AutoCC: Automatic Discovery of Covert Channels in Time-Shared Hardware

Marcelo Orenes-Vera
Princeton University
Princeton, NJ, USA
movera@princeton.edu

Hyunsung Yun
Princeton University
Princeton, NJ, USA

Nils Wistoff
ETH Zurich
Zurich, Switzerland

Gernot Heiser
UNSW Sydney
Sydney, Australia

Luca Benini
ETH Zurich
Zurich, Switzerland

David Wentzlaff
Princeton University
Princeton, NJ, USA

Margaret Martonosi
Princeton University
Princeton, NJ, 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, branch 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 manually identify all vulnerabilities.

To systematically 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 in a time-multiplexed fashion. 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 our tool to generate new FPV testbenches applying the AutoCC methodology, we evaluated AutoCC on four open-source hardware projects, including two RISC-V cores and two accelerators. Without hand-written code or directed tests, AutoCC uncovered both known covert channels (within minutes instead of many hours of test-driven emulations) and new 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 improve both the security and reliability of hardware designs.

1. INTRODUCTION

The end of Moore's law has given rise to increasingly complex and heterogeneous System-on-Chip (SoC) designs, which are composed of diverse hardware blocks and intricate software systems [6, 12, 20, 24, 41, 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 [5, 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 [35].

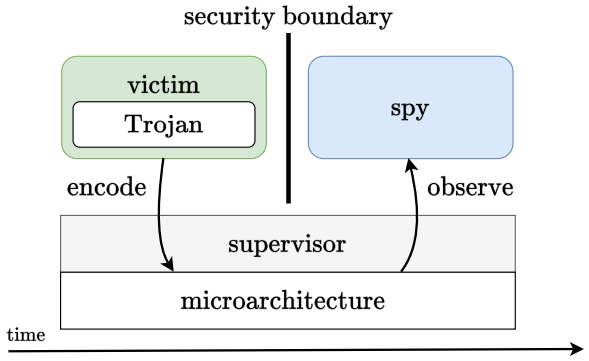

Figure 1: A microarchitectural covert channel. The Trojan in the victim process modifies—via permitted operations—microarchitectural state so as to encode a secret. The spy process observes this modification, either directly or via a timing difference, to infer the secret. Sec. 2.1 exemplifies using a covert channel to leak a secret.

Uncovering covert channels in heterogeneous SoCs with 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. Verifying the effectiveness of the RTL fix is also challenging, as the fix may change 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 *challenges*, such as a steep learning curve, the difficulty of posing the security problem into FPV to find the desired behavior as property counterexamples (CEXs), and the exponential growth of FPV tool runtime with the increase in hardware state size.

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 a 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 approach enables RTL designers to systematically explore data leaks between processes that time-multiplex the usage of a hardware IP block while not needing to reason about which states may leak. The modularity of our methodology makes it suitable for large designs—circumventing the exponential state growth—and the automatic generation of FTs makes it more accessible to RTL designers. 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 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 as 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 [35]; it enables information leakage across security boundaries of the operating system (OS) and between domains that should be isolated in violation of the system's security policy. For example, a spy process can leverage a covert channel to extract a secret key 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 [3, 61]. *Microarchitectural channels* exploit hardware states invisible to the instruction set architecture (ISA) to enable unauthorized information flow [18, 64]. Our paper focuses on the latter; for the rest of the paper when we say covert channels, we refer specifically to microarchitectural ones.

2.1 Covert Channels

Covert channels have been demonstrated via the L1-D [25] and L1-I caches [2], the last-level cache (LLC) [30, 38], the TLB [22, 27], the branch predictor [2], and the interconnect [48, 66]. The Spectre attack [32] 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: To motivate the threat scenario, let us assume a setup as shown in Fig. 1. The victim and the spy are two applications running concurrently on shared hardware. They are (supposedly) isolated by a supervisor using 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 we mentioned above, but potentially 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) [19]. Our threat model (detailed in Sec. 3.1) is based on time-sharing, because (a) it is common in specialized hardware, and (b) a security domain may prefer to not simultaneously share capacity- or bandwidth-limited resources (e.g., instruction cache, TLBs, predictors, etc.) with another one 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 arising from microarchitectural state that is dependent on the victim's secret [19]. 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 any traces dependent on the requested data. As Sec. 3 explains, AutoCC detects differences at RTL module interfaces, and thus it 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 sub-

set of the covert channels when 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 cryptographic software [13]. In a simultaneous multi-threading processor, *hardware partitioning* spatially partitions shared resources like caches or prediction tables. In a time-shared system, shared resources are temporally partitioned via a flush [64]—this is 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 [21]. 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 with 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 software.

