DISS. ETH NO. 30245

SOFTWARE-INSPIRED TECHNIQUES FOR DIGITAL HARDWARE SECURITY

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES
(Dr. sc. ETH Zurich)

presented by

FLAVIEN SOLT

Ingénieur diplômé de l'École polytechnique
MSc ETH ITET, ETH Zürich

born on May 16th, 1997

accepted on the recommendation of

Prof. Dr. Kaveh Razavi (Advisor)
Prof. Dr. Mathias Payer
Prof. Dr. Marco Guarnieri
Prof. Dr. Luca Benini

2024

Diss. ETH No. 30245
TIK-Schriftenreihe-Nr. 211

ABSTRACT

We entered an era where new hardware flourishes at an unprecedented pace and with unseen diversity. We are also living in an era where security and safety are paramount, and where the potential impact of a single bug can be catastrophic. Hence, we urgently need foundations to detect as many hardware bugs as possible before their deployment.

Hardware validation is universally recognized as complex, expensive and tedious. Despite genuine best efforts, the last decade has shown that the industry is incapable of producing non-trivial bug-free hardware. What will then happen with the rise of open-source hardware? Without effective and easy-to-adopt solutions for validation, it is hard to believe that the open-source hardware community will be able to produce safe and secure hardware, despite its best intentions.

Interestingly, the exact same situation occurred in the software world some decades ago. Software was plagued with myriads of bugs and security issues, after what the software community developed a formidable set of tools and methodologies to detect bugs and security issues. **Could we adapt some of these tools and methodologies to hardware?**

To answer this question, our plan is to first observe many CPU errata, deduce the most promising techniques from software security, and adapt them. To understand contemporary CPU bugs, we build the *RemembERR* database based on thousands of errata. We deduce two techniques inspired by software security that are particularly promising for hardware: dynamic information flow tracking and fuzzing. We introduce *CellIFT*, the first scalable hardware dynamic information flow tracking mechanism and showcase 4 new architectural or microarchitectural security applications. We then introduce *Cascade*, a black-box CPU fuzzer that found dozens of new bugs and outperforms other fuzzers' coverage. We finally demonstrate *MiRTL*, a new class of hardware attacks that relies on EDA software bugs, and propose *TransFuzz*, a fuzzer that produces complex hardware descriptions to find such bugs in popular open-source EDA software.

All these contributions demonstrate that when properly adapted, software security techniques can provide effective and easy-to-adopt solutions that will empower safer and more secure hardware.

RÉSUMÉ

Nous entrons dans une ère où le matériel (hardware) prospère à un rythme sans précédent et avec une diversité inédite. À une époque où la sécurité est cruciale, la détection précoce des bugs matériels devient impérative.

La validation matérielle est reconnue comme complexe, coûteuse et fastidieuse. Malgré des efforts sincères, la dernière décennie a révélé l'incapacité de l'industrie à produire un matériel sans bugs significatifs. Face à l'émergence du matériel open source, sans solutions de validation efficaces et faciles à adopter, il paraît invraisemblable que la communauté puisse garantir un matériel sûr et sécurisé malgré ses bonnes intentions.

Une situation similaire s'est produite dans le domaine du logiciel il y a quelques décennies. L'univers logiciel était affecté par de nombreux bugs et problèmes de sécurité, mais la communauté a développé des outils efficaces pour les détecter. **Pourrait-on adapter ces outils et méthodes au matériel ?**

Pour répondre à cette question, notre plan est d'observer de nombreuses erreurs CPU, déduire les techniques les plus prometteuses provenant de la sécurité logicielle pour le matériel, et les adapter. Pour comprendre les bugs CPU actuels, nous introduisons la base de données *RemembERR* basée sur des milliers d'erratas. Nous en déduisons deux techniques particulièrement prometteuses pour le matériel, issues du logiciel: le suivi dynamique de flux d'informations et le fuzzing. Nous présentons *CellIFT*, le premier mécanisme de suivi dynamique de flux d'informations matériel scalable, avec 4 nouvelles applications de sécurité architecturale ou microarchitecturale. Ensuite, nous présentons *Cascade*, un fuzzer de CPU boîte noire qui a permis de découvrir de nombreux bugs et excède la couverture des autres fuzzers. Enfin, nous présentons *MiRTL*, une nouvelle classe d'attaques matérielles basées sur des bugs logiciels EDA permettant de corrompre du hardware apparemment sain, ainsi que *TransFuzz*, un fuzzer qui identifie ces bugs dans des logiciels EDA open source populaires.

Toutes ces contributions démontrent que, lorsqu'elles sont correctement adaptées, les techniques de sécurité logicielle peuvent fournir des solutions efficaces et faciles à adopter qui renforceront la sécurité du matériel.

PUBLICATIONS

This dissertation is based on several papers published in conference proceedings presented hereafter.

RemembERR: Leveraging Microprocessor Errata for Design Testing and Validation

Flavien Solt, Patrick Jattke, Kaveh Razavi.

IEEE/ACM MICRO, Chicago, IL, USA, 2022.

In this work, I prepared and performed the study and classification, and the RemembERR database. I wrote most of the paper.

CellIFT: Leveraging Cells for Scalable and Precise Dynamic Information Flow Tracking in RTL

Flavien Solt, Ben Gras, Kaveh Razavi.

USENIX Security, Boston, MA, USA, 2022.

In this work, I designed and implemented CellIFT, and performed most of the evaluation. I wrote most of the paper.

Cascade: CPU Fuzzing via Intricate Program Generation

Flavien Solt, Katharina Ceesay-Seitz, Kaveh Razavi.

USENIX Security, Philadelphia, PA, USA, 2024.

In this work, I designed and implemented Cascade and performed the whole evaluation. I wrote most of the paper.

Lost in Translation: Confused Deputy Attacks on EDA Software

Flavien Solt, Kaveh Razavi.

Under review.

In this work, I designed and implemented TransFuzz and the MiRTL gadgets, and performed the evaluation. I wrote most of the paper.

The following publications were part of my PhD research, but they are not covered in this dissertation.

ProTRR: Principled yet Optimal In-DRAM Target Row Refresh

Michele Marazzi, Patrick Jattke, Flavien Solt, Kaveh Razavi.

IEEE Security & Privacy, San Francisco, CA, USA, 2022.

REGA: Scalable Rowhammer Mitigation with Refresh-Generating Activations

Michele Marazzi, Flavien Solt, Patrick Jattke, Kubo Takashi, Kaveh Razavi.

IEEE Security & Privacy, San Francisco, CA, USA, 2023.

ZenHammer: Rowhammer Attacks on AMD Zen-based Platforms

Patrick Jattke, Max Wipfli, Flavien Solt, Michele Marazzi, Kaveh Razavi.

USENIX Security, Philadelphia, PA, USA, 2024.

HiFi-DRAM: Enabling High-fidelity DRAM Research by Uncovering Sense Amplifiers with IC Imaging

Michele Marazzi, Tristan Sachsenweger, Flavien Solt, Peng Zeng, Kubo Takashi, Maksym Yarema, Kaveh Razavi.

IEEE/ACM ISCA, Buenos Aires, Argentina, 2024.

HybriDIFT: Scalable Memory-Aware Dynamic Information Flow Tracking for Hardware

Flavien Solt, Kaveh Razavi.

IEEE/ACM ICCAD, Newark, NJ, USA, 2024.

μCFI: Formal Verification of Microarchitectural Control-flow Integrity

Katharina Ceesay-Seitz, Flavien Solt, Kaveh Razavi.

Under Minor Revision for ACM CCS, Salt Lake City, UT, USA, 2024.

ACKNOWLEDGMENTS

This PhD thesis would not have been possible without the support and encouragement of many individuals, to whom I am deeply grateful.

First and foremost, I would like to express my deepest gratitude to my advisor, Prof. Razavi. Your unwavering trust, constant feedback, expertise, curiosity, humor, and encouragement were invaluable throughout this journey. Your presence and guidance provided a foundation that allowed me to navigate the complexities of hardware security research with confidence and clarity. Your dedication has been truly inspiring.

I am also immensely grateful to my PhD colleagues, who were an essential part of this journey. Michele, my Italian officemate and de facto psychoanalyst, offered a listening ear and wise counsel during challenging moments. Patrick, your kindness and patience in enduring my humor have made the office a brighter place. Finn, your engagement with controversial topics sparked many thought-provoking discussions that broadened my perspective. Katharina, your delicious cakes have been a sweet respite during this long journey. Johannes, your steady nightly presence and support have been much appreciated. Raphael, your freedom, optimism and generosity will remain in the group's memories. Silvan and Sandro, you pleasantly challenged my stereotypes and opened my mind. I am also grateful to all other colleagues: Jiahui, Carmine, Emmet, Jiantao, and Nils, and to Prof. Josipović for second-advising this PhD, and to Prof. Payer, Prof. Guarnieri, and Prof. Benini for their useful advice and feedback. Dr. Boris Köpf, Dr. Oleksii Oleksenko, Dr. Jana Hoffmann, Dr. Stavros Volos, and Dr. Cédric Fournet, you made the Microsoft Research internship a memorable experience. Also, a great thanks to the ASVZ team.

A special thanks to Frank K. Gürkaynak, Beat Futterknecht, Susann Arreghini, and Edoardo Talotti for all the administrative, personal, technical, and expert support throughout the years. I would like to thank Matej, Tobias, Quentin, Tristan, Adriel, Stijn, Maximilian, Valentin, and Guillaume who successfully completed projects under my supervision. Working with you was pleasant, rewarding and enriching as well as with all my students from the three years of computer engineering classes.

I would also like to extend my heartfelt thanks to my friends who have supported me in countless ways, providing laughter, spicy food, constant muscle or foot pain, language struggles, and encouragement.

To my family, words cannot express the depth of my gratitude for your patience and support. My parents, Carmen and Raphael, my siblings, Perrine and Virgile, and my grandparents, Antoinette and Wendelin, have been my rock throughout this journey.

Finally, I dedicate this thesis to all those who encouraged and supported me throughout this journey.

Thank you.

CONTENTS

1

INTRODUCTION

An exciting era has started where hardware diversity is key. Different electronic products have vastly different contradictory requirements, that one single product cannot always fulfill. Some devices must provide high performance, while others must be energy efficient. Some devices must always perform the same few types of operations, such as multiply-accumulate, while others must be versatile enough to adapt to new tasks. Some devices comply with some standard interface, e.g., some specific instruction set architecture, while others adopt application-specific interfaces.

