UNIVERSITY OF CALIFORNIA
SANTA CRUZ

A TIME AND PLACE FOR EVERYTHING: SIDE-CHANNEL VERIFICATION USING CO-SIMULATION

A dissertation submitted in partial satisfaction of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

in

COMPUTER ENGINEERING

by

Ramesh Krishna Jayaraman

December 2022

The Dissertation of
Ramesh Krishna Jayaraman is approved:

Professor Jose Renau, Chair

Professor Matthew Guthaus

Professor Alvaro Cardenas

Peter Biehl
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures

List of Tables

Abstract

A time and place for everything: Side-Channel verification using Co-Simulation

by

Ramesh Krishna Jayaraman

Since the advent of the modern microprocessor, the pursuit of better performance has led to increased design complexity. This increased complexity manifests due to adopting several design concepts like branch prediction, speculative execution, Out-of-Order execution, and their respective implementation choices. When implementing these design concepts in hardware, it is necessary to store information about the execution state of the processor in some form. By design, multiple processes can run on the same hardware. This leads to the execution state of any given process being influenced by one or more other processes. This creates massive security vulnerabilities through timing side-channel attacks, the most infamous classes belonging to Spectre, MDS, and Foreshadow. These are flaws inherent in the nature of the aforementioned design concepts due to their need to maintain information about the execution state to deliver increased performance. These vulnerabilities are found in most deployed modern processors. Most attempts at fixing or patching them through software incur huge performance penalties and require a hardware redesign to recoup them. This work presents a framework to be deployed during the design and verification of microprocessors that will utilize the timing and side-channel effects of these vulnerabilities to the designers' advantage to prove the existence of such vulnerabilities in designs that have been verified using conven-

tional design methodologies. We demonstrate the incidence of timing and side-channel effects in three RISC-V designs, Ariane, BlackParrot, and BOOM. We also prove the correctness of our framework using the patched version of these designs.

Acknowledgments

I thank the members of my committee, whose guidance has helped shape this thesis. In particular, I would like to thank my advisor Prof. Jose Renau for his unyielding support.

I also thank the current and past members of the MASC lab, especially Daphne, Nursultan, and Rafael.

Finally, I'd like to thank my family for their continuous encouragement and support.

Chapter 1

Introduction

Boosting computational power is one of the earnest endeavors of processor design. We have incorporated many significant advances to increase the computational power of microprocessors. Modern microprocessors have undergone significant incremental developments since the advent of integrated circuits. As the number of transistors in an integrated circuit increased, following Moore's Law [70], we have implemented several novel techniques to stick to the trend of increasing performance.

These techniques include, but are not limited to, instruction pipelining, caching, branch prediction, superscalar design, speculation, out-of-order execution, multithreading, and multiprocessing [39]. These ideas effectively deliver performance gains, and some of these techniques have several novel ways of implementing them in integrated circuits.

The inclusion of every novel technique, however, increases the complexity of the design. Implementing such features is a complex task. Modern design processes

break up the system-level design into smaller modules or blocks based on functionality to make this process easier. Due to an ever-increasing number of blocks in a given design, the possible interactions between them also increase, making it hard to note and verify all the interactions. Knowing exactly how the hardware will behave is essential for the software stack and hardware designers to make better implementation choices and fix mistakes in previous designs. This process is known as verification and is a crucial step in designing and creating microprocessors.

Verification, however, is no easy task. It is the most resource-consuming part of the design cycle of modern circuits- heavy on time, compute power, and human resources [37]. Current hardware design cycles rely on verifying the processor design using techniques such as dynamic simulation [50, 68, 88], formal verification [51], and power area verification [17]. While these are effective in their own right, as they check the functional correctness of their design, they do not provide complete coverage. They also ignore specific effects on the architectural state known as side-channels [58]. These are observable architectural side effects that do not impact the correctness of results during execution. Designers, prioritizing economics over laborious, repetitive verification flows, tend to ignore such effects as they do not contribute to the correctness of results. However, this has become a significant area of research as numerous techniques have been published that use side-channels to break security assumptions on the hardware level.

Side-channel attacks can leverage several types of side effects during the execution of a program. These range from physical effects like power consumption [55–57],

electromagnetic radiation [75], and acoustic noise [32] to microarchitectural timing attacks that exploit timing variability in caches [1, 36, 66], Branch Target Buffers [24, 62], and branch history [2,3]. Spectre [54] and Meltdown [65] belong to the later type of side-channel attacks; those that exploit timing variability and serve as the main motivation for this proposal.

Patching hardware after the fact is impossible and causes expensive recalls (monetary) or expensive software-based patches (performance) [10,83]. This necessitates finding these types of bugs during the design and verification phase. For Spectre and Meltdown, the patches so far have been via software [6, 42, 44, 45, 52]. There have also been some patches in hardware for the next generation of processors [5, 41]. This is not satisfactory, as even hardware patches with optimized performance penalties are only viable once a type of vulnerability has been discovered. On top of this, new classes/variants of speculative execution attacks have been discovered that work on fully patched hardware [101]. While there is no option but to issue patches for existing hardware, such vulnerabilities must be fixed during the design and verification phase.

To this end, we introduce a novel verification methodology to find timing side-channels in processor designs. We use co-simulation during the verification phase to run tests on the processor and an equivalent software model to check the design's correctness and introduce variations in the instructions in the speculative path. Any timing side-channels caused by the modified instructions are identified as the microarchitectural state of the processor changes.

We first review the current state of processor design, side-channel vulnerabil-

ities, and verification methods in the related works section. We then provide details about our methodology or framework. Then we provide our evaluation. Then we put forward future work building on our proposal and then present our conclusion.

Chapter 2

Background and Related Work

This Chapter presents the current state of processor design and verification, followed by an overview of Spectre-type vulnerabilities and their mitigations.

2.1 Modern Microprocessors

The modern microprocessor is a testament to accomplishments in aggregate and abstract design, incorporating tens of thousands to millions of lines of code (in RTL) into a synthesized integrated circuit with billions of transistors. Such projects result from many decades of continuous increase in transistor density [78]. This enabled designers to implement an increasing number of architectural features and techniques in a single design to increase performance [39].

Each of the following architectural features was implemented to gain performance and was facilitated by improved manufacturing processes resulting in higher transistor density.

2.1.1 Instruction Pipelining

Early processors would execute one instruction at a time, leaving large portions of the circuitry idle during the execution of a single instruction. Since most instructions have common steps that must be performed during their execution (fetch, decode, execute, write back), designers overlapped the fetch, decode, and execute stages across instructions. This technique has since become a staple in processor designs, regardless of the performance requirement. The latest processor designs usually feature anywhere from a few to more than two dozen pipeline stages, sometimes even more, accommodating a similar number of instructions in flight at any given time. Since pipelining allows multiple instructions to execute at a given time, designers can increase the overall execution speed. This is because smaller pipeline stages allow each pipeline stage to finish its work in shorter periods of time, enabling the entire pipeline to be clocked at a higher speed.

2.1.2 Cache

As the number of instructions that could be in flight at any given time increased with the use of deeper pipelines, so did the speed of execution. However, this increased execution speed and higher demand for instructions were not matched by memory [38]. This necessitated a mechanism to bridge the difference between the pipeline's clock rates and the memory connected to it, leading to the introduction of smaller memory units capable of higher clock speeds as a buffer between the pipeline and main memory. This technique, known as caching, is also a staple in modern processor designs, with multiple

levels of caches of different sizes.

2.1.3 Branch Prediction

With deeper pipelines and unused transistors (due to increasing transistor density), designers sought to increase instruction-level parallelism by minimizing the number of pipeline stalls caused mainly by conditional branches. This was achieved by implementing logic tracking program execution to predict future conditional branches. This technique is known as branch prediction and is implemented in many different ways, from simple two-bit counters to tagged geometric history (TAGE) based and complex neural network based predictors depending on performance needs [61, 81, 82].

2.1.4 Speculative Execution

The successful implementation of branch prediction, a particular case of an architectural technique known as speculative execution, led to the introduction of speculation in other areas of processor design. When using speculative execution and, in particular, branch prediction, the processor attempts to guess the direction of program flow when encountering a conditional branch in the program. When the correct choice is made, it results in increased performance compared to the case where the processor idled until the correct execution path was computed. If the incorrect choice is made, the processor can undo the changes made by guessing the wrong direction of the program flow. It then starts over with the correct program flow at the same time it would have started the correct program flow when idling.

By leveraging control flow speculation in the manner described above, a processor can execute more instructions at any given time when compared to when it idles to compute the correct execution flow, resulting in increased performance. Any increase in the accuracy of branch prediction directly leads to increased performance. For these reasons, speculative execution is a powerful architectural technique as, on the surface level, there are only gains to be made and no losses. Speculative execution encompasses other similar techniques used during execution when the processor would benefit from not idling by attempting to guess the program execution flow [84]. Some techniques that fall under speculative execution include value prediction [63, 97], memory prefetching [9, 46], and speculative loads and stores [76].

2.1.5 Superscalar

Until this point, even while implementing all techniques mentioned above, each pipeline stage executed only one instruction at any given time. As manufacturing technology improved and more transistors could be placed on a single chip, designers sought to further enhance instruction-level parallelism by allowing multiple instructions to be processed simultaneously at different pipeline stages. The replication of functional units in the pipeline facilitated this [47, 84]. Designs implementing this technique are known as superscalar processors and typically have higher throughput per clock cycle.

2.1.6 Out-of-Order Execution

Even when implementing all the aforementioned architectural techniques, stalling the pipeline is unavoidable for reasons such as cache misses, unavailability of specific functional units, etc. In such cases, other instructions in the pipeline could be executed when an instruction stalls. This led to the implementation of out-of-order execution, where the instructions are executed in a different order compared to the actual execution sequence of the program [74, 89, 100]. This technique further increases instruction-level parallelism and uses the available hardware to a greater extent.

2.1.7 Multithreading/Multiprocessing

The fundamental principle to improve the performance of a processor is to leverage the inherent parallelism in a program at some abstraction level. This can be done at bit level parallelism, instruction level parallelism, task level parallelism, and superword level parallelism. All the techniques discussed so far leverage Instruction Level Parallelism (ILP). One way to implement task level parallelism is to distribute the work done by the program into a set of concurrent sub-tasks, threads, or processes. This technique can be implemented in different ways and is supported at the software and hardware level.

One way to implement task level parallelism in hardware is to support multiple threads in a single CPU core. This is known as Simultaneous Multithreading (SMT) [22, 93]. This allows multiple independent threads to utilize the hardware resources better, simultaneously exploiting ILP and Thread Level Parallelism.

The definition of multiprocessing is not as rigid, as it depends on the system implementation; multiple cores on a single CPU die, multiple dies in one package, and multiple packages in a system sharing the same main memory. All these architectures can support the ability to run multiple processes concurrently.

These two approaches to task level parallelism are complementary. The critical difference is that in multithreading, the threads share the resources of one or more cores (compute units, caches, etc.). In contrast, multiprocessing tries to incorporate the hardware resources across multiple cores.

2.2 Existing verification techniques

There is an evergrowing need to ensure that the design functions as expected when processors have deep and complex pipelines, implementing some or all of the architectural techniques mentioned in Section 2.1. Designers employ several strategies to ensure that their designs perform as expected. This process is known as verification and involves many steps.

Designing a microprocessor is not a trivial task. It involves multiple steps, usually known as the design cycle. Figure 2.1 shows a typical design cycle. Verification needs to happen during each step of the design cycle. As the representation of the functionality in the design plan is transformed in each design step, it is necessary to ensure that the design remains within specification, that no existing functionality was lost, and that no unnecessary functionality was introduced due to the transformation

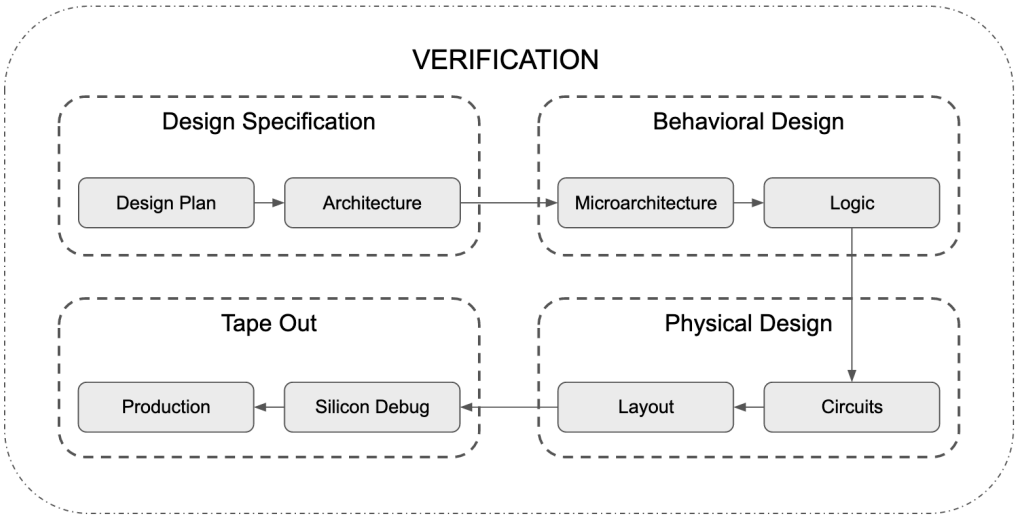

Figure 2.1: Design cycle involves verification at each step.

process. Thus, the verification of modern microprocessors is a long and arduous task. It consumes the most time, money, human and computational resources compared to all the other steps in the design cycle of a microprocessor [37].

