AutoCC: Automatic Detection of Microarchitectural Covert Channels

Hyunsung Yun

Adviser: Prof. Margaret Martonosi

Thesis

Submitted in partial fulfillment of the requirements for the degree of Bachelor of Arts in Computer Science

Princeton University

April 2023

Abstract

Microarchitectural state can be used as a covert channel to leak information across security domains. In hardware that is shared between multiple processes, these channels can be used in attacks that break process isolation. Existing methods for detecting these covert channels are largely based on manually analyzing security properties based on known attacks. The current era of heterogeneity and specialization make this approach increasingly difficult and suspect to failure.

AutoCC is an alternative, systematic approach that leverages formal verification to automatically discover covert channels in hardware that is time-shared between processes. Operating at the register-transfer level (RTL), AutoCC exhaustively detects executions in which the microarchitectural state left behind by one process could create an observable difference to another.

Validating AutoCC on four open-source hardware designs, we were able to detect known covert channels and discover new ones in a matter of minutes. Since AutoCC also produces a minimal trace showing how a covert channel is operated, we were further able to engineer both exploits and patches for some of the vulnerabilities we discovered. Finally, to both demonstrate its power as a general framework and to encourage its use, we developed a tool that, given an RTL design, automatically generates AutoCC testbenches.

Acknowledgements

If I have learned anything during my time here, it's that it really does take a village.

This thesis is certainly no exception. The lion's share of my thanks is certainly owed to Marcelo Orenes-Vera, my graduate student mentor, collaborator, and friend. I could not have asked for a better lab mate—thank you for your technical guidance, life advice, and Costco cold brew.

My thanks also to all the members of the Parallel Research Group: Rohan Prabhakar, August Ning, Grigory Chirkov, Haiyue Ma—you all saw my many blunders as a clueless undergrad and still welcomed me everyday into your lab. I look up to all of you and wish you the best on your PhD journeys! To Xuxu, thank you for the lunches and support every day at the end—yesterday, today, and tomorrow.

To the computer architecture faculty at Princeton, Professors David August, David Wentzlaff, and Margaret Martonosi—thank you for introducing me to so many wonders.

Finally, thank you to my parents, Kyeonghee Kim and Seungjoo Yun. My journey through Princeton has been one of seven years—not one day could I have done without your constant love and support. I hope I made you proud.

Contents

Chapter 1

Introduction

The contrast between function and procedure is a reflection of the general distinction between describing properties of things and describing how to do things.

—Abelson, Sussman, and Sussman, *Structure and Interpretation of Computer Programs*

A **side channel** is a source of information leakage. However clean a model of computation may be, it must ultimately be realized in the physical world, subject to concerns like time, radiation, and sound. An attacker may, in a manner much like a scientist, interpret these phenomena as data about a computational process that should appear as a black box, using its observations to obtain secret information.

Such a vulnerability is distinguished by the fact that it is not the product of a particular abstraction, be it an algorithm or an instruction set, but of a set of choices that were made in implementing an abstraction. Attacks that exploit such vulnerabilities are necessarily intimate with the details of the underlying system, familiar with not only the functional properties the system delivers, but also the means by which it delivers them.

1.1 Microarchitectural Covert Channels

This thesis is concerned with the security threats that exist within the implementing layer known as microarchitecture, the digital logic that realizes an ISA. More specifically, it targets **microarchitectural covert channels**, a class of side channels in which information is deliberately leaked across a security boundary via microarchitectural state.

1.2 Why Detecting Leaks Is Hard

Microarchitectural attacks have been at the forefront of hardware security concerns since the disclosure of Meltdown [1] and Spectre [2], two attacks that exploit speculative execution to break memory isolation in virtually all modern processors.

Traditionally, however, the semiconductor industry has paid much greater heed to performance and efficiency rather than security. Incredible intellectual and financial efforts have produced a series of faster and more efficient processors, each brimming under the hood with new performance optimizations.

Unfortunately, these optimizations have also introduced larger attack surfaces. Side channel attacks have been found in caches [3, 4], on-chip networks [5], branch predictors [6, 7], and memory controllers [8]. Shared structures and asymmetric access paths have often been found to be the source of such vulnerabilities, and these are common patterns in processor design [9].

The main challenge in discovering these leaks is complexity. Processors have become increasingly labyrinthine and as physical devices, some degree of information leakage is inevitable. A 1995 analysis of Intel x86 processors sums it up nicely: "We did not expect well-defined architectural features to cause undesirable security behavior. In retrospect, however, the very complexity of the architecture suggests that it was bound to include some unexpected feature interactions" [10].

In such a setting, finding side channels and assessing whether they represent viable attack vectors can be very difficult. The side channel may be only one component of a larger attack, as is the case in PACMAN [11]. It may also appear to be deceptively small; in his paper describing timing attacks on RSA, Kocher writes that although "intuition might suggest that unintentional timing characteristics would only reveal a small amount of information from a cryptosystem... attacks are presented which can exploit timing measurements from vulnerable systems to find the entire secret key" [12].

1.3 Contributions

With new vulnerabilities continuing to foil countermeasures in the industry state-of-the-art, a systematic method for detecting side channels is much needed and desired. To that end, this work proposes AutoCC, an automated method for detecting microarchitectural covert channels through formal verification (FV).

Our main technical contributions are the following:

1. A novel approach that leverages FV to exhaustively search a given RTL design for executions in which process isolation is violated,

2. A tool that, given an RTL file, automatically generates a formal testbench (FT) applying the approach above,

3. Discovery and disclosure of previously unknown covert channels in a RISC-V core and an accelerator,

4. Reproduction of known vulnerabilities using FV in a matter of minutes rather than days of RTL simulation,

5. Validation of a covert channel via a successful attack under RTL simulation.

1.4 Organization

The rest of the work is organized as follows:

Chapter 2 Background. A more in-depth discussion of covert channels, an overview of the formal verification techniques used in AutoCC, and related work.

Chapter 3 Design. AutoCC's threat model and approach, key implementation features for various hardware blocks, and automated FT generation.

Chapter 4 Evaluation. Getting up and running with AutoCC, refining the experimental model, and the covert channels we discovered.

Chapter 5 Conclusion Reflections on the relevance of our findings.

Chapter 2

Background

2.1 Covert Channel Attacks

Process isolation is fundamental to system security, and the primary mechanism by which information is confined to appropriate domains. While both side channels and **covert channels** can break isolation, the information leakage in the latter is deliberate. That is to say, there is a party—a Trojan horse, or Trojan—in the system that is actively using the covert channel to move information illegally across some security boundary. (In a side channel, the system does this on its own, and inadvertently.)

2.1.1 Trojans and Multilevel Security

Considering a covert channel in a multilevel security context more precisely characterizes the nature of this threat. A system implementing multilevel security should obey the *-property, which stipulates that no process may communicate to a level lower than its own (also known as no write down) [13, 14].

A user with a "secret" clearance may execute an untrusted process at its level to perform some function on privileged information. The executing process is considered a Trojan if, in addition to its advertised function, it also has a "clandestine function"—namely, communicating the information to a lower level, where it may be accessed by a user without the required clearance to do so [15].

This possibility was identified as early as in the 1970s when Lampson outlined several mechanisms by which a lower, untrusted process could leak client data to its owner, even when executing in a controlled environment [16].

2.1.2 Example: Spectre Variant I

In the Spectre vulnerabilities disclosed in 2018 [2], a snippet of code in the victim process—a so-called "gadget"—is manipulated by the attacker to serve as a Trojan. (Once located, it may be invoked by the attacker via the victim's interface.) The original paper gives the example of an indexed load wrapped in a conditional for Spectre variant 1:

```
if (x < array1_size)
    y = array2[array1[x] * 4096];
```

Figure 2.1: Spectre variant I gadget from [2].

Suppose that the attacker chooses `x` such that `array1[x]` resolves to a secret. The crux of the attack lies in speculative execution: if the read of `array1_size` in the first line incurs a cache miss, the processor may speculatively issue the read of `array2` in the second line to try and mask the miss latency. This read is transiently executed under the *false* assumption that the out-of-bounds check on `x` holds good. Although the check is ultimately resolved correctly, the speculative memory access has already affected the microarchitectural state of the machine—specifically, the value of `array2[array1[x] * 4096]` has been brought into the cache.

This creates a timing difference on cache access. Since the cache is a shared structure, the attacker may, within its own process, execute a Flush+Reload or Prime+Probe attack to determine which line contains the cached value of the `array2` access. Since the location of the line depends on the address that was accessed, from this the attacker may deduce the value of `array1[x]`, completing the attack.

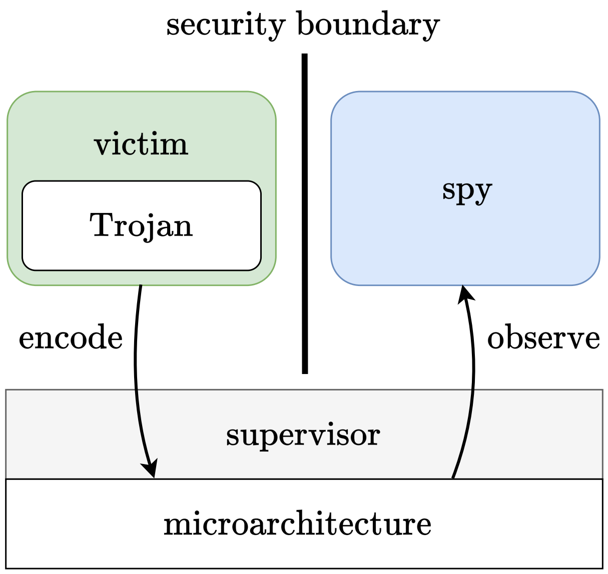


Figure 2.2: Operation of a microarchitectural covert channel. The Trojan in the victim process modifies—via permitted operations—microarchitectural state so as to somehow encode a secret. The spy process observes this modification, either directly or via a timing difference, to infer the secret.