New independent phenomena have been further accelerating the drive for hardware diversity. The first is the blazingly fast evolution of the typical workloads. For instance, increasing demand for machine learning inference will require new hardware as the algorithms evolve, may it be low-level or high-level evolutions. The second concerns geopolitics, where we are witnessing phenomena where entities may prevent others from using some technologies, pushing the latter to develop new ones. Finally, the rise in open-source hardware is an extremely powerful accelerator for design diversity as it fosters reuse and modifications of hardware. Furthermore, new hardware construction languages and libraries further ease reuse and connection of hardware components.

This push in diversity, however, puts hardware design validation to the test. Actual validation practices are often kept secret, internally, by hardware vendors, and are already imperfect. In the era of agile diverse hardware made by smaller entities, the challenge is pressing and opens safety and security concerns. Furthermore, community developers do not always have access to commercial software and may not have the incentive to put an enormous effort into validation, not to mention their potential lack of knowledge of what can be potential issues and how to find them.

Yet the rise of open-source hardware is also an inspiring opportunity for improving validation practices as it provides a formidable, unprecedented database of designs and of practical errors. Hence, used carefully, it may allow for the development of a new generation of tools and methodologies that are more open, more transparent, more efficient and more user-friendly.

Another opportunity is provided by the software world. Indeed, software already started becoming massively diverse and open source some decades ago. Like open-source hardware now, open-source software was for long plagued with myriads of bugs and security issues. Regarding these issues, researchers and the software community in general have developed a broad set of tools and methodologies to find and fix bugs and security issues. Hence, it is tempting to believe that these tools and methodologies could be adapted to hardware. For example, we will argue that dynamic information flow tracking and fuzzing, originally developed for software security, have a great adaptation potential to hardware. However, the initial attempts to bring these software-inspired techniques to hardware have been inconclusive, specifically regarding scalability and effectiveness. We hypothesize that these few existing adaptations do not take full advantage of the inherent properties of the target hardware.

Leading Research Question: Can we adapt software security techniques to build effective counterparts in the realm of digital hardware?

To answer this question, we start by exploring the characteristics of contemporary hardware bugs. We then identify software security techniques that are particularly relevant for hardware security and adapt them. Because EDA software conditions the result of the hardware design process, we also examine its potential security implications. Hence, we structure this dissertation around the following four intermediate research questions.

To produce well-suited hardware security solutions, we propose to first understand what issues affect contemporary hardware.

Research Question 1. What are characteristics of bugs that occur in contemporary CPUs?

In Chapter 2, we answer this question by introducing the *RemembERR* database. While established hardware vendors tend to develop their validation knowledge in-house, they do share some important aspects through errata. We exploit this tremendous but disorganized source of information to bridge the knowledge gap between vendors and the open-source hardware community. Concretely, we provide a comprehensive classification of thousands of errata of commercial CPUs, based on a novel methodology based on triggers, contexts and observable effects. Not only does this knowledge mining adventure provide immediate new insights concerning commercial

CPU bugs but it also provides, for the first time, machine-readable errata descriptions for these thousands of CPU errata.

Knowing typical CPU bugs and taking a closer look at specific techniques that seem particularly promising for hardware, we first identify dynamic information flow tracking as a candidate, theoretically, for detecting confidentiality and integrity breaches.

Research Question 2. How to adapt dynamic information flow tracking to complex digital hardware and what are ensuing applications?

In Chapter 3, we present *CellIFT*, the first hardware dynamic information flow mechanism that scales to complex CPUs and SoCs. One main aspect of dynamic information flow tracking for hardware is that the instrumentation itself must be expressible as hardware. The state of the art naively proposes a simple gate-level mechanism [1], not scalable because it requires breaking all macrocells down into logic gates. *CellIFT* proposes to implement the mechanism at the level of macro-cells, which are the building blocks of hardware description languages, and leverages three mathematical macrocell properties (monotonicity, transportability and translatability) in order to make the mechanism scalable, precise and complete. Given this new mechanism, we introduce four new applications for dynamic information flow tracking in hardware. First, finding design subcomponents that are prone to leak information. Second, detecting known bugs in a SoC that were reported to be undetectable using standard formal verification properties. Third, detecting Meltdown-type [2] microarchitectural leakages, where permission checks are mistakenly bypassed during speculative execution. Finally, detecting Spectre-type [3] microarchitectural vulnerabilities where confidentiality is breached through speculative execution.

Besides dynamic information flow tracking, fuzzing is another candidate for adaptation, but with very different challenges.

Research Question 3. How to design an effective CPU fuzzer?

In Chapter 4, we present *Cascade*, a black-box RISC-V CPU fuzzer that, applied across several significant open-source CPUs, discovered 29 new Common Vulnerabilities and Exposures (CVEs), which is more than all other existing hardware fuzzers [4–17] combined. *Cascade* operates by generating long and complex yet valid programs, in which the data flow and the control flow are tightly intertwined to exert maximal pressure on the CPU

under test. It can then detect bugs by relying on program non-termination as a proxy, and automatically reduces buggy programs to a small human-readable reproducer. Because *Cascade* does not rely on any instrumentation or coverage feedback, it is fast and easily portable. Interestingly, it achieves far more coverage than the existing coverage-guided CPU fuzzers [4, 5], measured on their own coverage metric. This unambiguously proves that all past attempts to construct coverage-guided CPU fuzzers have not been very successful. *Cascade* is not only an open-source and easy-to-use effective fuzzer, but also a solid base on which to build future hardware fuzzing research upon.

Finally, not only the hardware descriptions themselves, but also the open-source EDA tools that process them may be imperfect.

Research Question 4. How to effectively find bugs in RTL simulators and synthesizers, and how could they be exploited?

In Chapter 5, we introduce *TransFuzz*, a new fuzzer for EDA software that specifically targets translation bugs in RTL simulators and synthesizers, i.e., bugs that cause an RTL description to be mistranslated into a wrong design. Based on observations made on existing translation bugs, *TransFuzz* generates RTL designs that feature complex interconnections of diverse operators, and discovered 25 new CVEs. It employs various flavors of differential fuzzing to detect translation bugs and reduces them to small reproducer designs. Given the 20 newly detected translation bugs, we demonstrate a new class of attacks, *MiRTL* for Mistranslated RTL, where a seemingly correct RTL design translates into synthesized malicious hardware. We instantiate a *MiRTL* attack in the CVA6 RISC-V CPU by injecting malicious hardware transformations allowing kernel-to-user information leakage using seemingly benign RTL code, and demonstrate that such attacks can bypass dynamic information flow tracking based mitigations.

Answering these four intermediate research questions provides us with an introductory overview of capabilities of software techniques in hardware security. After gaining awareness of contemporary bugs, we successfully adapted two software security techniques to hardware that vastly outperformed existing hardware security techniques. We further discovered a new class of attacks and many EDA software bugs on which they may rely, again through fuzzing. Hence, we can conclude that yes, adaptations of software security techniques are likely to occupy the hardware security landscape in the near future. We have shown that these adaptations may be

challenging, yet yield significant benefits. This opens exciting new research opportunities and is likely to only be a first step toward a new generation of hardware security tools and methodologies that are more open, efficient and user-friendly, contributing to a safer and more secure digital hardware environment.

2

REMEMBERR: LEVERAGING MICROPROCESSOR ERRATA FOR DESIGN TESTING AND VALIDATION

Microprocessors are constantly increasing in complexity, but to remain competitive, their design and testing cycles must be kept as short as possible. This trend inevitably leads to design errors that eventually make their way into commercial products. Major microprocessor vendors such as Intel and AMD regularly publish and update errata documents describing these errata after their microprocessors are launched. The abundance of errata suggests the presence of significant gaps in the design testing of modern microprocessors.

We argue that while a specific erratum provides information about only a single issue, the aggregated information from the body of existing errata can shed light on existing design testing gaps. Unfortunately, errata documents are not systematically structured. We formalize that each erratum describes, in human language, a set of *triggers* that, when applied in specific *contexts*, cause certain *observations* that pertain to a particular bug. We present RemembERR, the first large-scale database of microprocessor errata collected among all Intel Core and AMD microprocessors since 2008, comprising 2,563 individual errata. Each RemembERR entry is annotated with triggers, contexts, and observations, extracted from the original erratum. To generalize these properties, we classify them on multiple levels of abstraction that describe the underlying causes and effects.

We then leverage RemembERR to study gaps in design testing by making the key observation that triggers are *conjunctive*, while observations are *disjunctive*: to detect a bug, it is necessary to apply *all* triggers and sufficient to observe only a *single* deviation. Based on this insight, one can rely on partial information about triggers across the entire corpus to draw consistent conclusions about the best design testing and validation strategies to cover the existing gaps. As a concrete example, our study shows that we need testing tools that exert power level transitions under MSR-determined configurations while operating custom features.

2.1 INTRODUCTION

What are the bugs that we could not discover before we sent the microprocessor design for fabrication? This is probably the most important question that design test engineers repeatedly ask themselves. The question is not getting any easier to answer with the ever-increasing complexity of modern microprocessors [18]. Despite advances in design testing tools and techniques [4, 12, 19–38], we still see plenty of post-production bugs after new microprocessors are released, indicating that there exist gaps in design testing and validation. In this Chapter, we identify these gaps and propose concrete actions to cover them by leveraging a new classification based on errata reported by Intel and AMD.

DESIGN TESTING AND VALIDATION. Before a microprocessor is shipped to customers, it goes through a variety of testing and validation steps. In the early stages, a design simulation using random or human-driven inputs may reveal bugs [19, 31]. Once the design matures, formal verification techniques ensure the correctness of selected design parts [32–34, 39, 40]. Finally, many bugs can only be found in post-silicon testing under real-world conditions [41–46]. These design testing and validation methods, unfortunately, do not scale to the complexity of today's microprocessor designs [31]. In the testing steps, the lingering question is whether the test cases are providing a sufficient coverage [47–49]. Similarly, expensive verification efforts should be targeted to those parts of the design where critical bugs are likely to lurk.

MICROPROCESSOR ERRATA. In response to the discovery of bugs after production, microprocessors vendors regularly publish *errata documents*: human-readable documents containing a list of errata [30, 50–53]. The goal of publishing this list of defects is to document known bugs and to provide system designers with workaround guidance where appropriate. The organization of errata differs across vendors, but the structure of each erratum entry remains similar. Each erratum, from both Intel and AMD, includes a *description* with information about the conditions under which the bug occurs and a brief discussion of its *implications* once triggered. Furthermore, each entry includes information about the proposed workarounds and whether or not the bug has been fixed. While the individual erratum is useful for keeping track of a bug and informing users about it, we argue that grouping them reveals precious information that can guide future design testing and validation.