This body of work will concentrate on verifying the behavioral design, i.e., verifying the correctness of the microarchitecture when represented at the Register Transfer Level (RTL).

Due to the complexity of a modern microprocessor design, complete functional verification of the design tends to be an impossible task. To reduce this complexity, designers have introduced many verification techniques to be as correct as possible. This involves the use of various proxy metrics based on statistics (coverage) [87], time (bugs per week) [102], functionality (special coverage models), and randomness (random testing) [4, 71]. Using these proxy metrics, the designers establish a verification plan

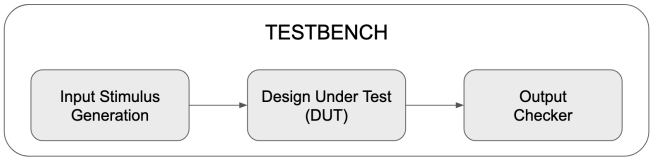

Figure 2.2: Architecture of a simple testbench.

to test the design [40]. This, added with regression testing, where designers test for possible bugs created by fixing any bugs previously found, results in gradually increasing coverage. Designers also incorporate the verification process results into the verification plan to find coverage holes missed by the previous step. By repeating these steps, designers increase their confidence that the design under test is relatively bug-free. This process is extensive but cannot guarantee completely bug-free designs.

We will now describe some of the techniques used in this verification step. Henceforth, any reference to an RTL design is assumed to be in Verilog/System-Verilog to keep things standardized.

2.2.1 Testbench

As mentioned above, while verifying the implemented design is essential, design complexity plays a significant role in determining how designs are verified. Verification can be done manually when implementing very small designs like a half adder or a flip-flop. This is because the number of inputs and outputs is small and the total number of states the design can be in is relatively small.

A testbench usually consists of non-synthesizable Verilog code that can generate inputs for the Design Under Test (DUT) and verify the outputs generated by the

design under test [85]. Figure 2.2 shows a high-level view of the testbench.

The process of writing a testbench usually consists of the following steps:

- **Declarations:** The interfaces needed to communicate with the DUT are declared.
- **DUT Instantiation:** The connections between the DUT and the testbench are instantiated.
- **Input Stimuli and Output Checking:** The inputs and outputs can be checked in two phases:
  - **Initialization:** In this phase, initial inputs are provided to the DUT using the interfaces defined above, and its outputs are checked. This is usually done in a `initial` block in Verilog.
  - **Delay Based Testing:** In this phase, the behavior of the DUT can be checked by varying input stimuli at different times. This is implemented based on different units of `delay`, assigning input stimuli based on current `delay` and checking the output generated by the DUT. For example, the clock and reset functionality can also be tested during this phase, along with design functionality. This is where most of the input stimuli-based testing is done. This is usually contained in some variation of an `always` block in Verilog and is represented by non-synthesizable logic to check the DUT's functionality.
- **Simulation:** This is the step where the DUT is simulated using the different

inputs generated, and its output is tested. The results are generated to show waveforms of the DUT behavior and check them against expected behavior. Although waveforms are the used for debuggin DUT behavior, other methods may be used, including simply dumping DUT output signals in human-readable form and checking against expected behavior.

The input stimuli need to be generated as needed for the DUT. The stimuli vary depending on the functionality of the design being tested. The same is true when checking the output of each design. For example, it would not be possible to use the same input stimuli generator or output checker used for a flip-flop when testing a half adder and vice-versa because these modules represent different functionality and use different inputs and outputs. Adopting existing test benches for other designs is as, if not more complicated, than designing a new testbench for the new design.

This process, while practical, cannot be applied in all cases. As the complexity of a design grows, incorporating many modules, it becomes infeasible to test or write specialized testbenches for each module manually. This is due to the exponential growth of the number of states for the entire DUT. Another factor is capturing complicated cases that arise as different modules interact. Yet another factor is the enormously long simulation time required to test a complex design. Hardware simulation is very slow, and the fastest simulators can achieve speeds of a few Million Instructions Per Second (MIPS) [15] for a reasonably small design and get a lot slower for more complex designs, simulating around tens to hundreds of thousands of instructions per second -

Kilo Instructions Per Second (KIPS).

Several techniques are used to be as thorough as possible to overcome the difficulty of testing increasingly complex designs. These are addressed in the following Sections.

2.2.2 Random Instruction Generators

Random Instruction Generators (RIG), also known as Test Program Generators or Instruction Stream Generators, are programs that can generate a random stream of instructions for a given set of configurations or parameters. These can be very effective in generating complex test cases that are hard to generate manually [7, 102].

While these tools are good at testing for a wide range of functionality implemented in the DUT, they are not the best at testing complex interactions between different modules. Some RIGs have additional inputs they can take that are known as test program templates. These templates are abstract descriptions of the test. They serve as a guide to the RIGs to match certain constraints when generating the instruction stream. This allows RIGs to be more effective at testing the interactions between different modules in the DUT [20, 95].

2.2.3 Reference Models

The reference model is a high-level implementation of the functionality in the DUT. It does not reflect the implementation-level details of the DUT. Instead, it reflects the changes in the architectural state at instruction-level granularity. These models

can be used to check the implemented DUT for the correctness of the execution path. The same benchmark or test code is run on the DUT and the reference model. The architectural state at any point of comparison must be the same. If this requirement is not met, the DUT is checked for correctness using conventional methods such as waveform analysis. This technique can be implemented at different levels of complexity, as outlined below.

2.2.3.1 End of Simulation Comparison

In end of simulation comparison, a test program is run on the DUT and the reference model. When the test is done, the architectural state of the DUT is checked against the reference model. The compared architectural states include register states and memory states of both models [50]. This type of comparison has apparent drawbacks. It is hard to debug as there may be a substantial difference at the end of the simulation due to a bug at the point of deviation. There may also be undetected bugs as they may be hidden by correct execution later in the simulation.

2.2.3.2 Trace Comparison

A better way to implement simulation and comparison is Trace Comparison. In this method, the DUT and reference model dump logs that contain information about the instruction, Program Counter flow, and register/memory accesses. The traces are compared, and any deviation is noted. This method facilitates a more straightforward debug process as points of deviation can be noted, and erroneous behavior can be

captured with more detail in contrast to End of Simulation Comparison. Though more useful, this method also has its drawbacks. Mainly this method is susceptible to external stimuli such as interrupts or debug requests [68]. Since the traces are taken by running the DUT and reference model independent of each other and interrupts can happen at random points during program execution, the traces can be vastly different when compared.

2.2.3.3 Co-Simulation

The DUT and the reference model need to be run in parallel to overcome the drawbacks of the aforementioned simulation comparison methods. This needs to be accompanied by the ability to communicate between the DUT and the reference model about the current execution state and any asynchronous interrupts that might occur. Such communication provides the ability to match execution states in the presence of asynchronous interrupts and allows the pausing/halting of simulation at the point of divergence. An infrastructure that supports this is known as a co-simulation infrastructure [88].

2.2.4 Coverage Based Testing

It is hard to continue writing testbenches for each module in a complex design and simulate all possible states for the reasons mentioned above. One set of techniques used to alleviate this is coverage analysis. This is done using a set of metrics to measure the verification effort's adequacy and progress. Doing this makes it possible to allocate

the available computational resources more efficiently. These metrics can also be used as a guide to generate input stimuli [18, 26, 87]. The next Sections cover the different classifications of coverage metrics.

2.2.4.1 Code Coverage

Code coverage metrics are very similar to their counterparts in software testing. These metrics identify structures in HDL code that can be used during simulation. These can be as simple as line coverage to more complex cases like branch, expression, and path coverage. The latter three leverage the control flow in hardware using Control Flow Graphs (CFG). Branch coverage requires exercising each possible direction of control flow at the branch. Expression coverage involves exercising each possible way an expression in the HDL can be evaluated. Path coverage involves the overall path of execution in the CFG.

2.2.4.2 Circuit Structure Based Coverage

The most basic coverage metric based on circuit structure is toggle coverage. This metric involves checking stuck-at or toggle (from 0 to 1 or 1 to 0) for each binary node in the circuit at some point during simulation.

More complex metrics based on circuit structure require the circuit to be separated based on data and control paths. Data path based metrics include checking structures like registers, counters, and other structures for different states. For example, registers are checked for initialization, loading, reading, and register-to-register

connections. On the other hand, counters are often checked for minimum and maximum values and reset conditions on top of those checks performed on registers. Similarly, each structure identified requires different checks. Control path based metrics are similar to path coverage metrics defined in code coverage. Additionally, there must be tests of communication between the control path and the data path. For example, checking control status interfaces between the data path and control path.

2.2.4.3 Finite State Machine Based Coverage

So far, we have discussed coverage metrics based on static representations of the RTL, like code and netlists (usually gate-level descriptions of the connectivity of an electronic circuit). Since these do not provide a complete test of the sequential behavior, another type of metric needs to be defined.

Metrics based on Finite State Machine (FSM) models of the DUT are better at testing sequential behavior. These models require state and transition information about the DUT. Since it is computationally intensive to model the entire DUT as an FSM, it is common to model parts of the DUT as FSM models. There are currently two ways to model parts of the design as an FSM model, each with advantages and disadvantages.

- **Hand Written FSMs:** This is done when testing specific hardware functionality at high levels of abstractions. These models usually capture the design more succinctly, but there are a few drawbacks since these are manually designed. They include its time-intensive nature and the fact that these models might not accu-

rately represent the design implemented in RTL.

- **Automatically Extracted FSMs:** This method uses a set of state variables upon which the RTL design is projected. These state variables can be manually selected from the design or be automatically selected based on heuristics. Once the state variables are selected, the RTL implementation can be used to determine the possible transitions between the different states. The biggest challenge in this method is determining the necessary coverage across the different states. The reason for switching to coverage-based testing is to reduce the total number of states tested. Selecting the critical state transitions in the automatically generated FSMs can be challenging.

2.2.4.4 Functional coverage

These coverage models are based on the computation performed by a specific module instead of its structure. These models are specific to functionality implemented in the design and require the selection of certain error-prone scenarios that might occur during execution. These scenarios can be defined manually, specific to the implementation, or can be extracted from the RTL by tools designed to search for given predetermined structures. Like FSM-based models, these models can test the behavior of modules over time (spanning several clock cycles, etc.).

2.2.4.5 Other Metrics

- **Error Models:** Each coverage metric mentioned above has an error model associated with it. These are simple for some cases and explicit for most. For example, toggle coverage has a simple error model (check the bit transitioned from 0 to 1 or 1 to 0). Some metrics are defined solely based on error models. These are called error-based or fault-based metrics.

- **Observability:** It is crucial to be able to observe the variables being checked while simulating the reference model against the DUT. This can vary based on when the check is performed, and the level of abstraction used when checking. Observability requires that the variable can be checked at different stages for correct behavior. Some variables cannot be checked at all times or at some levels of abstraction and are hence unobservable at those times.

- **Metrics Applied to Specification:** While testing based on all the metrics mentioned above, it is important also to test if all the design objectives mentioned in the design specifications are met in the DUT. This process involves metrics testing for design functionality that should exist in different modules and the DUT overall. These types of metrics are known as metrics applied to specification.

2.2.4.6 Test Generation

Adopting the aforementioned coverage models makes it possible to automate the generation of tests and input stimuli to a design. Designers can use the heuristics

gathered by these metrics to generate tests. This task can be challenging depending on the type of coverage model chosen. For example, test generation for code-based and circuit-based coverage models is easier as they involve working with static versions of the DUT. In contrast, FSM-based and Functional coverage models make it harder to generate inputs automatically due to modeling behavior over multiple cycles.

2.2.5 Formal Verification

All the techniques mentioned above are implemented at a particular stage in the verification process shown in Figure 2.1. As mentioned earlier in this Section, the hardware design process involves transforming the design into different representations. It is also essential to verify that no errors or unwanted features were introduced while transforming the design from one representation to another.

Formal verification [51] is a promising methodology to test the functional correctness of code, especially during these transformations. It is increasingly gaining popularity among hardware designers. This verification methodology tries to examine all possible execution paths in the design by proving or disproving the correctness of a formal model of a design. However, it does have its drawbacks, mainly due to its scalability; it is applied to designs on a modular level and can only be used for designs that are moderate in complexity.

2.2.6 Other Techniques

All the techniques mentioned above run predetermined programs or test the functionality according to some defined constraints. When implementing coverage-based tests, it is not only essential to test correct inputs based on test programs; it is also important to test some architectural states that might not occur when running these tests. Recent work has shown the effectiveness of such techniques, known as Fuzzing [48, 60, 92], where the test programs are run, and some architectural states are randomly changed. These techniques have proven to be effective in exposing bugs in the design that were previously undetected.