Detection: Information flow security in hardware has been actively explored since the early 2010s [4, 43, 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 [11,28,31,34,58], 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 Hw Verification

The first works to ensure RTL correctness through formal verification utilized model checking with SAT solvers and binary decision diagrams [7, 42, 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 [10, 65] to exhaustively search for property violations. 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 length of the shortest path between the two states furthest apart in the model [15]. 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.

FPV has been used for different purposes: RTLCheck verifies RTL implementations of CPUs against their memory consistency models [40]; ILA generates a Verilog model of the design from its functional specification and compares it against the RTL implementation [26]; 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, to detect data leakage across processes. Sec. 3 elaborates on how we frame this detection as a FPV problem.

Formal methods have also been used to detect security vulnerabilities. InSpectre [23] creates formal models of processors to detect Spectre-like attacks that combine speculative execution and a covert channel. UPEC [16] 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 in hardware RTL to automatically detect microarchitectural covert channels that arise from state dependent on a previous execution impacting the timing of future instructions. AutoCC complements empirical covert channel measurement frameworks such as *Channel Bench* [17], which show the (non-)existence of specific channels, but not all.

3. THE AUTOCC APPROACH

This section starts by presenting the threat model we tackle in this paper: time-multiplexed executions of processes on shared hardware. Second, it describes how we formalize that threat model into a problem that FPV engines can solve to automatically discover covert channels between these processes. Third, it shows how to apply this methodology to an RTL project using our automated flow to generate the FPV testbench and tool bindings. Fourth, it provides a viable path for applying AutoCC to large projects via modularity. Finally, it proposes two approaches that leverage AutoCC to aid the design of temporal protections that prevent 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 process executes in a controlled environment in which the OS restricts the clients with which the victim may communicate.

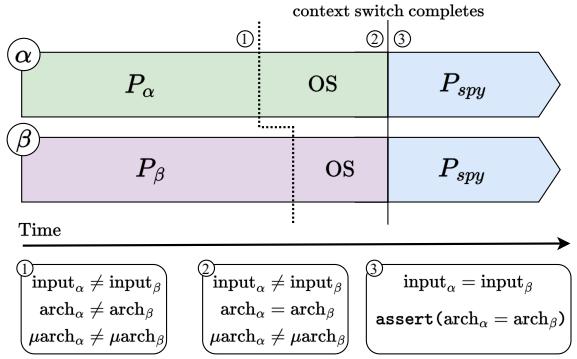

Figure 2: Overview of the AutoCC methodology. The victim processes P_α and P_β are free to take on any legal execution for an arbitrary number of cycles; the inputs to both are symbolic. At the end of this phase ①, both *arch* and $\mu arch$ of α and β may differ. The context switch then occurs, and once it completes at ② the *arch* states of both α and β are the same, but differences in $\mu arch$ state 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 $\mu arch$ state after the context switch cause observable differences P_{spy} execution.

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 illegally extract information. Its primary asset is a *Trojan*, some code in the victim process that leaks the secret data via the channel.

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 designer, placing the final determination of which ones pose real threats in the hands of the person who knows the system best. As such, we make no assumptions about the way the secret data is encoded into the state of the compromised hardware. 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 and fix covert channels regardless of the Trojan's nature allows us to prove stronger correctness assertions—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 fashion and they are also susceptible to covert channel vulnerabilities. The operations available to these specialized hardware blocks can be considered as their ISA [70]. For the rest of the paper, DUT will refer to the top-level module that 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 FPV problem, 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 section 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 or defines the program context.*

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 to which they belong is a responsibility shared by both 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 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 piece of 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 α and β—in the following way (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 constitutes a legal execution.)

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 α and β 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 α and β, 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, α and β 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 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, α and β 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—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 a tedious task. 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 α and β without reasoning about their internal signals. Any difference between $arch_\alpha$ and $arch_\beta$ on cycle k can, by a sequence of these instructions, be externalized by the FPV tool as a difference in outputs on cycle $k+n$ for some bounded n. This allows the AutoCC tool to generate a FPV testbench (FT) without any 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* that is 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 α and β and the symbolic *arch* of the spy (y-axis) are resolved by the end of the context switch. Although α and β 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 sequence of instruction that lead α and β 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 to not overconstrain the model, as this can miss exploring important behavior.

