# AutoCC : Automatic discovery of Covert Channels in Time-shared Hardware

Marcelo Orenes-Vera
Princeton University
Princeton, USA

Hyunsung Yun
Princeton University
Princeton, USA

Nils Wistoff
ETH Zurich
Zurich, Switzerland

Gernot Heiser
UNSW Sydney
Sydney, Australia

Luca Benini
ETH Zurich
Zurich, Switzerland

David Wentzlaff
Princeton University
Princeton, USA

Margaret Martonosi
Princeton University
Princeton, USA

## ABSTRACT

Covert channels enable information leakage between security domains that should be isolated by observing execution differences in shared hardware. These channels can appear in any stateful shared resource, including caches, predictors, and accelerators. Previous works have identified many vulnerable components, demonstrating and defending against attacks via reverse engineering. However, this approach requires much human effort and reasoning. With the Cambrian explosion of specialized hardware, it is becoming increasingly difficult to identify all vulnerabilities manually.

To tackle this challenge, we propose AutoCC, a methodology that leverages formal property verification (FPV) to automatically discover covert channels in hardware that is shared between processes. AutoCC operates at the register-transfer level (RTL) to exhaustively examine any machine state left by a process after a context switch that creates an execution difference. Upon finding such a difference, AutoCC provides a precise execution trace showing how the information was encoded into the machine state and recovered.

Leveraging AutoCC's flow to generate FPV testbenches that apply our methodology, we evaluated it on four open-source hardware projects, including two RISC-V cores and two accelerators. Without hand-written code or directed tests, AutoCC uncovered known covert channels (within minutes instead of many hours of test-driven emulations) and unknown ones. Although AutoCC is primarily intended to find covert channels, our evaluation has also found RTL bugs, demonstrating that AutoCC is an effective tool to test both the security and reliability of hardware designs.

## CCS CONCEPTS

• **Security and privacy** → **Side-channel analysis and counter-measures**; *Tamper-proof and tamper-resistant designs*; *Information flow control*; • **Hardware** → Best practices for EDA.
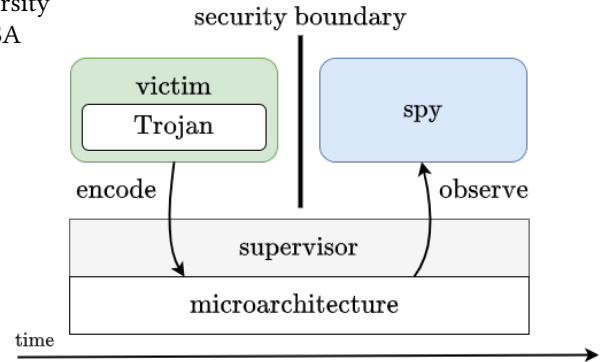
Figure 1: A microarchitectural covert channel. The Trojan in the victim process modifies—via permitted operations—microarchitectural state to encode a secret. The spy process observes this modification, directly or via a timing difference, to infer the secret. Sec. 2.1 exemplifies using a covert channel.

## KEYWORDS

FPV, formal, verification, covert channel, microarchitectural, timing channel, information flow, data leak, temporal partitioning, flush.

## 1 INTRODUCTION

The end of Moore's law has given rise to complex and heterogeneous System-on-Chip (SoC) designs, which are composed of diverse hardware blocks and intricate software systems [5, 10, 18, 22, 40, 54, 57, 60, 62]. Ensuring the security of these systems is becoming increasingly challenging due to the sheer number of hardware modules and their interactions [4, 47, 49]. In particular, microarchitectural covert channels, which exploit hardware state hidden by the instruction set architecture (ISA)[64], pose a significant threat to system security, allowing unauthorized information flow across security boundaries[33].

Uncovering covert channels in heterogeneous SoCs during simulation and emulation-based testing is akin to finding a needle in a haystack, requiring much engineering effort, time, and cleverness

to create tests that *exercise* all possible vulnerabilities. Moreover, upon empirically observing a channel, it is difficult to *find the root cause*, as the state that leaks information is often not directly observable [64]. Even when this cause is found, verifying the effectiveness of RTL fixes is challenging, as design changes may alter the execution that previously exercised the issue.

Formal property verification (FPV) is a promising alternative to exhaustively and precisely find covert channels without relying on tests. However, FPV also presents several *challenges*, such as a steep learning curve, the difficulty of formalizing the security problem to find the desired behavior as property counterexamples (CEXs), and the exponential growth of FPV tool runtime with the increase in hardware state size.

***Our approach:*** To tackle these challenges, we present AutoCC, a novel methodology that frames the problem of finding covert channels in time-shared hardware (as described in Fig. 1) into an FPV testbench (FT). We also introduce an automated flow that generates FTs implementing our methodology by simply providing the path to an RTL module and a target FPV tool. This **systematic** approach enables RTL designers to explore potential data leaks (between processes that time-multiplex the usage of a hardware IP block) without needing to reason about which states may leak. Our **modular** methodology makes it suitable for large designs—circumventing the exponential state growth. The **automatic** generation of FTs makes our methodology accessible to RTL designers without prior knowledge of formal methods.

The security of a hardware system depends on the security of each component; AutoCC enables designers to more efficiently and effectively identify and address covert channels in heterogeneous SoC designs, enhancing overall system security.

Our main technical **contributions** are:

- A modular FPV methodology that exhaustively searches for execution traces within a victim process that lead to execution differences observable to a spy process.
- An automated procedure to generate an FPV testbench that applies the above methodology without requiring any upfront user input or RTL details.
- Uncovering covert channels and hardware bugs in the mature open-source RISC-V CVA6 core and MAPLE accelerator.

We evaluate and demonstrate that AutoCC's methodology:

- Exercises previously known and new hardware issues in minutes (as opposed to hours of stress-test simulation).
- Finds the root cause of a CEX with little engineering effort since the length of the execution trace is minimal.
- Uncovers experimentally viable covert channels that we can validate in system-level RTL simulation.
- Validates that the RTL fixes to address covert channels are effective since they eliminate the CEXs.

## 2 BACKGROUND AND PRIOR WORK

Process isolation is fundamental to system security and the primary mechanism by which information is confined to appropriate domains. A covert channel is an information flow that uses a mechanism not intended for information transfer [33]; it enables information leakage across security boundaries of the operating system (OS) and between domains that should be isolated— violating the

system's security policy. For example, a spy process may leverage a covert channel to extract a secret from a victim process.

Covert channels can be categorized based on the source of their data leakage. For example, *physical channels* rely on measurable changes in the electromagnetic field or power draw to extract information [2, 61]. *Microarchitectural channels* exploit hardware states invisible to the instruction set architecture (ISA) to enable unauthorized information flow [17, 64]. Our work focuses on the latter; for the rest of the paper, when we say covert channels, we refer specifically to the microarchitectural ones.

### 2.1 Covert Channels

Covert channels have been demonstrated via the L1-D [23] and L1-I caches [1], the last-level cache (LLC) [28, 37], the TLB [20, 25], the branch predictor [1], and the interconnect [48, 66]. The Spectre attack [30] famously demonstrated the practicality of covert channels by combining them with speculation to a so-called *transient execution attack*. Similar attacks were later presented by exploiting additional covert channels [51, 59].

***Motivating Example:*** Let us assume a setup as shown in Fig. 1 to motivate the threat scenario. The victim and the spy are two applications running concurrently on shared hardware. They are (supposedly) isolated by a supervisor using an established mechanism for memory protection. However, this security boundary can be bypassed using a covert channel, for example, by a *prime-and-probe* attack on the L1 data cache: the spy first *primes* the data cache by accessing each element of a data array with the size of the data cache (*prime buffer*), filling the L1 data cache with it. During the victim's execution time slice, the embedded (malicious or unwitting) Trojan encodes a secret $S$ into the microarchitectural state, in this example, by evicting $S$ cache lines with its own data. Finally, the spy again accesses its entire prime buffer, measuring its execution time. Doing so, it observes a latency that linearly depends on the number of cache misses, through which it can infer the number of cache lines that the Trojan evicted and thus the victim's secret $S$.

***Resource Sharing:*** A microarchitectural covert channel is possible when the spy and the victim processes share a resource. Exploitable resources are those holding state that depends on execution history, and that can impact the timing or behavior of future instructions. This includes the hardware units mentioned above and also subtle ones like arbiters, buffers, and FSMs. Regarding how the processes share the resource over time, we distinguish between hardware threads *simultaneously sharing* a resource (e.g., a pipeline or a shared cache) and software threads *time-sharing* a resource (e.g., time-multiplexing a core or an accelerator) [17]. Our threat model (detailed in Sec. 3.1) is based on time-shared hardware because (a) it is common in specialized hardware, and (b) a security domain may already prefer not simultaneously sharing capacity- or bandwidth-limited resources (e.g., instruction cache, TLBs, predictors, etc.) to avoid contention-derived information leakage.

***Spy's Observation Model:*** For data to leak from the victim to the spy process, the spy must be able to observe some fraction of the victim's execution. *Timing channels* result from observable timing differences in the spy's execution originating from microarchitectural states whose value depends on the victim's secret [17]. Other channels might infer the contents of these states directly based

on the outcomes of executing unauthorized operations. The latter are frequently regarded as hardware bugs in security literature, as unauthorized access attempts should not leave traces dependent on the requested data. As Sec. 3 explains, AutoCC detects differences at RTL module interfaces, and thus, its observation model is applicable to all microarchitectural channels.

**Victim's Intent:** Regarding the intention of the execution trace within the victim process that enabled the information leakage, the literature considers *side channels* as the subset of the covert channels where the victim process leaks inadvertently, while the rest rely on a malicious function—a Trojan—to use the secret in a specific way that actively leaks information across the security boundary. Our methodology is agnostic to intent, as it explores every possible execution that enables the covert channel.

**Protections:** The literature in security offers two alternative protections against timing channels: partitioning of hardware resources and constant-time implementations of cryptography software [11]. In a simultaneous multi-threading processor, *hardware partitioning* spatially divides shared resources like caches or prediction tables. In a time-shared processor, shared resources are temporally partitioned via a flush [64]—this is the mechanism we evaluate in this work. *Constant-time programming* does not necessarily mean that the execution time is deterministic, but that it does not depend on the secret data [19]. This programming style avoids branches and array indexing based on secret data. This is done so that benevolent software does not inadvertently leak information (a side channel). Our methodology, by default, does not restrict the type of instructions that can be executed since we focus on finding covert channels to be closed in hardware. However, a user can also constrain the FPV environment generated by AutoCC to only explore executions that are allowed under constant-time programming. Such an environment would verify that a hardware design does not leak data while executing constant-time software. Sec. 5 further discusses the tradeoffs of protecting against covert channels in hardware versus restricting the software.

