

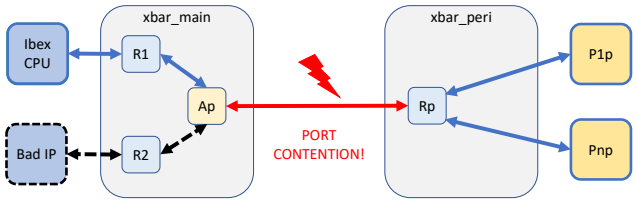
all of the SoC's IPs are trusted and no dynamic access control is implemented on the SoC level. However, our case study considers a scenario where a user designs a system by using the open-source Earl Grey platform as a starting point and adding more functionality through integration of third-party IPs. The implementation details of these IPs are unknown, which requires the SoC to have a rigorous access control mechanism in order to guarantee overall system security.

We have designed and integrated such an access control mechanism into Earl Grey based on the principles laid out in Sec. 3.1. The Ibex core was designated as the TCB and is the only IP that can modify the access control mechanism's configuration registers. These keep track of the security level $l$ for every IP with a one-bit value (1 for high, 0 for low). The routing logic in the crossbar has been modified to explicitly reject requests from low-security-level controllers to high-security-level peripherals. Communication in other directions was still allowed, as it was believed that the already existing crossbar microarchitecture would suffice to maintain operation integrity.

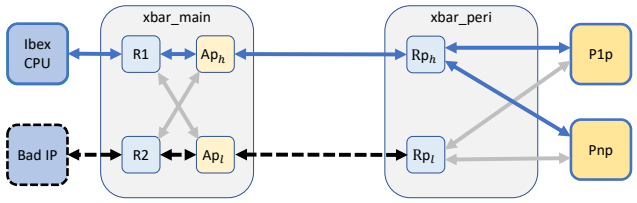
Next, we added a controller and a peripheral IP to the system and connected it to *xbar_main*. These IPs model potentially malicious components and can be seen in Fig. 9 as marked with dashed outlines. It should be noted that adding a controller IP is a realistic scenario in system integration, as some IP devices may require controller privileges to properly perform their functions. The actual RTL implementation of the added components is not important for the sake of our experiment, as they are modeling third-party IPs whose RTL is unavailable and which will be blackboxed during the actual verification stage. Finally, the modified SoC was verified regarding its functional correctness.

### 5.2 Vulnerabilities Found by UPEC-OI and RTL Enhancements

Our access control policy considers the two added IPs as low security level, while all other devices are considered as high-security-level IPs. This configuration makes sense, for




(a) Before




(b) After

Figure 10: Port contention vulnerability (a) and its fix (b).

example, during system boot-up to ensure that untrusted IPs cannot interfere with system initialization. Since the TL-UL protocol uses a valid-ready handshake, the *ready* output signals from both IPs were modelled as separate propagation sources, as they are not semantically related to the other interface signals. Therefore, the low-security-level domain was segmented into *four propagation sources*: (i) controller IP request signals, (ii) controller IP ready signal, (iii) peripheral IP response signals and (iv) peripheral IP ready signal.

The iterative induction procedure was run separately for every propagation source, as per the steps outlined in Sec. 3.4. Our verification methodology found 8 weaknesses in both the added access control RTL and already existing hardware. In the following, we present the most interesting vulnerabilities detected and the fixes that mitigate them.

### 5.2.1 Port Contention for Access to "xbar_peri"

As mentioned before, Earl Grey's interconnect comprises two crossbars, *xbar_ main*, which is connected to the CPU, high frequency peripherals, as well as our added malicious IPs, and *xbar_peri*, which is connected to slower peripherals. The two crossbars communicate through a bridge using the TL-UL protocol. As can be seen in Fig. 9, a special FIFO synchronizes messages from one crossbar's clock domain to that of the other.

Our access control mechanism uses the already existing address decode logic in the crossbars. After a request's recipient peripheral has been determined, the access control hardware checks the security levels of the issuing controller and peripheral before allowing the request to be forwarded. Since address decoding for requests designated to peripherals on *xbar_peri* is done within *xbar_peri* itself, both high- and low-security-level requests pass through the *xbar_main* - *xbar_peri* interface without any checks.