Measuring Context Switch Latency: For all of its advantages, taking the end of the flush as the synchronization point between α and β admits one blind spot, as it assumes that the flushes in both universes complete on the same cycle. This precludes any CEXs that originate from a difference in the latencies of the flush event itself. If a Trojan can modulate this latency and a spy can observe the difference, the flush

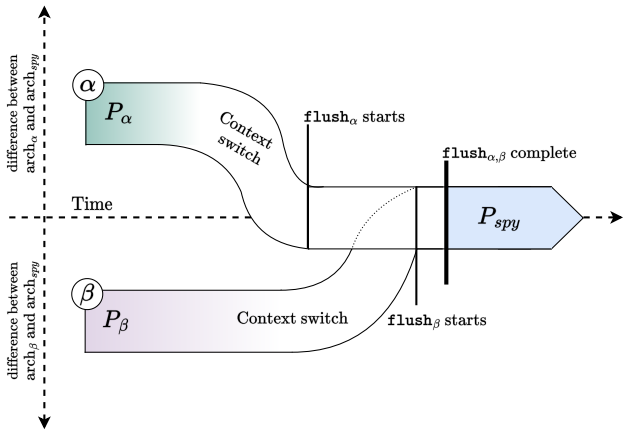

Figure 3: AutoCC model of the context switch event. Instead of enforcing a discrete jump to a sequence of OS instructions, we simply require that the victim processes in α and β eventually converge to the same *arch* (indicated here by P_α and P_β converging on the y-axis). This is then the state of 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 α and β; it is only required they complete together.

latency itself may function as a covert channel. AutoCC can further verify the DUT against this behavior by taking the start of the flush as the cycle on which α and β must converge. The flush event may then be considered as part of the spy process, and our existing assertions will generate a CEX for any differences between the flush event in α and β.

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 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 [1, 47] to parse the DUT interface.

3.3.1 Generating the DUT Wrapper

Based on the RTL we set as the DUT (e.g., core, accelerators, or subset of them, the flow generates a 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., α and β), except for the signals that 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 to not 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 α and β 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 internals of the DUT, as the properties generated solely use interface signals. Properties are written in SystemVerilog Assertions language (SVA) [29]. Assumptions are generated for DUT inputs, and assertions for outputs.

Transactions: When a group of signals is governed by a valid signal we call 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 makes use of 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 approach to automatically identify transactions [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 complete 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 is responsible to manage 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 α and β, 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 length of 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 of training to effectively use them. To ease their usage, our tool also generates the backend-specific commands and binding files required to use FPV engines based on their documentation [10, 65]. We have tested AutoCC with two different backends: JasperGold [9] and 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 involved in that 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 size of the RTL and the depth of the search (time in cycles). As a baseline mitigation, we adopt the standard technique of minimizing the size of modules that are parameterized, 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 bounded proof that is sufficiently deep to provide confidence in the correctness of the design. To that end, we adopt two additional techniques: blackboxing and modularity. (Since blackboxing is 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 α and β, while the inputs to the DUT are assumed equal.

To the verification engine, the internals of a blackboxed module do not exist; it does not follow any state evolution. A module should only be blackboxed, then, 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 complexity of the verification problem is reduced exponentially. Second, the depth of the exploration required to exercise the relevant features of the DUT is reduced, since the FPV tool is driving the inputs of the DUT directly.

Disadvantages: First, the CEXs that are found are less informative, since we do not know how the inputs of the DUT were produced. For the case of blackboxing, this refers to the outputs of the blackboxed module, which are driving the rest of the logic still within the DUT. Second, 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 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 to not include these assumptions in its RTL module if the rest of the SoC is untrusted (e.g., resulting from integrating third-party 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. However, we pose 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 avoids overconstraining the verification process.

SoC-level Verification: In order to apply AutoCC at the SoC level, we recommend first creating FTs for RTL modules with the simplest interfaces, e.g., modules that are 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 RTL modules can be independently verified to produce a deterministic output given an input sequence, regardless of their $\mu arch$. 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.

Algorithm 1: Incremental Flush Signal Construction

```
Flush ← ∅;
result ← FPV(DUT, Flush, AutoCC_FT);
while (result == CEX) do
    state ← FindCause(result);
    Insert(Flush, state);
    result ← FPV(DUT, Flush, AutoCC_FT);
```

Algorithm 2: Decremental Flush Signal Construction

```
Candidates ⊆ μarch;
Flush ← μarch;
for (state in Candidates) do
    Remove(Flush, state);
    result ← FPV(DUT, Flush, AutoCC_FT);
    if (result != Proof) then
        Insert(Flush, state);
```

3.5 AutoCC During RTL Development

The properties in Listing 1 are expressed using interface signals, making them implementation-independent. This, along with their modular nature, allows hardware designers to utilize AutoCC properties for test driven development (TDD), where CEXs help to refine the design [8, 56].