**Detection:** Information flow security in hardware has been actively explored since the early 2010s [3, 42, 52, 53, 71]. While these approaches focus on monitoring and controlling the flow of sensitive data through hardware components to mitigate security threats, they do so via RTL simulation. As such, they are as effective as the test cases provided. Although constrained-random testing and fuzzing can be used to generate a wide range of test cases [9, 26, 29, 32, 58], they are not as exhaustive as formal methods. Subtle timing differences can be exploited to extract secrets—if targeted efficiently, even a binary channel can leak a 256-bit AES key in under a second for a typical context switch frequency of 1kHz [64]. Thus, formal methods are key to finding *every* channel.

## 2.2 Formal Methods for Hardware Verification

The first works to ensure RTL correctness through formal verification utilized model checking with SAT solvers and binary decision diagrams [6, 41, 50]. For a given design under test (DUT), a model checker generates a state space of all possible executions of the DUT, given its inputs and the specified assumptions. *Assumptions* constrain the state space exploration by preventing some behaviors, while *assertions* check that properties hold on all the explored paths.

FPV backend tools use a variety of solver engines [8, 65] to search for property violations (counterexamples) exhaustively. Bounded Model Checking (BMC) is the method of choice for many solver engines today. In BMC, correctness properties are unwound to a bounded number of transitions $k$, reducing the problem of model checking to an instance of SAT. For AutoCC, this means proving the property for all $k$-cycle executions of the DUT—every successful proof increments $k$. *What does this mean for completeness?* A bounded proof of a property for $k$ cycles means that the property holds for executions of less than or equal to $k$ cycles—longer executions may still result in a property violation. To prove the property for unbounded executions, $k$ must reach a completeness threshold [55]. A naive threshold is the number of states in the model; a tighter one is the shortest path between the two states furthest apart in the model [13]. In practice, reaching this completeness threshold is not always possible; the checker may run out of time or memory, or the threshold itself may be hard to compute.

Prior work has leveraged FPV for different purposes: RTLCheck verifies RTL implementations of CPUs against their memory consistency models [39]; ILA generates a Verilog model of the design from its functional specification and compares it against the RTL implementation [24]; and AutoSVA checks the liveness properties of RTL module interactions [47]. Liveness properties specify that "something good will happen," e.g., a request is eventually acknowledged, while safety properties specify that "nothing bad will happen," e.g., a response must have had a request. In the context of covert channels, we are interested in safety properties that detect data leakage across processes. Sec. 3 elaborates on how AutoCC frames this detection as an FPV problem.

Formal methods have also been used to detect security vulnerabilities. InSpectre [21] creates formal models of processors to detect Spectre-like attacks that combine speculative execution and a covert channel. UPEC [14] uses FPV to detect memory leakages via side effects of non-permitted operations. However, UPEC is limited to uncovering memory leakages (e.g., through stale microarchitectural state) and does not consider leakage due to execution time.

To extend the scope of prior work based on formal methods, AutoCC uses FPV on hardware RTL to automatically detect microarchitectural covert channels originating from states whose value depends on a previous execution and impacts the timing of future instructions. AutoCC complements empirical covert channel measurement frameworks such as *Channel Bench* [16], which show the (non-)existence of some specific channels, but not all.

## 3 THE AUTOCC APPROACH

This section first presents the threat model we tackle in this paper, i.e., time-multiplexed executions of processes on shared hardware. Sec. 3.2 then describes how we formalize that threat model (into a problem that FPV engines can solve) in order to discover covert channels between these processes automatically. Sec. 3.3 explains how to apply this methodology to an RTL project using our automated flow, which generates the FPV testbench and tool bindings. Sec. 3.4 proposes a viable path for applying AutoCC to large projects via modularity. Finally, Sec. 3.5 introduces two strategies that leverage AutoCC to assist the correct design of temporal protections against covert channels.

## 3.1 The AutoCC Threat Model

The AutoCC threat model assumes two processes, an *attacker* and a *victim*, executing on time-shared hardware and separated via a context switch enforced by the operating system (OS). Both processes are untrusted, and the victim runs in a controlled environment where the OS restricts with whom the victim may communicate.

The attacker process possesses no special privileges and executes in a security domain of its own. In theory, no hardware state should leak data from the victim to the attacker since the processes are located in different security domains. However, an attacker could use a covert channel to extract information illegally. Its primary asset is a *Trojan*, i.e., a piece of code in the victim process that enables the data leak (as depicted in Fig. 1).

As a tool for hardware designers, AutoCC's emphasis is on sensitivity. That is to say, its goal is to expose the full set of possible covert channels to the designers, who then decide the course of action (Sec. 5 discusses decision tradeoffs). As such, we place no constraint on how the secret data is encoded into the state of the compromised hardware, i.e., the Trojan can be a malicious hidden function of the victim process or innocent code that leaks data inadvertently as a side effect of a legitimate operation. Aiming to find every covert channel—regardless of the intent of the code enabling it—allows us to prove stronger correctness assertions, i.e., hardware free of covert channels must also be free of side channels.

We further note that this threat model is not restricted to CPUs. Accelerators and other specialized hardware blocks are often shared between processes in a time-multiplexed manner, and they are also susceptible to covert channels. The operations available to these specialized hardware blocks can be considered as their ISA [70]. For the rest of the paper, design under test (DUT) refers to the top-level module we are testing, regardless of its level of specialization.

## 3.2 Formalizing the Threat Model for FPV

Having defined the threat model, we now explain how we formalize it as a problem for FPV by pushing the FPV tool closer and closer to modeling the scenario described above.

For our formalization, we consider the following definitions:

**Definition 1** (State). *The state of a DUT is the set of all flip-flops, registers, and memory cells contained within that hardware module and its instantiated submodules.*

The DUT defines our universe of discourse; any RTL outside of the DUT is not considered. This distinction is especially relevant for our discussion on modularity in Sec. 3.4.

**Definition 2** (Architectural State). *The architectural state (arch) of a DUT is the subset of the state that is readable via ISA instructions.*

**Definition 3** (Microarchitectural State). *The microarchitectural state ($\mu$arch) is the subset of the state that is not part of arch (not directly readable via ISA instructions).*

A process executing on a DUT will naturally alter the values of both *arch* and $\mu$*arch*. Accordingly, the isolation of these states (to the processes they belong to) is a responsibility shared by software and hardware. A well-implemented OS (1) guards the *arch* that is only accessible via privileged mode and (2) swaps the values of *arch* before another process begins. Well-designed and secure
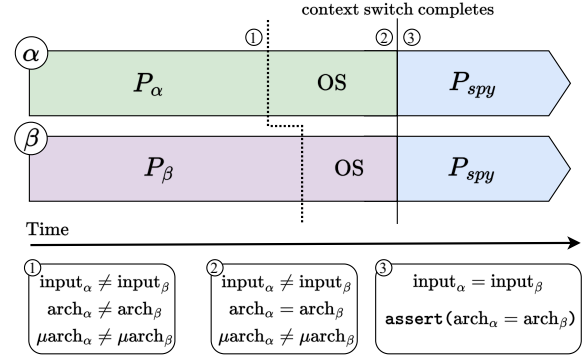


Figure 2: Overview of the AutoCC methodology. The victim processes $P_\alpha$ and $P_\beta$ are free to take on any legal execution for an arbitrary number of cycles; the inputs to both processes are symbolic. At the end of this phase ①, both *arch* and $\mu$*arch* of $\alpha$ and $\beta$ may differ. The context switch then occurs, and once it completes at ②, the *arch* of both $\alpha$ and $\beta$ are the same, but differences in $\mu$*arch* may remain. (See Fig. 3 for details of the context switch.) We assert our *arch* condition once $P_{spy}$ begins execution ③. Holding inputs to both universes equal, AutoCC checks whether differences in $\mu$*arch* after the context switch cause observable differences $P_{spy}$ execution.

hardware will either partition or flush any $\mu$arch that could leak data from one process to another. In these terms, AutoCC *assumes* the correctness of the OS and *checks* the isolation of $\mu$arch.

***Data Leakage:*** Two conditions must be met for data leakage to occur. First, the values of $\mu$arch at the beginning of the spy process are determined from the behavior of the victim process. That is, based on different values of a victim's data, there exist at least two executions of the victim process that lead to different values of $\mu$arch. Second, there exist at least two executions of the same spy program starting from the same values of *arch* that lead to different *arch*, solely because of that difference in $\mu$arch. The goal is to set up an environment where the FPV tool explores any possible execution of victim and spy processes where these conditions are met.

AutoCC achieves this by setting up two instances of the DUT—universes $\alpha$ and $\beta$—as follows (see also Fig. 2): Both universes start from an identical reset state; Each universe has its own set of input and output signals; Because each set of input signals is driven separately by the FPV tool, each universe can take on any legal execution. (Sec. 3.4 elaborates on what makes an execution legal.)

Fig. 2 also defines three events that occur during the execution of the DUT. The first event is the end of the victim process (and the beginning of the context switch), where $\alpha$ and $\beta$ can be in any reachable state after an arbitrary number of cycles. These states represent all possible executions of the victim process. Although the start of the context switch may be staggered, the end of it serves as a synchronization point between $\alpha$ and $\beta$, forcing the two universes (with hitherto different executions) into convergence. To do so, the context switch must ensure that upon completion (1) $arch_\alpha$ and $arch_\beta$ are identical, and (2) the microarchitectural flush mechanism has been executed if it exists. With these two conditions met, $\alpha$ and $\beta$ are assumed to now both be executing the same process, namely the spy process that was just switched

in. The inputs for both universes are forced equal to ensure that any observed divergence is only the result of different values of $\mu arch_\alpha$ and $\mu arch_\beta$. In this post-switch world, we assert that on every cycle, $arch_\alpha$ and $arch_\beta$ must be equal.

***What would it mean if this assertion were violated?*** A counterexample (CEX) to this assertion means that on some cycle following the switch, $\alpha$ and $\beta$ diverged in an observable way—at the resolution of a cycle—and that this discrepancy was caused by their differing executions before the switch. That is to say, there is a mechanism by which some code in the victim process can affect the execution of the spy process, i.e., a covert channel. Analyzing the CEX and determining the root of this divergence reveals how the channel is operated; we showcase how this encoding and observation occurs in Sec. 4.

***Observation Model:*** In our threat model, the spy is a software program, so for a covert channel to be exploitable, it must be observable by software. In practical terms, this implies that the program's visible state is impacted, which is why Fig. 2 displays an assertion on *arch*. However, given the variety of modern hardware designs, determining which states belong to *arch* can be unclear, and manually specifying all the relevant signals becomes tedious. We pose that as long as there exist ISA instructions that allow a process to expose *any* subset of *arch* to the DUT output interface, we can assert an equivalent correctness condition just on the DUT outputs of $\alpha$ and $\beta$ without reasoning about their internal signals. Any difference between $arch_\alpha$ and $arch_\beta$ on cycle $n$ can, by a sequence of these instructions, be externalized by the FPV tool as a difference in outputs on cycle $n + k$ for some bounded $k$. This allows the AutoCC tool to generate an FPV testbench (FT) without user input beyond providing the path to the DUT. Sec. 3.3 elaborates on how the FT is generated and how the user might need to manually specify the subset of *arch* expected to be handled by the OS.