REMEMBERR. To extract the relevant information from the errata, we created RemembERR, a comprehensive, annotated database of all errata of Intel Core and AMD microprocessor families since 2008, with a total of 2,563 entries. Creating this database itself presented challenges since the errata are not machine-readable:

(a) they lack an identical structure between documents,

(b) they contain a significant number of errors such as duplicate entries in the same document, reused errata numbers, and erroneous Model Specific Register (MSR) numbers, and

(c) a lack of classification and consistency in notations.

To facilitate guiding testing and validation, we create a new classification of errata for RemembERR. We manually annotate each RemembERR entry with its necessary *triggers*, the *contexts* to which the bug applies, and the *observations* that can be made once the bug is triggered. We call this level the *concrete* level of our classification. Although useful, the *concrete* level can sometimes be too erratum-specific to generalize. For example, a particular offset inside a certain machine-specific register must be written to trigger a bug. To study causes and effects in an aggregate manner, we further classify and annotate RemembERR entries at two higher levels of abstraction, which we call the *abstract* and *class* levels.

Equipped with RemembERR, we then study trends to identify design testing gaps. We make a key observation that in almost every erratum, trigger conditions are *conjunctive*, while contexts and observations are *disjunctive*. This means that to discover a bug, *all* triggers must be activated (e.g., a misaligned load that causes a page fault), in *any* of the applicable contexts (e.g., in user mode), and observing *any* behavior deviating from the expected behavior (e.g., a machine check exception) is sufficient to detect the bug. This powerful insight allows us to extract valuable information from aggregated errata, regardless of how vague each individual erratum may be on its triggers and/or observations (the contexts are usually clear). Importantly, this information about triggers, contexts, and observations is necessary for directing design testing and validation campaigns to discover bugs that are not currently missed by the existing tools and techniques.

Our study shows that more than 40% of bugs are uncovered only when two distinct trigger types are combined. Moreover, most triggers do not interact with each other, while others seem to be closely related and together, they bring up new bugs. Exploiting these interactions is crucial to boost

future design testing and validation of microprocessors and to keep up with their increasing complexity.

CONTRIBUTIONS. We make the following contributions:

- We propose a new classification of design flaws based on necessary triggers, and sufficient contexts and observations.
- We create RemembERR, a comprehensive database created from 2,563 public errata across all 12 first generations of Intel Core and 13 current AMD microprocessor families.
- Using RemembERR, we study trends in post-production microprocessor bugs and develop testing and validation guidelines that relate triggers, contexts, and observations.

OPEN SOURCING. We make the entire RemembERR database, including our annotations, publicly available[1] so that researchers and design test engineers can draw conclusions specific to their goals and automate their tools.

2.2 BACKGROUND

This section provides some brief background on existing hardware bug detection techniques (Section 2.2.1) and errata documents (Section 2.2.2).

2.2.1 *Bug detection methods*

We provide background on the three commonly used techniques for detecting hardware bugs [31]: simulation, formal methods, and silicon testing.

SIMULATION. Design simulation is a traditional testing technique that is already used early in the design's development cycle [31]. During a simulation, the design is given sequences of inputs. The outputs and the resulting state of the design are then compared against a golden model [4] or inspected manually [31, 54]. Modern simulators provide a rich set of features, such as undefined values (i.e., *don't care* values), signal injection, and various coverage metrics [55–57].

Testing with simulation has two major shortcomings. First, simulation-based testing is extremely slow. Therefore, it can process only a few inputs

1 https://github.com/comsec-group/rememberr

in a reasonable time, making it challenging to reach all possible system states. For example, the open-source CVA-6 64-bit RISC-V core requires four days to boot Linux in simulation [58]. We expect more complex CPUs, tailored towards high performance, to be even more complex by several orders of magnitude. Simulation becomes mostly ineffective for complex modern microprocessors without limiting the test cases to those effective in triggering bugs. Second, simulation cannot expose issues related to the physical design, such as timing violations, data loss after power gating [59], or interaction with real-world peripherals and memories. Emulation, in spite of being significantly faster, suffers from the latter shortcoming as well.

FORMAL METHODS. Unlike simulation, which may suffer from limited input coverage, formal verification methods aim to prove that certain properties always hold given some allowed inputs. This approach makes it possible to prove correctness for all expected inputs — achieving completeness. However, these powerful formal methods have three weaknesses. First, they typically do not scale to complex designs with many stateful elements [31, 40, 60, 61]. Therefore, a typical approach is to verify only selected design parts while modeling the rest [62–64]. Second, properties may be difficult to express formally, and there can exist many properties for complex designs [65, 66]. Third, properties related to power management or other physical effects may be difficult to reason about [59, 67–70]. As each property is proven exhaustively, quickly rendering verification time infeasible, the test and validation engineers must carefully decide which properties to prove.

SILICON TESTING. Complex bugs often escape traditional pre-silicon testing and validation [41–46, 71]. Therefore, silicon testing remains a crucial part of design validation, and takes up to 50% of the testing cost for commercial designs [72]. In contrast to simulation, silicon testing achieves far higher throughput, but it does not reach the completeness offered by formal methods. Furthermore, silicon testing makes the design's internals inaccessible.

2.2.2 *Errata*

For each design generation (Intel) or family (AMD) of microprocessors, vendors typically provide a *specification update* document, also known as *errata*, for listing known bugs after a product has been shipped. When a customer

ID: ADL001

Title: X87 FDP Value May be Saved Incorrectly

Description: Execution of the FSAVE, FNSAVE, FSTENV, or FNSTENV instructions in real-address mode or virtual-8086 mode may save an incorrect value for the x87 FDP (FPU data pointer). This erratum does not apply if the last non-control x87 instruction had an unmasked exception.

Implications: Software operating in real-address mode or virtual-8086 mode that depends on the FDP value for non-control x87 instructions without unmasked exceptions may not operate properly. Intel has not observed this erratum in any commercially available software.

Workaround: None identified. Software should use the FDP value saved by the listed instructions only when the most recent non-control x87 instruction incurred an unmasked exception.

Status: For the steppings affected, refer to the Summary Table of Changes.

TABLE 2.1: An erratum for Intel Core 12th generation.

observes that a microprocessor deviates from its original specification, they can look through the errata documents to verify whether it is a known bug. The errata also provide information on how to avoid triggering unwanted behavior. Notably, the bugs described in errata documents can no longer be fixed and remain for the lifetime of the affected microprocessors.

ORGANIZATION. Following their intended purpose, errata documents produced by Intel and AMD are human-readable PDF documents listing the individual bugs. Each erratum has a *title*, a *description*, *implications*, *workarounds*, and a *status* indicating whether a fix is available for current or future releases of the same CPU generation or family. Intel released separate erratum documents for the Mobile and Desktop version of its Core microprocessors until generation 5. After that, they released only one document per generation. AMD uses a single document per CPU family (i.e., per CPU microarchitecture).

ERRATA EXAMPLES. We provide two recent errata examples. In Table 2.1, we show the first erratum for Intel Core 12th generation CPUs, and in Table 2.2, the most recent erratum for AMD Zen 3 family CPUs.

ID: 1361

Title: Processor May Hang When Switching Between Instruction Cache and Op Cache.

Description: Under a highly specific and detailed set of internal timing conditions, running a program with a code footprint exceeding 32 KB may cause the processor to hang while switching between code regions that consistently miss the instruction cache and code regions contained within the Op Cache.

Implications: System may hang or reset.

Workaround: System software may contain the workaround for this erratum.

Status: No fix planned.

TABLE 2.2: An erratum for AMD Zen 3 family.

2.3 MOTIVATION: LEARN FROM THE PAST

The number of published errata has not significantly decreased over time, as we show in Section 2.4. Strikingly, we will show that some bugs require years to be reported, while similar bugs were already found in previous designs. These trends point to gaps in existing design testing and validation tools and techniques. A data-driven approach using the information contained in the errata can shed light on these gaps and provide directions for covering them.

ACCESSIBILITY Each erratum is specific to one bug in a particular design, complying with a certain Instruction Set Architecture (ISA). This makes deriving any valuable insights from a series of individual errata difficult. Further, it does not incite communities that build and verify other microprocessors to read and learn from known pitfalls and spots that require special testing focus. This is becoming increasingly more important as the complexity of community-driven microprocessors is progressively catching up with their proprietary and closed-source counterparts [73–75]. By aggregating errata and building an annotated database, we intend to make this information more accessible than it currently is.

STRUCTURE The way errata are structured is suitable for reading by an experienced human but is not optimized for automated data mining. A clear

specification of what each field contains or implies is missing. The useful information is often spread across the title, description, and implication (and sometimes workaround) fields, with a high degree of redundancy. This observation calls for creating and maintaining an improved erratum structure, scheme and tooling support that would be more adapted for data mining and to rule out redundancy while remaining human-readable.

GUIDING DESIGN TESTING AND VALIDATION In complex CPU designs, all testing and validation methods must be directed. Formal methods require knowing the bug type to target and prioritizing the parts of the design that are most susceptible. Furthermore, formal properties must be local and specific to minimize the impact of state explosions. For dynamic methods such as simulation and silicon testing, it is crucial to know which input signals to provide in which context and what effects to expect if a bug is triggered [54, 76]. In Section 2.6, we provide an in-depth discussion on how the annotated errata information can enhance existing validation methodologies.

For example, errata reveal that specific bugs require ongoing PCIe communication. Is connecting a PCIe device enough to discover all PCIe-related bugs? Looking at all the errata, we observe that some PCIe-related bugs require triggering a reset signal. Furthermore, how can we efficiently observe whether a bug was triggered? This knowledge of the interaction between different input types, contexts, and effects is crucial for maximizing a testing campaign's efficiency and efficacy.

2.4 REMEMBERR

In this section, we introduce RemembERR, an annotated database of 2,563 errata from AMD and Intel microprocessors. In Section 2.4.1, we first describe the scope and our methodology. Based on this (yet unannotated) database, we present essential observations about the current state of microprocessor errata in Section 4.3.1.