TDD is particularly useful to design the $\mu arch$ flush mechanism. The overall flush mechanism would be correct if every module involved in the victim process is effectively flushing 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 needs to be flushed to provide full temporal partitioning (i.e., no observable differences).

Algorithm 1 incrementally constructs the flush signal by inserting states that cause a CEX to AutoCC properties when not flushed. 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 as there is no incentive to remove a state flush if it has no impact on 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 [39]; application-class 64-bit CVA6 core [44,68]; MAPLE memory access engine [45,46], and an accelerator for AES encryption [39]. 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).

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: 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
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.

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.

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 that the AutoCC methodology works 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 leaks in RTL and re-ran AutoCC to confirm that the CEXs were no longer found, and (c) communicated with the maintainers of these projects, and most of the fixes have now been merged.

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 (a potential user) through how we applied the AutoCC methodology to it. First, we create the FT with the following command: `python autocc.py -f vscale_core.v`. Second, we run the generated FT using JasperGold: `jg ft_vscale_core/FPV.tcl`. Note that this first run uses the default values for the flush signal and the architectural state signal (see Listing 1). The CEXs shown in Table 2 are the result of refining the definition of the architectural state.

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 there are many registers inside the CSR module, it was more convenient to black box the module 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 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 [44]. Particularly, our fifth CEX of Table 2 showed a case where an interrupting instruction in the write-back stage of α—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 FV engine kept searching until it timed out after our time limit of

24 hours. At that moment, it had reached a bounded proof of depth 21. Since Vscale does not have caches or other deep units, and the previous CEX had a depth of 9, we believe it would not find any more CEXs even if the tool 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), that has been taped out multiple times into silicon [14, 69]. CVA6 offers several 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 (FPU) to lighten the FV 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 $\mu arch$ flush 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$) has invalidated its lines. One of the first CEXs we found (after we added the PC, register file, and CSR into the *arch* signal) was caused by executions where α had an outstanding AXI (Advanced eXtensible Interface) request going into the flush while β did not. Since the arrival of the flush signal kills all outstanding AXI transactions, α's instruction cache (I$), which was making the request, transitioned to a `KILL_MISS` state while β's remained in `IDLE`. This divergence of $\mu arch$ 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 complete. We found another cause after assuming that all AXI requests are satisfied before the flush event, where the page table walker (PTW) takes longer to flush in α because it had an active memory request to the D$. These CEXs confirm and extend the findings about *full flush* `fence.t` made in [63]. The observation that subtle, hard-to-find components may produce a covert channel if not cleared systematically was their primary motivation for the third implementation of CVA6's $\mu arch$ flush: *microreset*.

Evaluating the safest configuration. Unlike the full flush, microreset targets the entire $\mu arch$ 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 $\mu arch$ and padding to a constant latency is the most thorough temporal partition that 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 α and β. 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 its output back to the pipeline—based on one bit of this payload without knowing that the payload did not come from a valid I$ line. The difference in the output of the realigner then results in a PC difference in α and β. 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 α and β, 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` goes high, 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 α does. However, once β is in `WAIT_RVALID`, it takes an exception, causing `flush` to go high again. As a result, β's FSM transitions to `IDLE` on the next cycle, terminating the 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 α 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 up with a valid line after the flush completes. This CEX shows that a sequence of events initiated before the flush lead to an effect observed 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 proposed 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 TLB between processes. The `cleanup` operation is performed as a first step of the initialization process.

[1] https://github.com/openhwgroup/cva6/pull/1184

[2] https://github.com/pulp-platform/cva6/commit/ae79ec5

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 found was caused by several other requests being in the NoC protocol's output buffer in α when the flush signal was asserted. 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 second CEX found in 28 seconds at depth 14, was caused by the TLB in α being disabled while the TLB in β 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 that indicates whether the TLB is 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 tentatively fixed this in the RTL by setting this flip-flop during the flush.

M3. The third CEX, found after 158 minutes at depth 23, 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 clarify 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)) & 0
     x00FF;
    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: Code 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, we first built the RTL simulation environment of MAPLE integrated with the OpenPiton SoC [5] following the tutorial in the MAPLE repository. Then, we performed the test bare-metal using VCS O-2018.09-SP2. It took about a minute for VCS to simulate the test program on the OpenPiton SoC with MAPLE, where the spy recovered a 128-bit secret over 16 iterations in a total of 112,000 clock cycles.