***Modeling the OS:*** Our threat model assumes that the OS is trusted and correctly implements the context switch. Rather than reasoning about the sequence of instructions that the OS uses to switch between processes, we assume that its goal is achieved by the end of it. This is represented in Fig. 3 by showing that *arch* differences between $\alpha$ and $\beta$ and the symbolic *arch* of the spy (y-axis) are resolved by the end of the context switch. Although $\alpha$ and $\beta$ are in different symbolic *arch* and $\mu arch$ during the execution of the victim process, because we consider that the spy process begins when the *arch* is the same in both universes, the FPV tool is only interested in exploring executions of the victim process that lead to this condition. The victim process and the OS are only separated for conceptual purposes, as hinted in Fig. 2 with the dashed line. In practice, there is no bright line between the execution of the victim process and that of the OS; we are agnostic to the timing and specific instruction sequence that lead $\alpha$ and $\beta$ to the same *arch*. This may result in CEXs that present covert channels that are not exploitable under a specific OS implementation, but we argue that it is useful for a hardware designer to be aware of them. Moreover, in FPV, it is best practice not to overconstrain the model, as this can miss exploring important behavior.

***Measuring Context Switch Latency:*** For all its advantages, taking the end of the flush as the synchronization point between $\alpha$ and $\beta$ admits one blind spot, as it assumes that the flushes in
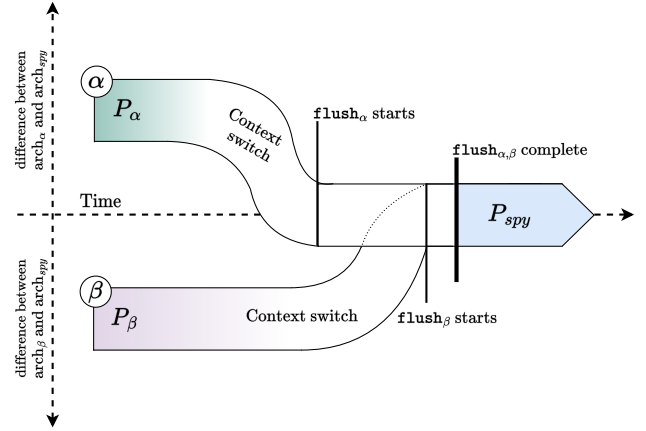


**Figure 3: AutoCC model of the context switch event. Instead of enforcing a discrete jump to a sequence of OS instructions, we simply require that the victim processes in $\alpha$ and $\beta$ eventually converge to the same *arch* (indicated here by $P_\alpha$ and $P_\beta$ converging on the $y$-axis). This is then the state of the incoming spy process. Since the microarchitectural flush is the last thing that executes before $P_{spy}$ begins, this convergence must occur by the start of the flush. Note that the flush is free to start on different cycles in $\alpha$ and $\beta$; it is only required they complete together.**

both universes finish on the same cycle. This precludes any CEXs originating from a difference in the latencies of the flush event itself. If a Trojan can modulate the flush latency and a spy can observe the difference, this latency may enable a covert channel. Nonetheless, AutoCC can further verify the DUT against this behavior by considering the start of the flush as the cycle on which $\alpha$ and $\beta$ must converge. The flush event may then be considered part of the spy process, and our existing assertions will generate a CEX for any differences between the flush event in $\alpha$ and $\beta$.

## 3.3 FPV Testbench (FT) Generation Flow

To make AutoCC accessible to hardware designers, we have developed a tool flow that requires minimal effort to set up. It creates—in under a second—a working FPV testbench (FT) from the path to the DUT and the choice of target FPV backend (Sec. 3.3.3). This FT has three major components: (1) a wrapper containing two instances of the DUT, (2) a property file that defines the properties to be checked, and (3) a backend-specific command file to invoke the FPV engines with the appropriate parameters. We implemented this FT generation flow in Python, leveraging the AutoSVA framework [44, 47] to parse the DUT interface.

### 3.3.1 Generating the DUT Wrapper.

Based on the top-level RTL module we set as the DUT (e.g., core, accelerators, or subset of them), the flow generates an FT in 3 steps.

First, the flow parses the interface signals of the DUT to create the wrapper's interface. The input and output signals of the wrapper are two sets of the DUT signals, each with a unique suffix (e.g., $\alpha$ and $\beta$), except for the signals we do not want to replicate, such as the clock and reset signals.

Second, the flow instantiates the DUT twice—as submodules of the wrapper—with different names, i.e., $u\alpha$ and $u\beta$.

Third, it connects each set of the independent, duplicated interface signals to the corresponding submodule and the common, non-duplicated signals to both submodules. If users want other interface signals of the DUT not to be replicated (e.g., a debug interface), they can specify them via a Verilog comment (//AutoCC Common) above each signal. This is equivalent to assuming that an input signal is equal throughout the entire execution, which may be useful to deal with illegal inputs, as we elaborate in Sec. 3.4. Making a signal common to $\alpha$ and $\beta$ helps improve the FPV tool runtime at the cost of not searching the space state derived from that signal being different in both universes.

### 3.3.2  Generating the AutoCC Property file.

```
localparam THRESHOLD = 4;
//eq_cnt counts the number of consecutive cycles the transfer
    condition holds since the flush finished
reg [$clog2(THRESHOLD):0] eq_cnt;
wire transfer_cond;
reg spy_mode; //Set when the eq_cnt reaches THRESHOLD
wire spy_starts = transfer_cond && eq_cnt>=THRESHOLD;
wire flush_done = 'x; //Set free by default (anytime) USER may set
    the conditions that indicate the flush has finished for both
    universes.
always_ff @(posedge clk)
    if (reset) begin
        spy_mode   <= '0;
        eq_cnt    <= '0;
    end else begin
        spy_mode <= spy_starts || spy_mode;
        eq_cnt <= (flush_done || eq_cnt>0) &&
      transfer_cond ? eq_cnt + 1 : '0;
    end
// There is an assumption per input signal to the DUT
wire input1_eq = ua.input1 == ub.input1;
assume property (spy_mode |-> input1_eq);
// There is an assertion per output signal of the DUT
wire output1_eq = ua.output1 == ub.output1;
assert property (spy_mode |-> output1_eq);
//If some output signals are grouped by a transaction with a valid
     signal, then the assertion for the payload has the valid
     signal as a precondition
wire out_transact_valid_eq = ua.out_transact.valid == ub.
    out_transact.valid;
assert property (spy_mode |-> out_transact_valid_eq);
wire out_transact_pld_eq = !ua.out_transact.valid || ua.
    out_transact.payload==ub.out_transact.payload;
assert property (spy_mode |-> out_transact_pld_eq);

wire architectural_state_eq = 1'b1; // The USER includes
    conditions here based on the architectural state of the DUT
// Conditions to be met before starting spy_mode
assign transfer_cond = architectural_state_eq && input_signal_eq
    && output_signal_eq && out_transact_valid_eq &&
    out_transact_pld_eq;
```

**Listing 1: Property file created generated by the AutoCC tool. It uses the signal that indicates that $\mu$arch flush has finished in both universes, to start the equality condition that defines the transfer period. After the transfer period is done, the spy process begins, i,e, inputs are assumed equal in both universes, and outputs are checked.**

Listing 1 shows the template of the property file generated by AutoCC. Users are not required to provide a priori information about the internals of the DUT, as the properties generated solely use interface signals. Properties are written in SystemVerilog Assertions language (SVA) [27]. Assumptions are generated for DUT inputs and assertions for DUT outputs.

***Transactions:*** When a valid signal governs a group of signals, we name it a transaction. We use this valid signal as a precondition for the properties reasoning about the payload of the transaction. This means that we do not check whether the payload of an outgoing transaction (from the DUT perspective) changes values while the transaction is not valid. However, if the RTL module to which the DUT is outputting wrongly uses an invalid payload, this would be detected by AutoCC when applied to this incorrect module since the input payloads are only assumed equal when the input transaction is valid. This careful management of interface transactions is crucial when verifying a large design via modularity (Sec. 3.4). We reuse AutoSVA's method to identify transactions automatically [47].

***Defining the Architecture and Flush Conditions:*** By default, AutoCC does not identify the $\mu$arch flush event or the set of *arch* signals. Users can modify these signals depending on the DUT to determine when a flush is considered finished and which state elements belong to *arch*. As we showcase in the evaluation section, we recommend adding states to the `architectural_state_eq` condition as CEXs are found to avoid overconstraining in advance. However, states that are clearly architectural because the OS manages them, e.g., the register file, may be added upfront.

***Flush Completion:*** The flush event can be tricky to nail down as some DUTs do not have a well-defined signal for when the flush completes, and some do not have a flush operation at all. For instance, certain accelerators are designed under the assumption that when a new process begins utilizing the accelerator, there are no ongoing operations within its pipeline. That is to say, each stage of the pipeline must be idle when a new process begins; for these DUTs, flush completion can simply be defined as an idle pipeline.

***Transfer Period:*** This concept is introduced to ease the definition of the flush completion on DUTs that have neither a flush nor an idle signal. The condition defining the transfer period is that for some cycles after the flush has finished, both *arch* and the interface signals are identical for $\alpha$ and $\beta$, giving time for the pipeline stages in both universes to converge. As shown in Listing 1, the length of this transfer period is configurable via the THRESHOLD parameter. In theory, a transfer period of $n$ cycles would eliminate CEXs that could only exercise within the first $n$ cycles of the new process. In practice, as long as $n$ remains smaller than the length of the OS operations between the flush completion and the transference of control to the spy process, these CEXs would not correspond to exploitable covert channels. As a heuristic, the length of the transfer period may be set to the longest path through the pipeline.

***Spy Mode:*** The properties in Listing 1 only apply when the spy process is executing and the transfer period has elapsed (spy_mode is asserted). Until then, the inputs to both universes are free to be different, and the outputs are not checked.

### 3.3.3  AutoCC's FPV Backend Support.

The adoption of formal methods is frequently hindered by the access to FPV engines, as the need for training to use them effectively. To ease their usage, our tool also generates the backend-specific commands and binding files required to use FPV engines—based on their documentation [8, 65]. We have tested AutoCC with two different backends: JasperGold [7] and YosysHQ's SBY [65, 67]. Once the properties and bindings are generated, our tool invokes the backend to start the property-checking process. Our methodology

only uses single-cycle properties, which are efficient for FPV engines to verify and are supported by the open-source part of SBY. Thus, our tool is potentially amenable to an end-to-end open-source tool flow via SBY when applied to Verilog projects.