2.3 Dromajo

One of the verification techniques mentioned in Section 2.2 is co-simulation. In this section, we talk about one tool that allows us to do this. Developed by Esperanto Technologies, Dromajo [88] is a RISC-V RV64GC emulator primarily designed for RTL co-simulation.

It is fast. Dromajo enables executing applications, such as benchmarks running on Linux, under fast software simulation (around 17 MIPS). It is flexible. It can interface with a variety of RISC-V RTL cores using API calls. It contributes towards experiment reproducibility through its ability to generate checkpoints after a given number of cycles and resume such checkpoints for HW/SW co-simulation. Due to these factors, Dromajo is a powerful tool to capture bugs in combination with randomized tests [48].

2.3.1 Checkpoints

Dromajo facilitates the creation of checkpoints with information about the processor architectural state (registers, Control and Status Registers(CSRs)), the memory, program interrupts (PLIC / CLINT), and performance counters such as the cycle counters and the number of instructions executed.

This information is made available as two images, a bootrom image and a memory image. The memory image restores the state of the memory during checkpoint generation. The bootrom image is used to restore the architectural state of the processor when the checkpoint is created. This is done by leveraging the RISC-V debug specification [29], which allows setting different supervisor registers. The RISC-V debug specification is supported by most RISC-V CPUs. This makes this approach compatible with most existing designs.

2.3.2 Co-simulation

Dromajo can be compiled as a shared library and linked to an RTL simulator. This allows for interaction between the RTL and Dromajo during simulation through API calls such as DPI calls from Verilog. These API calls are simple yet powerful, facilitating RTL co-simulation with Dromajo.

2.4 Spectre

In section 2.1 we have discussed several microarchitectural features implemented in modern processor designs. In this section, we will discuss how the implementation of these features culminates in the primary motivation for this proposal; timing variability transient execution attacks [13, 54, 65, 80, 94].

To recap and summarize section 2.1, almost all modern processors implement pipelining, have caches, and at least a rudimentary branch predictor. Processor designs geared towards high performance usually implement more complicated speculative execution techniques and usually have superscalar out-of-order pipelines. It is highly unlikely to achieve the performance seen in the current generation of processors without implementing such techniques. Most current processors can achieve an Instructions Per Cycle (IPC) greater than 1 during most execution phases of most benchmarks. Speculative execution is one of the leading architectural features contributing to such high performance. This is when the processor guesses which program execution path is more likely to happen at a given program execution state and starts executing that path before the correct execution path is computed. This works out most of the time but can lead to scenarios where the wrong execution path was chosen. When this is the case, the processor discards any changes made to some critical architectural structures like registers, re-order buffer, etc. This is done by creating a checkpoint each time a speculative path is executed. The instructions in the speculative path, which were executed and then discarded, are known as *transient instructions*.

While some architectural structures are updated to reflect true program execution, the effect of these transient instructions is not limited to those structures. When executed, they are treated like any other instruction (except for the checkpoint) and can influence hardware structures like caches, branch prediction tables, special registers, and functional units. These act as side-channels and have been shown to break several hardware-level security constructs guaranteed by hardware designers and assumed by software developers.

There have been multiple such transient execution security vulnerabilities that have been disclosed in the last few years. All of these vulnerabilities expose `secret data` belonging to other processes. This effect is demonstrated in several Spectre vulnerabilities [54], which include Meltdown [65], which exposes kernel memory, Foreshadow [94], which exposed trusted execution environments, and Microarchitectural Data Sampling (MDS) [80], which can leak data from other threads. Each of these vulnerabilities has multiple variants, exploiting the primary mechanism of the vulnerability in different ways.

Spectre-type attacks use the presence of these transient instructions to their advantage. They can trick the processor into executing instructions that will not be executed during the correct program execution. This is achieved by tuning the attacker program to exhibit specific behavior, which influences the state of some architectural structures like branch prediction structures, return address stack, etc. This then influences which transient instructions are speculatively executed. The result is the ability to leak information from other processes through techniques such as forced cacheline

replacement.

2.5 Mitigation Strategies

As these vulnerabilities have been detected in existing processor designs from different vendors spanning multiple Instruction Set Architectures, strategies to mitigate them have become a prime area of research. Several strategies have been proposed, with some of them seeing real-life implementations. These range from code recompilation [6], making patches to the Operating System (OS)/Virtual Machine Hypervisor (VM) [42, 44, 52], implementing microcode patches [45], to redesigning hardware [5, 41, 77] to mitigate these vulnerabilities. Table 2.1 shows different transient execution attacks confirmed by manufacturers of commercial processors and patches applied to mitigate them. A few transient execution attacks have been published but have not been acknowledged by manufacturers. For example, Spectre-NG (v1.2) [53], a Read-only Protection Bypass attack, has not been acknowledged by Intel, and Take a Way [64], a cache-way prediction based attack has been disputed by AMD and has not yet been patched.

Each mitigation strategy mentioned above has drawbacks [10, 83]. Code recompilation requires recompiling every software program to mitigate these vulnerabilities. This is not practical. This method usually incurs high performance reduction as the mitigation is performed in software while the hardware is still vulnerable. While patching the compiler ensures that programs can no longer be written to exploit such

Vulnerability (version)	Exploit type	Mitigation
CROSSTalk	Special Register Buffer Data Sampling	Microcode
Fallout	Microarchitectural Store Buffer Data Sampling	OS, Microcode
Foreshadow	L1 Terminal Fault/L1 Rouge Access	Microcode
Foreshadow-OS	L1 Terminal Fault/L1 Rouge Access	OS, Microcode
Foreshadow-VMM	L1 Terminal Fault/L1 Rouge Access	OS, Microcode
Load Value Injection (LVI)	Load Value Injection	Code Recompilation
Meltdown	Rogue Process Data Cache Load	Microcode
Rogue In-Flight Data Load (RIDL)/CacheOut	L1D Eviction Sampling	OS, Microcode
Rogue In-Flight Data Load (RIDL)	Microarchitectural Load Port Data Sampling	OS, Microcode
Rogue In-Flight Data Load (RIDL)	Microarchitectural Data Sampling Uncacheable Memory	OS, Microcode
Rogue In-Flight Data Load (RIDL)	Vector Register Sampling	OS, Microcode
Rogue In-Flight Data Load (RIDL)/ZombieLoad	Microarchitectural Fill Buffer Data Sampling	OS, Microcode
Rogue In-Flight Data Load (RIDL)/ZombieLoad v2	Transactional Asynchronous Abort	OS, Microcode
Spectre (v1)	Bounds Check Bypass	Code Recompilation
Spectre (v2)	Branch Target Injection	Hardware, OS/VM, Microcode
Spectre SWAPGS	Bounds Check Bypass	Code Recompilation
Spectre-NG	Lazy FP State Restore	OS/VM
Spectre-NG (v1.1)	Bounds Check Bypass Store	OS/VM
Spectre-NG (v3a)	Rogue System Register Read	Microcode
Spectre-NG (v4)	Speculative Store Bypass	Hardware, OS/VM, Microcode
SpectreRSB	Return Mispredict	OS

Table 2.1: Transient Execution attacks which have been confirmed by commercial processor manufacturers and have since been patched.

vulnerabilities, the hardware remains vulnerable.

The second strategy of patching the Operating System/Virtual Machine Hypervisor is a more feasible strategy to pursue. These patches are often successful but still introduce a small to significant performance hit as the patch is still handled at the software level. These patches are easier to implement as they require updating the OS/kernel. This requires significantly fewer resources to implement and distribute these patches.

The third strategy of implementing microcode patches requires updating the microcode versions of each affected CPU. This might seem as effective as the Operating System/Virtual Machine patches, but it involves significantly more resources. Patches of this type need specific implementations for each CPU type affected (there are plenty) and require integrating the microcode patches to every kernel deployed. These patches usually incur a similar performance penalty to the OS/VM level patches.

The final type of proposed patches involves implementing at least a partial redesign of the affected architectural structures. This can include changes like when updates on specific structures are performed, ISA level changes [23], implementing secure running modes in hardware for secure code, and implementing changes in hardware behavior [5,77] to name a few. Some of these proposed hardware redesigns also consider the effects on caches, proposing new update strategies and newly introduced special caches for speculative accesses. These changes require the most resources due to reasons discussed in section 2.2 and also take considerable time to manifest in the next generation of hardware. It also does not fix current hardware that has already been deployed.

There have also been proposals to secure timing side-channel attacks enabled by the Translation Lookaside Buffer [21]. Another approach is introducing mechanisms in both the OS/VM and hardware to protect specific memory ranges selectively [31]. There have also been proposed microarchitecture changes to update policies for registers that have been shown to prevent timing side-channel attacks [79]. Other proposals to mitigate timing side-channels include adding checks to control flow instructions and restricting them within legal address ranges [59]. Additionally, static program analysis can also be used to detect branches that can be exploited by timing side-channel attacks [96]. On top of this, dynamic program analysis can also be used to show timing side-channels [73].

It should be noted that there is no single best strategy to implement patches for each of the vulnerabilities mentioned in section 2.4. The best strategy depends on the type of vulnerability and its variant. It is also important to note that all these strategies

focus on fixing the problem's effect rather than addressing the cause of the problem. These vulnerabilities exist as there is no verification infrastructure/framework that can integrate easily with existing modern verification techniques to handle such timing side-channels. As described in section 2.2, current verification methodologies only check for functional correctness. Transient execution is not considered, as even with the latest co-simulation flows, only the correct program execution path is considered. To address these issues, we propose a verification framework to detect the timing side-channel effects of transient execution.

2.5.1 Detecting Side-channels

The most comparable publications to this work are not mitigation strategies but verification techniques. They can be classified into three main categories, as explained below.

2.5.1.1 Model-Based

Model-based approaches like [12,16,91] create a model of the CPU and perform different tests/checks to verify that the model does not have leaks. This approach is quite different, as it requires a model and does not work at the CPU/RTL level like our proposed approach. Revizor [72] is another model-based testing approach for commercially implemented CPU designs. Another model-based approach is shown in [11], where the biggest contribution of the authors is to reduce the search space of the model using observation refinement for cache coloring and speculative leakage. All model-

based approaches have the disadvantage that the model might not be the same as the RTL under test. Building an ISA model for the co-simulator is much simpler in terms of verifying the correctness and complexity of the model. An ISA model has a further advantage in that it can be used to test multiple CPU designs that use the same ISA.

2.5.1.2 Formal

Formal approach is showcased by UPEC [25]. They propose a formal method to detect when a transient instruction does not allow data from a protected region to leak outside. Such formal methodologies are also quite different in approach and scope from the proposed work. Formal techniques use formal SAT solvers to check for side-channel attacks. The scope is also different because UPEC restricts only protecting leaks from a "confidential" memory region and does not consider leaks outside the region. This does not cover side-channels that avoid loads. For example, a `retpoline` [49] attack could create a sequence of transients that leak contents from registers. While such an attack may be difficult to implement, it is possible. In summary, UPEC is an important but different approach (formal) with advantages like mathematical guarantees but disadvantages like scalability to large designs. While both approaches provide the ability to verify the design at verification time, our approach has the advantage of being much more scalable for large designs.

2.5.1.3 Simulation

The alternative *Simulation* based approach is IntroSpectre [33]. It has a similar goal of detecting speculative side-channel leaks, but the approach is very different. It leverages known attack code snippets, fuzzes them and simulates them with RTL, and has checks to detect leaks. This is complementary to our approach. IntroSpectre cannot protect against unknown attacks, but it has the advantage of verifying against specific attacks. While similar, we consider being able to protect against unknown attack vectors a significant advantage.

2.5.1.4 Other approaches

A possible related alternative is to have a new [27, 105] Hardware Description Language (HDL) to verify against leaks. Such approaches require new languages, not reusing existing designs, and high levels of investment in new HDLs. Osiris [98] is a fuzzing-based approach to detecting side-channels in CPUs. They use fuzzing to generate instruction sequences and run them on commercially available CPUs to detect side-channels, including those caused by transient execution. While this approach is valid, our framework works during the verification stage of the CPU's design to eliminate possible speculative side-channels altogether. Another fuzzing-based approach is Medusa [69]. In this paper, the authors use fuzzing to mutate existing Meltdown variants' basic blocks to generate new Meltdown sub-variants. This requires knowledge of existing time side-channels to find similar ones. Unlike the approaches mentioned above, our work still has the advantage of being verified during the design and verification of

the processor.