UPEC-OI found a counterexample demonstrating how the malicious controller IP could exploit this to slow down a security-critical operation involving a peripheral connected to *xbar_peri*. Fig. 10a displays the scenario. Both controllers, the high-security-level Ibex core and the low-security-level malicious IP attempt to send a request to *xbar_peri* in the same clock cycle, causing port contention. Since no access control is imposed on the interface between crossbars, both requests reach the arbiter node $Ap$ for *xbar_peri*, which implements a round-robin arbitration scheme. The IPC solver assumed a starting state where it is the malicious IP's turn to be prioritized in the arbitration. Therefore, the CPU's security-critical request is put on hold for at least one clock cycle. This seemingly insignificant delay is still enough to cause a stall in the CPU's pipeline, or, as seen through the UPEC-OI counterexample, is enough to allow difference propagation to the CPU's program counter. The delay can become much greater if the malicious controller floods *xbar_peri* with many requests and fills up the asynchronous FIFO. In this case, a secure request from the CPU would first have to wait for all unsecure requests in the FIFO to be processed, which can cause significant stalls in the operation time.

UPEC-OI also showed that simply changing the arbitration scheme to a fixed priority for high-security-level requests was not enough, as it did not solve the issue of low-security-level requests being ahead of a secure request in the FIFO. In order to completely remove this vulnerability, all timing dependency between high-security-level requests and low-security-level messages had to be eliminated. We, therefore, introduced another interface between the two crossbars. Now, the different security domains send requests through separate interfaces, with a fixed priority for servicing the high-security-level interface on *xbar_peri*'s side. This way, the low-security-level messages can still reach *xbar_peri*, but cannot interfere with the timing of high-security-level messages. Fig. 10b shows the implemented enhancement.

### 5.2.2 Stalling by Keeping the "ready" Signal Low

Suppose some peripheral IPs were in the middle of fulfilling a request for our malicious controller when they were elevated to the high-security-level domain. If they are not allowed to respond to the malicious controller, they will never become available for the pending secure operation. This is why communication in the direction high-security peripheral $\rightarrow$ low-security controller was still allowed by our access control mechanism. However, as already mentioned, the TL-UL protocol requires a *ready* signal be sent from the receiving side to complete a transaction. UPEC-OI showed how this allows the malicious controller to block the response indefinitely by keeping its *ready* signal low.

To remove this vulnerability we introduced a FIFO just before the response interface of the malicious controller. We also added control logic in the crossbar which limits the amount of requests the malicious controller can send at the same time

to be less than or equal to the depth of its new FIFO. This way, all responses to the malicious controller are stored in the FIFO regardless of its *ready* signal value, eliminating the possibility of stalling the security-critical operation this way.

#### 5.2.3 Sending Orphan Responses

The vulnerability described in the following is part of the original RTL code of OpenTitan. The counterexample found by UPEC-OI shows how the implemented hardware for message routing can be abused by a malicious peripheral to completely starve the SoC until reset.

Earl Grey's crossbar logic is constructed using a collection of router and arbiter nodes, just like in our example SoC in Fig. 1. Each router node uses two microarchitectural state variables to store the current communication state of its respective controller. One of these variables, *dev_select_outstanding*, stores the destination peripheral's ID after a request has been routed and sent successfully. The other variable, *num_req_outstanding*, records the number of requests sent by the controller that are currently pending. The node allows its controller to send multiple requests to only one peripheral at the same time. If the controller attempts sending messages to other peripherals while still having outstanding requests to the first one (i.e., *num_req_outstanding* $\neq 0$), such messages will be rejected.