Spectre variant 1 thus relies on a microarchitectural covert timing channel. *Microarchitectural* since the secret is encoded in the cache, *covert* since a Trojan is illicitly sending messages to an attacker, and *timing* since each message introduces and is identified by a timing difference.

2.2 Mitigating Covert Channels

A variety of countermeasures have been proposed to guard against covert channel leakage. Implementation details aside, a covert channel is fundamentally an information channel, one by which a Trojan sends a signal to be analyzed and received by an attacker. A countermeasure may then be broadly classified as one that eliminates the channel entirely or limits its bandwidth to some non-zero bound. (These classifications were codified by the U.S.

Department of Defense in the 1990s in its “Rainbow Series” [17].)

2.2.1 Elimination

§1.2 briefly alluded to asymmetric access paths—that is, the existence of fast and slow paths in a processor—as a source of information leakage. Enforcing the invariant that *all* operations take constant time is one way to eliminate leakage of this sort, effectively removing any dependence on the input and preventing an attacker from modulating the timing behavior of the processor. Constant-time implementation of cryptographic functions [18] is a prominent example of this approach in software.

Shared structures, and consequently shared state, are also known to leak information. The evident solution here is to fully and truly partition shared resources between processes, either by means of a flush in a concurrent system or by partitioning hardware in a truly parallel one [19]. This is the primary mitigation strategy evaluated in this work, and a specific implementation is discussed in more detail in §3.3.1.

2.2.2 Bandwidth Limitation

An alternative approach is to introduce noise into the channel so as to make an attacker’s measurements largely worthless [19]. Placement in a cache can be randomized [20] and high-resolution timers can be degraded and made “fuzzy” [21]. Adding delay between the sender and receiver is another method by which a channel’s bandwidth may be limited [17].

2.2.3 Mitigation Challenges

A significant difficulty across all these methods is maintaining performance. Asymmetric access paths and shared structures exist in the first place as optimizations; the covert channel mitigations we have discussed effectively disable them, often degrading performance to an unacceptable extent.

Mitigations are also very difficult to get right. This is especially true when working around silicon—a constant-time implementation of AES found it exceedingly difficult to exert the microarchitectural control necessary to close timing channels in the cache [22]. Hardware-based countermeasures fare no better: Meltdown and Spectre mitigations have been successfully circumvented by new attacks such as EchoLoad [23] and Foreshadow-NG [24].

Much of the difficulty may be attributed to the fact that these attempts are indeed *mitigations* rather than part of a more systematic security-centered design. Mitigations that are effective in one generation may fail in the next—a study of hardware supported mitigations across several vendors found that they inevitably fell short of completely closing intra-core channels [25].

As the previous paragraph hints at, finding the covert channels to mitigate in the first place remains the main challenge. Techniques for detecting covert channels like information flow tracking (IFT) often rely on RTL simulation. Such simulation is only as effective as the test cases provided; even a wide range of test cases will never cover all possible inputs. Furthermore, the execution traces generated are usually at the system level, making it difficult to pinpoint the specific hardware component that originated the leak.

2.3 A Word on Formal Verification

The approach we outline in Chapter 3 is based on formal methods. Although it relies on a tool with a closed source implementation, it is still worthwhile to say something of the basic theory behind model checking and distinguish it from other verification methods.

Given a system, what constitutes convincing evidence that it satisfies a specified property? A prover can either:

1. Exhaustively examine all possible states, or

2. Exhibit a proof.

Either one of these approaches, when successful,[1] represents the gold standard of correctness. Insofar as the formalization of the system is accurate, a positive result for a property is evidence on the strength of mathematics that it holds true. Properties may then be chosen and verified separately such that together, they imply the correctness of the system as a whole.

We will focus on model checking, a type of the first, state-based approach that is widely used in hardware verification. It offers two significant advantages: first, the logical task can be largely automated and carried out by a machine. Second, when a property violation is detected, a corresponding counterexample (CEX)—that is, an execution trace that led to an error state—can be produced.

2.3.1 Formalization

As a first step, both the system and any properties we wish to demonstrate must be formalized. It is natural to think of a system as a set of states S evolving over time, beginning from an initial state in $I \subset S$ and transitioning according to some relation $R \subset S \times S$. Given a set of atomic propositions P, the *semantics* of the model is obtained by labeling each state with a subset of P. (That is, if the mapping is $L : S \to 2^P$ and $s \in S$, $L(s)$ is the set of propositions that are true in s.) Together, the tuple $\langle S, I, R, L \rangle$ constitutes a **Kripke structure** that constitutes a formal model of the system [28].

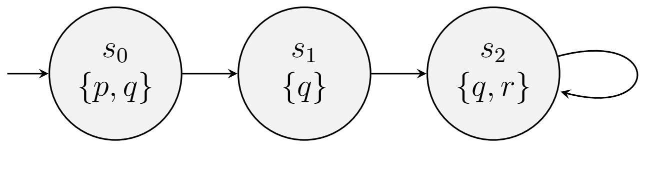


Figure 2.3: A Kripke structure over $P = \{p, q, r\}$. Note the self-loop is necessary as R must be left-total.

If we consider an execution in the Kripke structure, a **path** is the sequence of interpretations that corresponds to the sequence of states in the execution. Since these interpretations

[1]The property that a program halts is famously undecidable [26]. More generally, a theorem from Rice shows that any non-trivial semantic property of a program is undecidable [27].

are the truth values for propositions in P, we can extend propositional logic over the notion of a path, which is precisely what linear temporal logic (LTL) does.

Let π denote a path, and π^i the subsequence of interpretations $(\pi_i, \pi_{i+1}, ...)$. LTL allows us to express temporal properties about the system by extending the usual set of propositional logic operators with the following [28]:

1. **G**lobally ϕ: $\mathbf{G}\phi \iff \forall i.\pi^i \vDash \phi$
2. **F**inally ϕ: $\mathbf{F}\phi \iff \exists i.\pi^i \vDash \phi$
3. ne**X**t ϕ: $\mathbf{X}\phi \iff \pi^1 \vDash \phi$
4. ϕ **U**ntil ψ: $\phi\mathbf{U}\psi \iff \exists j.\pi^j \vDash \psi$ and $\forall i < j.\pi^i \vDash \phi$

An LTL formula is of the form $\mathbf{A}\phi$, where ϕ is given by the grammar of propositional logic extended with the operators above. Semantically, **A** captures the notion of system-level correctness: $\mathbf{A}\phi$ is true if and only if all paths from a given state satisfy ϕ.

AutoCC's correctness properties are expressed using SystemVerilog Assertions (SVA), a decidable fragment of LTL. As an example, we may wish to ascertain that every read request is eventually transmitted. This is expressed in SVA as:

```
assert property (axi_read_outstanding_req |-> s_eventually(axi_r_hsk))
```

which is equivalent to the following LTL formula:

$$\mathbf{AG}(\texttt{axi\_read\_outstanding\_req} \implies \mathbf{F}(\texttt{axi\_r\_hsk})).$$

The full set of relevant properties is discussed in §3.3

2.3.2 Bounded Model Checking

The problem of model checking is that of determining the validity of an LTL formula in a Kripke structure [29]. It asks: given a Kripke structure M and an LTL formula $\mathbf{A}\phi$, does

$M \models \mathbf{A}\phi$? That is, does $\pi \models \phi$ for all paths π with π_0 corresponding to $s \in I$?

Even for finite-state systems, the number of states can be prohibitively large. In general, the model state size grows exponentially with the number of state variables; this so-called state explosion problem is the main difficulty in model checking [30]. The problem is intractable in the worst case, but we can fare much better by checking the model symbolically rather than explicitly enumerating every state.

Bounded model checking (BMC) was a significant breakthrough in this area, and has come to be the predominant method used by industrial-strength model checkers today [30]. Instead of operating on the states of a Kripke structure directly, the BMC algorithm produces a propositional formula by "unwinding" the LTL formula to a number of transitions k. This reduces the problem of model checking to an instance of propositional satisfiablity (SAT), for which powerful decision procedures—namely CDCL [31]—exist [29].

To see this in more detail, suppose that we want to check the property $\mathbf{AG}p$, where p is an atomic proposition, for a bound of length 2. Since $\mathbf{AG}p$ is valid if and only if $\neg\mathbf{AG}p = \mathbf{EF}\neg p$ is not, the BMC procedure searches for a path *prefix* of length 2 that satisfies $\mathbf{EF}\neg p$; success means that the property fails to check. Let $I(s)$ be true if and only if $s \in I$, and $p(s)$ be true if and only if $p \in L(s)$. The equisatisfiable propositional formula for $\mathbf{EF}\neg p$ is then

$$I(s_0) \wedge R(s_0, s_1) \wedge R(s_1, s_2) \wedge (\neg p(s_0) \vee \neg p(s_1) \vee \neg p(s_2)).$$

The above is satisfied when p is false in at least one state in a valid path prefix of length 2. The formula may be generalized to paths of length k as

$$I(s_0) \wedge \bigwedge_{i=0}^{k-1} R(s_i, s_{i+1}) \wedge \bigvee_{i=0}^{k} \neg p(s_i).$$

As Figure 2.4 illustrates, the BMC procedure can use the SAT solver as a fully automated black box. (Greater performance, however, usually means tighter integration with the SAT solver.) Since a SAT result returns a satisfying assignment for the BMC formula, we can easily recover the counterexample that led to the property violation.