One possible defense against speculative side-channel attacks is to selectively forward the speculative instructions by identifying the covert channel and forwarding the results only if it is not going to the covert channel [103]. This dynamic approach adds almost 15% overhead. Another approach is to analyze programs to determine if the leak happens during the speculative or non-speculative instruction and only perform execution once it is determined that the execution is non-speculative [19]. Another dynamic approach is DOLMA [67]. In this work, the authors use the principle of transient non-observability by ensuring that within a given time slice on a core for a program, they can provide isolation from existing attack vectors. While all the aforementioned dynamic approaches are valid, they can have very high performance overheads during runtime. While some works only has a 15% overhead [103], others can have up to 45% and 42% overhead [19] [67]. While this is still better than existing dynamic approaches, the problem of speculative side-channels needs to be fixed during verification. This is the significant advantage our approach has over dynamic approaches.

One of the mitigations for the original Spectre variants was `retpoline` [49]. It is a software construct that prevents *branch target injection* by isolating indirect branches from speculative execution. While there were concerns that the `retpoline` patch could be exploited to create another class of speculative execution attacks, it was deemed impractical to exploit [90]. As it turns out, dismissing possible attacks as impractical is not the correct approach, and the `retpoline` patch has been exploited by a new class of speculative execution attacks called `retbleed` [101]. We believe other

patches to different Spectre variants could be exploited similarly. It is not enough to make attacks improbable or patch them once they are found. Time side-channels must be eliminated. To achieve such a goal, it is insufficient to use only known attack vectors like IntroSpectre [33] to achieve it. *We propose using random test generation to maximize coverage and flag any and all time side-channels, however unlikely they may be to exploit.*

Chapter 3

Verification Framework for Timing Side-Channels

The raison d'être of this proposal is to establish a verification framework to detect side-channels. We concentrate on timing side-channels (side-channels cause differences in the performance of the test program depending on the transient instructions executed) since Spectre-type side-channels can be detected based on changes in the execution time of the instructions executed by the test program. This framework can also detect other side-channels, like changes in branch prediction structures, cache states, etc., that are visible due to transient execution.

On a high level, we first identify the transient instructions of interest. This is done by running the test program on the DUT and a co-simulation tool. Once the transient instructions of interest are identified, we rerun the test program, changing the transient instructions in different ways (changing their PC, instruction, and data

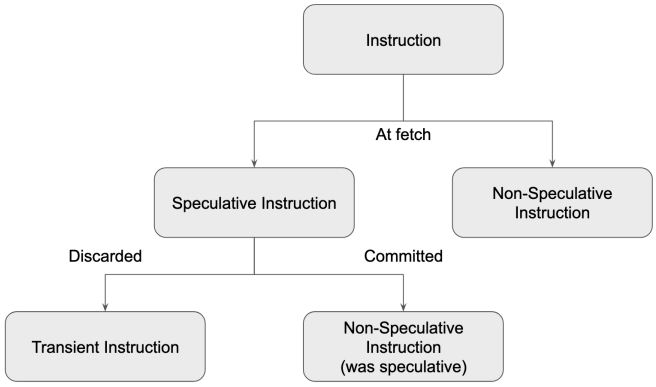

Figure 3.1: Instruction life cycle with respect to speculation.

accessed by the instruction). We examine the effects of these changes with the help of the co-simulation tool, capturing changes in the timing of commit and changes in architectural structures like branch prediction structures and caches.

3.1 Speculative Execution and Transient Execution

To recap section 2.4, Spectre-type attacks are timing-based transient execution attacks. We will first look at instruction behavior in the pipeline to define this accurately.

Since modern CPU designs use speculative execution and pipelining, they can have multiple instructions in the pipeline at various stages of execution. An instruction in the pipeline can be classified based on its speculative nature. Very few instructions are non-speculative, i.e., very few instructions belong to the category where they will always be executed regardless of which control flow path is chosen at any point during execution. The majority of instructions belong to the category of speculative instructions, i.e., instructions that are determined to be on the correct program execution flow at fetch

by the speculative logic of the processor. These speculative instructions can be further classified based on what happens to them on commit. Depending on the correct program execution flow, they will either be committed, becoming non-speculative instructions, or can be squashed/discarded, becoming transient instructions. This lifecycle of an instruction with respect to speculative execution is shown in Figure 3.1.

Extending this terminology, more than one consecutive transient instruction can be identified in the pipeline when we identify such transient instructions while using our framework. We name such phases in program execution with more than one consecutive transient instruction, a transient phase. In this proposal, we are primarily interested in the transient instructions/phases, and any description of changes made to instructions on the speculative path is made on the transient instructions. Henceforth, when referring to speculative instructions/phases/execution, we refer to the classification at fetch, as described above and shown in Figure 3.1. Similarly, when referring to transient instructions/phases/execution, we refer to the classification at commit, i.e., the discarded instructions as described above and in Figure 3.1.

Since transient instructions are not committed, there should be no effects on the processor's architectural state due to their presence/execution. However, this is generally not true, and there are some side effects due to the execution of transient instructions. Hardware free from the side effects of transient instructions would be complicated and impractical to build and consume an unacceptable amount of power. Instead, designers elect to implement more straightforward solutions like checkpoints that protect critical hardware structures from the effects of transient execution. This

includes solutions like reverting the register file, reorder buffer, undoing the updates caused by transient execution to the branch predictor structures, and so on. There have been a few proposed methods to handle transient updates on caches [5, 77] though their performance and design implications on hardware that has been taped out have not been verified. Designers also implement features like updating the branch predictor structures before the branch instruction is retired- before the branch instruction has computed the actual branch. This is usually done for performance reasons during speculative execution - to predict future branches in the speculative phase.

3.2 Timing Side-Channel Effects

Implementing more straightforward choices mentioned in Section 2.1 leads to multiple observable side effects during execution. These side effects can manifest in different forms and can be used to leak information in different ways. These are known as side-channels [58]. These can be detected based on how they manifest as described in Chapter 1. In this work, we are primarily interested in timing side-channels caused by transient execution. These side-channels cause differences in the performance of the test program depending on the transient instructions executed. The performance of the test program can be measured in terms of the timing of the instructions executed, hence timing side-channel. These timing side-channels are also called transient timing side-channels or speculative timing side-channels as they are caused due to speculative execution of transient instructions.

Spectre-type vulnerabilities leverage these speculative timing side-channels to leak information from victim processes. These timing side-channels can be detected when simulating the DUT during the verification stage of the hardware design cycle. It is also true that each instruction type can have different execution latencies in the pipeline. We take advantage of both these facts in the proposed framework to expose timing side-channel effects in a design, our primary focus. This framework can also expose other side-channels, like effects on the branch predictor structures and caches, but this is not the main focus of our work. Any side-channel effects of transient execution that do not affect the timing of the correct execution trace with respect to the reference model are not considered timing side-channels.

Speculative side-channel attacks can be classified under many categories and variants. However, in all cases, they can be broken down into three main components: *Access*, *Leak*, and *Measurement*. In this Section, we expound on these ideas in the context of our framework.

Access mandates that the transient should be able to access protected data. An attack can have transients reading register file contents or memory locations. Nevertheless, the instructions with access to protected data can be executed during transient phases. Any access to data the programmer did not expect is an access violation. From an RTL/CPU point of view, most transients can access protected data because we do not know what register or memory data is protected and what is not.

Due to the complex nature of predictors and speculation, the programmer has little control over the possible sequence of instructions that become transients.

Compounding this problem, in ISAs like RISC-V and x86, a branch/jump's predicted target PC can be in the middle of instruction sequences, effectively creating new code fragments that are not what the programmer/assembler expected. The result is that protecting access is quite a challenge unless speculation is avoided.

Leak is the next necessary component after *Access*. A leak indicates that there is some measurable side-channel effect from a transient that is data-dependent. If the side-channel effect is constant, i.e., independent of the data accessed, there is no way to perform an attack. This is because the third key step, *measurement*, cannot be performed by the attacker.

The leak can happen in many ways, depending on the measurement capabilities, but this work focuses on time and any performance counters. It can also have a power/energy/temperature impact, leading to other side-channel leaks, which is out of this work's scope.

Most established attacks leverage caches to trigger cache misses to perform leaks. Leaks can have many other aspects. For example, it could affect a performance counter like cache misses or have a micro-architectural side-effect. Examples of micro-architectural states that could be speculatively updated leading to such side-channel attacks include any branch predictor state, any predictor like a way predictor, memory dependence predictor, value speculation, cache replacement, variable latency units, prefetchers, and any other state that plays a role in speculative execution.

Similar to committed instructions, transients will have side-effects, but the key observation is that the leak must be data-dependent for an attack to occur. This is the

critical point that our framework targets with co-simulation and fuzzing.

```
1   ; x1 protected data
2   ; x2 enough permission level
3   ; x3 some memory pointer
4
5   beq x2,x0,no_permission
6
7   shl x4,x1,8     ;; access
8   add x3,x3,x4
9   ldb x4,(x4)     ;; leak
10
11 no_permission:
12  ; some other code
```

Listing 3.1: Access to protected data

We consider leaks not only caused by individual instructions but any combination of instructions on the transient path. For example, in the case of a function return instruction (`ret`), if the `ret` updates the return address stack predictor (RAS), the update is data-independent, so the `ret` is not a leak per se. Nevertheless, if other transients can create data-dependent pipeline stalls, the RAS update due to the `ret` instruction can result in a leak. This is because the other transient instructions before the `ret` RAS update can be used to leak information.

To summarize, there is a leak only when the transients can have a data-dependent leak, which can result from the interaction of multiple transients.

Measurement is the last step needed to perform an attack. The transients could access protected data and have a data-dependent leak, but they are deemed a leak only if they are measurable. This was the point architects used to dismiss this type of attack before Spectre/Meltdown. They thought that they could not be measured if transients were flushed. The problem is that the leak can affect resources that will impact performance. This performance impact can be measured in many ways.

A typical attack triggers cache misses and scans the cache to account for hit/miss regions. The truth is that any execution time or performance counter change due to a leak transient can be measured if enough access is provided. Imagine the operating system call with a small number of cycle variations based on the leak data. The code calling the OS `syscall` can measure the time before and after the call. This effectively allows for measuring the leak. An example of a performance counter that can be used to measure leaks is the `page_walks` counter in Intel x86 architectures [43]. This performance counter tracks the number of core cycles during a page walk. If the attacker can read the counter, or the overflow for page walk triggers a performance counter overflow, it can be used as an address-dependent point of attack. These attacks are similar to data-dependent attacks where the address of some memory location is used as the data to trigger the attack.

The goal is to have no measurable transient leaks. This implies that even one cycle over the whole program execution is considered a leak. The same for any slight change in any performance counter exposed.

Putting all this together with a RISC-V example, in the Listing 3.1 the branch

is taken if no permission is available. The *access* is triggered by the access of register `x1` in the `shl` instruction. The leak is triggered by the `ldb` instruction, which loads a byte based on the secret data accessed earlier. This load can update the cache state, which can then be *measured* by the attacker by checking for cache hits/misses.

3.3 Detecting Leaks

A successful attack requires *Access*, *Leak*, and *Measurement*. Transient instructions are responsible for the access and leak, and the measurement can be done afterward. The proposed verification framework focuses on the leak. This is because most transient instructions can perform the access. The measurement can happen in many forms, including those that do not involve architectural states like measuring OS syscall performance.

If the verification infrastructure can guarantee that transients do not have any data-dependent leaks, it can be guaranteed that the CPU/RTL is safe from speculative side-channel attacks.

Theoretically, it would be possible to find speculative side-channel attacks by only fuzzing the data accessed by the transient instruction as this would mimic the secret data *access* performed by transient instruction. While this is a valid approach, it is also inefficient in the context of verification as the transient instructions do not change, restricting the coverage in terms of code fragments accessed and conventional coverage metrics like line and toggle coverage. A key step in verification is to increase

coverage as much as possible. To this end, we propose the following steps:

- **Transient detection:** Our framework leverages the existing set of tests and co-simulation platform for verification. As each test executes, we detect the transients. We call this step transient detection.
- **Transient fuzzing:** For each test, we perform one run where the transients are randomized. Here, we mean the transient instructions are randomized to increase speculative path coverage. We call this step transient fuzzing.
- **Data fuzzing:** For each fuzzed transient instruction, we fuzz the instruction's operands. We call this step data fuzzing.

Transient detection requires one run of the test program. Afterward, creating many *transient fuzzing* tests is possible. For each *transient fuzzing* test, several *data fuzzing* tests can be created. This is shown in Figure 3.2.

As the different *data fuzzing* tests run, we compare the performance counters and the instruction commits. All the data fuzzing tests should have precisely the same performance counters and instruction commits at every cycle for any given instruction fuzzing test. If there is any mismatch, there is a leak.