## 3.4 Reducing the State Space via Modularity

Covert channels can potentially be exploited from any state that a victim touches. Thus, AutoCC should be applied to all the RTL modules impacted by that software process. Proving the assertions of Listing 1—or achieving a deep-enough bounded proof—is often infeasible for SoC designs of realistic size.

The space state exploration in FPV (and thus backend tool runtime) grows exponentially with the RTL size and the search depth (time in cycles). As a baseline mitigation, we adopt the standard technique of minimizing the size of parameterized modules, such as TLBs, caches, etc [55]. Provided that the downsized module is still able to exercise all the relevant features, this technique would not affect the coverage of evaluation. However, this technique is often not enough to achieve a sufficiently deep bounded proof to provide confidence in the correctness of the design. To that end, we adopt two techniques: blackboxing and modularity. (Since blackboxing is a form of modularity, we discuss them together.)

The implications of both techniques are very similar, but they differ in the location of the abstracted module. Blackboxing means that a submodule of the DUT is abstracted away from the verification engine, while modularity means that we create a new FT where the DUT is a submodule of the former top module. In practice, blackboxing can be thought of as if the submodule was moved outside the DUT while the wires that connect it to the DUT are left intact. These wires now become part of the DUT interface and are subject to the same constraints as the other DUT inputs and outputs, i.e., upon entering the spy mode, the wires that output the DUT (and input the blackboxed module) are checked to be equal in $\alpha$ and $\beta$, while the inputs to the DUT are assumed equal.

To the FPV engine, the internals of a blackboxed module do not exist; it does not follow any state evolution. Thus, a module should only be blackboxed if the user does not care about any leaks originating from within it. (This could be because the OS is assumed to flush the module's state or the module has already been verified.)

***Advantages:*** First, since the DUT contains less state, the combinatorial search size is reduced exponentially. Second, the exploration depth required to exercise the relevant features of the DUT is reduced since the FPV tool is driving the inputs of the DUT directly.

***Disadvantages:*** The CEXs found are less informative since we do not know how the inputs of the DUT were produced. For blackboxing, this refers to the outputs of the blackboxed module, which drive the rest of the logic within the DUT. Moreover, the CEXs are more likely to be spurious since inputs to the DUT may be illegal.

**Definition 4** (Illegal Input Sequence). *An input sequence to the DUT is considered illegal if it is unreachable when the DUT is instantiated within the full SoC (driving the DUT inputs).*

Based on the above definition, the user could create assumptions to limit the inputs to legal values, e.g., do not receive a memory response if a request was not sent. A hardware designer may decide not to include these assumptions in its RTL module if the rest of the SoC is untrusted (e.g., resulting from integrating third-party

---

**Algorithm 1:** Incremental Flush Signal Construction

$Flush \leftarrow \emptyset$;
$result \leftarrow$ FPV(*DUT, Flush, AutoCC_FT*);
**while** *(result == CEX)* **do**
    $state \leftarrow$ FindCause(*result*);
    Insert(*Flush, state*); // Add to the Flush process
    $result \leftarrow$ FPV(*DUT, Flush, AutoCC_FT*);

---

**Algorithm 2:** Decremental Flush Signal Construction

$Candidates \subseteq \mu arch$;
$Flush \leftarrow \mu arch$;
**for** *(state in Candidates)* **do**
    Remove(*Flush, state*);
    $result \leftarrow$ FPV(*DUT, Flush, AutoCC_FT*);
    **if** *(result != Proof)* **then**
        Insert(*Flush, state*);

---

IP). Alternatively, one may add individual assumptions to the FT to limit the inputs to legal values. To ease the modeling of DUT's outgoing transactions, our tool flow can also generate that from AutoSVA annotations [44]. However, we argue that in FPV, it is good practice to add assumptions and modeling upon encountering spurious CEXs, as it is a good way to learn about the design and avoid overconstraining the verification process.

***SoC-level Verification:*** To apply AutoCC at the SoC level, we recommend first creating FTs for RTL modules with the simplest interfaces, e.g., modules connected to the network-on-chip (NoC). This makes it much easier to deal with illegal inputs, as the NoC protocol is usually well-defined. Our properties in Listing 1 are designed to be modular so that RTL modules can be independently verified for the absence of covert channels. However, modularity results in more effort, not because of creating the FTs (which is automated in AutoCC), but because the DUT inputs are arbitrarily driven by the FPV tool, making the CEXs more prone to be spurious.

## 3.5 AutoCC during RTL Development

Listing 1 properties are expressed using interface signals, making them implementation-independent. This, along with their modular nature, allows designers to utilize AutoCC properties for test-driven development (TDD), where CEXs help to refine the design [56].

TDD is particularly useful for designing the $\mu arch$ flush mechanism. The overall flush mechanism would be correct if every module involved in the victim process effectively flushes exploitable $\mu arch$ and the orchestration of the flush signals across modules is properly implemented. We propose two methods that use AutoCC to identify the minimal set of $\mu arch$ states that need to be flushed to provide full temporal partitioning (i.e., no observable differences).

Algorithm 1 incrementally builds the flush mechanism by adding flushes to the states that cause CEXs to AutoCC properties.

Algorithm 2 starts with the assumptions that the entire $\mu arch$ is being flushed and AutoCC properties achieve a proof. Then it iteratively takes a state from the set of candidates and removes it

from the flush signal as long as proof is still achieved. The candidate set is a subset of flush since there may not be an incentive to remove a state flush if it does not impact performance. Both approaches assume that FPV returns in a finite amount of time, and the user is responsible for determining when a bounded proof yields confidence.

## 4 EVALUATION AND RESULTS

This section presents our evaluation of AutoCC on four open-source projects: 32-bit RISC-V Vscale core [38]; application-class 64-bit CVA6 core [43, 68]; MAPLE memory access engine [45, 46], and an accelerator for AES encryption [38]. We chose these projects because they represent a diverse set of designs in terms of complexity and pipeline depth. Table 1 lists the valuable CEXs we found. We consider a CEX valuable if it uncovers (a) a behavioral difference in the execution of a spy process based on the state left by a victim process or (b) unexpected or unintended behavior in the RTL based on legal execution. Alternatively, a spurious CEX is caused by an illegal input sequence (see Definition 3).

**Table 1: Description, DUT execution depth, and FPV tool runtime (in minutes and hours) of the CEXs found in Vscale (V), CVA6 (C), MAPLE (M), and AES (A) that uncover hardware bugs or possible covert channels.**

| Description | Depth | Time |
|---|---|---|
| **V5.** Interrupt in the WB stage stalls pipeline | 9 | < 10 min. |
| **C1.** Leaks invalid I-Cache data to the next PC | 76 | < 30 min. |
| **C2.** Wrong transition in the FSM of the PTW | 80 | < 6h |
| **C3.** Valid D$ line after flush caused by PTW | 80 | < 6h |
| **M2.** Leak whether the TLB was disabled | 21 | < 30 min. |
| **M3.** Leak the value of a configuration register | 23 | < 3h |
| **A1.** Request in the pipeline during the switch | 42 | < 1 min. |

Table 1 also shows the depth of the CEX (length of the execution trace) and the runtime of the FPV tool. Although we have validated the AutoCC methodology with both SBY and JasperGold, we chose to perform evaluations with the latter due to familiarity with its GUI and because we are also evaluating SystemVerilog projects.

During the rest of the section, we walk the reader through the steps of applying AutoCC to the RTL projects listed above, including generating the FTs, refining the architectural state signal upon CEXs, and finding the CEXs indicated in Table 1. In the case of CVA6 and MAPLE, we (a) found hardware bugs and exploitable covert channels and reproduced a leak in system-level RTL simulation, (b) fixed these bugs and vulnerabilities in RTL and re-ran AutoCC to confirm that the CEXs were no longer found, and (c) merged these fixes into the upstream repositories of these open-source projects.

### 4.1 The 32-bit Vscale RISC-V core

**Step-by-step use-case.** Because Vscale is the first DUT presented, we will walk the reader (as a potential user) through how we applied the AutoCC methodology to it (see specific commands on Sec. A.5).

First, we create the FT by running the AutoCC python script indicating the path to the top-level module of Vscale (`vscale_core.v`). Second, we start the exhaustive exploration by running JasperGold and indicating the path to the generated FT. Note that this first run uses the default values for the flush and architectural state signals (see Listing 1). The CEXs shown in Table 2 result from iteratively refining the definition of the architectural state.

**Table 2: Description, depth, and FPV tool runtime (in seconds) of every CEX found in our experiments with Vscale starting from the default AutoCC FT, in order.**

| Description | Depth | Time |
|---|---|---|
| **V1.** Jump to address read from the reg. file | 6 | <10 sec. |
| **V2.** Jump to address read from CSR | 6 | < 10 sec. |
| **V3.** PC different throughout the pipeline | 7 | < 10 sec. |
| **V4.** Decode Stage registers different | 7 | < 10 sec. |
| **V5.** Interrupt in the WB stage stalls pipeline | 9 | < 100 sec. |

*V1.* The first CEX we observed was caused by a jump to an address in a register. Recall that the default assertions in the FT only check whether the output interfaces of the DUT are equal. Thus, the formal engine searches for an execution path to expose different internal states at the output interfaces. We refined that CEX by adding a condition to `architectural_state_eq` to check that `pipeline.regfile.data` is equal in both instances of the Vscale core. We could have added this condition from the beginning, but we chose to add them as we were finding CEXs for three reasons: (1) because we had not looked inside the core's internal state before, and so the CEX helped us find the path to each signal name; (2) to validate that the methodology can find covert channels based on an unflushed state; and (3) because it is good practice to start with the simplest precondition possible to make sure we do not overconstrain the state exploration.

*V2.* The second CEX was caused by a jump to a register previously fetched from the CSR module. The OS is responsible for protecting and managing the CSR registers, so these should be considered part of the architectural state. Since the CSR module contains many registers, it was more convenient to blackbox it and follow the procedure described in Sec. 3.4.

*V3.* The third CEX was caused by the PC being different in both universes, causing the next instruction fetch to have a different address. We refine this CEX by adding the PC registers along the core's pipeline to the architectural state.

*V4 & V5.* The fourth and fifth CEXs are caused by the fact that the Vscale core does not have a temporal fence like the version we used for CVA6 [43]. Particularly, our fifth CEX of Table 2 showed a case where an interrupting instruction in the write-back stage of $\alpha$—from the execution before the context switch—was causing stalls in the fetch stage of the pipeline for the spy process. However, since the OS code that manages the context switch has more instructions than pipeline stages of Vscale, it seems reasonable to consider that all instructions inside the pipeline should be equal in both universes when the spy process is about to start. For this

evaluation, we assume a trusted and correct OS. Nonetheless, if an AutoCC user prefers not to assume that, this CEX could constitute a covert channel in that threat model.