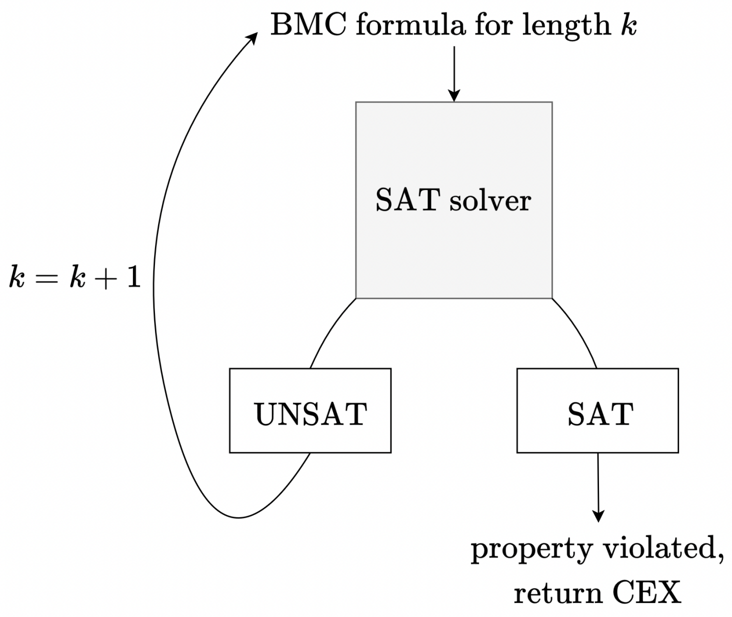


Figure 2.4: Overview of the BMC procedure. On every iteration that the SAT solver returns UNSAT, the formula is unwound to one more transition.

While it is clear that bounded model checking is sound, its completeness is not as apparent. An UNSAT result from the SAT solver implies that the LTL formula is valid for path prefixes of length less than or equal to k, but a counterexample of length greater than k may yet exist. To conclude that $M \models \mathbf{A}\phi$ and definitely solve the model checking problem, k must reach a completeness threshold. Clearly $|S|$ is one such threshold; a tighter one is the diameter of the system, that is, the length of the shortest path between the two states furthest apart in M [32]. In practice, reaching this completeness threshold is not always possible; the checker may run out of time or space, or the threshold itself may be hard to compute. In such cases BMC still retains much value as a bug hunting tool.

2.4 Related Work

AutoCC's formal verification approach was first proposed by Marcelo Orenes-Vera in a submission to the Young Architect Workshop (YArch) [33].

The AutoCC testing framework is built upon a formal verification testbench from AutoSVA, a tool that automatically generates SystemVerilog assertions to evaluate control logic for liveness and safety properties [34].

Prior work applying formal verification to the problem of detecting hardware security vulnerabilities include InSpectre [35] and Unique Program Execution Checking (UPEC) [36]. InSpectre creates a formal model of a processor—including its microarchitectural state—to detect Spectre-like attacks that combine speculative execution with a covert channel. UPEC uses FV to detect memory leakage stemming from the side effects of non-permitted operations. AutoCC extends the scope of prior work by seeking to detect *any* information leakage that can occur via microarchitectural state.

Chapter 3

Design

This chapter details the actual design and implementation of AutoCC, beginning with the assumptions in our threat model and working to a high-level overview of an AutoCC experiment. It concludes with a more detailed discussion of how some of AutoCC's key features were implemented and extended.

3.1 The AutoCC Threat Model

The AutoCC threat model assumes two processes, an **attacker** (spy) and a **victim**, separated by a context switch and executing on the same physical core. More precisely, the processes time-share the core in a manner governed by a **supervisor** such as an operating system. Both processes are untrusted, and the victim process executes in a controlled environment much like the one discussed in §2.1.1. In such an environment, the supervisor restricts the clients with which the victim may communicate.

The attacker process possesses no special privileges and executes in a security domain of its own. Its primary asset is a **Trojan** in the victim process that attempts illicit communication via a microarchitectural covert channel. Executions of the victim and the attacker are separated by a context switch provided by the supervisor. Since the processes are located in different security domains, the microarchitectural state should not leak data from the victim to the attacker.

No assumptions are made on whether the Trojan is a malicious hidden function of the victim process or innocent code that leaks data inadvertently. This allows us to prove stronger correctness assertions—hardware free of covert channels must also be free of side channels.

At the most fundamental level, AutoCC is concerned with the flow of information between two sets across time. Let the **state** of a core be defined as the set of all the memory cells contained within the design and each of its instantiated submodules. The ISA, by exposing certain operations to software, partitions the state of a core into **architectural state** and **microarchitectural state**.

Definition. The **architectural state** of a core consists of all the elements in the state that are directly readable via the ISA.

Definition. The **microarchitectural state** of a core consists of all the elements in the state that are not directly readable via the ISA.

A process executing on a core will naturally alter both. Accordingly, but not always, the isolation of these states to the processes to which they belong is a responsibility shared by both software and hardware. Well-implemented software (i.e. a supervisor) will (1) guard architectural state that is accessible only under privileged mode and (2) swap architectural state before another process begins execution. Well-designed hardware will partition or flush any microarchitectural state that could leak data from the architectural state of one process to that of another. In these terms, AutoCC *assumes* the correctness of the supervisor and *asserts* the correctness of the hardware mechanism for microarchitectural state isolation. Since time-shared hardware is the focus of this work, the specific mechanism that we evaluate is a microarchitectural flush.

3.2 The AutoCC Approach

AutoCC leverages a model checker to detect the existence of illicit communication between a victim process and a spy process. There exists a considerable variety of software today that offers the model checking capabilities discussed in §2.3; our implementation uses JasperGold, an industrial-strength verification tool from Cadence.

An experiment begins by initializing two identical instances of the Design Under Test (here CVA6) from reset in universes α and β. The inputs to both instances are set free and independent of each other and for some arbitrary number of cycles, each may take on any legal execution. These executions represent all possible behaviors of the victim process, in particular that of the embedded Trojan.

At some point—and not necessarily synchronously—a context switch begins in α and β. Although the start of the context switch may be staggered, the end of the context switch 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) the architectural states of both α and β are the same and (2) the microarchitectural states have been flushed.

With these two conditions met α and β are assumed to now both be executing the same process, namely the attacker process that was just switched in. The inputs to both are set equal to ensure that any observed divergence is only the result of state internal to α and β. It is in this post-context switch world that we assert our correctness condition: on every cycle, the architectural state of α should equal the architectural state of β.

What would it mean if this assertion were violated? This would mean that on some cycle following the context switch α and β diverged in an observable way, and that this discrepancy must be attributed to their differing executions prior to the context switch. That is to say, there was microarchitectural state that persisted through the context switch and influenced subsequent execution—a microarchitectural covert channel.

We made brief mention of the completeness of bounded model checking in §2.3.2, and

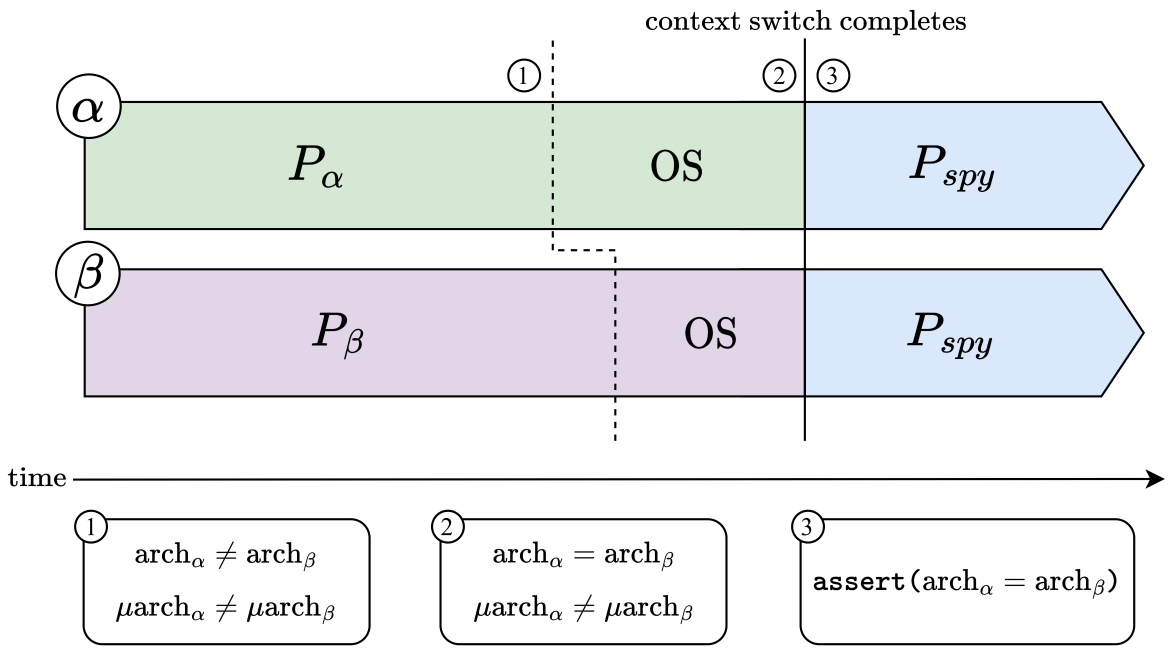

context switch completes
1
2
3
α
P_α
OS
P_{spy}
β
P_β
OS
P_{spy}
time
1 $\text{arch}_\alpha \neq \text{arch}_\beta$ $\mu\text{arch}_\alpha \neq \mu\text{arch}_\beta$
2 $\text{arch}_\alpha = \text{arch}_\beta$ $\mu\text{arch}_\alpha \neq \mu\text{arch}_\beta$
3 assert($\text{arch}_\alpha = \text{arch}_\beta$)

Figure 3.1: An overview of our verification methodology. The victim processes P_α and P_β are first free to take on any legal execution for an arbitrary number of cycles. At the end ① of this phase, the architectural and microarchitectural states of α and β may differ. AutoCC then models a context switch, modeling both the OS operations that restore the architectural state of the incoming spy process and the microarchitectural flush. At the end ② of this phase, the architectural states of both α and β are guaranteed to be the same, but differences in microarchitectural state may still remain. We assert our correctness condition once P_{spy} begins execution ③, ensuring that any differences in microarchitectural state at ② do not cause observable differences.

it is worth revisiting that discussion in the context of an AutoCC experiment. If JasperGold detects a violation of our correctness property, it will return a CEX on some cycle n, representing a path of length n in our model—which includes both α and β—that led to a state where the negation of our property held true. A verification engineer may then analyze the CEX, stepping backwards in time to see where the difference originated in the execution preceding the flush. If JasperGold fails to detect such a violation before running out of space or time, or reaching a completeness threshold, all that we may conclude is that no n-cycle execution of our system suffices to successfully operate any microarchitectural covert channel that may exist.