2.4.1 *Methodology*

Figure 2.1 presents an overview of our methodology. Our approach can be summarized into four steps: 1a First, we acquired the latest errata documents from Intel and AMD, and 1b analyzed duplicate errata. This

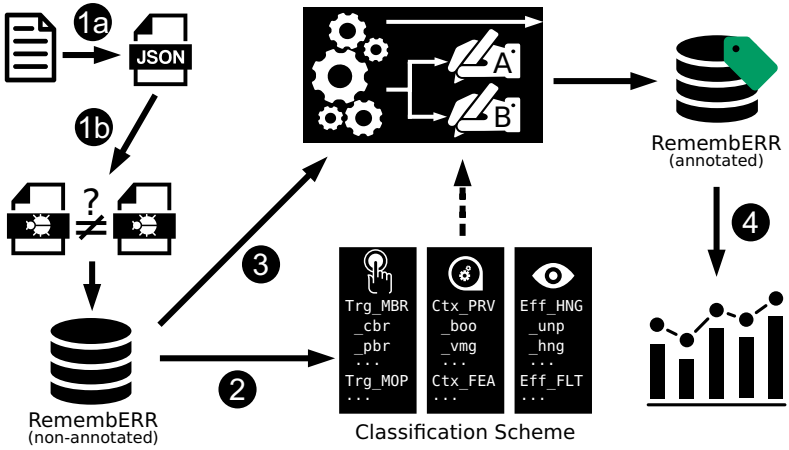

FIGURE 2.1: Overview of our methodology.

already allows us to make general observations about errata's current state (Section 4.3.1). ❷ We then generalized the triggers, contexts, and observable effects to derive a universal classification scheme for errata (Section 2.5.1). ❸ Using automation, we classified a portion of the errata, and for the rest, we used four-eyes manual classification. The result is the annotated RememberR database. ❹ Lastly, we leveraged RememberR to derive novel insights for filling the gaps in existing design testing and validation (Section 2.5.2).

EXAMINED DOCUMENTS. We comprehensively examined all the errata documents listed in Table 2.3. Vendors usually withdraw errata documents once the processor line is not supported anymore, which makes finding the errata documents not always straightforward. We scraped the web thoroughly and took the most recent findable document for each generation (Intel) or family (AMD). We examined all the errata from the Intel Core series and all the errata from AMD CPUs since 2008.

ERRATA IN ERRATA. Errata documents contain many errors themselves. Examples are two revisions pretending to have added the same erratum (affects 8 errata across 3 documents), some errata are never mentioned in the revision notes (affects 12 errata across 2 documents), the same name refers to two different errata (affects an erratum named AAJ143), there are missing or duplicate fields in errata (affects 7 errata across 4 documents), or there are errors in the MSR numbers (affects 3 errata across 3 documents). In

Intel		AMD		
Gen.	Reference	Fam.	Models	Reference
1 (D)	320836-037US	10h	00-0F	41322-3.84
1 (M)	322814-024US	11h	00-0F	41788-3.00
2 (D)	324643-037US	12h	00-0F	44739-3.10
2 (M)	324827-034US	14h	00-0F	47534-3.18
3 (D)	326766-022US	15h	00-0F	48063-3.24
3 (M)	326770-022US	15h	10-1F	48931-3.08
4 (D)	328899-039US	15h	30-3F	51603-1.06
4 (M)	328903-038US	15h	70-7F	55370-3.00
5 (D)	332381-023US	16h	00-0F	51810-3.06
5 (M)	330836-031US	17h	00-0F	55449-1.12
6	332689-028US	17h	30-3F	56323-0.78
7/8	334663-013US	19h	00-0F	56683-1.04
8/9	337346-002US			
10	615213-010US			
11	634808-008US			
12	682436-004US			

TABLE 2.3: Inspected errata documents. Left: Intel Core CPUs, right: AMD CPUs. (M): Mobile, (D): Desktop.

rare cases, errata may be repeated inside the same errata document (affects 11 errata pairs across 6 documents). These errors are a clear indicator that the writing of errata is a manual process. Humans not only express errata in a human language, but they also seem to be responsible for non-systematically (redundantly) distributing information across errata fields.

DUPLICATES. As we will show, it is common that two (or multiple) designs from the same vendor with different release dates are affected by the same erratum. RemembERR contains all the duplicates as often as they appear across documents. This provides useful information about bugs shared among generations or families. However, to allow filtering for unique entries, RemembERR features a keying mechanism that assigns a unique identifier to each cluster of identical errata.

AMD identifies errata across microprocessor families using a unique numeric identifier: two families are affected by the same erratum if both

have an erratum with the same number in their corresponding errata document. This mechanism protects against intra-document duplicates. Besides different errata numbers, some cases are indistinguishable given the limited information in the errata's fields. For example, errata no. 1327 and no. 1329 only differ in their suggested workaround but may originate from distinct root causes. *In total, we collected 506 errata from AMD, of which 385 are unique.*

Intel errata documents do not provide a simple way to identify duplicates across generations. Instead, we base our duplicate detection on the errata titles. As a first step, we marked all errata with the same title as duplicates. Extensive manual inspection of all the candidate duplicates shows that when the titles are (nearly) identical, all other fields are identical as well. Except for minor phrasing variations or slightly different levels of detail. As a second step, we manually analyzed remaining errata that have not been marked yet as duplicates, sorted by decreasing title similarity, given that title similarity is a strong indicator of potential duplicates. We could manually identify 29 pairs as duplicates. *In total, we collected 2,057 errata for Intel, of which 743 are unique.*

Because Intel and AMD use different identifiers for their errata, it is difficult to determine whether a bug is common between products of these two vendors; at least, we could not find any occurrence giving strong evidence. Arguably, Intel and AMD designs are proprietary; hence they might not share hardware blocks. It is, hence, unlikely for identical bug instances to occur across vendors.

2.4.2 *Observations*

The data gathered in RemembERR allows us to make several novel observations about the current state of errata.

Timeline

We first analyze the number of reported bugs accumulated over time. Unfortunately, bug discoveries are not timestamped; hence, we approximate the timestamp of each erratum by identifying in which revision of the errata document it first appeared. We then use the errata document's release or update date to approximate the timestamp.

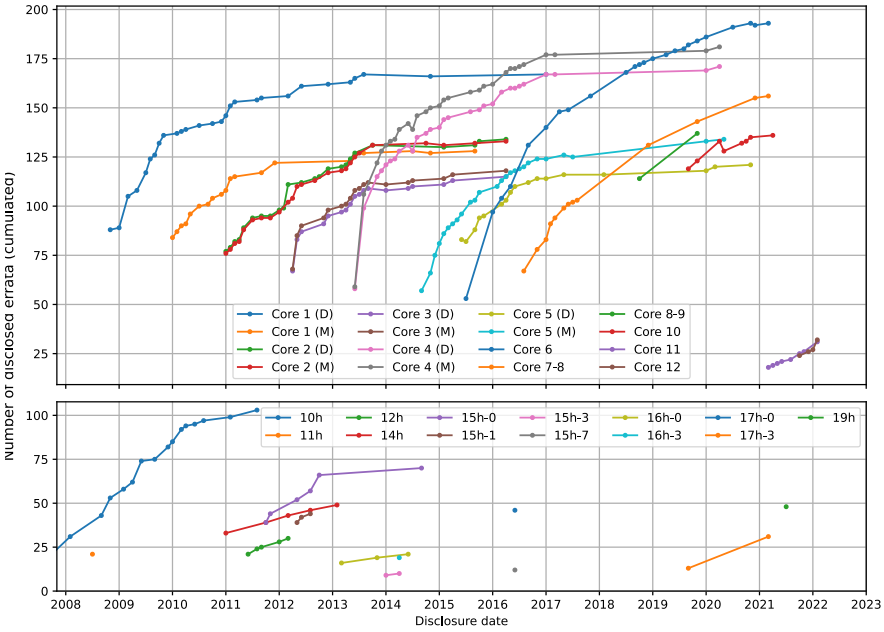

FIGURE 2.2: Disclosure dates of Intel Core errata (top) and AMD CPU errata (bottom). The y-axis represents the cumulative number of disclosed errata. The data point represents the errata's release date.

In some cases, the revision summary does not indicate in which revision a certain erratum was added. Fortunately, errata are sequentially numbered. Hence, we can approximate the date by assuming that the subsequent erratum was added simultaneously. In rare cases, we observed contradicting dates: revision logs falsely pretend that the same erratum was added in two consecutive revisions. In this case, we consider the date of the earlier of the two revisions as the correct one.

Figure 2.2 shows the cumulative growth of errata over time, where duplicate entries are counted individually. We observe that Intel updates its errata documents significantly more frequently than AMD. Desktop and mobile processors released at close dates have very similar curves, for example, Intel Core 2, 3, and 4 during the year 2013. This may suggest that the same bugs tend to affect multiple generations. In Section 2.4.2, we study this bug *transmission effect* across design generations in more detail.

Figure 2.2 further demonstrates that vendors keep introducing new bugs into their products. While the latest microarchitectures seem to be less affected, it is likely that many bugs have not yet been discovered or reported.

(O1) Observation. The number of reported errata does not significantly decrease over time with new designs.

All cumulative curves tend to be concave. The more time passes, the fewer bugs are found in a given period. In most older designs, the curve stagnates towards the end, where only very few new bugs are discovered after many years from the initial release of the CPU, especially for Intel Core designs. This observation confirms the intuition that finding new bugs in a design becomes increasingly more difficult or that older designs are not as rigorously tested anymore compared to newer designs.

(O2) Observation. The increase in errata for a given design is usually concave.

Heredity

It is known from well-studied bugs such as Meltdown [2], Foreshadow [77], RIDL [77] and ZombieLoad [78], that different designs may suffer from exactly the same bug. One cause for this phenomenon may be the reuse of microarchitectural blocks across design generations. We study this phenomenon to provide an answer to the questions:

(a) How often are bugs transmitted across generations or families?

(b) Are transmitted bugs rediscovered multiple times?

TRANSMISSION. By definition, distinct AMD families have distinct microarchitectures. Our data corroborates that, as we find fewer shared errata between AMD families, compared to Intel Core generations. Furthermore, AMD provides limited chronological information, as depicted in Figure 2.2. Hence, we focus this part of our study on Intel errata.