Closing M2 and M3 channels. We communicated our findings to the MAPLE maintainers, who already made two patches to the open-source RTL to close these channels. For fabricated chips that include MAPLE, 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 also 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 α contained several ongoing requests, while β had none. Since the flush signal (set free) appeared while the accelerator pipeline in α was processing requests, a timing difference appears when α eventually responds and β 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 the accelerator cannot be used by another process 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 both universes having no ongoing requests. Once this condition was added, the tool found **full proof** in 5 hours.

Heterogeneous SoCs may lead to subtle vulnerabilities. We would like to point out that 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 assumption made by other designers. Otherwise, integrating a 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 (waiting for all ongoing requests) may create 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 performance and security; by identifying covert channels, our methodology can help 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. With this information, a hardware vendor can better decide whether to fix the covert channel or issue warnings to programmers of cryptographic libraries, so that they avoid using them if that goes against their security goals.

For instance, if a hardware-based division operation is susceptible to a covert channel and fixing it would significantly slow down the operation for non-security-critical applications, programmers prioritizing security should avoid using divisions on sensitive data. However, if addressing the channel in hardware has minimal performance impact, a simple flush during context switching can prevent the covert channel, as long as the flush process does not create another one.

The Cost of Flushing Microarchitectural State: Although analyzing the performance impact of flushing $\mu arch$ is out of the scope of this paper, we can make some observations. Flushing $\mu arch$ may affect performance 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). Regarding the second one, the concern is the performance lost because the state is not available after the context switch. For example, more cache 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 length of the period between context switches and the size of these structures [63]. We also expect that since on-core caches are small—typically much smaller than the working set of a program [17]—chances are that the cache lines that were interesting for the second process were already replaced anyways, and so there is no performance impact in this regard. We regard the problem of preventing covert channels as a challenge in hardware-software co-design. Hardware must provide the required means to partition shared resources, such that an OS can use these as necessary when reallocating those resources from one security domain to another. To that end, AutoCC assists the design and verification of temporal partitioning mechanisms in RTL modules.

6. FURTHER RELATED WORK

Information flow tracking (IFT) monitors the flow of sensitive data through hardware components via RTL simulation [4, 43, 52, 53]. Like AutoCC, IFT techniques provide a precise trace of the leakage; however, they rely on input tests and user-provided security properties to operate. Prior works in IFT are in part orthogonal to AutoCC, as they focus on system-level evaluation, while AutoCC formally verifies SoC components—during or at the end of 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 [37] is one such example that statically analyzes designs written in its language to guarantee noninterference. Sapper [36] 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 values at runtime. 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.

Simarel [33], like AutoCC, uses bounded model checking to verify relational invariants between core executions. The emphasis is on using the relational invariants as inductive invariants to prove information isolation. However, Simarel reasons generally about flows between levels in a security lattice; no testing occurs against a formalized context switch.

Moreover, while prior work is effective at tracking hardware state being read and propagated, they do not directly reason about how timing in the program execution may also leak information.

7. CONCLUSION

Our research introduces an FPV methodology that, given an RTL module, exhaustively searches for execution traces within a victim process, which can lead to execution differences observable by a supposedly isolated spy process. We also developed an automated procedure to generate FPV testbenches implementing this methodology, eliminating the need for upfront user input or RTL details. We demonstrated the effectiveness and efficiency of AutoCC's methodology by applying it to four used open-source projects. Particularly, we found that AutoCC: (1) exercises previously-known issues within minutes, compared to lengthy stress-test simulations; (2) finds 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 open-source RISC-V CVA6 core and the MAPLE accelerator; (4) uncovers experimentally-viable covert channels as we validated one via system-level RTL simulation; (5) validates that the RTL fixes to close covert channels are effective.

AutoCC holds much value for hardware designers, empowering them to perform systematic searches for covert channels in their RTL modules during the development stage. This paper also presented a test-driven approach to assist in designing hardware modules that require temporal isolation, i.e., flushing the $\mu arch$ state between processes. AutoCC is open-source, available at *github.com/morenes/AutoCC*.

Acknowledgements

This material is based upon work supported by (while was serving at) the National Science Foundation (NSF), and based on research sponsored by the Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under agreement FA8650-18-2-7862. [3] The work of Wistoff and Benini has been supported in part by 'Fractal' project under grant agreement No 877056 that receives funding from ECSEL-JU as part of the EU Horizon 2020 research and innovation programme, and in part by the ETH4D Humanitarian Action Challenges Application on "Secure Infrastructure for Humanitarian Organizations".

[3]The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of NSF, AFRL and DARPA or the U.S. Government.