3.3 Implementing AutoCC for CVA6

AutoCC must necessarily be configured for a given RTL design. Our first implementation was built around CVA6 (formerly known as Ariane), a 6-stage, in-order, 64-bit RISC-V core that boots Linux and was taped out in GlobalFoundries 22nm process in 2017 [37].

3.3.1 The CVA6 Processor

§2.2.1 discussed several approaches to eliminating covert channels. CVA6 adopts the strategy of **temporal partitioning**, which separates processes in time by clearing stateful components on a context switch. This is accomplished via a temporal fence instruction `fence.t`; CVA6 implements three different versions of `fence.t` that flush varying degrees of microarchitectural state. Our work evaluates two of these three versions; detailed findings are reported in Chapter 4.

CVA6 offers several advantages that make it well-suited for AutoCC beyond its open source RTL. As an application-class processor capable of running a full operating system its complexity is significant, bringing it closer to the motivating cases in §1.2 than, say, a smaller embedded system. Its cache subsystem is easily configurable in size, which is useful for controlling the size of the search space in formal verification.

The rest of this section describes how some of AutoCC's key features were implemented for CVA6. Much of the work was in separating the herrings from the marlins, that is to say, in guiding the verification tool by way of assumptions and preconditions to only those parts of the search space that correspond to the execution model described in §3.2.

3.3.2 Defining the Context Switch

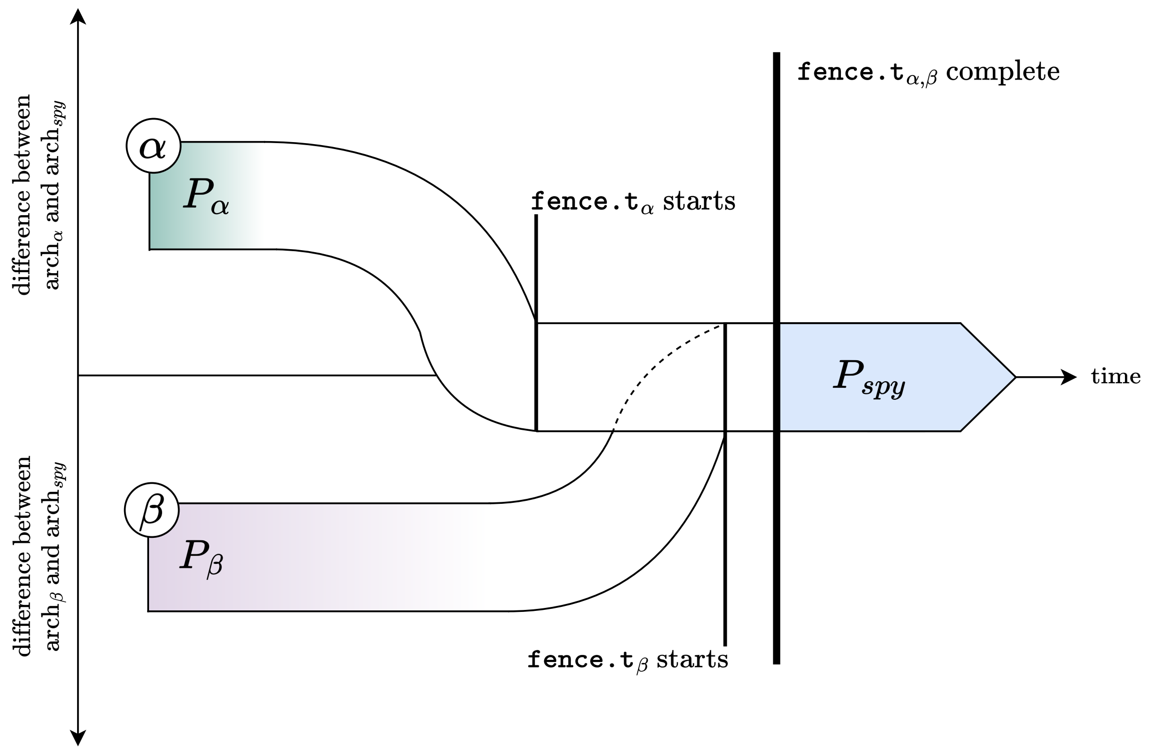
As mentioned earlier, the end of the context switch is meant to serve as a synchronization point between α and β. While flushing the microarchitectural state is a hardware operation, the architectural state must be handled by a supervisor. Specifically, CVA6 assumes that any state in the architectural whitelist—namely the register files and the CSRs in Figure 4.1—is saved and restored by an OS on a context switch. As such, there is no flush signal tied to the microarchitectural flush for any of these modules; their transition across a security domain is delegated to the supervisor.

With processes in JasperGold effectively running on bare metal, it is necessary to mimic the supervisor services that are involved in a context switch. In doing so, we make the key observation that the operations required by a context switch will depend on the prior executions of α and β. As such, the context switch itself may be modeled as part of the victim process's execution; there is no need to draw a bright line between the execution of the victim process and that of the supervisor.

Following such an approach offers two significant advantages. First, it allows AutoCC to remain agnostic to the details of the supervisor implementation; there is no need to specify a particular procedure for how the OS should handle its part of a context switch. Second, it allows AutoCC to keep constraints to a minimum; no assumptions are made on what instructions comprise a context switch or even when such a context switch should begin. All that is required is that α and β each follow *some* execution that *eventually* leads them to converge on the same architectural state, at which point a microarchitectural flush occurs. (And even this microarchitectural flush need not start on the same cycle in both universes.)

It is worth noting here that choosing not to specify a routine for a context switch necessarily results in an underconstrained system model. That is, AutoCC may generate CEXs that are not exploitable given a particular OS implementation. On balance, however, we think the choice to remain OS-agnostic is a valuable one; hardware is not always yoked to the same supervisor implementation, and it is useful for the hardware designer to be aware of the channels that exist, even if they are guarded by software.

The end of the microarchitectural flush is a natural synchronization point for α and β. As an event that is guaranteed to occur as part of the context switch in both universes, aligning α and β on the cycle on which it occurs gives us a clean starting point for the execution of



difference between $arch_\alpha$ and $arch_{spy}$
difference between $arch_\beta$ and $arch_{spy}$
α
P_α
β
P_β
`fence.t`$_\alpha$ starts
`fence.t`$_\beta$ starts
`fence.t`$_{\alpha,\beta}$ complete
P_{spy}
time

Figure 3.2: How AutoCC models the context switch event. Instead of enforcing a discrete jump to a specific sequence of OS instructions, we simply require that the victim processes in α and β eventually converge to the architectural state of the incoming spy process. (Indicated here by P_α and P_β converging to 0 on the y-axis.) Since the temporal fence is the last thing that executes before P_{spy} begins, this convergence must occur by the start of the `fence.t` instruction. Note, however, that `fence.t`$_\alpha$ and `fence.t`$_\beta$ are free to start on different cycles; we only require that they complete together.

the spy process. Interpreting this point as the end of the context switch also allows us to much more easily express the invariants that should hold after such a switch completes.

3.3.3 Measuring Context Switch Latency

For all of its advantages, taking the end of the flush as the synchronization point between α and β does admit one significant blind spot. Specifically, the assumption that the flushes in both universes complete on the same cycle 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 latency itself may function as a covert channel.

AutoCC can still verify 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. A discussion of these CEXs can be found in Chapter 4.

3.3.4 Partitioning the Pipeline State

In our model, we consider the instructions that begin executing immediately after the microarchitectural flush as part of the spy process. In reality, the flush would be followed by further supervisor operations to transfer control to the incoming spy process. When we arrive at this point, however, our preconditions guarantee that the architectural state of the incoming process has been restored and that the microarchitectural flush has completed. That is to say, all of the relevant functions of the context switch have been performed; we may consider the OS operations immediately after `fence.t` as part of the spy process, just as we consider the OS operations immediately before `fence.t` as part of the victim process. Conceptually, the `fence.t` instruction should move through the pipeline like a wavefront, serving as a barrier between the instructions before and after it and a dividing line between the victim and spy processes.

This interpretation necessitates special care in managing the state of the pipeline. If the instruction following the flush is the first instruction in the spy process, it is clear that this instruction should be the same in both α and β. The fetch of the instruction's address, however, has already occurred by the time that α and β synchronously commit the microarchitectural flush. Since this cache access occurs before any inputs are constrained, α and β are primed with different addresses for this instruction, leading to a spurious CEX when it executes on a later cycle.

In essence, the gap in the pipeline between when the flush is decoded and when the flush commits must be covered by an SVA assumption; we accordingly assume that when the flush commits, all of the instructions behind it in the pipeline are the same in both α and β.

3.3.5 Improving Performance via Blackboxing

The specter of the state explosion problem §2.3.2 is always present in formal property verification. As a baseline mitigation, AutoCC adopts the standard technique of minimizing any modules that are instantiated with a size parameter, namely any TLBs and caches.

We also blackbox certain modules, effectively removing them and their submodules from the Design Under Test (DUT). To the verification engine, the internals of a blackboxed module do not exist, reducing its search space. Accordingly, a blackboxed module does not follow any meaningful state evolution; any input it receives is effectively discarded and any output it produces is random.