These leaks can be detected based on the performance of the programs (time) or the changes in the performance counter values. Time-based program performance can be detected by tracking the commit times of the committed instructions. The performance counters include the time-based counters but also several other counters. In RISC-V, there are 32 performance counters, but in theory, we should compare all

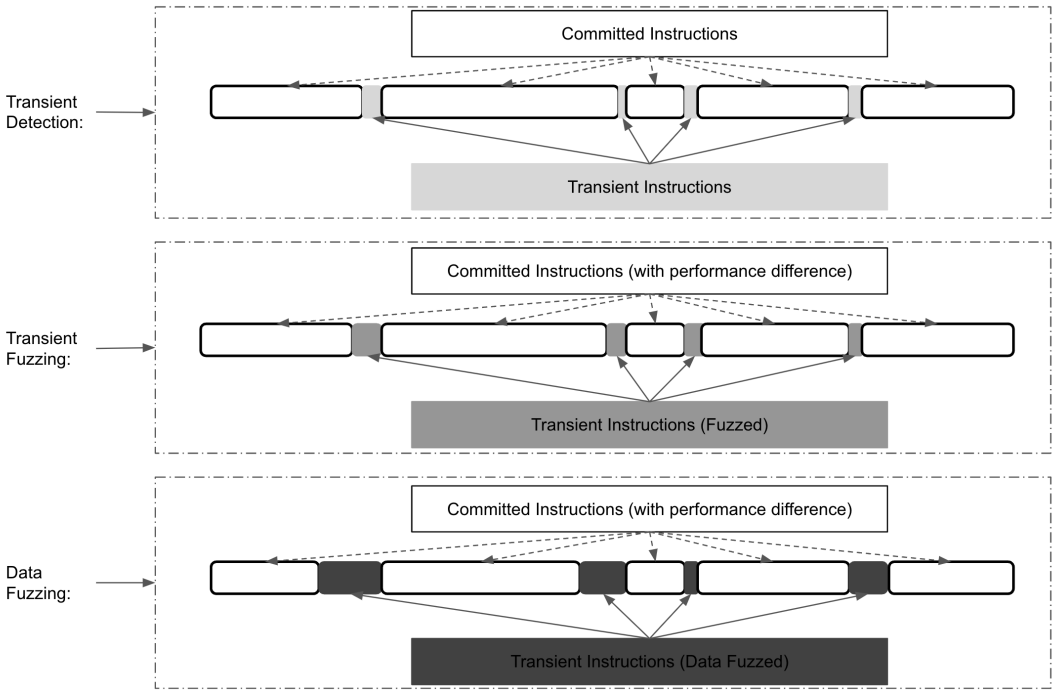

Figure 3.2: The three types of proposed runs.

the possible performance counters available in the CPU. Some RISC-V CPUs have programmable performance counters at boot time [29,35]. For example, counter 22 [29, 35] could be used as an L1 miss counter or for the number of TAGE updates. If these counters are programmable and exposed, the counter should be included in the verification. This is similar to the `page_walk` counter described in Section 3.2.

Comparing the hundreds of counters the CPU has at every cycle may have some performance overhead, but we can have a hash function of the counters. It is enough to create a hash of the whole execution and compare the hashes across data fuzzing tests, simplifying the comparison.

3.3.1 Protection Set

One of the contributions of this work is to use a simple and hence easy-to-understand method to avoid speculative side-channel attacks. This work claims that any speculative side-channel attack can be avoided if there are *no data-dependent transient leaks with a measurable side-effect*. In other words, it is not possible to have an attack unless transient instructions can have different measurable side-effects based on data.

This definition is a superset of what is needed for an attack. Some examples of why this is a superset:

- **Impractical:** Most attacks require multiple measurements. It may not be a practical vector if this is a very noisy attack.
- **Complete:** It protects against all data, while some implementations may want to protect only a range of data.
- **Broken:** Not completely protected systems may still be better than no protection.
- **Fuzzed transients:** We propose fuzzing transients which may not happen in real systems.

The **impractical** protection is the point that most randomization techniques argue. Although reasonable, it makes verification a much more complicated problem. Instead of no side-channel leaks, it is required to verify not enough leaks, with a difficult to quantify significance because it is quite sensitive to measuring techniques and the type of attack used (combination of transients). This may be a potential future work, but

for the moment, we consider a more straightforward definition of zero leaks allowed as it is easier to handle and verify.

The **complete** vs. "only a subset" is one of the differences with work like UPEC. A complete approach protects all the memory and registers. Works like UPEC protect only a memory region. We consider that everything should be included. It is possible to have contents on the register file that should not be visible. Namely, a thread register's contents should not leak. Protecting the register file and a programmable memory region is not much different from protecting everything. If the CPU designers had an implementation that does not protect a memory region, the proposed work would still be compatible and applicable by ensuring that the fuzzing does not generate addresses to those unprotected regions.

The **broken** system may still be possible to verify. The fuzzer generates changes in the transient instructions to measure their impact. If something is not protected in the CPU, the verification infrastructure is still valid as long as the broken portion of the design is not fuzzed. For example, the fuzzer could avoid the division input/outputs if the division is a telescopic unit. We do not advise this approach, but it shows the potential to at least verify that parts of the CPU are safe from attacks.

The **fuzzed transient** is a technique that we propose to increase the coverage and reduce the verification time. However, conceptually, this can lead to impossible to execute paths if we control the binary being tested. For example, the fuzzed transients may use RISC-V CSR instructions, but the binary may not have any such instructions. Hence, no matter what the hardware did, the software would have never triggered such

a condition. Notice that this also requires having "software" control. Any possible transient sequence can be created if we do not control what the compiler generates for the speculative path. This implies that a more restrictive definition can exist if we control the software stack. This work focuses on CPU/RTL verification without requiring software control. It may be an exciting restriction for future work, but we think that being software-independent is a better approach for CPU/RTL verification.

These are some more apparent examples of why our approach is a superset. We argue that this is a good superset for two reasons: (1) It is easy to explain and verify; (2) existing cores that protect against any speculative side-channel attack adhere to this. As the evaluation shows, BlackParrot and CVA6 have speculative and non-speculative safe. The verification infrastructure can detect leaks in the non-speculative safe versions and verify that the speculative-safe configurations do not have any data-dependent transient leak.

3.3.2 Speculation Fences

A potential solution to avoid speculative side-channel attacks is to extend the ISA to insert fences to avoid speculation [23]. These speculative fences effectively mean that there is no side-channel speculation. In an architecture without leaks, the fence can become a no-op. Alternatively, if the CPU only protects speculation against branches without fences, the proposed verification should apply only to the fenced branches. The others are allowed to have leaks.

The proposed co-simulation can be deployed in a CPU with only speculative

fence leak protection. In such CPUs, the transient fuzzing would happen only when the speculation fence notifies the region to protect.

3.3.3 Non-Transient Leaks

This work focuses on leaks by transient instructions, but there is an interesting case when a non-transient instruction is the source of transients. Many instructions can trigger pipeline flushes, for example, branch mispredictions or exceptions. The interesting case happens because the pipeline flush can be data-dependent. For example, a branch can be predicted correctly or incorrectly depending on the branch input source values. The pipeline flush will have a number of cycles impact on the execution, and hence it can be used as a side-channel attack. We do not consider this a transient leak because the branch itself is non-transient.

In the late 90s [58], it became known that many encryption algorithms were susceptible to non-transient side-channel attacks. Specifically, it was observed that different data had different branch prediction performances, and this change in performance could be observed by other applications or code sections, which compromised the algorithms. Since then, most encryption algorithms have been implemented with constant execution time to avoid leaking information through the branch prediction timing side-channel. This work does not protect against these non-transient side-channel leaks, only for transient leaks. Even if no transient is executed, this variable latency is a source of leak independent of speculative side-channel attack.

3.4 Fuzzing Transients

Fuzzing transients is the critical step in co-simulation to verify that the CPU/RTL does not have speculative side-channel attacks. As previously stated, this is done by checking that no transients can have data-dependent side-effects in execution time or performance counters.

The high-level process is to leverage co-simulation. As usual, the CPU/RTL is verified against the Instruction Set Simulator (ISS) model, but we introduce some additional steps to Fuzz Transients as shown in Figure 3.2. We detect the transients during the first co-simulation run, which serves as a *baseline* to increase speculative path coverage. A second run fuzzes the transient *opcodes* to increase coverage, and its output is kept as a reference. Additional runs fuzz the transient *data* sources to provide different data values effectively. There is a *leak* whenever there is any mismatch between the second and successive runs with data fuzzing.

3.4.1 Transients Opcodes

A random instruction generator (RIG) like Google DV [34] is a standard tool used together with co-simulation. The verification team keeps creating random programs and performs RTL co-simulation until some metrics like coverage are satisfied.

We must create many possible transient sequences to create any potential past/future attack. In theory, if we run randomly generated programs from RIGs enough times, we should hit any possible attack. In reality, achieving such coverage in modern

CPUs is not easy. For most CPUs, the transient path is branch predictor dependent, and it is harder to create extensive coverage than in typical programs. The intuition is that only a few possible random paths may exist for a given branch.

We propose to fuzz the speculative path at co-simulation time to speed up the verification steps. We propose to fuzz the transient opcodes. The decode stage of most CPUs is an accessible place to perform such fuzzing. Instead of using the instruction cache data, we provide a fuzzed opcode.

3.4.2 Transient Data

While fuzzing transient opcodes helps to reach high coverage faster, fuzzing transient data is what increases leak coverage.

After the fuzzed transient opcode is created, we do additional runs fuzzing the data. There are several ways to fuzz the data, but generally, we generate different input values (operands) for the transient instructions in all cases.

The easiest way is to fuzz the instruction source registers to random sources. This allows us to use the same fuzzing location in the decode stage for transient opcode and transient data fuzzing. This fuzzing is dependent on the register value coverage. An approach with faster coverage is to provide a register file read with fuzzed data for transient instructions. This requires further changes at the decode stage in addition to the transient opcode fuzzing.

3.5 Flow

Co-simulation is an RTL verification technique that couples a simpler golden model to check an RTL model. In CPU verification, the simpler golden model is an Instruction Set Simulator (ISS) without any timing information. As instructions update the micro-architectural state in the RTL, the golden model advances to the next instruction, and the result is verified.

Several pieces are needed to have an effective co-simulation environment. The most important is the capacity to generate test inputs and decide when to stop testing. The Dromajo [48] work explains in more detail the open-source RISC-V co-simulation infrastructure that we use as a starting point for this paper.

3.5.1 Random Instruction Generation

A Random Instruction Generator generates randomized assembly instruction streams for a given set of configurations. The tests generated by the RIG sweep a broad range of implemented functionality. It can create complex test cases that are hard to come up with for an engineer [7, 102].

3.5.2 Enough Testing

A crucial decision in co-simulation is when there have been enough RIG tests. Different teams have different approaches, but metrics like toggle coverage and line coverage are usually used to detect enough bugs. As [48] shows, bugs can hide under perfect coverage cases. This is why new tools like Coverity [86] try to test the quality

of testing infrastructure.

This work does not introduce any new novelty in deciding when there is enough testing. The same metrics and methodologies used for traditional co-simulation, like line coverage, are applied when detecting transient leaks.

3.5.3 Performance Counters

The golden model or ISS is typically not able to verify performance counters. The reason is that many factors, like branch miss predictions, are RTL-specific, not architectural. To verify against speculative side-channel attacks, we must check if transients do not have data-dependent performance counter updates. Section 3.4 shows that we verify performance counters against previously fuzzed transients, not against the ISS model.

3.5.4 Flow Runs

A simple approach will perform two runs: baseline and transient data fuzzing. The baseline run gathers performance counters and is compared against the run with transient data fuzzing. Any mismatch is a leak.

This approach is not as efficient because it is very sensitive to the RIG to generate potential attacks, and the branch predictor can select those RIG-generated transients.

The proposed framework performs three types of runs: *Transient detection* (baseline run), *transient fuzzing* (where the opcode is fuzzed), and *data fuzzing* (where

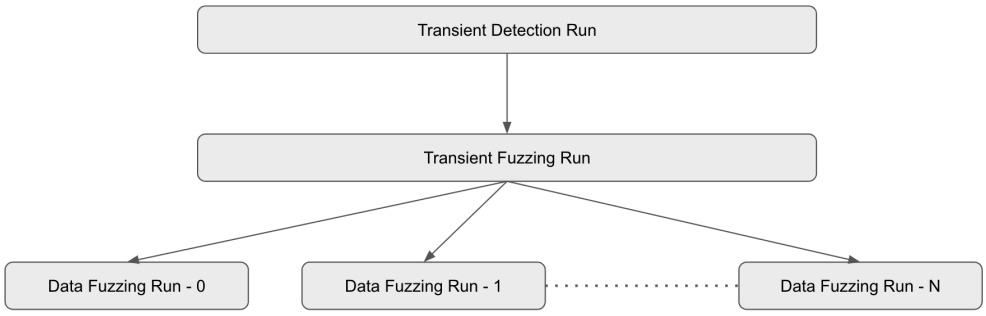

Figure 3.3: Overall flow with the three types of proposed runs.

the operands of the fuzzed opcode are fuzzed).

The first baseline run is to find the transient paths. The second run is to fuzz the transient opcodes. The following runs use the same fuzzed transients opcodes but fuzz the transient data. A verification error is raised if any measurable difference in time or performance counters happens between the last runs. The advantage of this methodology is that high coverage can be achieved much faster. The overall flow with *transient detection*, *transient fuzzing*, and *data fuzzing* is shown in Figure 3.3.