In our scenario, a trusted controller IP has last communicated with a malicious peripheral IP before being elevated to the high security level domain. Even though communication between the two IP devices is finished at this point (*num_req_outstanding* $= 0$), the secure controller's router node still records the ID of this peripheral in its *dev_select_outstanding* buffer. The router node forwards response messages from the connected peripherals iff their ID corresponds to the value of *dev_select_outstanding*. This provides a time window for the malicious, low-security-level peripheral to send a new response message, even though no requests are pending. Since Earl Grey's controller IPs have their TL-UL *ready* signal hardwired to "1", any such orphan response message will be accepted. Even worse, the router node will further decrement the *num_req_outstanding* variable, causing it to overflow. Because of this, the controller IP is now blocked from communicating with any other part of the SoC, effectively starving the system.

We successfully performed the attack and managed to simulate starvation of the Earl Grey SoC. The results were reported to the OpenTitan development team, which acknowledged the weakness as a security risk. A possible fix to this vulnerability simply involves checking for *num_req_outstanding* to be larger than 0 alongside *dev_select_outstanding* in order to forward a response message from a peripheral.

Table 1: CPU time and memory consumption for different values of $k$ and $Z_{ps}$.

| $Z_{ps}$ | Time (hh:mm:ss) | Peak memory (MB) | $k$ |
|---|---|---|---|
| C request | 06:41:16 | 23902 | 12 |
| C ready | 11:14:05 | 25483 | 13 |
| P response | 00:00:26 | 6978 | 3 |
| P ready | 00:00:03 | 6146 | 1 |

### 5.3 Resource Consumption

Iterating the Earl Grey SoC through the verification-driven design flow consumed, overall, around 3 person months. This includes the effort spent on implementing the access control logic, its functional and UPEC-OI verification. During this time, the flow was iterated 19 times.

For analyzing the efficiency of UPEC-OI we measure the use of two main resources: computation time and memory. As is the case with any IPC model, consumption of these resources depends to a large degree on the temporal length of the property. Tab. 1 compares these three metrics for each of the four propagation sources (cf. Sec. 5.2). The reported values are from the last run of the UPEC-OI procedure in our design verification flow, i.e., from the run that proved the system as secure. The time presented here is the sum of all property check computation times in the inductive procedure for the particular propagation source. It should also be noted that all property checks were executed sequentially. Some steps in our procedure can be parallelized, such as, e.g., iterations of the *for* loop in Algorithm 2, line 5. The memory peak represents the highest RAM usage during any of the procedure's property checks. Finally, the last column shows the maximum temporal length $k$ of the base property that had to be reached before no more P-alerts were found.

The results show a big difference in the time and memory used to verify the controller and peripheral propagation sources. This is mostly due to the amount of cycles the IPC solver had to unroll in order to find all P-alerts in the base procedure, which greatly differs between the controller and the peripheral case. It can be seen from the peripheral case that, even though the modified SoC contains almost 14 million state bits, the design's size is not a limiting factor to the methodology. This clearly is a merit of sound blackboxing, which removes the majority of these state bits.

Most of the total procedure time elapsed during the property are checks that found no counterexamples. In our case study, all property checks that failed returned a counterexample in under 5 minutes. This makes the approach quite fast in finding security bugs. Even the overall time and memory used to certify the security of the design stays well within reasonable limits.

Our experiments show that the exhaustive verification of the global *OI* property in an SoC of a realistic size is feasible with moderate manual efforts. This is a clear advantage over conventional functional verification techniques. Those primarily rely on the formalization of selected design behaviors, as well as on a priori knowledge of attack surfaces and scenarios. By their nature, these techniques are inherently prone to missing corner cases and previously unknown attack patterns. There also exist approaches [27, 36] exhaustively covering functional design specification by sets of properties. This exercise has to be applied to every peripheral of the SoC, which, however, can be prohibitively laborious in practice.

We review the state of the art in secure SoC-level access control mechanisms and their verification in more detail in the following section.

## 6 Related Work and Discussion

There have been numerous works in the past on secure SoC access control architectures. A popular approach involves adding security wrappers around untrusted IP devices [18, 32]. These wrappers monitor their respective IPs to prevent any kind of illegal messages from reaching the security-critical part of the system. However, their monitoring logic is statically implemented in hardware, and relies on the designer and/or hardware analysis tools to determine unsecure access patterns and specify them explicitly, which is prone to error. In contrast, our methodology implicitly covers all attack patterns. Prior knowledge of possible attack scenarios is not required and there is no need for explicit specification of allowed behavior.