**Bounded proof.** After refining the last CEX, the FPV engine kept searching until it reached our limit of 24 hours. At that moment, it had reached a bounded proof of depth 21. Since Vscale does not have caches or deep units, and the previous CEX had depth 9, we believe it would not find more CEXs even if it ran longer.

## 4.2 The 64-bit CVA6 RISC-V core

CVA6 is a mature application-class RISC-V core, fully implementing I, M, A, F, D, and C extensions (ISA v2.3) and three privilege levels (M, S, U). CVA6 has been taped out numerous times into silicon [12, 15, 34, 69] and offers several cache, MMU, and core configurations, including 32-bit and 64-bit variants.

**Configurations.** We used the 64-bit one with all the extensions, defined by their `cv64a6_imafdc_sv39_config_pkg` configuration file. However, we shrank the size of caches (16 lines), TLB (4 lines), and branch predictor table (16 entries) to reduce the state size while still exercising their functionality. Leveraging the modularity of AutoCC, we disabled the floating-point unit to lighten the FPV process, as this IP block could be evaluated separately. There are three adaptations of CVA6 that implement different versions of the `fence.t` instruction—a *µarch* temporal partitioning mechanism—with increasing levels of flush exhaustiveness [63].

**Validating previously-found covert-channels.** Our work began with the second implementation—*full flush*—which clears the caches, TLBs, branch predictors, and other states in smaller units, such as arbiters. We set the *flush_done* condition as the `fence.t` has completed in both universes, i.e. when the write-back data cache (D$) has invalidated its lines. One of the first CEXs we found (after we added the PC, register file, and CSR into the *arch* signal) was caused by executions where $\alpha$ had an outstanding AXI (Advanced eXtensible Interface) request going into the flush while $\beta$ did not. Since the arrival of the flush signal kills all outstanding AXI transactions, $\alpha$'s instruction cache (I$), which was making the request, transitioned to a `KILL_MISS` state while $\beta$'s remained in `IDLE`. This divergence of *µarch* can lead to an observable timing difference after the flush event, for instance, by issuing another cache request. A natural solution is to stipulate that the flush must first wait for all outstanding AXI requests to be completed. We still found another CEX after assuming that all AXI requests are satisfied before the flush. In this new CEX, the page table walker (PTW) takes longer to flush in $\alpha$ because it had an active memory request to the D$. These CEXs confirm and extend prior findings about *full flush* `fence.t` in Wistoff et al. [63]. The observation that subtle, hard-to-find components may produce a covert channel (when not cleared systematically) was their primary motivation for the third implementation of CVA6's *µarch* flush: *microreset*.

**Evaluating the safest configuration.** Unlike the full flush, microreset targets the entire *µarch* rather than attempting to identify a subset of vulnerabilities (only *arch* is left unflushed). Microreset also enforces the `fence.t` latency be independent of any previous execution, padding it to the worst-case: the latency of a full D$ write-back. Flushing all *µarch* and padding to a constant latency is the most thorough temporal partition a designer can do against

covert channels in hardware, so we were not expecting to find any relevant CEXs; however, we found three, presented below.

**C1.** First, we found a CEX where an I$ fetch results in an exception in both $\alpha$ and $\beta$. Since the exception is a valid response for this transaction, `icache_dreq_i.valid` is asserted even though the fetch did not hit the I$. In the frontend, CVA6 loads `icache_data` with whatever data payload it receives from the I$, as long as the response is valid. This payload is an input into the instruction realigner; the crux of the CEX is that the realigner sets its valid signal (for the output back to the pipeline) based on a bit of this payload without knowing that the payload came from an invalid I$ line. The difference in the output of the realigner then results in a PC mismatch in $\alpha$ and $\beta$. We tentatively fixed this to continue exploring by zeroing out the data payload if we do not hit in the I$.

**C2.** Second, we faced a CEX caused by an invalid FSM transition in the PTW. This CEX begins with a TLB miss in both $\alpha$ and $\beta$, resulting in both universes going on a page table walk; the flush signal from `fence.t` arrives while the walk is ongoing. The FSM logic for the PTW dictates that if the PTW looks up a page table entry (PTE) when `flush` gets set, it should wait for a response before going to IDLE. (The intended transition is `PTE_LOOKUP` to `WAIT_RVALID`, then `WAIT_RVALID` to `IDLE` on receiving a valid response.) This is exactly what $\alpha$ does. However, while $\beta$ is in `WAIT_RVALID`, $\beta$ also handles an exception, causing `flush` to get set again. As a result, $\beta$'s FSM transitions to `IDLE` on the next cycle, terminating the page walk before it gets a response. We reached out to the CVA6 maintainers to discuss this corner case and proposed a fix, which has been merged upstream. [1] This CEX showcases that AutoCC not only finds potential covert channels but also errors in the design.

**C3.** Third, we hit a CEX where $\alpha$ observes a chain of events involving the I$, TLB, PTW, and D$. Initially, the I$ experiences a miss, whose memory translation also results in a TLB miss. Subsequently, the PTW starts fetching PTEs, which results in a D$ request, right when the flush signal arrives. Although the TLB and PTW eventually get flushed, the D$ ends with a valid line after the flush completes. This CEX shows that a sequence of events initiated before the flush leads to an effect observable after the flush ends, constituting a potential covert channel. Based on this CEX, we find that draining D$ transactions after writing back the D$ and before clearing the design's flip-flops is insufficient; D$ transactions need to be drained before *and* after the write-back. We have made a corresponding fix for *microreset*. [2]

## 4.3 The MAPLE Memory-Access Engine

MAPLE is an accelerator for fetching memory patterns that supports fetching single array elements, array ranges, and indirect memory accesses. It also contains a memory-management unit (MMU) for virtual memory translation. In addition to `load` and `consume` operations, the API offered by MAPLE exposes several registers to configure the hardware queues and the MMU. Particularly, the API offers a `init` operation to allocate a MAPLE instance (by mapping its memory-mapped configuration registers into virtual memory), a `close` operation to de-allocate the instance, and a `cleanup` operation to invalidate these configurations and flush the

---

[1] https://github.com/openhwgroup/cva6/pull/1184
[2] https://github.com/pulp-platform/cva6/commit/ae79ec5

TLB between processes. The cleanup operation is performed as a first step of the initialization process.

***Flush mechanism.*** We used the FSM that controls the invalidation process to set up the flush signal—when the invalidation state transitions to idle. Although MAPLE queues could be considered architecturally visible, these are flushed by the cleanup operation, so we did not add them in the architectural state condition.

***M1.*** The first CEX we quickly found was caused by several other requests being in the NoC protocol's output buffer in $\alpha$ when the flush signal was set. Although this could potentially yield a covert channel under special timing conditions (an old request being backpressured from the NoC), we chose to continue exploring CEXs by assuming that this buffer is empty during the context switch.

***M2.*** The next CEX was caused by the TLB in $\alpha$ being disabled while the TLB in $\beta$ was enabled. The TLB is enabled by default at reset, but MAPLE's API allows disabling it. We found from the CEX trace that the flip-flop of TLB being enabled is not flushed during the context switch. This flip-flop could be used as a binary covert channel, provided that the Trojan could disable the TLB and the spy observe a page fault. We fixed this in MAPLE's RTL by resetting this flip-flop during the flush.

***M3.*** The third CEX, found after a couple of hours, was caused by another register not being flushed. This one is the base address of the array for which subsequent data fetches can be offloaded to MAPLE by indicating an array index. To better describe this covert channel and how to exploit it in practice, we recreate a data leak with a test written in C.

```
void leak(int iteration){ // Trojan inside victim's process
    int qid = dec_init();
    uint16 leak_byte = (secret >> (iteration*8)) & 0x00FF;
    uint16 offset = leak_byte << 2; // 4-byte aligned
    dec_set_array_base(qid, VADDR + offset);
    dec_close(qid);
}
// The spy process has an 256-element array allocated using mmap()
    to start at VADDR. The array contains consecutive elements
    from 0 to 255.
void observe(int iteration){ // Inside Spy Process
    int qid = dec_init();
    dec_open_producer(qid);
    dec_open_consumer(qid);
    // Tells MAPLE to fetch the 0th array element starting from
     the configured base address, i.e, array[leak_byte]
    dec_load_word_async(qid,0);
    // Consume array value from MAPLE's queue,
    uint32 spy_byte = dec_consume_word(qid);
    recovered = recovered | (spy_byte << (iteration*8));
    dec_close(qid);
}
```

**Listing 2: Pseudocode of the program that lets a spy process recover the secret that a Trojan is actively leaking. MAPLE has a function (dec_set_array_base) that sets the base address of an array so that subsequent loads from it are offloaded to MAPLE by simply indicating the array index to load (dec_load_word_async). Since AutoCC found that this base address is not properly flushed, we can use it to leak the secret. The secret is leaked a byte at a time, by using it as an offset to set the base address of the array. Since the spy has allocated an array where array[index]==index, this offset is inferred from the loaded value.**

***Exploiting M3 at system-level.*** Listing 2 shows the leak function that allows a Trojan to encode a byte of the secret per iteration

and the observe function that allows the spy to recover it. To evaluate this test[3], we first built an RTL simulation environment of MAPLE integrated with the OpenPiton SoC [4] following the tutorial in the MAPLE repository. Then, we performed the test bare-metal using VCS O-2018.09-SP2. It took under a minute for VCS to simulate the test on the OpenPiton SoC with MAPLE, where the spy recovers 8 bits per iteration, e.g., a 32-bit secret could be recovered with 4 iterations in less than 6,000 clock cycles.

***Closing the covert channels.*** We have merged the RTL fixes to close M2[4] and M3[5] covert channels into the upstream repository of MAPLE. For fabricated chips that include MAPLE [15], these channels could be closed in software by writing these registers explicitly to the reset value during the invalidation process.

## 4.4  An AES Accelerator

The AES accelerator we evaluated takes a 128-bit plain text and a 128-bit key as input and produces a 128-bit cipher text as output. It is a pipelined accelerator with 40 stages. We applied our methodology by following the same steps as in the previous section. We first ran the default FT generated by AutoCC, without specifying the flush signal. This accelerator does not contain any architecturally visible state but rather follows a request-response protocol.

***A1.*** We found a CEX at depth 42 in a few seconds; universe $\alpha$ contained several ongoing requests, while $\beta$ had none. Since the flush signal (set free) appeared while the accelerator pipeline in $\alpha$ was processing requests, a timing difference appears when $\alpha$ eventually responds, and $\beta$ does not.