Since α and β will both contain an instance of the blackboxed module, it is necessary that both instances are consistent in their randomness to avoid triggering false CEXs. To that end, we set the outputs of any instances of a blackboxed module to be equal after the context switch has completed.

In choosing which modules to blackbox, we consider both the size of the module and the nature of the state that the module contains. These dual considerations lead us to blackbox the Control and Status Registers (CSRs) in CVA6. First, the CSRs are quite large—CVA6 implements 71 CSRs, each as a 64-bit register. Second, the CSRs are defined as part of the processor's architectural state; since our threat model assumes that the supervisor is functioning as intended, CEXs that originate in the CSR module are not valid for our purposes. Blackboxing allows us to easily and concisely remove these CEXs without placing constraints on the internal state of the CSR via SVA assumptions.

The fine print here is that blackboxing alone is too blunt an instrument for the task; it not only removes any CEXs that originate in the CSR module, but also any that *traverse* it. That is, we will fail to detect valid CEXs that originate in the microarchitectural state of the core and become visible via a path through the CSRs. This is remedied by asserting that each input to the CSR module (which is an output from the core) is the same in both α and β after the context switch completes.

If no assertions fail, then there is no output from CVA6's microarchitectural state that can create a timing difference through the CSR module. If one of these assertions does fail, it is *possible* that the CSR module is on the path of a valid covert channel. A verification engineer can then either (1) bring the CSR module back into the DUT or (2) wait for the covert channel to generate a CEX via another pathway that does not touch the blackboxed module. In the latter case the channel may still be analyzed by its operation on the alternative access path; it is only when AutoCC fails to generate an alternative CEX that instantiating the CSR module becomes necessary.

Although possible in theory, we chose not to apply the same approach to the register file, mainly because of how tightly integrated it is with the pipeline. Since we expected most valid CEXs to inevitably touch the register file, the advantages of blackboxing seemed minimal, especially given its small size.

As a last performance consideration we explored the idea of modifying the reset signal to assign random values to the cache lines and registers, in the hopes of generating shallower CEXs. In practice we found running times to be short enough to make this unnecessary.

3.3.6 Observation Model

Keeping with our threat model of separate processes time-sharing the same hardware, we have cast the AutoCC correctness condition in terms of architectural state consistency between α and β. This leads to the following two observations.

First, what AutoCC really detects is not just microarchitectural timing channels—the original motivation of the work—but *any* microarchitectural covert channel that can be exercised given a set of user-defined assumptions on the system. That is to say, it detects any same cycle difference (modulo assumptions) in process-observable state that originates from a difference in microarchitectural state. The mechanism for this data leakage need not always be a timing difference; Chapter 4 includes covert channels that can operate without this further level of indirection.

Second, as long as there are ISA instructions that allow a process to expose *any* subset of architectural state to output, we can assert an equivalent correctness condition just on the outputs of α and β without reasoning about their internal states. Any difference in architectural state on cycle k can, by a sequence of these instructions, be externalized by the SMT solver as a difference in outputs on cycle $k + n$ for some bounded n.

Of course, this abstraction is not always valuable for designs like CVA6 where the architectural state must be identified and constrained in defining the context switch. However, there are hardware modules where this may not be the case; that is, where the presence of a supervisor is not assumed and the hardware is fully responsible for the state that must be flushed.

3.4 To Hardware IPs Beyond CVA6

Nothing intrinsic to our threat model or experimental setup limits AutoCC to CVA6, or even CPUs more generally. If we consider the instructions executing before and after the context switch as simply belonging to separate processes, AutoCC may naturally be extended to any functional unit that is accessed by multiple processes concurrently. An accelerator, for example, is often shared between processes in a time-multiplexed fashion. By considering the set of operations available to such a specialized hardware block as an ISA [38], we can define architectural and microarchitectural state as in §3.1 and apply the very same methodology.

Chapter 4 thus contains results not only for CVA6, but also for the following:

1. Vscale, a 32-bit RISC-V core [39],
2. MAPLE, a memory access engine [40],
3. An accelerator for AES encryption [41].

The rest of the work will accordingly use the term Design Under Test (DUT) to refer to the top-level hardware module that AutoCC is testing, regardless of its level of specialization.

3.4.1 *Auto*CC: Automated Property Generation

The AutoCC configurations for these hardware blocks do not differ significantly from the one for CVA6 discussed in detail above. In fact, the approach is general enough to be almost fully automated. Implemented as an extension of AutoSVA [34], a tool that generates formal testbenches (FTs) for RTL module transactions, AutoCC can generate a minimal FT for covert channels in under a second.

Given a DUT, AutoCC instantiates two copies of the design as submodules α and β in a top-level wrapper. Then, following the generalized observation model in §3.3.6, AutoCC parses the DUT's interface, generating the appropriate SVA properties for each input and output signal. Once the spy process begins executing, any inputs to α (`a`) and β (`b`) are assumed to be equal:

```
// Every input to the DUT generates an assumption.
wire input_i_eq = (a.input_i == b.input_i);
assume property (spy_exec |-> input_i_eq);
```

and any outputs are asserted to be equal:

```
// Every output from the DUT generates an assertion.
wire output_i_eq = (a.output_i == b.output_i);
assert property (spy_exec |-> output_i_eq);
```

3.4.2 Assertions on Transactions

RTL annotations indicate to the tool any signals—such as clock and reset—that should be shared between α and β. Annotations also identify any signals that are part of a **transaction**, that is, a group of signals governed by a valid signal. The correctness assertions generated for outputs in a transaction must be qualified with their accompanying valid signals—differences in invalid data should not generate a CEX.

```
// Any assertion on an output signal in a transaction should be
↪  preconditioned on the transaction's valid signal.
wire output_transact_valid_eq =
    (a.output_transact.valid == b.output_transact.valid);
assert property (spy_exec |-> output_transact_valid_eq);
wire output_transact_data_eq =
    !(a.output_transact.valid) ||
     (a.output_transact.data == b.output_transact.data);
assert property (spy_exec |-> output_transact_data_eq);
```

For inputs in a transaction, we only assume that the data payloads are equal in α and β if the transaction is valid; this allows AutoCC to detect RTL modules that incorrectly make use of invalid data.

3.4.3 Introducing a Transfer Period

By default, AutoCC does not identify the microarchitectural flush event or the set of architectural state:

```
// High when a and b both complete their flush operations. Set
↪  free by default (can occur at any time).
wire flush_done = 'x;

// High if the architectural states of a and b are the same. Set
↪  to TRUE by default.
wire arch_state_eq = 1'b1;
```

Depending on the DUT, the user can modify these signals to reflect

1. When a flush is deemed to have completed and
2. What state elements are architectural.

The first of these can be tricky to nail down—unlike the security-minded CVA6 core, some DUTs do not have a well-defined signal for when the flush completes, and some do not have a flush instruction at all.[1]

Even so, certain DUTs are easier than others. For instance, the AES accelerator we evaluate is designed under the assumption that there are no ongoing operations in the pipeline when a new process begins execution. That is to say, each stage of the pipeline must be idle when a new process begins; for this DUT and similar ones, flush completion can simply be defined as an idle pipeline.

The task is harder for DUTs that have neither a flush nor an idle signal. For these DUTs, we heed the old adage that time heals all wounds. That is to say, instead of (1) searching for a flush completion signal and (2) constraining the pipeline state around it, we simply pad out the end of the context switch to give α and β more time to converge. More precisely, after the flush completes—however nebulously that event is defined—we observe a transfer period during which the inputs, outputs, and architectural states of α and β remain the same, allowing the pipeline stages in both universes to converge.

The length of this transfer period is a configurable 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 cycle count of the OS operations between the end of the flush and the start of 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.

```
// Length of the transfer period.
localparam THRESHOLD = 4;

// The architectural states, inputs, and outputs of a and b are
↪  all held to be equal during the transfer period.
wire transfer_cond = arch_state_eq &&
                     input_i_eq && output_i_eq &&
                     output_transact_valid_eq &&
                     output_transact_data_eq;
```

[1]In the case of CVA6, we essentially use the flush commit as a proxy mechanism for partitioning the pipeline (§3.3.4). Without the assumptions tied to the flush commit, there is nothing to prevent two processes in the pipeline from interacting in a manner that lies outside of our threat model.

```
// Counts the number of cycles we have been in the transfer
↪  period so far (i.e. the number of consecutive cycles on which
↪  transfer_cond has held following the flush).
reg [$clog2(THRESHOLD):0] eq_ctr;

// High once the flush has completed in a and b AND the transfer
↪  period has elapsed.
reg spy_exec;

always_ff @(posedge clk) begin
    if (reset) begin
        spy_exec <= '0;
        eq_ctr   <= '0;
    end else begin
        spy_exec <= (transfer_cond && (eq_ctr>=THRESHOLD)) ||
                    spy_exec;
        eq_ctr   <= transfer_cond && (flush_done || eq_ctr>0) ?
                    eq_ctr+1 : '0;
    end
end
```

3.4.4 Modular AutoCC

Many of the specialized hardware IPs we now target are likely to be part of a larger heterogeneous system-on-chip (SoC). Since covert channels can originate from any microarchitectural state that a victim process touches, AutoCC should ideally be applied to all of the RTL modules a process can access during execution. On SoC designs of realistic size, however, state explosion makes this infeasible.

In addition to minimizing module sizes, in such cases we may apply AutoCC in a modular fashion. This is very similar to the blackboxing approach we discussed in §3.3.5. Conceptually, blackboxing may be thought of removing a submodule from a DUT while keeping the wires between them intact. Modular application, on the other hand, creates a new testbench in which a submodule of the DUT has been promoted to the top-level.

In both cases, a subset of the DUT is abstracted away and the AutoCC assertions are placed on the inputs and outputs of the subset that remains. Since the DUT contains less state, this reduces the complexity of the verification problem significantly. Furthermore, the depth of exploration required to exercise the relevant features of the DUT is reduced, since the FV tool drives any inputs directly.