3.5.5 Sample Leaks

This Section showcases some sample cases to illustrate how the flow detects leaks:

Non-flushable variable latency instruction: Any long latency transient instruction with a data-dependent latency is a problem if it can not be flushed with constant time. Imagine a load or a division that takes more time based on a given input value. If any resource is reserved or blocked after the transients are flushed, we can have a potential source of an attack.

The proposed framework finds this issue by fuzzing the data from all the transients. The framework will find a leak if data values impact future instructions differently.

Any data-dependent micro-architectural state update: If a transient has data-dependent micro-architectural updates, it can have a performance impact on future instructions. For example, updating the cache or branch predictor will affect the execution time. Again, by fuzzing the transient data sources, this can be found.

Variable latency transients: Variable latency transients can complicate the whole design even if they have no side-channel leaks. This is because chaining variable latency instructions could stall the pipeline, effectively creating a data-dependent set of transient instructions. If the CPU allows this, then even data-independent leaks can become a source of leaks. For example, a function return (`ret`) can update the return address stack all the time. This is a data-independent leak. However, if we can insert variable latency before the `ret` instruction, we can avoid fetching or not fetching the `ret` instruction. Since it leaks, we have leveraged a variable latency transient without side-effects to use a data-independent transient to create the problematic data-dependent leak.

The proposed flow captures this because the variable latency transient will allow adding more or less fuzzing transient opcodes. The first run detects the non-transient instructions. All the new instructions fetched are transient and subject to opcode fuzzing.

Data-dependent priority inversion: An interesting case that the proposed

framework captures is a priority inversion. By priority inversion, we refer to the case that a younger transient should not affect the performance of an older one. If an older transient can affect a non-transient, there is a source of performance leak. This means that a transient load without any micro-architectural impact can still leak. For example, if an older non-transient load is issued after the transient load, the older one should not stall due to the younger transient one.

Again by fuzzing the transient data and combine with RIGs, we can create corner cases that will exercise this case.

3.5.6 Sample Non-Leaks

Data prefetchers in most modern processors are a potential source of leaks as they use the transient memory operation's address. If the prefetcher were triggered once the memory operation is no longer speculative, there could still be speculation due to the prefetcher, but it is not transient data-dependent. Such prefetchers will not trigger an issue in the proposed framework because we only fuzz transients.

3.6 Other Considerations

One caveat to remember is that even if this proposed framework does not find any timing side-channels, it does not mean that non exist. Similar to many verification methods mentioned in Section 2.2, this is true for the proposed framework. However, The proposed framework is not standalone. It should be used along with conventional verification techniques. We run the proposed framework along with coverage based tests.

The two methods are complementary and will increase the overall coverage. We can only try to maximize coverage and our confidence in our framework and the verification process. Like with most verification methods, it is incredibly hard to prove a design is completely bug-free.

Some attacks utilize sophisticated, hand-crafted gadgets to be executed on the speculative path. While the proposed framework can support sophisticated, hand-crafted gadgets to be executed in the prescribed manner, it defeats one of the proposal's goals to identify new timing side-channels. Having sophisticated hand-crafted gadgets only allows us to check for vulnerabilities that have been already discovered. Our objective is to utilize pseudo-randomness to our advantage and maximize coverage and reach hard to encounter cases. This also lets us integrate better with the conventional verification methods and use similar coverage metrics.

3.7 Implementation setup

This section discusses the implementation of the framework described earlier in this chapter using existing designs and tools. We have implemented this framework on three RISC-V cores, Ariane [104] (CVA6), BlackParrot [8], and BOOM [14] using Dromajo [88] as the co-simulation tool.

Two of the CPUs we chose, Ariane and BlackParrot, both use in-order commit logic, whereas BOOM is an out-of-order core. We have performed numerous runs on several benchmarks.

3.7.1 Ariane

Ariane, also known as CVA6, is a 6-stage, single issue, in-order CPU design implementing the RISC-V instruction set. Ariane has a speculative frontend. It implements the I, M, A, and C RISC-V 64-bit extensions, privilege extension, and debug specifications. It is capable of booting Linux, and its architectural structures like the Branch History Table (BHT), Branch Target Buffet (BTB), Return Address Stack (RAS), and Translation Lookaside Buffer (TLB) are configurable. We have used the default configuration. It is primarily implemented in System Verilog.

The architectural features of Ariane make it ideal for testing the proposed framework as the speculative frontend allows us to test our proposed methodology and the in-order commit allows us to have a deterministic reference trace, ensuring reproducibility of the results.

3.7.1.1 Dromajo integration with Ariane

As described in Section 2, Dromajo, developed by Esperanto Technologies, is a RISC-V RV64GC emulator that was primarily designed for RTL co-simulation. We leverage its ease of use and speed and integrate it into the Ariane core through the System Verilog Direct Programming Interface (DPI). As Dromajo is linked to the simulator as a shared library, it also allows us to instantiate multiple instances of Dromajo during simulation.

Conventional co-simulation flows integrate the co-simulation tool only at the commit stage of the pipeline to check the committed instructions. We integrate Dromajo

at two different pipeline stages in the processor. This is done in the following stages:

- **Commit Stage:** Like other conventional co-simulation methodologies, we integrate Dromajo at the commit stage of the DUT. This is done to verify the DUT's execution trace against the reference model. This also helps us determine if any illegal commits have been caused by manipulating the fetch instruction in the transient path.

- **Frontend:** We have a separate instance of Dromajo integrated to run at the frontend, at the fetch stage of the pipeline. This allows us to identify transient instructions during *Transient Detection* and is independent of the check that is performed at commit. This makes sure that any manipulation performed does not influence the original reference trace and that the instance of Dromajo at the commit stage does not run ahead of the actual commit performed by the DUT.

At the frontend, during *Transient and Data Fuzzing*, when the transient fetches are identified, we use a RIG (Google-DV) to replace the transient instructions. Changing just one instruction is not enough to expose complex cases that cause timing side-channels. We manipulate the transient fetches at different identified transient phases. We have designed the framework to implement manipulation at all transient fetches, specific transient fetches, and random transient fetches. This maximizes the potential cases exposed over time, over multiple runs, and using different random instruction streams.

Once any timing side-channels are found, we can analyze the entire execu-

tion trace until that point using conventional methods like waveform analysis using the generated `.vcd` files. This allows us to analyze the operation of this verification framework and manipulate it at cycle level granularity. It is also possible to configure the framework to continue execution after detecting the timing side-channel if necessary. To the best of our knowledge, this would yield no additional benefit. Once a timing side-channel is identified, the execution traces differ, and there is no correlation with the *Transient Detection* trace.

3.7.2 BlackParrot

BlackParrot is an open-source, Linux-capable, cache-coherent, RV64GC multicore capable design. It implements the I, M, A, F, and D RISC-V 64-bit extensions and the SV39 Virtual Memory specifications. It is designed to be tiny, modular, and friendly to use. It can be configured to be a single core or multicore system, with the support for a race-free MESI based coherent cache for the multicore configuration.

Building on the project's goal to be modular, the design of the core comprises a Front End (FE), a Back End (BE), and a Memory End (ME). The Front End is primarily responsible for PC generation (speculative as necessary) and instruction memory access and is controlled by the Back End to match execution behavior. The Back End is responsible for non-speculative, in-order execution and receives the speculative instruction stream from the Front End. The Back End is controlled using the Detector (hazard detection, stalling), Director (controlling the Front End for Speculative cases), Scheduler (dispatching instructions), and Calculator (non-stalling pipeline for execu-

tion). The Memory End interfaces both these modules through the instruction and data caches. This modularity is configurable to be single core or multicore. Another critical feature of BlackParrot is tracking speculative memory accesses using a flag and having different running modes to support or ignore them.

3.7.2.1 Dromajo Integration with BlackParrot

Integrating the Dromajo co-simulation tool with BlackParrot is a similar process to that of Ariane mentioned in Section 3.7.1.1. We integrate Dromajo at two locations, the Frontend, between PC generation and fetch, and the Back End, at Commit. The transient instructions are identified similarly and are dumped as a reference trace during *Transient Detection*. The identified transient instructions are manipulated during *Transient and Opcode Fuzzing* using the methods mentioned in Section 3.3.

3.7.3 BOOM

The Berkeley Out-of-Order Machine (BOOM) [14] is a synthesizable and parameterizable open-source RISC-V out-of-order core. BOOM implements the open-source RISC-V ISA and utilizes the Chisel hardware construction language to construct generator for the core. BOOM is a family of out-of-order designs rather than a single instance of a core, as different configurations can be generated based on specifications provided to the generator. BOOM supports RV64GC and the privileged ISA which includes single-precision and double-precision floating point, atomics support, and page-based virtual memory.

3.7.3.1 Dromajo Integration with BOOM

Once again, integrating the Dromajo co-simulation tool with BOOM is a straightforward process similar to the other CPU designs mentioned above. The same process of integrating Dromajo at the Frontend and at Commit and *Transient Detection, Transient Fuzzing,* and *Opcode Fuzzing* are used.

Chapter 4

Evaluation

We describe the three CPU designs CVA6, BlackParrot, and BOOM, in Section 4.1. In Section 4.2 we provide the results of using our framework with these three cores.

4.1 Setup

We evaluate our proposal using three RISC-V processor designs, Ariane, BlackParrot, and BOOM. We use Google's RISCV-DV [34] to generate our random instruction streams. We tested the processor designs using RISCV Tests and Benchmarks and random instruction streams generated by RISCV-DV.

BlackParrot [8] has an optional flag to avoid speculative side-channel attacks. From a high level, when enabled, it treats the loads as accessing non-cacheable requests; hence, non-speculative requests are allowed. It also avoids updates on the branch predictor. In the evaluation, we consider additional flags like making all the memory requests

non-cacheable. The evaluation uses the names BP-safe (speculative side-channel safe configuration), BP-base (default or baseline configuration), and BP-NC (non-cacheable memory requests) for three variations of BlackParrot.

CVA6 [104] is sensitive to speculative side-channel attacks. A version of the core was modified by UPEC [25] to make the core safe. This patch handles speculative side-effects from the i-cache and other minor issues. The evaluation uses the names CVA6-safe and CVA6-base.

Being an out-of-order design, BOOM is also vulnerable to speculative side-channel attacks. It can also be configured to have different complexities. To the best of the author's knowledge, there are no side-channel safe versions of BOOM available.

The co-simulation uses Dromajo [48]. For CVA6, instead of performing the first run to detect transient instructions, we perform two simultaneous Dromajo executions so that a single pass can detect transient instructions. This is a small change to avoid requiring creating traces, but it is conceptually the same.

4.2 Results

Since both CVA6 and BlackParrot have variants that are safe (BP-safe, CVA6-safe) and vulnerable (BP-base, BP-NC, CVA6-base) to speculative side-channel attacks. This makes them good platforms for checking the effectiveness of our approach and how fast we can detect transient leaks. Evaluating our framework with BOOM proves the framework's validity for out-of-order designs.

4.2.1 Correctness

The proposed framework verified that BP-safe and CVA6-safe did not have any issues.For the other configurations, the following issues show some that were problems detected:

4.2.1.1 CVA6-base

The default CVA6 version (CVA6-base) is sensitive to side-channel attacks. The co-simulation was able to find the following issues:

- Division: CVA6 has a telescopic division unit with a maximum latency of 64 cycles. Due to this, we found time side-channels by performing fuzzing with division transients. This has similarities to the attack explained by SpectreRewind [30] but instead of the floating-point unit with the division unit. The leak could have been avoided if the telescopic unit had been flushed when the transient instruction was discarded. CVA6 does not do that, and as a result, the program IPC is affected by the data in the transient path if the following instruction tries to use the division unit.

- CSR: read and write CSR transient fuzzing revealed time side-channels in CVA6-base. We are unaware of RISC-V-specific CSR attacks in the speculative path, but time variations due to CSR operations are potential sources of attack in CVA6. Both transient CSR read and writes show time dependence impacts on future instructions.

- Fence: Opcode fuzzing with fence instructions revealed time side-channels in CVA6-base.

4.2.1.2 BP-NC and BP-base

BlackParrot has several configuration options. Useful for this work is the BP-NC or non-cacheable. Enabling this essentially means that all the memory operations are not speculative. BP-base is the baseline/default BlackParrot design. One leak was found in both configurations:

- CSR write: Speculative side-channels were identified with CSR write transients. This is similar to the CVA6 CSR leak, but the CSR read did not produce a side-channel.

The fuzzing did not find any leaks with BP-safe and CVA6-safe.

4.2.1.3 Out-of-Order designs

We also integrated our framework with BOOM [14] to show that this framework works with out-of-order designs. BOOM does not have protections or configurations to disable transient side-effects like CVA6 or BlackParrot. As a result, just running Dhrystone on BOOM is enough to detect transient leaks. Despite that, tests were run using RISCV Tests and Benchmarks and random instruction streams as the test program. Most inserted transients caused a measurable change in performance due to its OoO and highly speculative nature.