APPENDIX

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. They can be accessed here: *github.com/LGTMCU/vscale.git*, *github.com/morenes/cva6.git*, *github.com/PrincetonUniversity/maple.git*, *github.com/morenes/aes.git*.
- **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 *github.com/morenes/AutoCC*. The repository contains a README with detailed instructions for installing AutoCC and reproducing our results.

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 running times 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

Clone the repository for AutoCC:

```
git clone
 ↪  https://github.com/morenes/AutoCC.git;
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

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
./fixes/fix_combo_loop_vscale_rtl.sh
```

Generate the Vscale formal testbench using AutoCC.

```
export
 ↪  DUT_ROOT=$PWD/vscale/src/main/verilog;
python 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 9-cycle CEX 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 cycle 9 because the opcode was different in cycle 8 (`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 constrain the register file data to be the same in both universes when the `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 &
```

A.5.2 CVA6: Uncovering and fixing hardware bugs

Clone CVA6 and checkout 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__AXI_ar_valid_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 being different because `instr_compressed` had a different value. This 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 `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.[4] 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 &
```

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

A.5.3 MAPLE: Engineering a covert channel exploit

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. Run 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. Run the attack to reveal the secret key `0xdeadbeef`:

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

Apply the patch to close the covert channel and run the system-level test again:

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

The recovered secret should be `0x0`, indicating that the secret cannot be extracted using this channel anymore.

A.5.4 AES Accelerator: Achieving full proof

Clone the AES repo:

```
cd $AUTOCC_ROOT
git clone
 ↪  https://github.com/morenes/aes.git
```

We run JG on the AES testbench, with the DUT being the RTL of the AES accelerator:

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

This testbench already includes the architectural modeling described in Section 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] "AutoSVA," https://github.com/PrincetonUniversity/AutoSVA.

[2] O. Acıiçmez, S. Gueron, and J.-P. Seifert, "New branch prediction vulnerabilities in OpenSSL and necessary software countermeasures," in *IMA International Conference on Cryptography and Coding*. Springer, 2007, pp. 185–203.

[3] M. Alam, H. A. Khan, M. Dey, N. Sinha, R. L. Callan, A. G. Zajic, and M. Prvulovic, "One&Done: A Single-Decryption EM-Based Attack on OpenSSL's Constant-Time Blinded RSA." in *USENIX Security Symposium*, vol. 8, 2018, pp. 585–602.

[4] A. Ardeshiricham, W. Hu, J. Marxen, and R. Kastner, "Register transfer level information flow tracking for provably secure hardware design," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 2017, pp. 1691–1696.

[5] J. Balkind, K. Lim, F. Gao, J. Tu, D. Wentzlaff, M. Schaffner, F. Zaruba, and L. Benini, "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, 2019.

[6] J. Balkind, K. Lim, M. Schaffner, F. Gao, G. Chirkov, A. Li, A. Lavrov, T. M. Nguyen, Y. Fu, F. Zaruba *et al.*, "BYOC: a 'bring your own core' framework for heterogeneous-ISA," in *ASPLOS'25*, 2020, pp. 699–714.

[7] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *International conference on tools and algorithms for the construction and analysis of systems*. Springer, 1999, pp. 193–207.

[8] J. Buckingham, "Formal for designers," in *Agile Test Driven Dev. for ASIC*, 2016.

[9] Cadence Design Systems Inc., "JasperGold Apps User Guide," 2015.

[10] Cadence Design Systems Inc., "JasperGold Engine Selection Guide," 2016.

[11] S. Canakci, L. Delshadtehrani, F. Eris, M. B. Taylor, M. Egele, and A. Joshi, "Directfuzz: Automated test generation for rtl designs using directed graybox fuzzing," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE Press, 2021, p. 529–534. [Online]. Available: https://doi.org/10.1109/DAC18074.2021.9586289

[12] L. P. Carloni, "The case for Embedded Scalable Platforms," in *Proceedings of the 53rd Design Automation Conference (DAC)*, Jun. 2016, pp. 17:1–17:6.

[13] S. Cauligi, G. Soeller, B. Johannesmeyer, F. Brown, R. S. Wahby, J. Renner, B. Grégoire, G. Barthe, R. Jhala, and D. Stefan, "Fact: a DSL for timing-sensitive computation," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 174–189.

[14] G. K. Chen, P. C. Knag, C. Tokunaga, and R. K. Krishnamurthy, "An Eight-Core RISC-V Processor With Compute Near Last Level Cache in Intel 4 CMOS," *IEEE Journal of Solid-State Circuits*, vol. 58, no. 4, pp. 1117–1128, 2023.

[15] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, MA, USA: MIT Press, 2000.

[16] M. R. Fadiheh, D. Stoffel, C. Barrett, S. Mitra, and W. Kunz, "Processor hardware security vulnerabilities and their detection by unique program execution checking," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 994–999.

[17] Q. Ge, Y. Yarom, T. Chothia, and G. Heiser, "Time protection: the missing OS abstraction," in *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019, pp. 1–17.

[18] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," *Journal of Cryptographic Engineering*, vol. 8, pp. 1–27, 2018.

[19] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A survey of microarchitectural timing attacks and countermeasures on contemporary hardware," *Journal of Cryptographic Engineering*, vol. 8, no. 1, pp. 1–27, 2018.

[20] D. Giri, K.-L. Chiu, G. Eichler, P. Mantovani, and L. P. Carloni, "Accelerator integration for open-source soc design," *IEEE Micro*, vol. 41, no. 4, pp. 8–14, 2021.

[21] K. v. Gleissenthall, R. G. Kıcı, D. Stefan, and R. Jhala, "Iodine: Verifying constant-time execution of hardware," in *Usenix Security*, vol. 19, no. 10.5555, 2019, pp. 3 361 338–3 361 436.

[22] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation leak-aside buffer: Defeating cache side-channel protections with {TLB} attacks," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 955–972.

[23] R. Guanciale, M. Balliu, and M. Dam, "Inspectre: Breaking and fixing microarchitectural vulnerabilities by formal analysis," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1853–1869.

[24] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture," *Communications of the ACM*, vol. 62, no. 2, pp. 48–60, 2019.

[25] W.-M. Hu, "Lattice scheduling and covert channels," in *Proceedings 1992 IEEE Computer Society Symposium on Research in Security and Privacy*. IEEE Computer Society, 1992, pp. 52–52.

[26] B.-Y. Huang, H. Zhang, P. Subramanyan, Y. Vizel, A. Gupta, and S. Malik, "Instruction-level abstraction (ILA): A uniform specification for system-on-chip verification," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 24, no. 1, pp. 1–24, 2018.

[27] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space ASLR," in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 191–205.

[28] J. Hur, S. Song, D. Kwon, E. Baek, J. Kim, and B. Lee, "Difuzzrtl: Differential fuzz testing to find cpu bugs," in *2021 IEEE Symposium on Security and Privacy (SP)*, 2021, pp. 1286–1303.

[29] *IEEE Standard for SystemVerilog–Unified Hardware Design, Specification, and Verification Language*, IEEE 1800-2012 Std., 2013.

[30] G. Irazoqui, T. Eisenbarth, and B. Sunar, "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, 2015, pp. 591–604.

[31] R. Kande, A. Crump, G. Persyn, P. Jauernig, A.-R. Sadeghi, A. Tyagi, and J. Rajendran, "TheHuzz: Instruction fuzzing of processors using Golden-Reference models for finding Software-Exploitable vulnerabilities," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 3219–3236. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/kande

[32] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1–19.