Other works propose similar monitors that are reprogrammable during the chip's lifetime in order to patch up the system for any newly discovered vulnerabilities [9, 11, 25, 35]. These works are based on the assumption that some security vulnerabilities will inevitably be missed during the design phase. The goal of this paper, however, is to not allow any access control integrity vulnerabilities to be missed. Having formal security guarantees at the end of pre-silicon development reduces the need for such monitors to only mitigate post-silicon security threats. Restuccia et al. [30] also propose a wrapper-style access control system, and emphasize the need to verify their access control hardware. In their approach, security properties are formulated based on the MITRE CWE database [1]. Although this greatly reduces the risk of such an access control mechanism being compromised, it still does not provide a formal guarantee that the system is secure, as the properties only cover already known weaknesses.

A hardware Trojan detection technique based on property checking is proposed in [29]. Despite the improved coverage, the required effort and expertise for manually formulating several security properties is a major weakness for this approach. Wang et al. propose an automated method of generating such security assertions for finding hardware Trojans in [38]. The approach relies on extracting *RTL invariants* from numerous simulations traces that are produced during the system's functional validation phase. These invariant properties are then checked by a formal tool in an attempt to find uncommon behavior that could be caused by a hardware Trojan. Although it is a promising approach, it still relies on previous security expertise to manually select signals that are to be checked based on their likelihood of being a Trojan, which leaves room for missing a bug.

Rajendran et al. use Bounded Model Checking (BMC) to look for SoC integrity vulnerabilities in [28]. They formulate security properties which make sure that security-critical buffers can only be updated in *legal* ways. This requires manual specification of what is allowed, which can be cumbersome and is prone to error. Furthermore, the approach uses general BMC which can only provide a bounded proof of a property.

Other works [19, 37] propose security verification methods for hardware described on higher abstraction layers. Although this greatly reduces complexity, it is ignorant of the fine details of the hardware microarchitecture which can also be a cause of security vulnerabilities. We believe that verification at the RTL is necessary to guarantee pre-silicon security for as long as RTL is used as the golden model for hardware implementation in chip design.

### 6.1 UPEC-OI for Weaker Notions of Integrity

None of the aforementioned approaches formulate precise security requirements that their access control mechanisms and/or verification methodologies try to satisfy. The main integrity objective for the security wrappers in [9], [18], [30] and [35] is restricting illegal modification of trusted buffers of the SoC by untrusted IPs. Similarly, [28] formulates properties which intend to detect the same kinds of illegal modifications of protected data. This security aspect is fully covered by our *OI* hyperproperty and the UPEC-OI methodology. Any possible modification of the high-security-level domain's data by the propagation source is detected by a UPEC-OI counterexample. In addition, UPEC-OI also covers any timing interference that the propagation source can have with the protected parts of the SoC, as demonstrated by the counterexamples presented in Sec. 5.

There may exist cases where some of the UPEC-OI restrictions on the information flow could be deemed too conservative and detrimental to the desired SoC performance goals. The possible impact of low-security hardware on the timing of the protected part of the SoC is not an integrity issue under all circumstances [33]. Therefore, timing interference by the low-security-level domain of the SoC can be permissible in some applications. Although UPEC-OI is intrinsically sensitive to any interference with the trusted subsystem's cycle-accurate behavior, the property can be modified such

that timing interferences are neglected. The verification engineer can identify control signals in the untrusted hardware which regulate timing of operations, and constrain them to be equal between the two SoC instances. Meanwhile, the data signals would be left unconstrained. In this way, UPEC-OI would only consider cases where the untrusted IP's control signals, such as e.g., *valid* and *ready* signals in the TL-UL protocol, cause a transaction with identical timing in the two SoC instances, but with possibly different data. If such data reaches any T-alert candidate buffers, a counterexample is generated. However, cases such as port contention, message delays, denial-of-service attacks, etc. would be ignored in the modified approach. In such a case the verified design conforms to a weaker notion of integrity, compared to the UPEC-OI hyperproperty (cf. Sec. 3.2).