A leak detected showed that transients updated the branch predictor, effectively leaking state by affecting future branch outcomes. Updating the branch predictor at retirement instead of execute will avoid this source of the leak. Another leak showed that transients updated the return address stack. This then affected future return instructions. Yet another leak showed that transient load and store instructions updated the cache behavior, leading to another side-channel. While fixing such easily triggered side-channels is exciting future work, patching BOOM is not a trivial problem. This proposed framework opens the opportunity to do so incrementally.

4.2.1.4 Comparison with competing approaches

When verifying the CVA6-base version, we were able to replicate the findings of UPEC [25], where the instruction cache allowed a user-level process to load cachelines from inaccessible addresses. When testing with the CVA6-safe version, we verified that the patches provided by the UPEC team were sufficient, and the problem did not persist. We did not find any additional problems with the CVA6 cores. This furthers the validity of our framework.

4.2.2 Coverage

	CVA6-base	**BP-base**	**BOOM**
Transient Fuzzing Coverage	51.80%	75.10%	61.90%
Data Fuzzing Coverage	52.10%	75.80%	63.10%

Table 4.1: Line coverage metrics for CVA6-base, BP-base and BOOM.

We evaluated the CVA6-base, CVA6-safe, BP-base, BP-safe, and BOOM de-

signs for coverage analysis. We tracked the Line Coverage metrics through Verilator simulation. We compared the coverage numbers for running a single test iteration, i.e., one Transient Detection run, one Transient Fuzzing run, and one Data Fuzzing run. Table 4.1 shows the increase in coverage between the Transient Fuzzing and the Data Fuzzing runs. We see a minor increase in overall coverage as we fuzz the data via operand fuzzing. This is caused by different execution paths triggered by changing the operands. The increase in coverage is minor because we only fuzz the operands, causing corner cases otherwise not present.

	CVA6-safe	**BP-safe**
Transient Fuzzing	52.20%	75.70%
Data Fuzzing	52.60%	76.30%

Table 4.2: Line coverage metrics for CVA6-safe and BP-safe.

Table 4.2 shows the increased coverage between the Transient Fuzzing and the Data Fuzzing runs for the CVA6-safe and BP-safe configurations. Once again, like their base counterparts, we see a minor increase in overall line coverage as we fuzz the data via operand fuzzing.

While it can be argued that such minor changes in coverage is a drawback, we firmly disagree. The reason for this is that we are able to find side-channels in most of the tests with such a minor increase in coverage. We see this as an advantage of the proposed framework.

4.2.3 Efficiency

Traditional co-simulation requires many RIG tests to verify a core. One of the main decisions in co-simulation verification is how many tests to run. A similar question is how many additional tests are required for verifying against speculative side-channel attacks.

Config	Total Execution
CVA6-base (time - s)	10,199.97
BP-base (time - s)	14,983.84
BOOM (time -s)	13,841.42
CVA6-base (cycles simulated)	5,916,132
BP-base (cycles simulated)	3,684,228
BOOM (cycles simulated)	5,321,984

Table 4.3: Total execution time and cycles simulated for CVA6-base, BP-base, and BOOM configurations.

The proposed co-simulation requires three passes: the original, the transient opcode fuzzing, and the transient data fuzzing. It is possible to overlap the opcode fuzzing and transient fuzzing, but we found it unnecessary. The reason is that for each baseline configuration (BP-base and CVA6-base), Dhrystone [99] is enough to find leaks. While Dhrystone may not find speculative side-channels due to Floating Point instructions, it did prove the existence of leaks. Traditional co-simulation requires many tests, but fuzzing quickly exposes the bugs, and for BP-base and CVA6-base, even Dhrystone is enough. As a reference, BP-base ran Dhrystone in 216 seconds for the first pass in a Xeon CPU E5-2689 v1. The transient fuzzing run was even faster at 171 seconds because it finished when an issue was detected. The fuzzing overhead is negligible compared with the RTL and co-simulation overhead. If the fuzzing error

termination is not triggered, fuzzing adds a 5.2% co-simulation slowdown. The result is that BP-safe co-simulation leaks in less than 10 minutes. CVA6 was slower, requiring 276 seconds per run. In total, it also finished in around 10 minutes.

The clear implication is that having transient opcode and data fuzzing is a very effective way to find leaks. This is one of the reasons for not running Google DV [34] in this evaluation. RISC-V tests [28] and RISC-V benchmarks are enough to find bugs. A more powerful Google DV will help but does not seem as needed compared to traditional co-simulation verification for BlackParrot and CVA6 type of cores.

Table 4.3 shows the total runtime of the simulations for the CVA6-base and BP-base configurations. To declare the safe variants as secure, we ran simulations to match a similar number of cycles simulated using our framework.

4.2.4 Data-insensitive Transients

One interesting observation from BlackParrot is that it has side-effects from transient instructions, but it is not a source of leaks. We modified the verification infrastructure to capture if any transient has data-independent side-effects. This can be done by performing multiple transient opcode fuzzing. By comparing different opcode fuzzing passes, we can detect if there is any side-effect or leak.

If any of the different transient opcode fuzzing passes have a new leak not seen with the default data fuzzing, then the inserted opcodes have data-independent leaks. Another way to say the same is that if executing a transient affects future instructions, we have a leak that may be data-dependent or data-independent. Since we tested for

data-dependent leaks, then the leak must be data-independent. CVA6-safe did not have any data-independent leaks, but BP-safe had data-independent leaks only for CSR RISC-V opcode type.

BP-safe had a performance change when a RISC-V CSR opcode replaced a transient instruction. This did not enable a speculative side-channel attack because BP-safe had no data-dependent variable latency instruction. In BP-safe, loads are non-cacheable, hence always inefficient but at the same latency. The branches are delayed until becoming non-speculative. The result is that the CSR data-independent leak cannot be leveraged.

If the BP-safe were to implement cacheable transient loads but without side-effects, the CSR and fences could be a source of a speculative side-channel attack. The reason is that long-latency loads could stall the pipeline before reaching the CSR. Thus short-latency loads would not reach the CSR, effectively enabling a leak. In other words, if the transient can affect the number of transients in the wrong path, then no transient can have any side-effect. In BP-safe, there is no data-dependent transient affecting the number of transients; therefore, the leak by a transient like CSR or memory fence is not an issue.

This is a complicated pattern that BP-safe allows, but there is nothing special in this work proposed methodology flow to handle it. By fuzzing the transient data, we can automatically observe these combined interactions. The leak would have been detected if BP-safe had allowed a transient instruction to affect the number of transients based on some register or memory data.

An interesting corollary from the previous observation is that CPUs must protect from data-insensitive transient leaks when the transients can trigger data-sensitive pipeline stalls. This type of core can have an even faster speculative side-channel co-simulation verification. This is because it is possible just to fuzz transient opcodes and compare against them, i.e., there is no need to fuzz transient data. CVA6 is a core that also passes this faster verification. The advantage is more efficient fuzzing and fewer runs, but as previously mentioned, the current fuzzing is fast enough that this does not seem necessary.

4.2.5 Gaining insights

As we record the changes made to the instructions and operands during the Transient Fuzzing and Data Fuzzing steps, and we stop the simulation as soon as any deviation in performance counters or performance of the program is noticed, we are able to identify performance differences caused by the introduced changes. For example, in CVA6, when a `division` instruction was inserted in the transient path, it caused a delay in the commits for the tests that used the division functional unit. Simple cases like these are easy to identify. Similarly, micro-architectural changes were easy to pinpoint using logs dumped during simulation. For example, a transient store instruction fuzzed used a CSR as its operand indirectly, causing changes in the cache line replacement, which was reflected in a non-transient load instruction a few cycles later. The most complex case would be interactions involving all the inserted transient instructions until the point of divergence. We did not encounter this case.

Choosing what metrics to track is tricky. We chose to track all implemented performance counters in the designs we tested. Performance counters can vary based on implementation given the same ISA. We recommend tracking all performance counters regardless of the design.

4.2.6 Other Side-Channels

We observed other side-channels in all three CPU designs that are not timing side-channels. These are discussed below.

- **Cache:** We observed changes in caching behavior in all three CPU designs due to fuzzed transient loads/stores. These fuzzed instructions resulted in cache lines being replaced.
- **Branch Prediction Structures:** In all three designs, we observed changes in the branch prediction structures like branch target buffers and branch history tables when the fuzzed transient was a branch type instruction. These changes were essentially new entries in the structures corresponding to the PC of the Fuzzed Branch instruction.
- **Return Address Stack:** Similar to the branch prediction structures, when the Fuzzed Transient was a return instruction, the return address stack was updated in all three processors.

While these are not timing side-channels, we did notice the side effects which later caused divergence in the timing of the commits. For example, when we targeted

a specific cacheline which was already cached and was used by a non-transient load instruction, the result of the cacheline being evicted due to the fuzzed transient load led to the non-transient load having to stall as the cacheline had to be brought back to the L1 cache. This resulted in the commit timing of the non-transient load instruction being delayed. Our framework can capture complex scenarios like these, using random testing and hand-crafted gadgets.

Chapter 5

Conclusion

This work presents a novel way to verify CPU/RTL designs against speculative side-channel attacks. For the industry to design cores safe from attacks, they need ideas to implement but also a methodology to verify the design. This work proposes a verification infrastructure that can detect speculative leaks. Since it protects against any potential speculative side-channel attack or leak, it does not require attack vectors.

Starting with a co-simulation setup, we add additional passes to generate any potential source of attack efficiently. This is achieved by having a transient opcode fuzzing followed by multiple transient data fuzzing.

To evaluate the correctness of the proposed design, we leverage that there are two cores (BlackParrot and CVA6) with patches or configuration options to have a baseline design and a speculative side-channel attack safe mode. In both cores, we instrument the decode to fuzz transients. Even co-simulating the simple Dhrystone is enough to detect leaks in the base designs. We run many more tests, and the safe

version of the cores has no leaks. We also showed that our framework can be integrated with complex out of order designs like BOOM.

In summary, the framework presented allows verifying CPU/RTL designs when focusing on performance and performance counters speculative side-channel attacks. The major advantage of this framework is the ability for designers to make design changes and quickly check for speculative side-channel attacks. This is achieved by extending the commonly used co-simulation platforms already used by academia and industry.

Bibliography

[1] Onur Aciiçmez. Yet another microarchitectural attack: exploiting I-cache. In *Proceedings of the 2007 ACM workshop on Computer security architecture*, pages 11–18, 2007.

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

[3] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *Cryptographers' Track at the RSA Conference*, pages 225–242. Springer, 2007.

[4] Allon Adir, Eli Almog, Laurent Fournier, Eitan Marcus, Michal Rimon, Michael Vinov, and Avi Ziv. Genesys-pro: Innovations in test program generation for functional processor verification. *IEEE Design & Test of Computers*, 21(2):84–93, 2004.

[5] Sam Ainsworth and Timothy M Jones. Muontrap: Preventing cross-domain

Spectre-like attacks by capturing speculative state. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 132–144. IEEE, 2020.

[6] AMD. Software techniques for managing speculation on amd processors.

[7] W. Anderson. Logical verification of the NVAX CPU chip design. In *Proceedings 1992 IEEE International Conference on Computer Design: VLSI in Computers Processors*, pages 306–309, 1992.

[8] Azad, Zahra; Delshadtehrani, Leila; Zhou, Boyou; Joshi, Ajay; Gilani, Farzam; Lim, Katie; Petrisko, Daniel; Jung, Tommy; Wyse, Mark; Guarino, Tavio; Veluri, Bandhav; Wang, Yongqin; Oskin, Mark; Taylor, Michael. The blackparrot processor: An open-source industrial-strength RV64G multicore processor. Mar 2019.

[9] Jean-Loup Baer and Tien-Fu Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 176–186, 1991.

[10] Lucy Bowen and Chris Lupo. The performance cost of software-based security mitigations. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ICPE '20, page 210–217, New York, NY, USA, 2020. Association for Computing Machinery.

[11] Pablo Buiras, Hamed Nemati, Andreas Lindner, and Roberto Guanciale. Validation of side-channel models via observation refinement. In *MICRO-54: 54th*

Annual IEEE/ACM International Symposium on Microarchitecture, pages 578–591, 2021.

[12] Gianpiero Cabodi, Paolo Camurati, Fabrizio Finocchiaro, and Danilo Vendraminetto. Model-checking speculation-dependent security properties: Abstracting and reducing processor models for sound and complete verification. *Electronics*, 8(9):1057, 2019.

[13] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In *28th USENIX Security Symposium USENIX Security 19)*, pages 249–266, 2019.

[14] Christopher Celio. *A Highly Productive Implementation of an out-of-order Processor Generator*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2018.

[15] Christopher Patrick Celio. *A highly productive implementation of an out-of-order processor generator*. University of California, Berkeley, 2017.

[16] Kevin Cheang, Cameron Rasmussen, Sanjit Seshia, and Pramod Subramanyan. A formal approach to secure speculation. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*, pages 288–28815. IEEE, 2019.

[17] Wen Chen, Sandip Ray, Jayanta Bhadra, Magdy Abadir, and Li-C Wang. Chal-

lenges and trends in modern soc design verification. *IEEE Design & Test*, 34(5):7–22, 2017.