Figure 2.3 shows the number of identical errata between pairs of Intel errata documents. We can observe that Desktop and mobile processors share the vast majority of bugs, matching our observation of similar curves in Section 2.4.2. The processors that share a substantial part of their microarchitecture are salient in this diagram, such as Intel Core generations 6 to 10. Note that if a security bug is discovered only after multiple generations, an

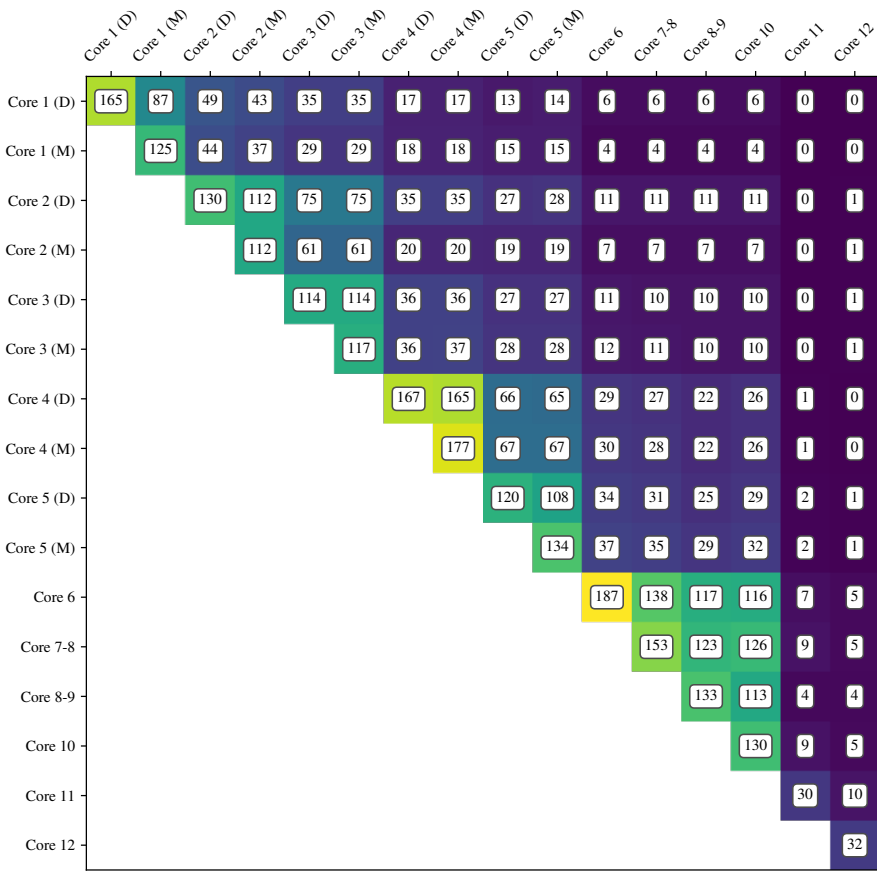

FIGURE 2.3: Bug heredity: number of common bugs across Intel microprocessor generations.

attacker could have exploited it for years without being uncovered. Therefore, the duration between bug introduction and discovery is not a suitable proxy for estimating criticality, especially regarding security. In Figure 2.3, long non-zero horizontal lines indicate long-lasting bugs. *6 bugs stayed from Core 1 to Core 10, and one erratum from Core 2 was still identified 11 generations later, more than 10 years after its initial discovery.*

(O3) Observation. Bugs are often shared between generations of microprocessors. Shared bugs may stay for up to 11 generations.

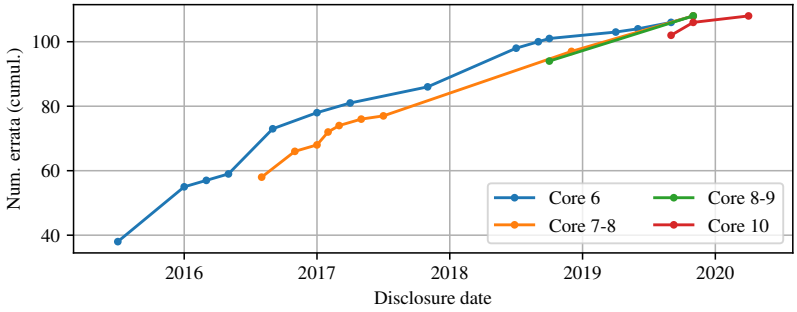

FIGURE 2.4: Disclosure dates of Intel Core errata for bugs that are shared by all Intel Core generations from 6 to 10.

REDISCOVERY. We conducted a further study to answer the question: from errata shared between microprocessors, which proportion was already reported in an earlier generation at the time of release?

Figure 2.4 shows the reporting date for the 104 bugs shared by all Intel Core generations from 6 to 10. This set of bugs corresponds to a salient region of common bugs in Figure 2.3. The first data point corresponds to the release date of each generation. Clearly, most of the shared design errors were known *before* the release of the subsequent generation, some even many years before.

This raises the question of where bugs are first discovered: in older designs and then confirmed on more recent ones (*forward*), or are they usually first found on more recent designs and then confirmed on older ones (*backward*)? While Figure 2.4 provides a qualitative insight, to answer this question, we define a *forward-latent* erratum as an erratum that was reported in one design and (strictly) later reported in a later design. Similarly, we say that an erratum is *backward-latent* if it was reported in a design (strictly) before being reported in an earlier design.

Figure 2.5 shows the forward-latent and backward-latent errata for Intel Core CPUs (again, the AMD errata documents lack sufficient chronological information for such an analysis). The salient portion of backward-latent errata around the year 2015 *may* represent a period at Intel where less resource was allocated to testing older CPU generations, for example, to prepare for the release of the Skylake microarchitecture. The increasing forward-latent numbers typically denote cores sequences with similar microarchitectures, where a bug has not been fixed, although it was known

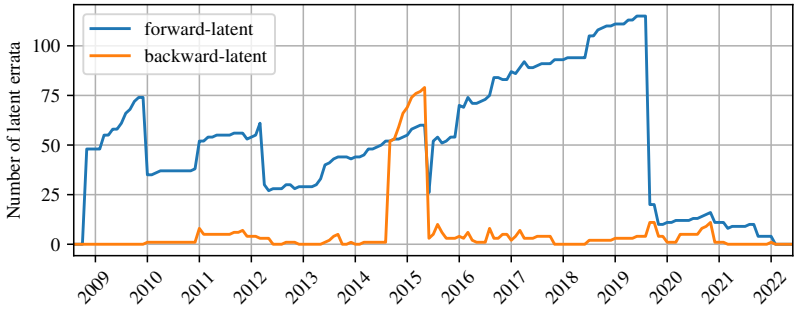

FIGURE 2.5: Forward-latent and backward-latent errata among Intel Core generations.

before the official CPU release. The number of forward-latent errata has always tended to increase, and this trend has accelerated since 2015. Note that these curves, for the time interval displayed here, may increase in the future with the rediscovery of more errata in existing or future Intel Core generations.

These results suggest that either the test and validation cycles are very long (in order of many years), making it difficult to react to newly discovered bugs during this phase, or these bugs are difficult to mitigate without fundamentally changing the microarchitecture.

(O4) Observation. Most of the design flaws that are shared between generations were already known before releasing the subsequent generation.

Observation *O4* indicates a correlation between long CPU development cycles and the difficulty of finding complex bugs.

Workarounds

The vendors propose different workaround types, depending on *where* the workaround should be applied, i.e., which actor should (not) perform a specific action to ensure proper functionality. Based on this, we classify the workarounds into five categories: BIOS, software, peripherals, absent, and None. The category *absent* indicates existing workarounds without any specific information, such as "Contact [...] for information on a BIOS update." Instead of absent, whenever possible, we classify the workaround

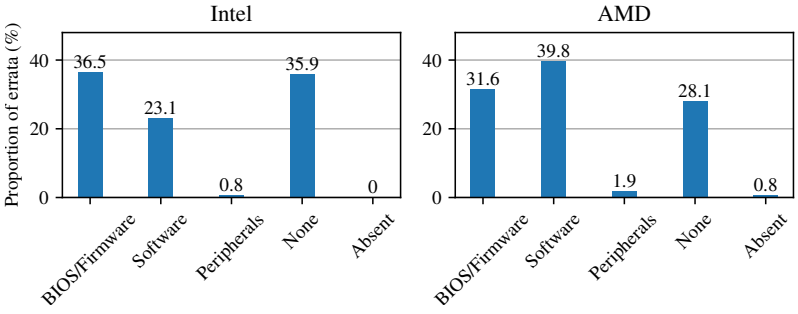

FIGURE 2.6: Suggested workarounds of errata by category.

into a specific category even if the exact information is missing. Vendors use an additional category *documentation fixes* to describe originally intended behavior that was wrongly documented. This category is negligible in size as it represents less than 0.5% of the total number of errata.

We summarize our results in Figure 2.6, where identical errata are merged. The errata that can be mitigated in the BIOS are arguably the least critical, as long as the mitigation does not substantially affect performance or security. Errata requiring conditions in the peripherals or the software are more challenging to mitigate due to the plethora of legacy hardware and software. In total, 28.9% (AMD) and 35.9% (Intel) of all unique errata do not have any suggested workaround at all.

(O5) Observation. A substantial number of errata do not have any suggested workaround.

As we discuss soon, this is not because bugs were fixed but because most bugs are deeply rooted in the design, limiting possible workarounds.

Fixes

In some cases, the vendors fix the root cause of a bug, as indicated explicitly in dedicated parts of the errata documents (in dedicated tables or in a status field). Fixes are distinct from workarounds as the former rules out the bug from the design completely, while the latter dynamically aims at preventing the bug from interfering with proper design functionality. Fixes may require a re-spin of the processor, which is an update of the CPU's

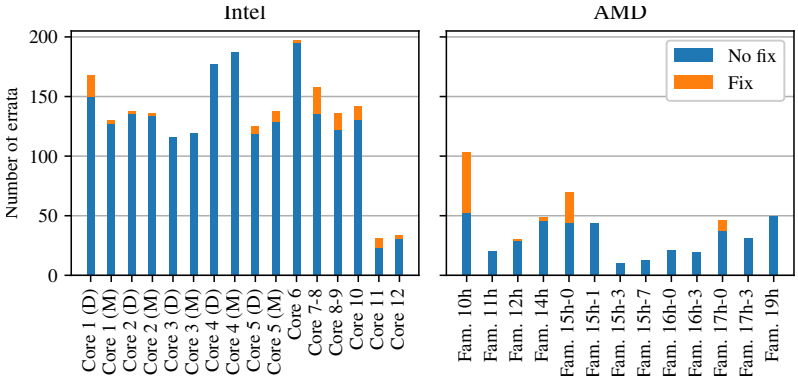

FIGURE 2.7: Proportion of fixed vs. unfixed bugs.

design masks. We could not find any requirements driving the decision to fix a bug rather than proposing a workaround. Most likely, the decision is made based on the bug's criticality by considering functionality or security impact, and also the complexity of fixing the bug.