[33] H. Kwon, W. Harris, and H. Esmaeilzadeh, "Proving flow security of sequential logic via automatically-synthesized relational invariants," in *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*. IEEE, 2017, pp. 420–435.

[34] K. Laeufer, J. Koenig, D. Kim, J. Bachrach, and K. Sen, "Rfuzz: Coverage-directed fuzz testing of rtl on fpgas," in *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE Press, 2018, p. 1–8. [Online]. Available: https://doi.org/10.1145/3240765.3240842

[35] B. W. Lampson, "A note on the confinement problem," *Commununications of the ACM (CACM)*, vol. 16, pp. 613–615, 1973.

[36] X. Li, V. Kashyap, J. K. Oberg, M. Tiwari, V. R. Rajarathinam, R. Kastner, T. Sherwood, B. Hardekopf, and F. T. Chong, "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*, 2014, pp. 97–112.

[37] X. Li, M. Tiwari, J. K. Oberg, V. Kashyap, F. T. Chong, T. Sherwood, and B. Hardekopf, "Caisson: a hardware description language for secure information flow," *ACM Sigplan Notices*, vol. 46, no. 6, pp. 109–120, 2011.

[38] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE symposium on security and privacy*. IEEE, 2015, pp. 605–622.

[39] A. Magyar, "VSCALE," https://github.com/LGTMCU/vscale.

[40] Y. A. Manerkar, D. Lustig, M. Martonosi, and M. Pellauer, "RTLCheck: Verifying the memory consistency of RTL designs," in *2017 50th Annual IEEE/ACM MICRO*, 2017, pp. 463–476.

[41] O. Matthews, A. Manocha, D. Giri, M. Orenes-Vera, E. Tureci, T. Sorensen, T. J. Ham, J. L. Aragón, L. P. Carloni, and M. Martonosi, "MosaicSim: A Lightweight, Modular Simulator for Heterogeneous Systems," in *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2020, pp. 136–148.

[42] K. L. McMillan, "Symbolic model checking," in *Symbolic Model Checking*. Springer, 1993, pp. 25–60.

[43] A. Meza, F. Restuccia, R. Kastner, and J. Oberg, "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)*, 2022, pp. 1–4.

[44] OpenHW Group, "CVA6," https://github.com/openhwgroup/cva6.

[45] M. Orenes-Vera, "MAPLE," https://github.com/PrincetonUniversity/maple.

[46] M. Orenes-Vera, A. Manocha, J. Balkind, F. Gao, J. L. Aragón, D. Wentzlaff, and M. Martonosi, "Tiny but Mighty: Designing and Realizing Scalable Latency Tolerance for Manycore SoCs," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 817–830. [Online]. Available: https://doi.org/10.1145/3470496.3527400

[47] M. Orenes-Vera, A. Manocha, D. Wentzlaff, and M. Martonosi, "AutoSVA: Democratizing Formal Verification of RTL Module Interactions," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, 2021, pp. 535–540.

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

[49] L. Piccolboni, D. Giri, and L. P. Carloni, "Accelerators & security: The socket approach," *IEEE Computer Architecture Letters*, vol. 21, no. 2, pp. 65–68, 2022.

[50] Ping Yeung and K. Larsen, "Practical assertion-based formal verification for SoC," in *2005 Intl. Symposium on System-on-Chip*, 2005, pp. 58–61.

[51] X. Ren, L. Moody, M. Taram, M. C. Jordan, D. M. Tullsen, and A. Venkat, "I See Dead μops: Leaking Secrets via Intel/AMD Micro-Op Caches," *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pp. 361–374, 2021.

[52] F. Restuccia, A. Meza, and R. Kastner, "Aker: A Design and Verification Framework for Safe and Secure SoC Access Control," in *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, 2021, pp. 1–9.

[53] F. Restuccia, A. Meza, R. Kastner, and J. Oberg, "A framework for design, verification, and management of soc access control systems," *IEEE Transactions on Computers*, vol. 72, no. 2, pp. 386–400, 2023.

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

[55] E. Seligman, T. Schubert, and M. A. K. Kumar, *Formal verification: an essential toolkit for modern VLSI design*. Morgan Kaufmann, 2015.

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

[57] Texas Instruments, "OMAP4 mobile applications platform," *Product Bulletin*, 2011.

[58] T. Trippel, K. G. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo, and M. Hicks, "Fuzzing hardware like software," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 3237–3254. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/trippel

[59] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "RIDL: Rogue In-Flight Data Load," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 88–105.

[60] A. Venkat and D. M. Tullsen, "Harnessing ISA diversity: Design of a heterogeneous-ISA chip multiprocessor," in *ISCA*. IEEE Press, 2014.

[61] Y. Wang, R. Paccagnella, E. T. He, H. Shacham, C. W. Fletcher, and D. Kohlbrenner, "Hertzbleed: Turning power Side-Channel attacks into remote timing attacks on x86," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 679–697. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/wang-yingchen

[62] T. Wei, N. Turtayeva, M. Orenes-Vera, O. Lonkar, and J. Balkind, "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*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 105–117. [Online]. Available: https://doi.org/10.1145/3582016.3582059

[63] N. Wistoff, M. Schneider, F. K. Gürkaynak, G. Heiser, and L. Benini, "Systematic prevention of on-core timing channels by full temporal partitioning," *IEEE Transactions on Computers*, vol. 72, no. 5, pp. 1420–1430, 2023.

[64] N. Wistoff, M. Schneider, F. K. Gürkaynak, L. Benini, and G. Heiser, "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)*, 2021, pp. 627–632.

[65] C. Wolf, "SymbiYosys," https://github.com/YosysHQ/SymbiYosys.

[66] F. Yao, M. Doroslovacki, and G. Venkataramani, "Are coherence protocol states vulnerable to information leakage?" in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 168–179.

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

[68] F. Zaruba and L. Benini, "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*, vol. 27, no. 11, pp. 2629–2640, 2019.

[69] F. Zaruba, F. Schuiki, S. Mach, and L. Benini, "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)*, 2019, pp. 767–770.

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

[71] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A hardware design language for timing-sensitive information-flow security," *Acm Sigplan Notices*, vol. 50, no. 4, pp. 503–516, 2015.