### 6.2 UPEC-OI for Different Variants of Access Control Mechanisms and Policies

In Sec. 3 we introduce UPEC-OI based on a general model of access control. Variants of access control mechanisms with different practical features can be mapped to UPEC-OI, as described in the following.

*More complex security lattices:* In practice, security lattices enforced by SoC-level access control mechanisms can have more than two security levels, as well as mutually exclusive sandbox environments. UPEC-OI can still map hardware in such a lattice into its two-security-level model: all security levels and domains that are allowed to influence the security-critical operation currently under verification are considered high-security-level, while all the others are considered low-security-level. For example, let us consider a security lattice for an SoC with 3 security levels, ordered from highest to lowest: *machine, supervisor* and *user*. Furthermore, let us assume that the *user* security level contains two mutually isolating execution environments: *local* and *remote*. If we are verifying hardware for integrity of operations in the *local* environment, then the high security level includes all security levels and subdomains that are allowed to influence operations in the *local* subdomain: the entire *machine* and *supervisor* security level, as well as the *local* subdomain of the *user* security level. The *remote* subdomain is assigned the low security level.

*Read/Write permissions:* A common feature in access control mechanisms is the distinction between read and write permissions. In the context of integrity, UPEC-OI should only verify information flow restrictions related to *write* permissions.

*Address-level granularity:* Many SoCs feature memory-mapped peripherals, i.e., different registers within a single IP can have distinct addresses. In such systems, the access control mechanism can also have the same address-level granularity, so that a controller IP might have access to some registers of a peripheral, but not to others. This class of access control mechanisms can also be verified with UPEC-OI by categorizing individual state variables of an IP into the low- or high-security-level domain, as opposed to entire IPs (cf. Sec. 3.2). It should be noted that a separate proof for each register may be unnecessary, since the configuration of the access control can be represented symbolically, which implicitly covers all possible situations with different trust boundaries. A work employing symbolic representation of security configurations for confidentiality verification is reported in [26].

## 7 Conclusion

This paper shows that ensuring integrity of an SoC's access control mechanism using a formal verification approach is possible and feasible. The considered property targets the integrity of any security-critical operation within the system's secure domain against all activities of an untrusted third-party IP. We present proof procedures that decompose this global verification problem such that it can be solved for SoCs of realistic size. An important element of our proof procedures is sound blackboxing. It does not only turn out to make a major contribution to handling computational complexity but also allows for giving integrity guarantees even at the presence of third-party IPs whose internal microarchitecture is not disclosed by the provider.

We show how our verification approach can drive the design of effective SoC-level access control mechanisms such that the final design is guaranteed to be secure w.r.t. the considered integrity property. This is practically demonstrated at the example of the security-conscious OpenTitan's Earl Grey SoC. To the best of our knowledge, this is the first time that such strong guarantee can be given for a design of this kind.

Future work includes expanding the methodology to cover also other security targets and to create a verification-driven design framework with formal guarantees of resilience against transient execution attacks [15], functional confidentiality bugs [26], data-dependent computing [13], and more.

## 8 Acknowledgements

We are grateful to all our reviewers for substantial feedback and help with revising the paper. We further thank Jörg Bormann from Siemens EDA OneSpin, as well as Sayak Ray and Jason Fung from Intel for fruitful discussions and suggestions related to this research. The work reported in this paper was partially supported by research grants BMBF ZuSE (Scale4Edge), 16ME0122K-16ME0140+16ME0465, DFG SPP Nano Security, KU 1051/11-1 and by the Intel Corp. Scalable Assurance Program.

## References

[1] Common weakness enumeration. https://cwe.mitre.org/.

[2] Ibex core documentation website. https://ibex-core.readthedocs.io/en/latest/index.html.

[3] OpenTitan GitHub repository. https://github.com/lowRISC/opentitan.

[4] OpenTitan website. https://opentitan.org/.

[5] UPEC-OI enhanced OpenTitan GitHub repository. https://github.com/nodix95/opentitan.