Figure 2.7 shows the number of bugs that are fixed in different designs. Clearly, the vast majority of bugs are never fixed. For Intel CPUs, there has been a weak trend over the last few generations toward fixing bugs.

(O6) Observation. Bugs are rarely fixed.

2.5 CLASSIFICATION

This section introduces an errata classification scheme based on triggers, contexts, and effects. We start by describing the methodology we applied to design our classification scheme in Section 2.5.1, after which we explain the classification scheme's categories. Using the classified data, we present new insights about bugs based on our classification results in Section 2.5.2.

2.5.1 *Categories*

A crucial part of our classification is the definition of concrete categories. We first made an exploratory pass over the errata documents to determine appropriate categories for triggers, contexts, and effects. To make the classi-

fication useful for our intended purpose, we require our concrete categories to be

(a) *unambiguous*: a category should clearly be distinctive from other categories to improve our classification's reliability,
(b) *usable*: categories should be helpful to guide the design testing process,
(c) and *self-explanatory*: a one-sentence description should be sufficient to understand the category.

For instance, requiring a reset signal to observe faulty behavior is an unambiguous trigger (i.e., unlikely to be misclassified or misunderstood). It is a usable trigger because it is necessary to trigger observable behavior, and it is self-explanatory. If reset signals are not needed to find some bugs of interest, we should apply more relevant, directed test cases to increase effectiveness and close design testing gaps.

Classification methodology

In the following, we describe our systematic approach for designing our errata classification scheme. After that, we explain how we efficiently classified the errata consistently and reliably.

GOAL. We designed a hierarchical classification scheme that allows us to seamlessly switch between different levels of abstraction. These abstraction levels are crucial for making the necessary observations and recommendations for improving design testing and validation. If the recommendations are too precise, methods cannot easily generalize the insights when looking for new bugs. If they are too abstract, however, then limited guidance will hamper efficiency and coverage.

Our classification scheme is composed of three levels: the *concrete* level, the *abstract* level, and the *class* level. We explain them for the example of triggers. First, the *concrete* level represents the exact action that is described in the erratum. For example, "the core resumes from the C6 power state" is an action described at the *concrete* level. Second, the *abstract* level represents a slightly higher level of abstraction. As an example, a transition between core power states is an action described at the *abstract* layer. The abstract level is crucial since design testing and validation tools must achieve generality to maximize coverage. In the example before, considering only transitions from the core C6 power state may not catch unknown bugs that only manifest when transitioning from other power states. Finally, the

class level represents the highest level of abstraction; in our example, power management is the representation of the action at the *class* level. This last level of abstraction provides even more generality, contributes to better readability and allows us to make more general conclusions about the bugs triggered by a particular trigger class.

METHODOLOGY. We define the categories for triggers, contexts, and observable effects in an iterative way. We process all unique errata to extract *concrete* triggers, contexts, and observable effects. For each of them, we check if we already have a corresponding *abstract* category. If so, we then label the erratum with this *abstract* category; otherwise, we create a new *abstract* category, and we check if we have a corresponding *class* category. If a corresponding *class* category exists, we add the new *abstract* category to the existing *class* category; otherwise, we create a new *class* category and attach the new *abstract* category to it. We provide a detailed overview of *class* and *abstract* categories in Table 2.4, Table 2.5, Table 2.6, Table 2.7, Table 2.8, Table 2.9, Table 2.10, Table 2.11, Table 2.12 and Table 2.13.

RemembERR is a cross-ISA database as typically, only items (i.e., triggers, contexts, or effects) at the *concrete* level may be ISA-specific. Therefore, RemembERR can naturally be extended with errata from designs implementing other ISAs (e.g., POWER, ARM).

FOUR-EYES CLASSIFICATION. Some errata contain expressions that are specific enough to be classified automatically using regular expressions into some categories, but many errata-category pairs require manual analysis for classification. Besides being time-consuming, manually extracting and annotating such an immense database of complex items is error-prone. To significantly improve the reliability of our results, two of the researchers involved in this Chapter independently classified the errata. After completing the classification, they discussed and resolved each mismatch individually. To improve the classification and clarify our understanding of the categories, the discussions were made iteratively in seven successive steps for each design, using the same method but with the next batch of individually classified errata. Figure 2.8 shows the cumulative number of errata in each classification step. Figure 2.9 shows the evolution of the agreement of the decisions of the two humans. Note that, since the AMD errata were classified after the Intel errata, the data provided in Figure 2.9 is chronological. There are multiple reasons for mismatches, notably

(i) human errors as classification is a tedious, long, and difficult process;

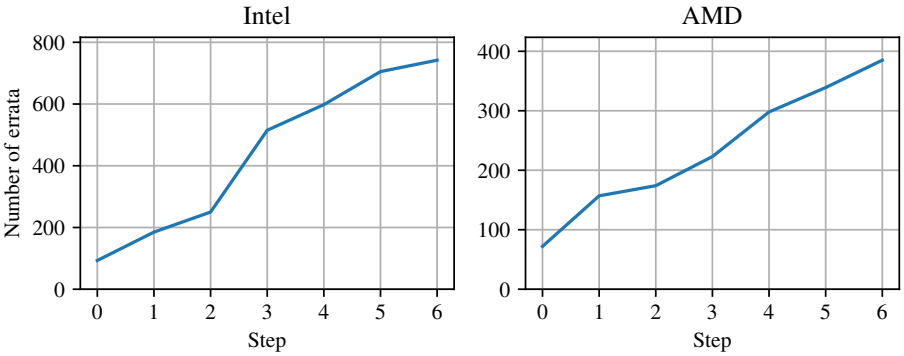

FIGURE 2.8: Number of errata per errata classification discussion step.

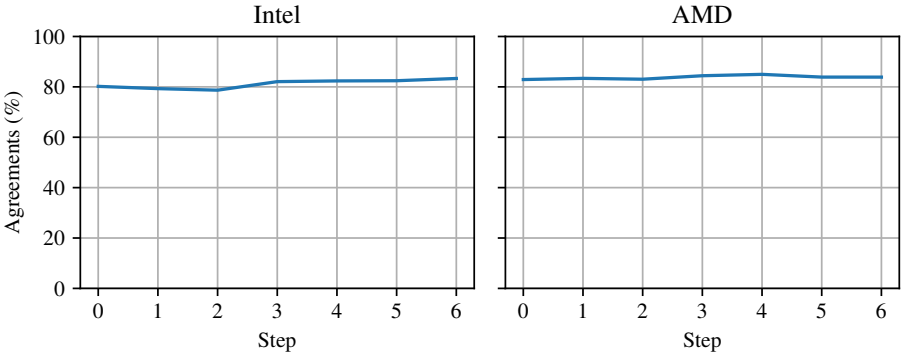

FIGURE 2.9: Percentage of human-classified errata-category pairs classified identically by both humans before the discussion.

(ii) imprecise description of the trigger, contexts, or effects in errata, leaving room for interpretation; and

(iii) ambiguous classification categories.

We note that the agreement percentage is generally above 80%.

SOFTWARE-ASSISTED CLASSIFICATION. The cumulative number of categories for triggers, contexts, and observable effects is large: in total, we defined 60 categories. First, we merge identical unique errata in the decision-making process, resulting in 1,128 remaining errata. This still amounts to $1128 \times 60 = 67{,}680$ classification decisions per human, even without considering the discussions for mismatches yet. We measured a typical average duration of 30 seconds per classification decision, which

Trg_MBR	*a data operation on a...*
- _cbr	cache line boundary.
- _pbr	page boundary.
- _mbr	memory map boundary such as canonical.

TABLE 2.4: Classification of MBR triggers.

Trg_MOP	*a memory operation involving...*
- _mmp	an interact. with a memory-mapped element.
- _atp	an atomic/transactional memory operation.
- _fen	a memory fence or a serializing instruction.
- _seg	a condition on segment modes.
- _ptw	a core page table walk.
- _nst	translation on nested page tables.
- _flc	flushing some cache line or TLB.
- _spe	a speculative memory operation.

TABLE 2.5: Classification of MOP triggers.

amounts to more than 560 hours of high-focus work per human merely for the individual classification part.

Fortunately, some classes can be automatically filtered out as irrelevant for a given errata, given the text describing it. Some others can be automatically said to be clearly relevant to an erratum. With conservative filtering based on regular expressions, we could reduce the number of decisions to 2,064 per human in the individual phase. These remaining decisions are difficult to make automatically and reliably. For example, if a reset signal is a trigger or an effect in an erratum based on its description. For guiding the human-based classification, we designed a syntax highlighting engine with regular expressions to emphasize parts of the errata descriptions relevant to a given category. With this tool's assistance, we could reduce the amount of pure classification work and discussion to approximately 30 hours per human in total. We release all code along with the RemembERR database and envision that such computer-assisted classification tools will encourage further contributions to errata classification.

Trg_FLT	*related to exceptions and faults*
- _ovf	a counter overflow.
- _tmr	a timer event.
- _mca	a machine check exception.
- _ill	an illegal instruction.

TABLE 2.6: Classification of FLT triggers.

Trg_PRV	*related to privilege transitions*
- _ret	a resume from System Management or OS mode.
- _vmt	a transition between hypervisor and guest.

TABLE 2.7: Classification of PRV triggers.

Trg_CFG	*related to dynamic configuration*
- _pag	a paging mechanism interaction.
- _vmc	a virtual machine configuration interaction.
- _wrg	a configuration register interaction.

TABLE 2.8: Classification of CFG triggers.

Trg_POW	*related to power states*
- _pwc	a transition between power states.
- _tht	a change in thermal or power supply conditions, or throttling.

TABLE 2.9: Classification of CFG triggers.

Triggers

Inputs that cause an exceptional observable effect are often not clearly stated or unspecified. At first sight, this renders the majority of errata unusable as they cannot easily be reproduced. However, we tackle this major challenge by designing a trigger classification scheme based on conditions that are *necessary* to cause an observable effect. Effectively, this means we define the required conditions under which suitable inputs can trigger a certain bug. This new classification method comes with several

Trg_EXT	*related to external inputs*
- _rst	a (cold or warm) reset.
- _pci	an interaction with PCIe.
- _usb	an interaction with USB.
- _ram	a specific DRAM configuration.
- _iom	an access through the IOMMU.
- _bus	system bus (HyperTransport, QPI, etc.).

TABLE 2.10: Classification of EXT triggers.

Trg_FEA	*related to features*
- _fpu	floating-point instructions.
- _dbg	debug features such as breakpoints.
- _cid	design identification (CPUID reports).
- _mon	monitoring (MONITOR and MWAIT).
- _tra	tracing features.
- _cus	other specific features (SSE, MMX, etc.).