On the other hand, the CEXs that are found are less informative, since we do not know how the inputs to the DUT were produced. In addition, the CEXs are more likely to be spurious, since the inputs are not constrained to only legal values. An AutoCC user, however, could create assumptions on the range of permitted legal values, e.g. the DUT does not receive an AXI response without first sending an AXI request.

As stated before, both blackboxing and modular application are blunt instruments; in general, their usage will require recovering the relevant details of the modules that have been abstracted away.

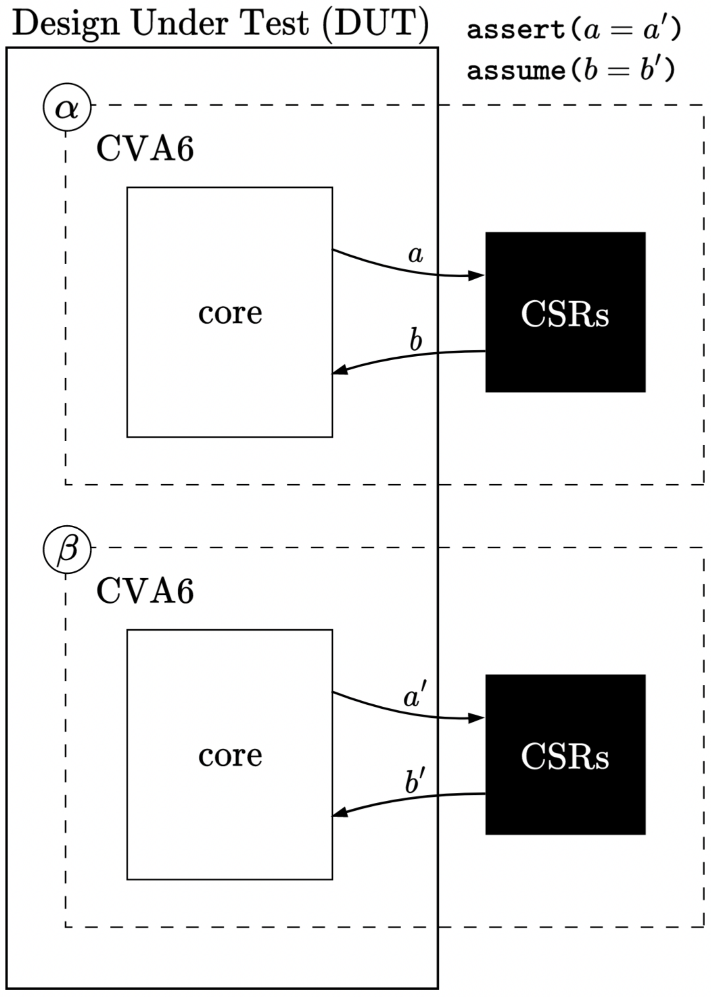


Figure 3.3: Blackboxing the CSRs. The top-level module contains two instances of CVA6, one for α and one for β. The CSR modules in both instances are blackboxed; since arbitrary inputs to the core from the CSRs are now permitted, we must assume that these inputs are the same in both universes to avoid spurious CEXs. Outputs to the CSRs are now considered outputs of the DUT, and must be checked accordingly.

Chapter 4

Evaluation

During this period, the business of discovering new TEMPEST threats progressed more swiftly than the art of suppressing them. Perhaps the attack is more exciting than the defense. At any rate, when they turned over the next rock, they found the acoustic problem under it.

—National Security Agency,
TEMPEST: A Signal Problem

We begin this chapter with an example of how to get AutoCC up and running, using Vscale as our DUT. We then discuss several notable CEXs that were encountered while evaluating CVA6, MAPLE, and the AES accelerator. A summary of these results, along with their cycle depths and runtimes, is given in Table 4.1. CVA6 and MAPLE remain under active development; the vulnerabilities we discovered were disclosed to the respective maintainers of each project.

4.1 Vscale: Getting AutoCC Up and Running

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

Table 4.1: CEXs detected by AutoCC in various DUTs (FV tool: JasperGold 2021.03). CVA6† is the full flush and CVA6 is microreset.

DUT	**Description**	**Depth (cycles)**	**Runtime (min)**
Vscale	V5. Interrupt in WB stage stalls pipeline	9	1.6
CVA6†	C1. Flush masked by earlier flush	20	1.2
CVA6†	C2. Outstanding AXI request	25	2
CVA6	C3. Leaks invalid I$ data	64	24
CVA6	C4. Buggy FSM transition in PTW	68	344
MAPLE	M1. Leaks whether TLB was disabled	14	0.5
MAPLE	M2. Leaks value of a configuration register	23	158
AES	A1. Request in the pipeline during switch	42	0.1

determination of which ones pose real threats in the hands of the person who knows the system best. By keeping the set of constraints minimal and remaining OS-agnostic, AutoCC imposes the minimum standard of exploitability on the DUT; anything further runs the risk of masking a real covert channel on another design.

This emphasis on sensitivity does, however, lead to "red herrings"—CEXs that result from input sequences to the DUT that are unreachable in the full system. (By full system we mean the DUT operating alongside its interfaced modules and a supervisor.) These false CEXs may be seen as arising from a mismatch in the mapping between AutoCC's model of the system and the real world. Iterative analysis of these CEXs can lead an AutoCC user to sharper and sharper definitions of what is and what is not exploitable in practice.

As a concrete example, the first CEX we observed in Vscale was caused by a jump register instruction. Since AutoCC defines the set of architectural state to be empty by default, the register file—which was neither flushed nor restored—was directly leaking data to the spy process. The register files of α and β thus had different values coming out of the context switch, and the FV engine externalized this difference via a jump register instruction. This led us to add the constraint that `pipeline.regfile.data` is the same in α and β after the context switch.

This condition could certainly have been added without running AutoCC at all. However, we chose to follow a CEX-driven refinement procedure for the following three reasons:

1. The CEX led us to the exact signal that needed to be accounted for without us having to reason comprehensively about Vscale's internal design.

2. The CEX validates the methodology's ability to detect covert channels that stem from unflushed state.

3. It is a FV best practice to start with the simplest precondition possible so as to avoid overconstraining the state space exploration.

After adding this new constraint, the next CEX we observed was also caused by a jump register instruction. This time, however, the register had been loaded with a value from the CSR module instead of being written to directly. Since it is the supervisor's responsibility to manage the CSRs, they should also be considered as part of Vscale's architectural state. Instead of adding a constraint as we did for the register file, however, we blackboxed the CSR module in the manner described in §3.3.5.

CEX V3 was caused by a difference in the PCs between α and β, leading to the fetch of the next instruction to differ. (This is very similar to the CEX described in §3.3.4 for CVA6.) Adding the PC registers along the pipeline to Vscale's architectural state closed this channel.

CEXs V4 and V5 stemmed from Vscale's lack of a clearly defined temporal fence instruction, which led to illegal inter-process interaction in the pipeline itself. CEXs of this sort were precisely the motivation for introducing the notion of a transfer period, and we applied the technique described in §3.4.3 here. The FV tool struggled to generate further CEXs after CEX V5; when it reached our time limit of 24 hours it had achieved a bounded proof depth of 21 cycles. Considering how small Vscale is (no caches at all!) and how shallow the previous CEXs had been, we thought it unlikely that the FV tool would find further CEXs even if given more time.

Table 4.2: CEXs found in Vscale, starting from the default AutoCC testbench (FV tool: JasperGold 2021.03).

DUT	Description	Depth (cycles)	Runtime (sec)
Vscale	V1. Jump to address from RF	6	1.1
Vscale	V2. Jump to address from MTIMECMP CSR	6	2.9
Vscale	V3. Different PCs in pipeline	7	0.9
Vscale	V4. Different register values in Decode stage	7	1.1
Vscale	V5. Interrupt in WB stage stalls pipeline	9	99

A summary of the Vscale CEXs is given in Table 4.2.

4.2 Covert Channels in CVA6: Full Flush

As discussed in §3.3.1, CVA6 implements three different versions of its microarchitectural flush. Our work began with the second implementation, the **full flush**, which clears the "components used by the prime-and-probe attack... the L1 data and instruction caches, the TLBs, and the branch predictors" [42]. Any dirty state is written back and prefetchers, write-buffers, and the pipeline are cleared. The full flush also targets deeper and smaller components such as linear-feedback shift registers and arbiters that were empirically identified as being possible sources of data leakage.

4.2.1 Flush Masked by Earlier Flush

As we have just said, the `fence.t` instruction flushes the instruction cache. But what if the instruction cache is already in a flush?

This is precisely the scenario that AutoCC stages in the following CEX. The instruction caches in α and β receive the flush signal `flush_i` on the same cycle. In α, `flush_i` goes high when the instruction cache is in the `READ` state; accordingly, it transitions to `IDLE` then `FLUSH`. β, however, is in the middle of its own flush—initiated 3 cycles earlier—when `flush_i` is asserted. Its instruction cache is already in the `FLUSH` state, and the second flush signal from the `fence.t` instruction is masked by the first and lost.

Since CVA6 has a `fence.i` instruction that allows user code to flush the instruction cache, this behavior could be used by a Trojan to affect the latency of a context switch.

4.2.2 The AXI4 Bus Protocol

CVA6 implements the Advanced eXtensible Interface 4 (AXI4) bus protocol as its memory interface, defining five channels through which the processor may communicate with memory. To make a request, the processor places the request on the appropriate channels and waits for memory to respond. If a handshake occurs, the request is considered as having been completed. An outstanding request, however, may be killed before completing on account of a branch misprediction or a flush.

One of the first CVA6 CEXs that we found—and which showed up repeatedly—was caused by executions where α had an outstanding AXI 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, which was making the request, transitioned to a `KILL_MISS` state while β's remained in `IDLE`.