[6] Markuze Alex, Shay Vargaftik, Gil Kupfer, Boris Pismeny, Nadav Amit, Adam Morrison, and Dan Tsafrir. Characterizing, exploiting, and detecting DMA code injection vulnerabilities in the presence of an IOMMU. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 395–409. Association for Computing Machinery, 2021.

[7] Arm Limited. AMBA®AXI and ACE protocol specification, 2021.

[8] ARM Limited. Arm system memory management unit architecture specification®SMMU, architecture version 3, 2021.

[9] Abhishek Basak, Swarup Bhunia, and Sandip Ray. A flexible architecture for systematic implementation of SoC security policies. In *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 536–543, 2015.

[10] Michael R Clarkson and Fred B Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

[11] Atul Prasad Deb Nath, Sandip Ray, Abhishek Basak, and Swarup Bhunia. System-on-chip security architecture and CAD framework for hardware patch. In *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 733–738, 2018.

[12] Ghada Dessouky, David Gens, Patrick Haney, Garrett Persyn, Arun Kanuparthi, Hareesh Khattri, Jason M Fung, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. Hardfails: Insights into software-exploitable hardware bugs. In *USENIX Security Symposium*, pages 213–230, 2019.

[13] Lucas Deutschmann, Johannes Müller, Mohammad R. Fadiheh, Dominik Stoffel, and Wolfgang Kunz. Towards a formally verified hardware root-of-trust for data-oblivious computing. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, DAC'22, page 727–732, New York, NY, USA, 2022. Association for Computing Machinery.

[14] Mohammad Rahmani Fadiheh, Dominik Stoffel, Clark Barrett, Subhasish Mitra, and Wolfgang Kunz. Processor hardware security vulnerabilities and their detection by unique program execution checking. In *Design, Automation & Test in Europe Conf. (DATE)*, pages 994–999, 2019.

[15] Mohammad Rahmani Fadiheh, Alex Wezel, Johannes Mueller, Joerg Bormann, Sayak Ray, Jason M Fung, Subhasish Mitra, Dominik Stoffel, and Wolfgang Kunz. An exhaustive approach to detecting transient execution side channels in RTL designs of processors. *IEEE Transactions on Computers*, 2022.

[16] J. A. Goguen and J. Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy*, pages 11–11, 1982.

[17] Mathieu Gross, Nisha Jacob, Andreas Zankl, and Georg Sigl. Breaking TrustZone memory isolation and secure boot through malicious hardware on a modern FPGA-SoC. 2015.

[18] Festus Hategekimana, Taylor J L Whitaker, Md Jubaer Hossain Pantho, and Christophe Bobda. Secure integration of non-trusted ips in socs. In *2017 Asian Hardware Oriented Security and Trust Symposium (AsianHOST)*, pages 103–108, 2017.

[19] Bo-Yuan Huang, Sayak Ray, Aarti Gupta, Jason M Fung, and Sharad Malik. Formal security verification of concurrent firmware in SoCs using instruction-level abstraction for hardware. In *IEEE/ACM Design Automation Conference (DAC)*, pages 1–6, 2018.

[20] Intel Corporation. Intel® virtualization technology for directed I/O architecture specification, 2022.

[21] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *arXiv preprint arXiv:1801.01203*, 2018.

[22] Nick Kossifidis. Updates from the RISC-V TEE group, 2021.

[23] Hyung Gyu Lee, Naehyuck Chang, Umit Y. Ogras, and Radu Marculescu. On-chip communication architecture exploration: A quantitative evaluation of point-to-point, bus, and network-on-chip approaches. 12(3), may 2008.

[24] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. pages 973–990, 2018.

[25] Wei-Kai Liu, Benjamin Tan, Jason M. Fung, Ramesh Karri, and Krishnendu Chakrabarty. Hardware-supported patching of security bugs in hardware IP blocks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2022.

[26] Johannes Müller, Mohammad Rahmani Fadiheh, Anna Lena Duque Anton, Thomas Eisenbarth, Dominik Stoffel, and Wolfgang Kunz. A formal approach to confidentiality verification in SoCs at the register transfer level. In *IEEE/ACM Design Automation Conference (DAC)*. IEEE, 2021.