***Using accelerators concurrently.*** The design of this AES accelerator assumes that it will only be used by one process at a time, as it does not offer any invalidate or flush signals. This would work well in a scenario where another process cannot use the accelerator until all the requests have been responded to. This is a reasonable assumption in the context of a well-programmed allocation of system resources. Hence, we refined this CEX by defining the flush signal as the condition of both universes having no ongoing requests. Once this was added, the tool found **full proof** in 5 hours.

***Heterogeneous SoCs may lead to subtle vulnerabilities.*** In the era of heterogeneous hardware, system designers have to be very careful when integrating third-party IP blocks, as they might not be aware of the assumptions made by other designers. Otherwise, integrating an IP block similar to this AES accelerator (without hardware invalidation mechanism) in a system that does not assume the OS to shield the allocation of hardware resources (e.g., waiting for all ongoing requests) may enable a covert channel.

## 5  DISCUSSION: HW/SW PROTECTIONS

We understand that security is not the task of hardware alone. Designers often have to make trade-offs between PPA[6] and security; by identifying covert channels, our methodology helps them make informed decisions by knowing which hardware blocks, features, or optimizations may cause data leakage. Our approach also provides concise traces of the execution that led to a particular state and how that state led to an observable difference in the spy process.

---

[3]github.com/PrincetonUniversity/maple/blob/main/tests/autocc.c

[4]github.com/PrincetonUniversity/maple/commit/fa614fc

[5]github.com/PrincetonUniversity/maple/commit/04a54d5

[6]Performance, power, and area. These are key metrics of a hardware design.

***Tradeoffs:*** With this knowledge, a hardware vendor can better decide whether to close the covert channel in RTL or warn against it on its security specification.[7] For example, if a hardware-based division operation is found to be susceptible to a covert channel and fixing it would significantly slow down the operation for non-security-critical applications, the hardware vendor may decide not to fix it but flag it, so that programmers prioritizing security avoid using divisions on sensitive data. However, addressing the channel in hardware may be worth it if it has a minor impact on PPA. This is the case for the covert channels found in this paper, where enhancing the existing flush mechanism fixed them with negligible PPA implications. The hard part of fixing these channels was knowing about their existence, which is what AutoCC provided.

***The Cost of Flushing Microarchitectural State:*** Although analyzing the PPA impact of flushing $\mu arch$ is out of the scope of this paper, we can make some observations. Flushing $\mu arch$ may affect runtime in two ways: (1) the time it takes to flush the state, and (2) the time it takes to restore the state after the flush. The first one is impacted by the unit that takes the longest to flush; much of the state can be flushed in a single cycle, but some units may take longer (e.g., write-back caches). On the second one, the concern is the performance loss due to the unavailable state after the context switch, e.g., more misses may occur because the cache is flushed, or the branch predictor might need to relearn the branch history. Prior work found that this impact mostly depends on the period between context switches and the size of these structures [63]. For example, since on-core caches are small (typically much smaller than the program working set [16]), the lines interesting for the second process are likely evicted by the cache replacement policy anyway, and so there is no performance impact due to the flush.

We regard the problem of preventing covert channels as a challenge in ***hardware-software co-design***. Hardware must provide the means to partition shared resources so that an OS can use these as necessary when reallocating those resources from one security domain to another. To that end, AutoCC can assist in designing and verifying temporal partitioning mechanisms for RTL modules.

## 6 FURTHER RELATED WORK

Information flow tracking (IFT) monitors the flow of sensitive data through hardware components via RTL simulation [3, 42, 52, 53]. Like AutoCC, IFT techniques provide a precise trace of the leakage; however, they rely on input tests and user-provided security properties. Prior works in IFT are in part orthogonal to AutoCC since they focus on SoC-level simulation while AutoCC formally verifies hardware components—potentially early in the design phase.

Other works in the area of information flow security propose new hardware description languages that integrate aspects of type systems to prevent illegal information flows. Caisson [36] statically analyzes designs written in its language to guarantee noninterference. Sapper [35] offers the same static guarantee by automatically inserting runtime checks into a Verilog design. SecVerilog [71] extends Verilog with a label-based type system to allow for dynamic labels that depend on runtime values. All of these approaches must be applied end-to-end on the entire design and require significant

modification and annotation of existing RTL. This, in turn, requires reasoning about design internals and their security properties.

Like AutoCC, Simarel [31] uses bounded model checking to verify relational invariants between core executions. They focus on inductive invariants to prove information isolation. However, Simarel generally reasons about flows between levels in a security lattice, and no testing occurs against a formalized context switch.

While prior work is effective at tracking hardware state being read and propagated, they do not directly consider how timing in the program execution may also be used to extract information.

## 7 CONCLUSION

Our work introduces an FPV-based methodology that, given an RTL module, exhaustively searches for execution traces of a victim process that lead to execution differences observable by a supposedly isolated spy process. We demonstrated the effectiveness and efficiency of this methodology by applying it to four open-source hardware components. Particularly, ***we found that*** AutoCC: (1) exercises previously-known issues within minutes, compared to lengthy stress-test simulations or emulations; (2) helps find the root cause of a CEX with minimal engineering effort due to the short length of the execution trace; (3) exposes new hardware bugs and covert channels in the mature RISC-V CVA6 core and the MAPLE accelerator; (4) uncovers experimentally-viable covert channels as we reproduced one via system-level RTL simulation; (5) validates that RTL fixes to close covert channels are effective.

***Users:*** AutoCC holds much value for hardware designers, empowering them to systematically search for covert channels in RTL during or after development. We believe AutoCC is most useful for developers of RTL modules or for those integrating third-party modules into a larger system. To make AutoCC accessible and practical for our potential users, we have: (a) developed an automated flow to generate FPV testbenches implementing this methodology, eliminating the need for upfront user input or RTL details; (b) proposed a test-driven approach to assist the design of hardware that requires temporal isolation, i.e., flushing the $\mu arch$ state between processes; (c) open-sourced AutoCC and added its artifact evaluation to showcase how to apply AutoCC to more RTL modules.

---

[7]This specification informs programmers about which hardware features may leak data so that they avoid using them if that goes against their security goals.

---

# A ARTIFACT APPENDIX

## A.1 Abstract

This artifact applies the AutoCC methodology to each of the hardware components evaluated in this paper: the 32-bit RISC-V Vscale core, the 64-bit application-class RISC-V CVA6 core, the MAPLE memory-access engine, and the 128-bit AES encryption accelerator. The AutoCC methodology employs formal property verification (FPV) to exhaustively examine the state of hardware components to determine whether it may expose a covert channel,i.e., FPV engines would trigger counterexamples (CEXs) to the AutoCC assertions if there is any hardware state (left unflushed after a context switch) that leads to an execution difference observable from the output of the component.

This artifact evaluation performs three types of tasks: (a) given an RTL component, AutoCC generates a FPV Testbench (FT); (b) feeding an FT into JasperGold to obtain CEXs to the properties generated by AutoCC; (c) reproducing a covert channel that a CEX uncovered at system-level simulation (not as a standalone hardware component).

## A.2 Artifact check-list (meta-information)

- **Data set:** The four RTL components we evaluate in this paper serve as the data set. This encompasses the open-source projects of Vscale, CVA6, and MAPLE, and a 128-bit AES accelerator. Moreover, the OpenPiton repository is used to evaluate MAPLE at SoC-level. These can be accessed in GitHub at: $LGTMCU/vscale$, $morenes/cva6$, $PrincetonUniversity/maple$, $morenes/aes$, $PrincetonUniversity/openpiton$.
- **Run-time environment:** Running the FTs generated by AutoCC requires Cadence's JasperGold tool (JG). Reproducing the covert channel found with AutoCC requires Synopsys' VCS simulator.
- **Experiments:** There are four use cases, described in Section A.5, which are independent and can be evaluated in parallel.
- **Output:** Given the Vscale core as input, AutoCC will generate an FT for it. This and the FTs of the other components (provided in the AutoCC github) are fed into JG to obtain some of the CEXs shown in Tables 2 and 1. The system-level RTL simulation of OpenPiton+MAPLE would output the 32-bit word being transmitted using the covert-channel uncovered in this paper using AutoCC.
- **How much disk space required (approximately)?:** 2.5GB.
- **How much time is needed to prepare workflow (approximately)?:** Less than 1h.
- **How much time is needed to complete experiments (approximately)?:** The longest runs take 6h. The use cases are independent and can be performed in parallel in four terminals. Note that depending on the server that JG is running this may affect execution times.

## A.3 Description

### A.3.1 How to access.

This artifact can be accessed from *github.com/morenes/AutoCC*. The repository contains a README with detailed instructions for installing AutoCC and reproducing our results, which we also specify in this appendix.

### A.3.2 Hardware dependencies.

This artifact does not have any specific hardware dependencies. However, we recommend running on a machine with at least 16 cores to see similar runtimes as the ones we report in the paper.

### A.3.3 Software dependencies.

In addition to the source code of AutoCC and the projects to be tested, this' artifact evaluation requires:

- Cadence's JasperGold (JG), to obtain the CEXs to AutoCC assertions. We have performed our evaluation with version 2021.03, and we have checked that it also works with version 2019.12. Other versions would probably work too.
- Synopsys' VCS Simulator, to reproduce the covert channel on MAPLE at system-level.

## A.4 Installation

Make sure to use *bash* throughout the installation and evaluation process. Let's start by cloning the AutoCC repository:

```bash
git clone https://github.com/morenes/AutoCC.git;
git checkout v1.0;  # The release Tag for this artifact
cd AutoCC;
export AUTOCC_ROOT=$PWD;
```

Point to the JG binary:

```
which jg;
alias jg='<LIC_PATH>/jasper_2021.03/bin/jg';
# Or the version that you are using
```

## A.5 Experiment workflow and expected results

### A.5.1 Vscale: Generating FT and fixing constraints.

**Build.** Clone the Vscale repo and fix a combinational loop in the original RTL that prevents JG from running:

```
cd $AUTOCC_ROOT
git clone https://github.com/LGTMCU/vscale.git
export DUT_ROOT=$PWD/vscale/src/main/verilog;
./fixes/fix_combo_loop_vscale_rtl.sh
```

Generate the Vscale formal testbench using AutoCC.

```
python3 autocc.py -f vscale_core.v  -i
    vscale_ctrl_constants.vh;
```

Run JG on the generated testbench:

```
jg ft_vscale_core/FPV.tcl -proj projs/vscale_init &
```

**CEX V1.** The tool should find a CEX (of at least 6 cycles) to the assertion as__dmem_hwrite in a second of computation time.

**Waveform V1.** Clicking on the assertion in the GUI opens a waveform window. To visualize the CEX, we add a list of signals to the waveform window. We can use the signal list in the file vscale.sig. To load the signal list, go to File → Load Signal List, and select vscale.sig from the sigs folder. In the waveform, we would see spy_mode starting in cycle 5. Then, hwrite signal is different in the last cycle because the opcode was different a cycle before (ctrl.opcode). This is because the PC is different (PC_IF), since the branch was taken in one universe and not in the other because the register file data was different (regfile.data).