A natural solution is to stipulate that the microarchitectural flush must first wait for all outstanding AXI requests to complete. This requirement, however, can itself become a source of data leakage; if we consider the operation of a write-back cache, for example, the number of outstanding AXI requests—and by extension the latency of a context switch—would vary directly with the number of cache lines written by the Trojan.

The CEXs we see here generally stem from differences in the latency of the flush itself[1] across α and β. If a Trojan is able to modulate that latency as in the example above, a spy process with access to a relatively accurate timer, even a system clock, could measure the timing difference to establish a communication channel.

[1]These CEXs are detected by constraining the flush event to begin at the same time in both α and β; see discussion in §3.3.3.

4.3 Covert Channels in CVA6: Microreset

This problem of variable latency detected by AutoCC is corroborated by the more empirical findings in [42]. In fact, it is one of the main motivations for the third implementation of CVA6's microarchitectural flush: **microreset**. Unlike the full flush, microreset targets *all* non-architectural state by default rather than attempting to identify a subset of vulnerabilities (Figure 4.1). Only the architectural register files and the CSRs are excluded.

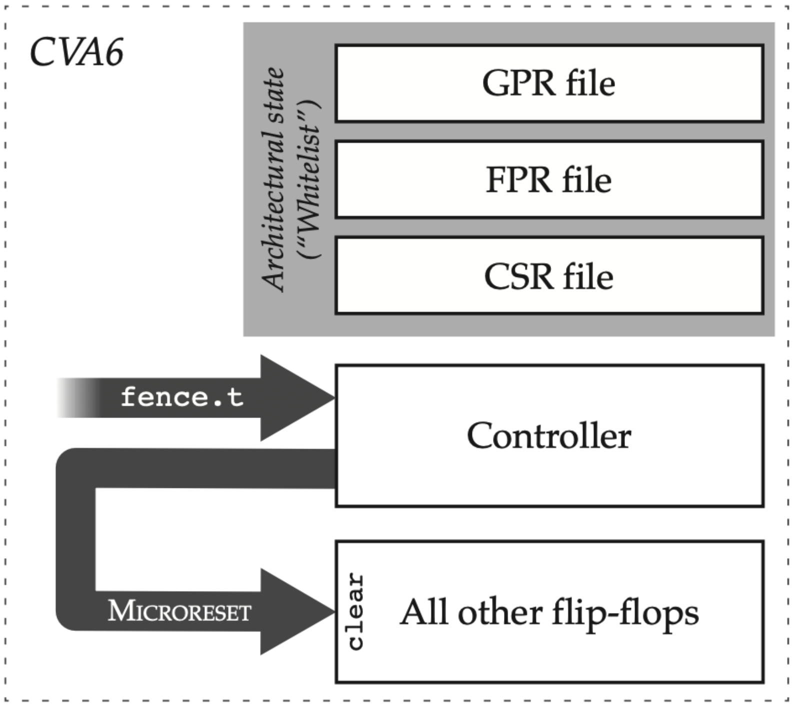


Figure 4.1: State cleared by the CVA6 microreset instruction. Figure taken from [42].

Microreset also enforces that the `fence.t` latency be independent of any previous execution. Although the RISC-V context switch routine is initiated by a timer interrupt generated on a fixed period, instructions following the timer interrupt may be delayed by memory transactions, mispredictions, and so on. To enforce a constant latency, microreset pads the interval between the timer interrupt and the completion of `fence.t` to the worst-case: the latency of a full write-back of the data cache.

Flushing all microarchitectural state on a constant latency is just about the best that a designer can do against covert channels in hardware. Accordingly, the vulnerabilities we present here stem from errors in implementation.

4.3.1 Leaking Invalid Instruction Cache Data

CEX C2 begins with an instruction fetch raising 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 in the cache. In the frontend, CVA6 loads `icache_data_q` with whatever data payload it receives from the instruction cache, as long as the transaction response is valid.

This register 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—without a check on whether the input data is valid or not. Specifically, the realigner sets `valid_o` simply based on whether the input instruction is compressed or not:

```
// the instruction is not compressed so we can't do anything in
↪  this cycle
if (!instr_is_compressed[0]) begin
    valid_o = '0;
    ...
// the instruction isn't compressed but only the lower is ready
end else begin
    valid_o = 1'b1;
end
```

Figure 4.2: Logic that drives the valid signal on the instruction realigner's output (from *instr_realign.sv* in [43]).

Invalid data is driving the output of `valid_o`, and as a result we see an execution where α's instruction is valid while β's is not.

A viable patch is to zero out the data payload if we do not hit in the cache, returning data to the pipeline only if it is valid (the buggy code is on line 2, and the patch on line 3):

```
if (cmp_en_q) begin
    // dreq_o.data = cl_sel[hit_idx];
    dreq_o.data = |cl_hit ? cl_sel[hit_idx] : '0;
```

4.3.2 A Faulty FSM in the Page Table Walker

This CEX begins with a TLB miss in both α and β, resulting in both universes going on a page table walk. During this walk, the page table walker (PTW) receives the flush signal from `fence.t`.

The controller logic for the PTW dictates that if the PTW is in the midst of looking up a page table entry (PTE) when it `flush_i` goes high, it should wait for a valid read before going to `IDLE`. (The intended transition is `PTE_LOOKUP -> WAIT_RVALID`, then `WAIT_RVALID -> IDLE` on receiving a valid read; see Figure 4.3.) This is exactly what α does; it transitions to `WAIT_RVALID` and waits for a valid read.

β also transitions to `WAIT_RVALID`. However, β then takes an exception, causing `flush_i` to go high again. This second flush signal is seen when the PTW is in the `WAIT_RVALID` state; that is, it is neither in `PTE_LOOKUP` nor is it waiting for a grant. As a result, the controller takes the `else` branch and transitions to `IDLE` on the next cycle, terminating the walk before it gets a valid read and transitioning illegally to `IDLE`.

Later on in execution, α and β both go to fetch the same instruction. In α, the instruction cache generates a fetch exception since the PTW is still waiting and returns a NOP. In β, the instruction cache sees no such obstacle—it returns what it believes to be valid data and β takes a jump where α does not. (This CEX also serves to validate our observation model by correctly exposing differences in architectural state to output.)

```
// --------
// Flush
// --------
// should we have flushed before we got an rvalid, wait for it
↪  until going back to IDLE
if (flush_i) begin
    // on a flush check whether we are
    // 1. in the PTE Lookup check whether we still need to wait
↪  for an rvalid
    // 2. waiting for a grant, if so: wait for it
    // if not, go back to idle
    if ((state_q == PTE_LOOKUP && !data_rvalid_q) || ((state_q ==
↪  WAIT_GRANT) && req_port_i.data_gnt))
        state_d = WAIT_RVALID;
    else
        state_d = IDLE;
end
```

Figure 4.3: Part of the FSM for the CVA6 PTW (from *ptw.sv* in [43]).

4.4 Covert Channels in MAPLE

MAPLE (Memory Access Parallel-Load Engine) is an accelerator for memory patterns that supports fetching single array elements, array ranges, and indirect memory accesses. MAPLE contains a memory-management unit (MMU) for virtual memory translation, and its API exposes several registers that configure the MMU and various hardware queues. In particular, the API offers (1) an `init` operation that allocates a MAPLE instance by mapping its configuration registers into virtual memory, (2) a `close` operation to deallocate a MAPLE instance, and (3) a `cleanup` operation to invalidate configuration registers and flush the TLB between processes.

We define a flush completing in MAPLE as the FSM controlling the invalidation process transitioning to `IDLE`. Although MAPLE's queues could be considered architecturally visible, we do not add them to AutoCC's architectural state condition since they are flushed by `cleanup`.

The first CEX (M1) stemmed from a TLB that was enabled in α but disabled in β. The TLB is enabled by default at reset, but MAPLE's API allows for it to be disabled. CEX M1 proved that the flip-flop that houses the enable bit is not flushed during the context switch. If the Trojan can disable the TLB and the spy can observe a page fault, this could be used as a binary covert channel.

4.4.1 Engineering a Trojan from a CEX

The second CEX (M2) was caused by a failure to flush the register that contains the base address of an array for array-indexed loads. MAPLE allows processes to set the value of this register via the `dec_set_array_base` instruction. To more explicitly present how this may function as a covert channel vulnerability, we wrote the following exploit that demonstrates how a Trojan and a spy process may communicate via this register:

```c
// Trojan executing inside the victim process. Leaks a byte of
↪  the secret into the base address register on every iteration.
void leak(int iteration) {
    int qid = dec_init();
    uint16 leak_byte = (secret >> (iteration*8)) & 0x00FF;
    uint16 offset = leak_byte << 2; // 4-byte aligned
    // Load the base address register with a byte of the secret.
    dec_set_array_base(qid, VADDR+offset);
    dec_close(qid);
}

// The spy process uses mmap() to allocate a 256-element array at
↪  VADDR where array[idx] == idx. Recovers a byte of the secret
↪  from the base address register on every iteration.
void observe(int iteration) {
    int qid = dec_init();
    dec_open_producer(qid);
    dec_open_consumer(qid);
    // Fetch the 0th element from the array located at the
↪  configured base address, i.e. array[leak_byte].
    dec_load_word_async(qid, 0);
    // Consume the value from MAPLE's queue. Since array[idx] ==
↪  idx, this is exactly the value of leak_byte.
    uint32 spy_byte = dec_consume_word(qid);
    recovered = recovered | (spy_byte << (iteration*8));
    dec_close(qid);
}
```

The Trojan leaks the secret one byte at a time, using each byte as an offset for the value of the base address register (line 7). Since the spy process has allocated an array where the value of each entry is equal to its index, it can infer what offset was added by simply loading the 0th element of the array via the `dec_load_word_async` instruction (line 17).

After building an RTL simulation environment with MAPLE integrated into the OpenPiton SoC [44], we ran the exploit on bare metal using Synopsys VCS. It took VCS about a minute to simulate the exploit; the spy process recovered a 128-bit secret over the course of 16 iterations, clocking in at a total of 112,000 cycles.