[27] Minh D. Nguyen, Max Thalmaier, Markus Wedler, Jörg Bormann, Dominik Stoffel, and Wolfgang Kunz. Unbounded Protocol Compliance Verification using Interval Property Checking with Invariants. *IEEE Transactions on Computer-Aided Design*, 27(11):2068–2082, November 2008.

[28] Jeyavijayan Rajendran, Vivekananda Vedula, and Ramesh Karri. Detecting malicious modifications of data in third-party intellectual property cores. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2015.

[29] Michael Rathmair, Florian Schupfer, and Christian Krieg. Applied formal methods for hardware trojan detection. In *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 169–172, 2014.

[30] Francesco Restuccia, Andres Meza, and Ryan Kastner. AKER: A design and verification framework for safe and secure SoC access control. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9, 2021.

[31] A William Roscoe. CSP and determinism in security modelling. In *Proc. IEEE Symposium on Security and Privacy*, pages 114–127. IEEE, 1995.

[32] Heba Salem and Nigel Topham. Trustworthy computing on untrustworthy and trojan-infected on-chip interconnects. In *2021 IEEE European Test Symposium (ETS)*, pages 1–2, 2021.

[33] Spyridon Samonas and David Coss. The cia strikes back: Redefining confidentiality, integrity and availability in security. *Journal of Information System Security*, 10(3), 2014.

[34] SiFive. Sifive tilelink specification, 2017.

[35] Benjamin Tan, Rana Elnaggar, Jason M. Fung, Ramesh Karri, and Krishnendu Chakrabarty. Toward hardware-based IP vulnerability detection and post-deployment patching in systems-on-chip. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 40(6):1158–1171, 2021.

[36] Joakim Urdahl, Dominik Stoffel, and Wolfgang Kunz. 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):291–304, Feb. 2014.

[37] Nandeesha Veeranna and Benjamin Carrion Schafer. Hardware trojan detection in behavioral intellectual properties (IP's) using property checking techniques. *IEEE Transactions on Emerging Topics in Computing*, 5(4):576–585, 2017.

[38] Chenguang Wang, Yici Cai, Qiang Zhou, and Haoyi Wang. ASAX: Automatic security assertion extraction for detecting hardware trojans. In *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 84–89, 2018.

[39] A. Waterman and K. Asanović. The RISC-V Instruction Set Manual, Version 20191213, 2019.

[40] Joseph Yiu. ARMv8-M architecture technical overview, 2015.

## A Example of *access_control_configured()*

In this appendix we provide a code snippet of the macro *access_control_configured()* from our OpenTitan case study. Here, the added controller and peripheral were considered to be low-security, while all other IPs were on the high-security level. As can be seen, the controller security level register lists the added controller as low security ("0"), while the peripheral security level register assumes all peripherals as secure (binary "1"), except for the added peripheral, whose security level is defined by the register's least-significant bit.

```
function automatic access_control_configured();
access_control_configured = (
miter.inst_1.acm.controller_cfgreg == 1'b0 &&
miter.inst_1.acm.peripheral_cfgreg == 19'h7fffe
);
endfunction
```

## B Proof of Correctness for the UPEC-OI Induction-Based Procedure

**Theorem:** For a given SoC design, if the UPEC-OI methodology does not detect any T-alert, the design fulfills the *OI* hyperproperty.

**Proof:** We observe the following characteristics of the UPEC-OI methodology:

1. The IPC solver used in the UPEC-OI methodology performs any-state proofs with a symbolic starting state, i.e., it explores all possible valuations of state variables that fulfill the assumptions of the property.

2. The macro *access_control_configured()* in the UPEC-OI properties does not restrict the state of any part of the system other than $L$. It restricts $L$ to hold a valuation that is an element of $\Lambda$.
3. All other assumptions of UPEC-OI only restrict the relation between the two instances in the miter model and not the individual instance's state space.
4. T-alert candidates (TC) and P-alert candidates (PC) together form the set of all whiteboxed high-security-level state variables in the SoC at the start of the UPEC-OI verification process. A variable is removed from PC only when it has already been detected as affected.