[18] Hana Chockler, Orna Kupferman, and Moshe Y Vardi. Coverage metrics for formal verification. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 111–125. Springer, 2003.

[19] Rutvik Choudhary, Jiyong Yu, Christopher Fletcher, and Adam Morrison. Speculative Privacy Tracking (SPT): Leaking information from speculative execution without compromising privacy. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 607–622, 2021.

[20] M. Chupilko, A. Kamkin, A. Kotsynyak, A. Protsenko, S. Smolov, and A. Tatarnikov. Test program generator microtesk for risc-v. In *2018 19th International Workshop on Microprocessor and SOC Test and Verification (MTV)*, pages 6–11, 2018.

[21] Shuwen Deng, Wenjie Xiong, and Jakub Szefer. Secure TLBs. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 346–359. IEEE, 2019.

[22] Susan J Eggers, Joel S Emer, Henry M Levy, Jack L Lo, Rebecca L Stamm, and Dean M Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE micro*, 17(5):12–19, 1997.

[23] Mathieu Escouteloup, Ronan Lashermes, Jean-Louis Lanet, and Jacques Jean-

Alain Fournier. Recommendations for a radically secure ISA. In *Workshop on Computer Architecture Research with RISC-V*, 2020.

[24] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over aslr: Attacking branch predictors to bypass aslr. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[25] Mohammad Rahmani Fadiheh, Alex Wezel, Johannes Muller, Jorg Bormann, Sayak Ray, Jason M. Fung, Subhasish Mitra, Dominik Stoffel, and Wolfgang Kunz. An exhaustive approach to detecting transient execution side channels in RTL designs of processors. *IEEE Transactions on Computers*, pages 1–1, 2022.

[26] Farzan Fallah, Srinivas Devadas, and Kurt Keutzer. Occom-efficient computation of observability-based code coverage metrics for functional verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 20(8):1003–1015, 2001.

[27] Andrew Ferraiuolo. *Timing-Safe Hardware-Level Information Flow Control*. PhD thesis, Cornell University, 2018.

[28] RISC-V Foundation, Jul 2013.

[29] RISC-V Foundation, March 2019.

[30] Jacob Fustos, Michael Bechtel, and Heechul Yun. Spectrerewind: Leaking secrets to past instructions. In *Proceedings of the 4th ACM Workshop on Attacks and Solutions in Hardware Security*, pages 117–126, 2020.

[31] Jacob Fustos, Farzad Farshchi, and Heechul Yun. Spectreguard: An efficient data-centric defense mechanism against spectre attacks. In *Proceedings of the 56th Annual Design Automation Conference 2019*, pages 1–6, 2019.

[32] Daniel Genkin, Adi Shamir, and Eran Tromer. RSA key extraction via low-bandwidth acoustic cryptanalysis. In *Annual Cryptology Conference*, pages 444–461. Springer, 2014.

[33] Moein Ghaniyoun, Kristin Barber, Yinqian Zhang, and Radu Teodorescu. Intro-Spectre: a pre-silicon framework for discovery and analysis of transient execution vulnerabilities. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 874–887. IEEE, 2021.

[34] Google. SV/UVM based instruction generator for risc-v processor verification, Jan 2019.

[35] RISC-V Foundation Compliance Task Group. *RISC-V Compliance Test Format Specification*, Dec 2018.

[36] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games–bringing access-based cache attacks on aes to practice. In *2011 IEEE Symposium on Security and Privacy*, pages 490–505. IEEE, 2011.

[37] Mentor Harry Foster. Part 8: The 2018 wilson research group functional verification study (IC/ASIC resource trends), 2019.

[38] John L. Hennessy and Norman P. Jouppi. Computer technology and architecture: An evolving interaction. *Computer*, 24(9):18–29, 1991.

[39] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[40] R. C. Ho, C. Han Yang, M. A. Horowitz, and D. L. Dill. Architecture validation for processors. In *Proceedings 22nd Annual International Symposium on Computer Architecture*, pages 404–413, 1995.

[41] Intel. Affected processors: Transient execution attacks and related security issues by cpu.

[42] Intel. Bounds Check Bypass Store (BCBS) vulnerability.

[43] Intel. Intel goldmont plus microarchitecture events.

[44] Intel. Lazy FP state restore.

[45] Intel. Microarchitectural Data Sampling advisory.

[46] Norman P Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. *ACM SIGARCH Computer Architecture News*, 18(2SI):364–373, 1990.

[47] Norman P Jouppi and David W Wall. Available instruction-level parallelism for superscalar and superpipelined machines. *ACM SIGARCH Computer Architecture News*, 17(2):272–282, 1989.

[48] Nursultan Kabylkas, Tommy Thorn, Shreesha Srinath, Polychronis Xekalakis, and Jose Renau. Effective processor verification with logic fuzzer enhanced co-simulation. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 667–678, 2021.

[49] Mohd Fadzil Abdul Kadir, Jin Kee Wong, Fauziah Ab Wahab, Ahmad Faisal Amri Abidin Bharun, Mohamad Afendee Mohamed, and Aznida Hayati Zakaria. Retpoline technique for mitigating Spectre attack. In *2019 6th International Conference on Electrical and Electronics Engineering (ICEEE)*, pages 96–101. IEEE, 2019.

[50] M. Kantrowitz and L. M. Noack. I'm done simulating; now what? verification coverage analysis and correctness checking of the DECchip 21164 Alpha microprocessor. In *33rd Design Automation Conference Proceedings, 1996*, pages 325–330, 1996.

[51] Christoph Kern and Mark R Greenstreet. Formal verification in hardware design: a survey. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 4(2):123–193, 1999.

[52] kernel.org. x86/speculation: Protect against userspace-userspace spectreRSB.

[53] Vladimir Kiriansky and Carl Waldspurger. Speculative buffer overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757*, 2018.

[54] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner

Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.

[55] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Annual international cryptology conference*, pages 388–397. Springer, 1999.

[56] Paul Kocher, Joshua Jaffe, Benjamin Jun, et al. Introduction to differential power analysis and related attacks, 1998.

[57] Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. Introduction to differential power analysis. *Journal of Cryptographic Engineering*, 1(1):5–27, 2011.

[58] Paul C Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

[59] Esmaeil Mohammadian Koruyeh, Shirin Haji Amin Shirazi, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Speccfi: Mitigating Spectre attacks using CFI informed speculation. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 39–53. IEEE, 2020.

[60] Kevin Laeufer, Jack Koenig, Donggyu Kim, Jonathan Bachrach, and Koushik Sen. RFUZZ: coverage-directed fuzz testing of RTL on fpgas. In Iris Bahar, editor, *Proceedings of the International Conference on Computer-Aided Design, ICCAD 2018, San Diego, CA, USA, November 05-08, 2018*, page 28. ACM, 2018.

[61] Johnny KF Lee and Alan Jay Smith. Branch prediction strategies and branch target buffer design. *Computer*, 17(01):6–22, 1984.

[62] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *26th USENIX security symposium (USENIX security 17)*, pages 557–574, 2017.

[63] Mikko H Lipasti, Christopher B Wilkerson, and John Paul Shen. Value locality and load value prediction. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 138–147, 1996.

[64] Moritz Lipp, Vedad Hažić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. Take-a-way: Exploring the security implications of amd's cache way predictors. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, pages 813–825, 2020.

[65] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *arXiv preprint arXiv:1801.01207*, 2018.

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

[67] Kevin Loughlin, Ian Neal, Jiacheng Ma, Elisa Tsai, Ofir Weisse, Satish Narayanasamy, and Baris Kasikci. DOLMA: Securing speculation with the principle of transient Non-Observability. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1397–1414, 2021.

[68] lowRISC. Ibex core verification, Aug 2019.

[69] Daniel Moghimi, Moritz Lipp, Berk Sunar, and Michael Schwarz. Medusa: Microarchitectural data leakage via automated attack synthesis. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1427–1444, 2020.

[70] G. E. Moore. Progress in digital integrated electronics. 21:11–13, 1975.

[71] Yehuda Naveh, Michal Rimon, Itai Jaeger, Yoav Katz, Michael Vinov, Eitan s Marcu, and Gil Shurek. Constraint-based random stimuli generation for hardware verification. *AI magazine*, 28(3):13–13, 2007.

[72] Oleksii Oleksenko, Christof Fetzer, Boris Köpf, and Mark Silberstein. Revizor: testing black-box CPUs against speculation contracts. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 226–239, 2022.

[73] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. Specfuzz: Bringing Spectre-type vulnerabilities to the surface. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1481–1498, 2020.

[74] Yale N Patt, Wen-mei Hwu, and Michael Shebanow. HPS, a new microarchitecture: Rationale and introduction. *ACM SIGMICRO Newsletter*, 16(4):103–108, 1985.

[75] Jean-Jacques Quisquater and David Samyde. Electromagnetic analysis (EMA): Measures and counter-measures for smart cards. In *International Conference on Research in Smart Cards*, pages 200–210. Springer, 2001.

[76] Anne Rogers and Kai Li. Software support for speculative loads. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 38–50, 1992.

[77] Gururaj Saileshwar and Moinuddin K. Qureshi. Cleanupspec: An "undo" approach to safe speculation. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '52, page 73–86, New York, NY, USA, 2019. Association for Computing Machinery.

[78] Robert R Schaller. Moore's law: past, present and future. *IEEE spectrum*, 34(6):52–59, 1997.

[79] Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. ConTExT: A generic approach for mitigating Spectre. In *NDSS*, 2020.

[80] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. Zombieload: Cross-privilege-boundary data

sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 753–768, 2019.

[81] André Seznec. A case for (partially)-tagged geometric history length predictors. *Journal of InstructionLevel Parallelism*, 2006.

[82] André Seznec. A new case for the TAGE branch predictor. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 117–127. ACM, 2011.

[83] Nikolay A Simakov, Martins D Innus, Matthew D Jones, Joseph P White, Steven M Gallo, Robert L DeLeon, and Thomas R Furlani. Effect of meltdown and spectre patches on the performance of hpc applications. *arXiv preprint arXiv:1801.04329*, 2018.

[84] Michael D Smith, Monica S Lam, and Mark A Horowitz. Boosting beyond static scheduling in a superscalar processor. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 344–354, 1990.

[85] Chris Spear. *SystemVerilog for verification: a guide to learning the testbench language features*. Springer Science & Business Media, 2008.

[86] Synopsys. Coverity sast software.

[87] S. Tasiran and K. Keutzer. Coverage metrics for functional validation of hardware designs. *IEEE Design Test of Computers*, 18(4):36–45, 2001.

[88] Esperanto Technologies. Dromajo - esperanto technology's RISC-V reference model, Dec 2019.

[89] James E Thornton. The CDC 6600 project. *Annals of the History of Computing*, 2(4):338–348, 1980.

[90] Linus Torvalds. Retpoline deemed impractical to exploit.

[91] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. Security verification via automatic hardware-aware exploit synthesis: The checkmate approach. *IEEE Micro*, 39(3):84–93, 2019.

[92] Timothy Trippel, Kang G. Shin, Alex Chernyakhovsky, Garret Kelly, Dominic Rizzo, and Matthew Hicks. Fuzzing hardware like software, 2021.

[93] Dean M Tullsen, Susan J Eggers, and Henry M Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the 22nd annual international symposium on Computer architecture*, pages 392–403, 1995.

[94] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 991–1008, 2018.

[95] I. Wagner, V. Bertacco, and T. Austin. Stresstest: an automatic approach to

test generation via activity monitors. In *Proceedings. 42nd Design Automation Conference, 2005.*, pages 783–788, 2005.

[96] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. OO7: Low-overhead defense against spectre attacks via program analysis. *IEEE Transactions on Software Engineering*, 2019.

[97] Kai Wang and Manoj Franklin. Highly accurate data value prediction using hybrid predictors. In *Proceedings of 30th Annual International Symposium on Microarchitecture*, pages 281–290. IEEE, 1997.

[98] Daniel Weber, Ahmad Ibrahim, Hamed Nemati, Michael Schwarz, and Christian Rossow. Osiris: Automated discovery of microarchitectural side channels. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1415–1432, 2021.

[99] Reinhold P Weicker. Dhrystone: a synthetic systems programming benchmark. *Communications of the ACM*, 27(10):1013–1030, 1984.

[100] Shlomo Weiss and James E Smith. Instruction issue logic for pipelined supercomputers. In *Proceedings of the 11th annual international symposium on Computer architecture*, pages 110–118, 1984.

[101] Johannes Wikner and Kaveh Razavi. Retbleed: Arbitrary speculative code execution with return instructions. In *31st USENIX Security Symposium (USENIX 2022)*, 2022.

[102] D. A. Wood, G. A. Gibson, and R. H. Katz. Verifying a multiprocessor cache

controller using random test generation. *IEEE Design Test of Computers*, 7(4):13–25, 1990.

[103] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W Fletcher. Speculative Taint Tracking (STT) a comprehensive protection for speculatively accessed data. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 954–968, 2019.

[104] 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*, 27(11):2629–2640, Nov 2019.

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