**Fix V1.** As described in the paper, this is an underconstraint in the testbench, since the testbench does not force the register file data to be the same in both universes when spy_mode starts. We fix this by adding conditions to the testbench and re-running JG:

```
./fixes/fix_underconstrain_vscale.sh;
jg ft_vscale_core/FPV.tcl -proj projs/vscale_fixed &
```

After refining the CEX, the FPV engine keeps searching until it reaches the time limit (24h in our evaluation).

### A.5.2    CVA6: Uncovering and fixing hardware bugs.

**Build.** Clone CVA6 and check out the commit without fixes:

```
cd $AUTOCC_ROOT;
git clone -b autocc https://github.com/morenes/cva6.git
```

Run JG on the CVA6 testbench:

```
jg ft_cva6/FPV.tcl -proj projs/cva6_orig &
```

**CEX C1.** The tool should find a CEX to the assertion as_PC_equal in under 30 minutes with a depth of 76 cycles (this may vary depending on the JG version).

**Waveform C1.** The waveform can be seen with the list of signals cva6_c1.sig from the sigs folder.

In the waveform, we would see the pc_q being different because instr_compressed had a different value. This difference propagated based on garbage data being read from the instruction cache during an exception.

**Fix C1.** Zero out data coming from the instruction cache if the line is not a hit. We apply the fix by checking out a branch with the patch already included.

```
cd cva6; git checkout autocc_fix_cex1;
cd ..;
jg ft_cva6/FPV.tcl -proj projs/cva6_c1 &
```

**CEX C2.** The tool should have found a CEX to the assertion as__AXI_ar_valid_equal in under 6 hours with a depth of 80 cycles.

**Waveform C2.** We add the list of signals cva6_c2.sig from the sigs folder. In the waveform we would see the signal ariane1.ex_stage_i.lsu_i.gen_mmu_sv39.i_cva6_mmu.i_ptw.state_q transitioning from WAIT_VALID to IDLE, which is an illegal FSM transition caused by ariane1.ex_stage_i.lsu_i.gen_mmu_sv39.i_cva6_mmu.i_ptw.flush_i being set while the PTW is waiting for a response.

**Fix C2.** Update the FSM to remain in WAIT_VALID even when flush_i is set.[9] We verify the fix by checking out a branch with the patch already included:

```
cd cva6; git checkout autocc_fix_cex2;
cd ..;
jg ft_cva6/FPV.tcl -proj projs/cva6_c2 &
```

The previous CEX trace should not be found anymore due to the fix. We have not continued debugging possible CEXs that may appear to this or other assertions.

---

[9]Fix applied upstream: github.com/openhwgroup/cva6/pull/1184

### A.5.3    MAPLE: Engineering a covert channel exploit.

**Build.** Install OpenPiton with MAPLE inside it:

```
cd $AUTOCC_ROOT
git clone -b openpiton-maple
    https://github.com/PrincetonUniversity/openpiton.git
cd openpiton;
source piton/ariane_setup.sh;
source piton/ariane_build_tools.sh;
# Building takes ~5-10 minutes
```

Clone and build the MAPLE repo:

```
source ../maple_setup_build.sh
# Building takes ~1 minute
```

**Uncovering a covert channel with AutoCC.** Start by running MAPLE's FT on JG:

```
cd $AUTOCC_ROOT
jg ft_maple/FPV.tcl -proj projs/maple_c1 &
```

In less than 30 minutes we should find a CEX at depth 21, where the assertion as__dev1_merger_vr_noc1_val fails. We can continue with the RTL simulation step while this experiment is running.

**Exploiting the covert channel in RTL simulation.** Start by running the attack to reveal the secret key:

```
cd openpiton/maple;
./run_test.sh 4;
```

The recovered secret should be 0xdeadbeef. The reported cycle count should be less than 6000 cycles.

**Closing the covert channel.** We now apply the patch to close the covert channel and run the system-level test again:

```
git checkout fa614fc;
source ../../maple_setup_build.sh
./run_test.sh 4;
```

The recovered secret should be 0x00000000. This indicates that the secret cannot be extracted using this channel anymore.

### A.5.4    AES Accelerator: Achieving full proof.

**Build.** Clone the AES repo:

```
cd $AUTOCC_ROOT
git clone https://github.com/morenes/aes.git
git checkout AutoCC-AE
```

**Achieving Full Proof.** We run JG on the AES testbench, with the DUT being the RTL of the AES accelerator:

```
jg ft_aes/FPV.tcl -proj projs/aes &
```

This testbench already includes the architectural modeling described in Sec. 4.4 of the paper to avoid spurious CEXs. The result of this run should be full-proof, i.e. no CEXs found, in less than 6 hours.

# REFERENCES

[1] Onur Acıiçmez, Shay Gueron, and Jean-Pierre Seifert. 2007. New branch prediction vulnerabilities in OpenSSL and necessary software countermeasures. In *IMA International Conference on Cryptography and Coding*. Springer, 185–203.

[2] Monjur Alam, Haider Adnan Khan, Moumita Dey, Nishith Sinha, Robert Locke Callan, Alenka G Zajic, and Milos Prvulovic. 2018. One&Done: A Single-Decryption EM-Based Attack on OpenSSL's Constant-Time Blinded RSA.. In *USENIX Security Symposium*, Vol. 8. 585–602.

[3] Armaiti Ardeshiricham, Wei Hu, Joshua Marxen, and Ryan Kastner. 2017. Register transfer level information flow tracking for provably secure hardware design. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 1691–1696.

[4] Jonathan Balkind, Katie Lim, Fei Gao, Jinzheng Tu, David Wentzlaff, Michael Schaffner, Florian Zaruba, and Luca Benini. 2019. OpenPiton+Ariane: The First Open-Source, SMP Linux-booting RISC-V System Scaling From One to Many Cores. In *Computer Architecture Research with RISC-V, CARRV*, Vol. 19.

[5] Jonathan Balkind, Katie Lim, Michael Schaffner, Fei Gao, Grigory Chirkov, Ang Li, Alexey Lavrov, Tri M Nguyen, Yaosheng Fu, Florian Zaruba, et al. 2020. BYOC: a 'bring your own core' framework for heterogeneous-ISA. In *ASPLOS'25*. 699–714.

[6] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. 1999. Symbolic model checking without BDDs. In *International conference on tools and algorithms for the construction and analysis of systems*. Springer, 193–207.

[7] Cadence Design Systems Inc. 2015. JasperGold Apps User Guide.

[8] Cadence Design Systems Inc. 2016. JasperGold Engine Selection Guide.

[9] Sadullah Canakci, Leila Delshadtehrani, Furkan Eris, Michael Bedford Taylor, Manuel Egele, and Ajay Joshi. 2021. DirectFuzz: Automated Test Generation for RTL Designs Using Directed Graybox Fuzzing. In *2021 58th ACM/IEEE Design Automation Conference (DAC)* (San Francisco, CA, USA). IEEE Press, 529–534. https://doi.org/10.1109/DAC18074.2021.9586289

[10] Luca P. Carloni. 2016. The Case for Embedded Scalable Platforms. In *Proceedings of the 53rd Design Automation Conference (DAC)*. 17:1–17:6.

[11] Sunjay Cauligi, Gary Soeller, Brian Johannesmeyer, Fraser Brown, Riad S Wahby, John Renner, Benjamin Grégoire, Gilles Barthe, Ranjit Jhala, and Deian Stefan. 2019. Fact: a DSL for timing-sensitive computation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 174–189.

[12] Gregory K. Chen, Phil C. Knag, Carlos Tokunaga, and Ram K. Krishnamurthy. 2023. An Eight-Core RISC-V Processor With Compute Near Last Level Cache in Intel 4 CMOS. *IEEE Journal of Solid-State Circuits* 58, 4 (2023), 1117–1128. https://doi.org/10.1109/JSSC.2022.3228765

[13] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. 2000. *Model Checking*. MIT Press, Cambridge, MA, USA.

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

[15] Fei Gao, Ting-Jung Chang, Ang Li, Marcelo Orenes-Vera, Davide Giri, Paul J Jackson, August Ning, Georgios Tziantzioulis, Joseph Zuckerman, Jinzheng Tu, et al. 2023. DECADES: A 67mm 2, 1.46 TOPS, 55 Giga Cache-Coherent 64-bit RISC-V Instructions per second, Heterogeneous Manycore SoC with 109 Tiles including Accelerators, Intelligent Storage, and eFPGA in 12nm FinFET. In *2023 IEEE Custom Integrated Circuits Conference (CICC)*. IEEE, 1–2.

[16] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. 2019. Time protection: the missing OS abstraction. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–17.

[17] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. 2018. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering* 8, 1 (2018), 1–27.

[18] Davide Giri, Kuan-Lin Chiu, Guy Eichler, Paolo Mantovani, and Luca P Carloni. 2021. Accelerator integration for open-source SoC design. *IEEE Micro* 41, 4 (2021), 8–14.

[19] Klaus v Gleissenthall, Rami Gökhan Kıcı, Deian Stefan, and Ranjit Jhala. 2019. IODINE: Verifying constant-time execution of hardware. In *Usenix Security*, Vol. 19. 3361338–3361436.

[20] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation leak-aside buffer: Defeating cache side-channel protections with {TLB} attacks. In *27th USENIX Security Symposium (USENIX Security 18)*. 955–972.

[21] Roberto Guanciale, Musard Balliu, and Mads Dam. 2020. Inspectre: Breaking and fixing microarchitectural vulnerabilities by formal analysis. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*. 1853–1869.

[22] John L. Hennessy and David A. Patterson. 2019. A New Golden Age for Computer Architecture. *Commun. ACM* 62, 2 (2019), 48–60.

[23] Wei-Ming Hu. 1992. Lattice scheduling and covert channels. In *Proceedings 1992 IEEE Computer Society Symposium on Research in Security and Privacy*. IEEE Computer Society, 52–52.

[24] Bo-Yuan Huang, Hongce Zhang, Pramod Subramanyan, Yakir Vizel, Aarti Gupta, and Sharad Malik. 2018. Instruction-Level Abstraction (ILA): A Uniform Specification for System-on-Chip Verification. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 24, 1 (2018), 1–24.

[25] Ralf Hund, Carsten Willems, and Thorsten Holz. 2013. Practical timing side channel attacks against kernel space ASLR. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 191–205.