Based on the described observations, we can prove the theorem by contradiction. Assume that there exist an execution trace (*CEX*) that violates the hyperproperty *OI* $d$ clock cycles after the initial property time point $t$, but is not detected by the UPEC-OI methodology. If we label the UPEC-OI_Base property's largest time window as $k$, we can distinguish the following two cases:

1. If $d \leq k$:

   CEX shows at least one high-security level output variable ($y_h \in Y_h$) that depends on the propagation source. For all the time windows smaller or equal to $k$ clock cycles, the UPEC-OI_Base method (Alg. 2) exhaustively checks whether T-alert candidates, which include $Y_h$ as a subset, depend on the propagation source (line 6 in Alg. 2). Therefore, there is definitely a call to the IPC solver with UPEC-OI_Base property for the time window of length $d$, in which $Y_h$ is compared between the two design instances in the miter. This IPC call may only fail to detect a T-alert if either the starting state of *CEX* for one of the SoC instances at $t_0$ is not included in the symbolic starting state of the UPEC-OI interval property, or the state of *CEX* for one of the SoC instances at $t_0$ violates the *access_control_configured()* macro assumptions of the interval property. The first case is in contradiction with the any-state proof of IPC, since the symbolic starting state includes, as a subset, all reachable states of the system. The second case is in contradiction with the *OI* hyperproperty since any trace that violates the macro does not represent a security-critical operation $\tau \in T$.

2. If $d > k$:

   There exists a starting subsequence (prefix) of *CEX* with a temporal length shorter than or equal to $k$. This prefix shows an execution trace in which a certain high-security level state variable ($z_h^p \in Z_h$) depends on the propagation source. Apart from T-alert candidates (TC), the UPEC-OI_Base method also checks the base property for all P-alert candidates (PC) for time windows up to $k$ clock cycles (Line 9 in Alg. 2). Therefore, there is definitely a call to the IPC solver with the UPEC-OI_Base property, in which $z_h^p$ is compared between the two design instances in the miter. Based on similar reasoning as in (1), the UPEC-OI_Base proof is guaranteed to find a P-alert that corresponds to the prefix trace. For this P-alert, UPEC-OI_Step may only fail to detect the subsequent T-alert if the states of the SoC instances reached at time later than $t+k$ in *CEX* are either not included in the symbolic starting state of the UPEC-OI step interval property or they violate the *access_control_configured()* assumption macro. This is again in contradiction with the underlying assumptions of the proof as explained above.

# C Soundness of the UPEC-OI-Driven Design Flow

We label the set of all states that are reachable by the SoC from *reset* as $R$. During a security-critical operation, even within $R$ there can exist illegal configurations of the system, caused by e.g., improper firmware commands, unexpected environment inputs, etc. We define a subset of $R$ that excludes such unwanted states of the system as the security-reachable state space, $S$. We make the following two observations:

1. All hardware invariants employed in the IPC proof must be supersets of $R$.
2. All other input and firmware constraints, as discussed in Chapter 4, restrict the proof to a state set, which can be a subset of $R$, but must be a superset of $S$.

Complying with these two requirements guarantees that the UPEC-OI-driven design methodology is sound and all practically relevant scenarios for security-critical operations are considered.

In view of the *OI* hyperproperty of Sec 3.2, the first iteration of the design flow considers the trace set $T$. The set of all states traversed by the traces in $T$ is a superset of $S$. This means that $T$ may also include some unwanted behaviors. In the sequence of iterations, counterexamples guide a refinement process, where these unwanted behaviors are removed by constraints and invariants. This corresponds to refining the set $T$ to a subset that is practically relevant. This subset still contains all traces that traverse states in $S$.

Ensuring the soundness of hardware invariants is a straightforward process and many verification tools feature dedicated solvers to prove their soundness. The soundness of input and firmware constraints can also be ensured by considering them as requirements for the system's firmware and/or the outside environment, which can be checked separately, e.g., by using firmware verification techniques. This compositional approach guarantees that the completeness of the UPEC-OI verification flow is maintained within our design methodology.