TABLE 2.11: Classification of FEA triggers.

benefits. First, it allows deriving valuable insights even if only a limited amount of information is available. This makes our scheme especially useful for newer microprocessors or ISAs where fewer errata are available. Second, the categories we defined are largely independent and not exclusive, which allows for a simple estimation of a bug's complexity: the more necessary conditions are involved, the more complex the bug is to trigger. Furthermore, this classification scheme can easily be augmented in the future with new trigger classes, if needed.

In Table 2.4, Table 2.5, Table 2.6, Table 2.7, Table 2.8, Table 2.9, Table 2.10 and Table 2.11, we show all the categories for trigger that we defined on the *abstract* and *class* levels. We write *class* descriptors as the concatenation of two elements:

(i) a prefix determining whether it refers to a trigger, context, or effect, and

(ii) a suffix determining the *class*, given the prefix.

Ctx_PRV	*related to privileges*
- _boo	booting or being in the BIOS.
- _vmg	being a virtual machine guest.
- _rea	operating in real mode.
- _vmh	being a hypervisor.
- _smm	being in SMM.
Ctx_FEA	***related to features***
- _sec	security feature enabled (SGX, SVM, etc.).
- _sgc	running in a single-core configuration.
Ctx_PHY	***non-digital conditions***
- _pkg	package-specific.
- _tmp	temperature-specific.
- _vol	voltage-specific.

TABLE 2.12: Classification of contexts.

For example, the *class* Trg_EXT consists of all triggers involving external input (e.g., a PCIe device). We write *abstract* descriptors as the concatenation of two elements as well:

(i) a prefix determining the *class* where the *abstract* category belongs to, and

(ii) a suffix determining the *abstract* category, given the prefix.

For example, the *abstract* category Trg_EXT_rst refers to applying cold or warm resets.

Contexts

Some bugs can only happen in specific settings, for example, in a virtual machine guest or during BIOS/UEFI initialization. Bugs that can be provoked from user mode represent a particular security risk as unprivileged user applications are usually executed in this mode. Contrary to triggers, contexts are disjunctive: there may exist multiple contexts in which the *same* bug can be triggered. That said, for a given erratum, it is sufficient to be in

Eff_HNG	*related to hangs*
- _unp	an unpredictable behavior.
- _hng	a hang of the processor.
- _crh	a crash of the processor.
- _boo	a boot failure.
Eff_FLT	***related to faults***
- _mca	a machine check exception.
- _unc	an uncorrectable error.
- _fsp	one or multiple spurious faults.
- _fms	one or multiple missing faults.
- _fid	a wrong fault identifier or order.
Eff_CRP	***related to corruptions***
- _prf	a wrong performance counter value.
- _reg	a wrong MSR value.
Eff_EXT	***related to physical outputs***
- _pci	issues observable on the PCIe side.
- _usb	issues observable on the USB side.
- _mmd	multimedia issues (e.g., audio, graphics).
- _ram	abnormal interaction with DRAM.
- _pow	abnormal power consumption.

TABLE 2.13: Classification of observable effects.

any of its contexts to observe the bug. In Table 2.12, we list all the *abstract* and *class* context categories that we derived from our considered errata.

Observable Effects

The main objective of our effect classification is to find an answer to the question: where to look at when testing a design against an erratum with multiple observable effects?

In Table 2.13, we describe all the *abstract* and *class* categories for observable effects that we derived from the errata under study. Similar to

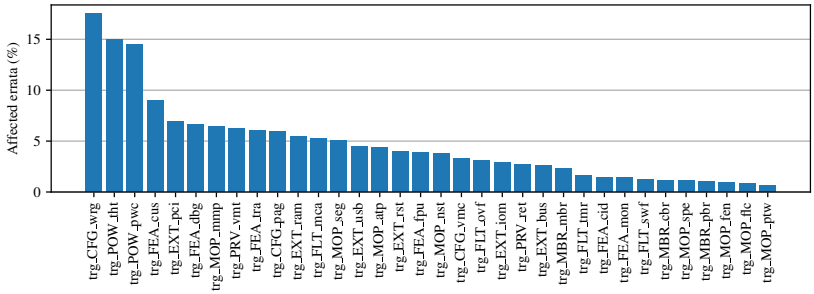

FIGURE 2.10: Most frequent triggers of all errata.

contexts, an erratum's observable effects are disjunctive. For example, if a corrupted configuration register *inevitably* leads to an unexpected fault, for instance, because its corruption triggers an exception, then this bug simultaneously belongs to two effect categories: wrong MSR value (`Eff_CRP_reg`) and spurious faults (`Eff_FLT_fsp`). There are also cases where an effect is observable in different ways. To give an example, an operation bringing the CPU to some incorrect power state can be observed either by reading a configuration register or by measuring the CPU's power consumption.

Only a few bugs can be considered non-critical: criticality generally depends on the assumptions made by the software running on the faulty CPU. Therefore, it is necessary to be conservative in this matter. For example, crashes and hangs are evidently critical: systems depending on the liveliness of the CPU would critically suffer. On the other extreme, seemingly innocuous wrong values in performance monitors could as well have critical consequences [79], not only on performance due to incorrect monitoring but also on security, since several recently proposed security defenses depend on the integrity of performance counters [80–89]. Wrong performance counter values open exploitable breaches in these defense systems.

2.5.2 *Insights*

In this section, we leverage RemembERR to present new insights about the most common triggers, contexts, and observations. To avoid any bias in our analysis, we use RemembERR with deduplicated (unique) errata.

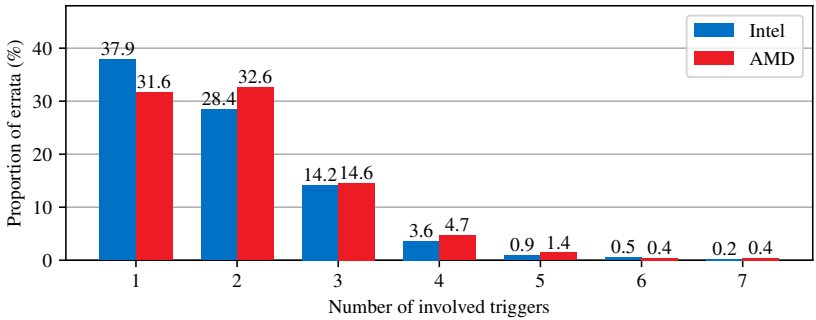

FIGURE 2.11: Number of errata by the number of triggers.

TRIGGERS. Our analysis starts by studying the most frequent triggers for Intel and AMD designs. We present the results in Figure 2.10. We can see that the most frequent triggers are either related to specific configurations set up by writing to model-specific registers (`trg_CFG_wrg`), power throttling (`trg_POW_tht`), or to power state transitions (`trg_POW_pwc`). More generally, many bugs require triggers related to power management, virtualization, external inputs, or features such as debugging or tracing. This suggests that implementing power management or communicating with other components (DRAM, memory-mapped components, peripherals such as PCIe) seems to be particularly challenging, thus resulting in many bugs. Such inputs correspond to stimuli that are difficult to supply to simulation or emulation prototypes that solely rely on the logical operation and that rule out power or peripheral physical layer considerations. Such bugs seem to be mostly discoverable by silicon testing. On the contrary, only five errata for AMD and one for Intel mention that the bug can only be triggered in simulation.

(O7) Observation. Most errata require specific MSR interaction or configuration combined with throttling, power state transitions, or peripheral inputs.

Figure 2.11 shows how many errata have a certain number of triggers. 14.4% of the errata do not specify any clear trigger or refer to trivial triggers such as usual load and store operations or intense workloads, and are therefore excluded from the figure. Mixing the errata from the two vendors, in total 49% of the errata require at least two combined triggers to cause a faulty behavior. However, we cannot derive whether bugs involving

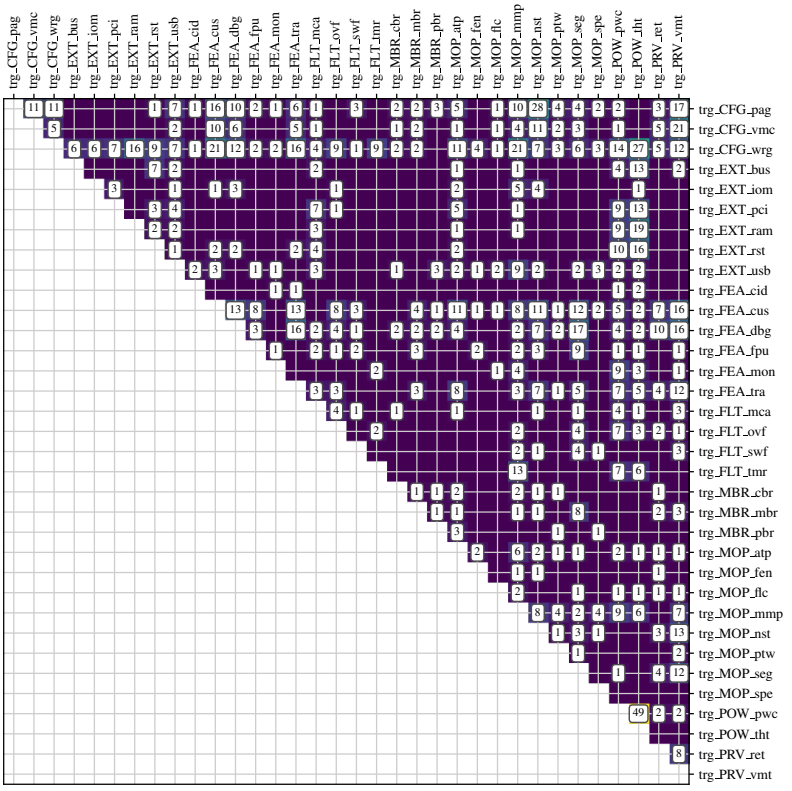

FIGURE 2.12: Pairwise cross-correlation between distinct abstract triggers. The values represent the number of errata documents that require *at least* these two triggers.

multiple triggers are rare or have been tested less. 8.7% of Intel and 20.8% of AMD unique errata mention that a "complex set of conditions" is required to trigger the bug. We ignored these indications as they are not precise enough to be exploited reliably.