[26] Jaewon Hur, Suhwan Song, Dongup Kwon, Eunjin Baek, Jangwoo Kim, and Byoungyoung Lee. 2021. DifuzzRTL: Differential Fuzz Testing to Find CPU Bugs. In *2021 IEEE Symposium on Security and Privacy (SP)*. 1286–1303. https://doi.org/10.1109/SP40001.2021.00103

[27] IEEE. 2013. Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language. IEEE 1800-2012, 1–1315. https://doi.org/10.1109/IEEESTD.2013.6469140

[28] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2015. A shared cache attack that works across cores and defies VM sandboxing–and its application to AES. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 591–604.

[29] Rahul Kande, Addison Crump, Garrett Persyn, Patrick Jauernig, Ahmad-Reza Sadeghi, Aakash Tyagi, and Jeyavijayan Rajendran. 2022. TheHuzz: Instruction Fuzzing of Processors Using Golden-Reference Models for Finding Software-Exploitable Vulnerabilities. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 3219–3236. https://www.usenix.org/conference/usenixsecurity22/presentation/kande

[30] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. 2019. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1–19.

[31] Hyoukjun Kwon, William Harris, and Hadi Esmaeilzadeh. 2017. Proving flow security of sequential logic via automatically-synthesized relational invariants. In *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*. IEEE, 420–435.

[32] Kevin Laeufer, Jack Koenig, Donggyu Kim, Jonathan Bachrach, and Koushik Sen. 2018. RFUZZ: Coverage-Directed Fuzz Testing of RTL on FPGAs. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* (San Diego, CA, USA). IEEE Press, 1–8. https://doi.org/10.1145/3240765.3240842

[33] Butler W. Lampson. 1973. A Note on the Confinement Problem. *Commununications of the ACM (CACM)* 16 (1973), 613–615. https://doi.org/10.1145/362375.362389

[34] Ang Li, Ting-Jung Chang, Fei Gao, Tuan Ta, Georgios Tziantzioulis, Yanghui Ou, Moyang Wang, Jinzheng Tu, Kaifeng Xu, Paul Jackson, August Ning, Grigory Chirkov, Marcelo Orenes-Vera, Shady Agwa, Xiaoyu Yan, Eric Tang, Jonathan Balkind, Christopher Batten, and David Wentzlaff. 2023. CIFER: A Cache-Coherent 12nm 16mm2 SoC With Four 64-Bit RISC-V Application Cores, 18 32-Bit RISC-V Compute Cores, and a 1541 LUT6/mm2 Synthesizable eFPGA. *IEEE Solid-State Circuits Letters* (2023), 1–1. https://doi.org/10.1109/LSSC.2023.3303111

[35] Xun Li, Vineeth Kashyap, Jason K Oberg, Mohit Tiwari, Vasanth Ram Rajarathinam, Ryan Kastner, Timothy Sherwood, Ben Hardekopf, and Frederic T Chong. 2014. Sapper: A language for hardware-level security policy enforcement. In *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems*. 97–112.

[36] Xun Li, Mohit Tiwari, Jason K Oberg, Vineeth Kashyap, Frederic T Chong, Timothy Sherwood, and Ben Hardekopf. 2011. Caisson: a hardware description language for secure information flow. *ACM Sigplan Notices* 46, 6 (2011), 109–120.

[37] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. 2015. Last-level cache side-channel attacks are practical. In *2015 IEEE symposium on security and privacy*. IEEE, 605–622.

[38] Albert Magyar. 2015. VSCALE. https://github.com/LGTMCU/vscale.

[39] Yatin A Manerkar, Daniel Lustig, Margaret Martonosi, and Michael Pellauer. 2017. RTLCheck: Verifying the memory consistency of RTL designs. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. 463–476.

[40] Opeoluwa Matthews, Aninda Manocha, Davide Giri, Marcelo Orenes-Vera, Esin Tureci, Tyler Sorensen, Tae Jun Ham, Juan L Aragón, Luca P Carloni, and Margaret Martonosi. 2020. MosaicSim: A Lightweight, Modular Simulator for Heterogeneous Systems. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 136–148.

[41] Kenneth L McMillan. 1993. Symbolic model checking. In *Symbolic Model Checking*. Springer, 25–60.

[42] Andres Meza, Francesco Restuccia, Ryan Kastner, and Jason Oberg. 2022. Safety verification of third-party hardware modules via information flow tracking. In *Proc. 1st Real-Time Intell. Edge Comput. Workshop (RAGE) Co-Located 59th Design Autom. Conf.(DAC)*. 1–4.

[43] OpenHW Group. 2023. CVA6. https://github.com/openhwgroup/cva6.

[44] Marcelo Orenes-Vera. 2021. AutoSVA. https://github.com/PrincetonUniversity/AutoSVA.

[45] Marcelo Orenes-Vera. 2022. MAPLE. https://github.com/PrincetonUniversity/maple.

[46] Marcelo Orenes-Vera, Aninda Manocha, Jonathan Balkind, Fei Gao, Juan L. Aragón, David Wentzlaff, and Margaret Martonosi. 2022. Tiny but Mighty:

Designing and Realizing Scalable Latency Tolerance for Manycore SoCs. In *Proceedings of the 49th Annual International Symposium on Computer Architecture* (New York, New York) (*ISCA '22*). Association for Computing Machinery, New York, NY, USA, 817–830. https://doi.org/10.1145/3470496.3527400

[47] Marcelo Orenes-Vera, Aninda Manocha, David Wentzlaff, and Margaret Martonosi. 2021. AutoSVA: Democratizing Formal Verification of RTL Module Interactions. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 535–540. https://doi.org/10.1109/DAC18074.2021.9586118

[48] Riccardo Paccagnella, Licheng Luo, and Christopher W Fletcher. 2021. Lord of the Ring (s): Side Channel Attacks on the CPU On-Chip Ring Interconnect Are Practical.

[49] Luca Piccolboni, Davide Giri, and Luca P Carloni. 2022. Accelerators & Security: The Socket Approach. *IEEE Computer Architecture Letters* 21, 2 (2022), 65–68.

[50] Ping Yeung and K. Larsen. 2005. Practical Assertion-based Formal Verification for SoC. In *2005 Intl. Symposium on System-on-Chip*. 58–61.

[51] Xida Ren, Logan Moody, Mohammadkazem Taram, Matthew Collin Jordan, Dean M. Tullsen, and Ashish Venkat. 2021. I See Dead µops: Leaking Secrets via Intel/AMD Micro-Op Caches. *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)* (2021), 361–374.

[52] Francesco Restuccia, Andres Meza, and Ryan Kastner. 2021. Aker: A Design and Verification Framework for Safe and Secure SoC Access Control. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 1–9. https://doi.org/10.1109/ICCAD51958.2021.9643538

[53] Francesco Restuccia, Andres Meza, Ryan Kastner, and Jason Oberg. 2023. A Framework for Design, Verification, and Management of SoC Access Control Systems. *IEEE Trans. Comput.* 72, 2 (2023), 386–400. https://doi.org/10.1109/TC.2022.3209923

[54] Karl Rupp. 2018. 42 Years of Microprocessor Trend Data. https://www.karlrupp.net/2018/02/42-years-of-microprocessor-trend-data/.

[55] Erik Seligman, Tom Schubert, and MV Achutha Kiran Kumar. 2015. *Formal verification: an essential toolkit for modern VLSI design*. Morgan Kaufmann.

[56] Stuart Sutherland. 2015. Who Put Assertions In My RTL Code? And Why? How RTL Design Engineers Can Benefit from the Use of SVA. *SNUG Silicon Valley* (2015), 1–26.

[57] Texas Instruments. 2011. OMAP4 mobile applications platform. *Product Bulletin* (2011).

[58] Timothy Trippel, Kang G. Shin, Alex Chernyakhovsky, Garret Kelly, Dominic Rizzo, and Matthew Hicks. 2022. Fuzzing Hardware Like Software. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 3237–3254. https://www.usenix.org/conference/usenixsecurity22/presentation/trippel

[59] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue In-Flight Data Load. In *2019 IEEE Symposium on Security and Privacy (SP)*. 88–105. https://doi.org/10.1109/SP.2019.00087

[60] Ashish Venkat and Dean M. Tullsen. 2014. Harnessing ISA Diversity: Design of a heterogeneous-ISA Chip Multiprocessor. In *ISCA*. IEEE Press.

[61] Yingchen Wang, Riccardo Paccagnella, Elizabeth Tang He, Hovav Shacham, Christopher W. Fletcher, and David Kohlbrenner. 2022. Hertzbleed: Turning Power Side-Channel Attacks Into Remote Timing Attacks on x86. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 679–697.

[62] Tianrui Wei, Nazerke Turtayeva, Marcelo Orenes-Vera, Omkar Lonkar, and Jonathan Balkind. 2023. Cohort: Software-Oriented Acceleration for Heterogeneous SoCs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) (*ASPLOS 2023*). Association for Computing Machinery, New York, NY, USA, 105–117. https://doi.org/10.1145/3582016.3582059

[63] Nils Wistoff, Moritz Schneider, Frank K Gürkaynak, Gernot Heiser, and Luca Benini. 2023. Systematic Prevention of On-Core Timing Channels by Full Temporal Partitioning. *IEEE Trans. Comput.* 72, 5 (2023), 1420–1430. https://doi.org/10.1109/TC.2022.3212636

[64] Nils Wistoff, Moritz Schneider, Frank K. Gürkaynak, Luca Benini, and Gernot Heiser. 2021. Microarchitectural Timing Channels and their Prevention on an Open-Source 64-bit RISC-V Core. In *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*. 627–632. https://doi.org/10.23919/DATE51398.2021.9474214

[65] Claire Wolf. 2023. SymbiYosys. https://github.com/YosysHQ/SymbiYosys.

[66] Fan Yao, Milos Doroslovacki, and Guru Venkataramani. 2018. Are coherence protocol states vulnerable to information leakage?. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 168–179.

[67] YosysHQ GmbH. 2023. YosysHQ. https://www.yosyshq.com/about.

[68] Florian Zaruba and Luca Benini. 2019. The Cost of Application-Class Processing: Energy and Performance Analysis of a Linux-Ready 1.7-GHz 64-Bit RISC-V Core in 22-nm FDSOI Technology. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27, 11 (2019), 2629–2640. https://doi.org/10.1109/TVLSI.2019.2926114

[69] Florian Zaruba, Fabian Schuiki, Stefan Mach, and Luca Benini. 2019. The Floating Point Trinity: A Multi-modal Approach to Extreme Energy-Efficiency and Performance. In *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. 767–770. https://doi.org/10.1109/ICECS46596.2019.8964820

[70] Yu Zeng, Aarti Gupta, and Sharad Malik. 2022. Automatic generation of architecture-level models from RTL designs for processors and accelerators. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 460–465.

[71] Danfeng Zhang, Yao Wang, G Edward Suh, and Andrew C Myers. 2015. A hardware design language for timing-sensitive information-flow security. *Acm Sigplan Notices* 50, 4 (2015), 503–516.