4.5 The AES Accelerator

The AES accelerator is pipelined across 40 stages and does not contain any architecturally visible state; rather, it follows a request-response protocol. The default AutoCC configuration (with a free flush signal) quickly converged on CEX A1, in which α completes its "flush" while several ongoing requests are still in its pipeline. (The pipeline for β is empty, leading to a difference in output when α responds and β does not.)

Without any invalidate or flush signals, the design of the accelerator seems to assume that it will not be used by another process until all the requests of the current process have completed. We view this as a reasonable assumption if system resources are well-allocated, and correspondingly define flush completion as a pipeline with no ongoing requests. With this condition, the FV tool proved the correctness assertions within 5 hours.

Chapter 5

Conclusion

Dilapidation's processes
Are organized Decays—
. . .
Ruin is formal—Devil's work
Consecutive and slow—

—Emily Dickinson,
Crumbling is not an instant's Act

This work presented AutoCC, a new formal verification-based methodology for detecting microarchitectural covert channels in RTL designs. Focusing on hardware that is shared between processes, AutoCC automatically creates formal testbenches that can detect any executions that may break process isolation. Evaluating this methodology against several open-source DUTs, we were able to detect known vulnerabilities in a matter of minutes. We further discovered new covert channels in projects that had already been thoroughly tested and even taped out, validating one with a successful attack under RTL simulation.

The current landscape of hardware security is one of playing catch-up ball; by and large, our understanding of previous attacks is what informs our analysis of current and future defenses. This case-by-case approach is a manual task that is both difficult and labor-intensive. Awesome machines are also infernal devices, and their very nature all but guarantees that ingenuity will always lose out to complexity.

This comes as particularly bad news in an era of heterogeneous computing. Systems today are composed of increasingly diverse and numerous processing elements. Spreading computation amongst a larger domain of hardware blocks means more modules to secure, more interactions to keep track of.

We Found Some Bugs. So What?

We want to emphasize that AutoCC is much more than just a case study in verifying research processors. In answer to the challenges above, AutoCC suggests a **systematic** and **practical** approach that puts security in front of evolving attack strategies and away from ad hoc solutions that have proven brittle time and time again.

Systematic. No modern processor is likely to be completely free of data leakage, and designers will inevitably have to make trade-offs between performance and security. But by leveraging formal verification to exhaustively test against *all* legal inputs, AutoCC enables more informed and methodical decision-making about what vulnerabilities to address and where in the system stack to do so.

Practical. AutoCC's most valuable feature is arguably the CEX it returns on a violation, providing the designer with a concise and exact trace of the execution that broke process isolation. By supplying the *how*, AutoCC allows the designer to identify the covert channel mechanism and assess how significant it is. As discussed in Chapter 3, AutoCC can also be applied in a modular fashion, making it particularly well-suited for heterogeneous systems composed of various hardware blocks. Finally, the fact that AutoCC is able to generate testbenches automatically makes it much easier to adopt and extend.

The vulnerabilities that AutoCC discovers are precisely the ones that present methods have trouble guarding against—ones uncommon and unimagined, originating largely from the unexpected interactions between modules. Although significant obstacles remain, the fact that AutoCC is able to detect these vulnerabilities in an accurate and performant manner is encouraging.

AutoCC will be open-sourced when our results are published; the DUTs we evaluate are already available online. In the meantime, AutoCC continues to faithfully run and generate CEXs.

Bibliography

[1] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading Kernel Memory from User Space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[2] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," in *40th IEEE Symposium on Security and Privacy*, 2019.

[3] 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*, pp. 605–622, 2015.

[4] S. Deng, W. Xiong, and J. Szefer, "A Benchmark Suite for Evaluating Caches' Vulnerability to Timing Attacks," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, March 2020.

[5] Y. Wang and G. E. Suh, "Efficient Timing Channel Protection for On-Chip Networks," in *2012 IEEE/ACM Sixth International Symposium on Networks-on-Chip*, pp. 142–151, 2012.

[6] O. Aciiçmez, c. K. Koç, and J.-P. Seifert, "On the Power of Simple Branch Prediction Analysis," in *Proceedings of the 2nd ACM Symposium on Information, Computer and*

Communications Security, ASIACCS '07, (New York, NY, USA), p. 312–320, Association for Computing Machinery, 2007.

[7] I. Vougioukas, N. Nikoleris, A. Sandberg, S. Diestelhorst, B. M. Al-Hashimi, and G. V. Merrett, "BRB: Mitigating Branch Predictor Side-Channels," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 466–477, 2019.

[8] Y. Wang, A. Ferraiuolo, and G. E. Suh, "Timing Channel Protection for a Shared Memory Controller," in *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pp. 225–236, 2014.

[9] J. Szefer, "Survey of Microarchitectural Side and Covert Channels, Attacks, and Defenses," *Journal of Hardware and Systems Security*, vol. 3, 09 2019.

[10] O. Sibert, P. Porras, and R. Lindell, "The Intel 80x86 Processor Architecture: Pitfalls for Secure Systems," in *Proceedings 1995 IEEE Symposium on Security and Privacy*, pp. 211–222, 1995.

[11] J. Ravichandran, W. T. Na, J. Lang, and M. Yan, "PACMAN: Attacking ARM Pointer Authentication with Speculative Execution," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA, p. 685–698, 2022.

[12] P. C. Kocher, "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems," in *Advances in Cryptology—CRYPTO '96* (N. Koblitz, ed.), (Berlin, Heidelberg), pp. 104–113, Springer Berlin Heidelberg, 1996.

[13] D. E. Bell and L. J. LaPadula, "Secure Computer Systems: Mathematical Foundations," tech. rep., MITRE CORP BEDFORD MA, 1973.

[14] R. J. Anderson, *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley Publishing, 3 ed., 2020.

[15] M. Schaefer, B. Gold, R. Linde, and J. Scheid, “Program Confinement in KVM/370,” in *Proceedings of the 1977 Annual Conference*, ACM ’77, (New York, NY, USA), p. 404–410, Association for Computing Machinery, 1977.

[16] B. W. Lampson, “A Note on the Confinement Problem,” *Commun. ACM*, vol. 16, p. 613–615, oct 1973.

[17] V. D. Gligor and J. K. Millen, “A Guide to Understanding Covert Channel Analysis of Trusted Systems,” Tech. Rep. NGSC-TG-030, National Computer Security Center, November 1993.

[18] 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*, PLDI 2019, (New York, NY, USA), p. 174–189, Association for Computing Machinery, 2019.

[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, pp. 1–27, Apr 2018.

[20] Z. Wang and R. B. Lee, “New Cache Designs for Thwarting Software Cache-Based Side Channel Attacks,” *SIGARCH Comput. Archit. News*, vol. 35, p. 494–505, jun 2007.

[21] R. Martin, J. Demme, and S. Sethumadhavan, “TimeWarp: Rethinking Timekeeping and Performance Monitoring Mechanisms to Mitigate Side-Channel Attacks,” *SIGARCH Comput. Archit. News*, vol. 40, p. 118–129, jun 2012.

[22] D. J. Bernstein, “Cache-timing attacks on AES.” `http://cr.yp.to/antiforgery/cachetiming-20050414.pdf`, 2005.

[23] C. Canella, M. Schwarz, M. Haubenwallner, M. Schwarzl, and D. Gruss, "KASLR: Break It, Fix It, Repeat," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, ASIA CCS '20, (New York, NY, USA), p. 481–493, Association for Computing Machinery, 2020.

[24] O. Weisse, J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch, and Y. Yarom, "Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution," *Technical report*, 2018.

[25] Q. Ge, Y. Yarom, F. Li, and G. Heiser, "Your Processor Leaks Information—and There's Nothing You Can Do About It," 2016.

[26] A. M. Turing, "On Computable Numbers, with an Application to the Entscheidungsproblem," *Proceedings of the London Mathematical Society*, vol. s2-42, no. 1, pp. 230–265, 1937.

[27] Z. Kincaid, "Lecture Notes on Program Analysis," September 2022.

[28] Z. Kincaid, "Lecture Notes on Temporal Logic," November 2021.

[29] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic Model Checking without BDDs," in *Tools and Algorithms for the Construction and Analysis of Systems* (W. R. Cleaveland, ed.), (Berlin, Heidelberg), pp. 193–207, Springer Berlin Heidelberg, 1999.

[30] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani, *Model Checking and the State Explosion Problem*, pp. 1–30. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[31] J. P. M. Silva and K. A. Sakallah, "GRASP—a New Search Algorithm for Satisfiability," in *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '96, (USA), p. 220–227, IEEE Computer Society, 1997.

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

[33] M. Orenes-Vera, N. Winstoff, D. Wentzlaff, and M. Martonosi, "AutoCC: Automatic Discovery of Covert-Channels in Time-Shared Hardware (draft)," 2021.

[34] 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)*, p. 535–540, IEEE Press, 2021.

[35] 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*, CCS '20, (New York, NY, USA), p. 1853–1869, Association for Computing Machinery, 2020.

[36] 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)*, pp. 994–999, 2019.

[37] 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, pp. 2629–2640, Nov 2019.

[38] 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)*, pp. 460–465, 2022.

[39] A. Magyar, "Vscale RISC-V CPU." `https://github.com/LGTMCU/vscale`.

[40] 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, ISCA '22, (New York, NY, USA), p. 817–830, Association for Computing Machinery, 2022.

[41] A. Salah, "AES-128 pipelined cipher module." `https://opencores.org/projects/aes-128_pipelined_encryption`.

[42] 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*, pp. 1–11, 2022.

[43] Parallel Ultra Low Power (PULP), "CVA6 RISC-V CPU." `https://github.com/pulp-platform/cva6`.

[44] 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.