Figure 2.12 shows pairwise correlations between triggers over all examined errata from AMD and Intel. This figure provides two specific insights: the relevant complex triggers and their interaction. Triggers typically interacting with other triggers are visible through highly populated lines. Regarding concrete interaction, a complex trigger can consist of debug features (`trg_FEA_dbg`) and virtual machine state transitions (`trg_PRV_vmt`), as we can see their intersection is salient. This insight is crucial for an efficient

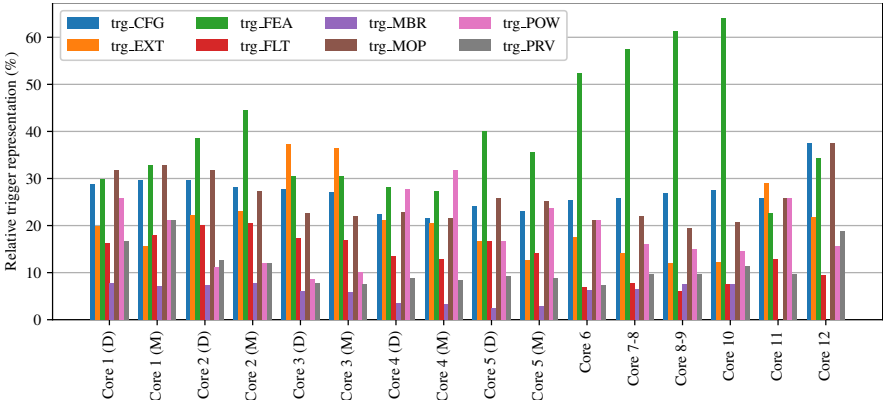

FIGURE 2.13: Trigger classes over Intel Core generations.

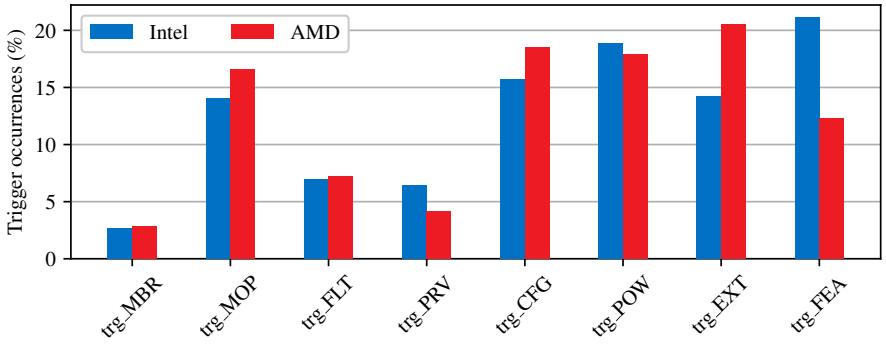

FIGURE 2.14: Relative representation of trigger classes between Intel and AMD.

and thorough testing campaign. For example, many bugs involving DDR (`trg_EXT_vmt`) or PCIe (`trg_EXT_pci`) will never be triggered until power levels change.

(O8) Observation. Some *abstract* triggers tend to correlate strongly, while most do not.

Figure 2.13 shows how the trigger classes evolved over different generations of Intel Core designs. Notably, errata triggered at memory boundaries (`trg_MBR`) are absent in the two latest Intel Core generations. This could be explained by different reasons: Intel's testing approach might have become more rigorous in this direction, or this kind of bug is now more difficult

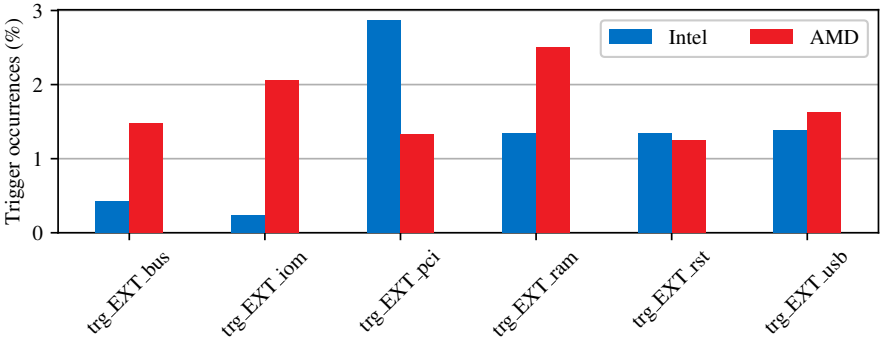

FIGURE 2.15: Relative representation of triggers related to external stimuli between Intel and AMD.

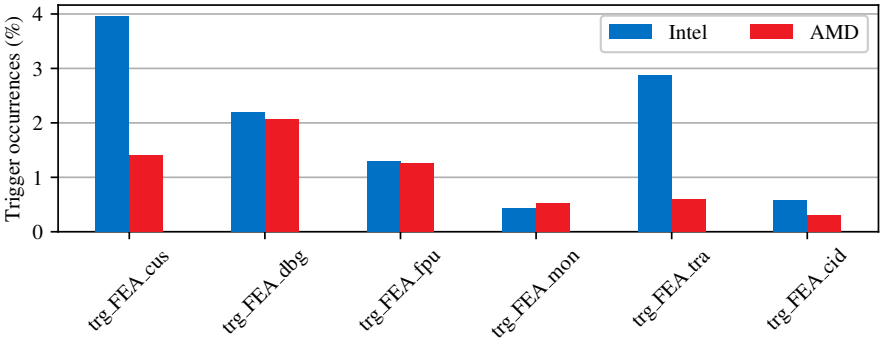

FIGURE 2.16: Relative representation of triggers related to specific features between Intel and AMD.

to find, or they have not yet been found and reported. Errata triggered by specific features or external communication have constantly been dominating. Without resorting to the former, more than 60% of the known errata cannot be reproduced in the 10th generation. Errata triggered by privilege transitions are gaining importance in the last generation. Importantly, all trigger classes are always necessary to trigger some bugs, except in the latest two generations. We additionally note that errata increasingly relate to specific features (`trg_FEA`), again, except for the latest two generations. Arguably, the latter generations may be too recent to draw conclusions, as we expect more errata to be released in the coming months and years.

(O9) Observation. It is necessary to apply all trigger *classes* to trigger all known bugs.

Figure 2.14 shows the relative trigger class representation between Intel and AMD errata. In this figure, for each vendor, we counted the total number of triggers for all unique errata and grouped them by the trigger classes. Overall, the representation of each trigger class is highly similar between the two vendors, which is interesting given that not only the designs are different, but also the vendors, testing and validation processes certainly substantially differ. Only the trigger classes related to external stimuli and specific features vary significantly between the two vendors.

(O10) Observation. The representation of trigger *classes* over the errata corpora is very similar for Intel and AMD.

Figure 2.15 and Figure 2.16 show a more specific analysis of the two latter trigger classes and clearly indicate the more specific differences between Intel and AMD errata. Concerning the triggers related to external stimuli, it is important to note that some CPUs offload certain peripheral functionalities to an external chipset whose errata are not necessarily included in the documents under study. Concerning the triggers related to specific features, we observe a clear overrepresentation of triggers related to custom features and tracing features in Intel compared to AMD.

CONTEXTS. In the next step of our study, we want to determine the context that most of the bugs require. Similar to the triggers, this knowledge is crucial for efficiently testing designs as certain bugs may only occur in specific contexts.

Figure 2.17 shows the most frequent contexts among Intel and AMD errata. Our data shows that running from within a virtual machine (`ctx_PRV_vmg`) is particularly prone to bugs. An explanation for this could be that today's hardware virtualization extensions (e.g., Intel's VT-x or AMD's SVM) are complex and deeply rooted in the CPU's design, making their rigorous testing more challenging.

(O11) Observation. Most errors occur in the context of hardware support for virtual machine guests.

EFFECTS. Next, we investigated which effects are the most valuable indicators for determining whether a bug was triggered. Figure 2.18 shows the

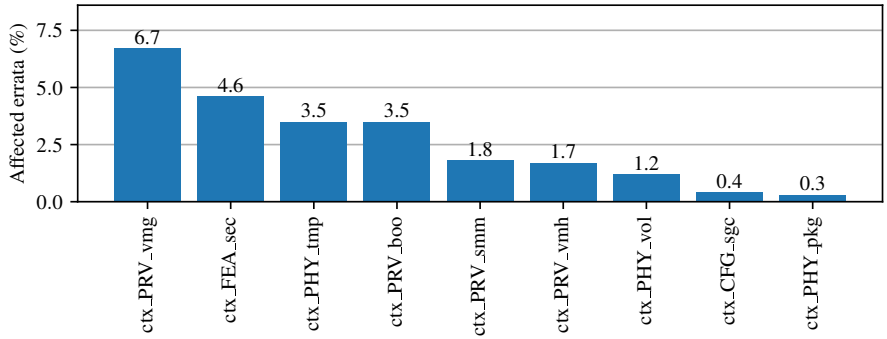

FIGURE 2.17: Most frequent contexts of all errata.

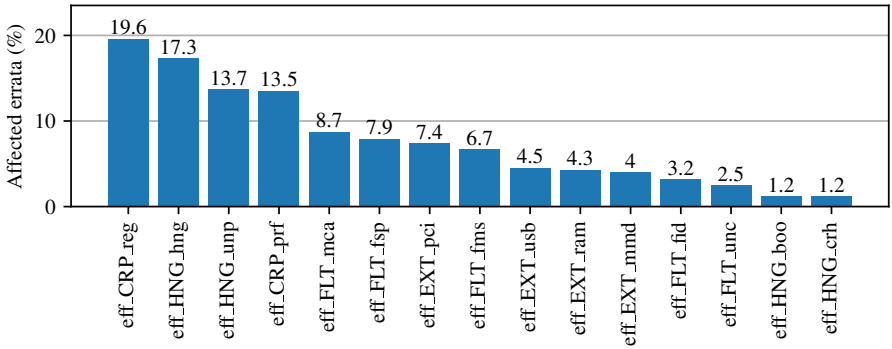

FIGURE 2.18: Most frequent effects for all errata.

most frequent observable effects in Intel and AMD designs. Most bugs manifest themselves as a corrupted register (`eff_CRP_reg`), a hang (`eff_HNG_hng`), or an unpredictable behavior (`eff_HNG_unp`). While an unpredictable behavior is not clear (vendors do not provide more information in these cases), the first two cases can easily be observed and may provide useful indicators for discovering new bugs.

(O12) Observation. Corrupted registers and hangs are the most common observable effect on Intel and AMD designs.

MODEL SPECIFIC REGISTERS. Based on our previous observation that corrupted registers are the most common observable effect, we wanted to know *which* registers provide information about unexpected behavior.

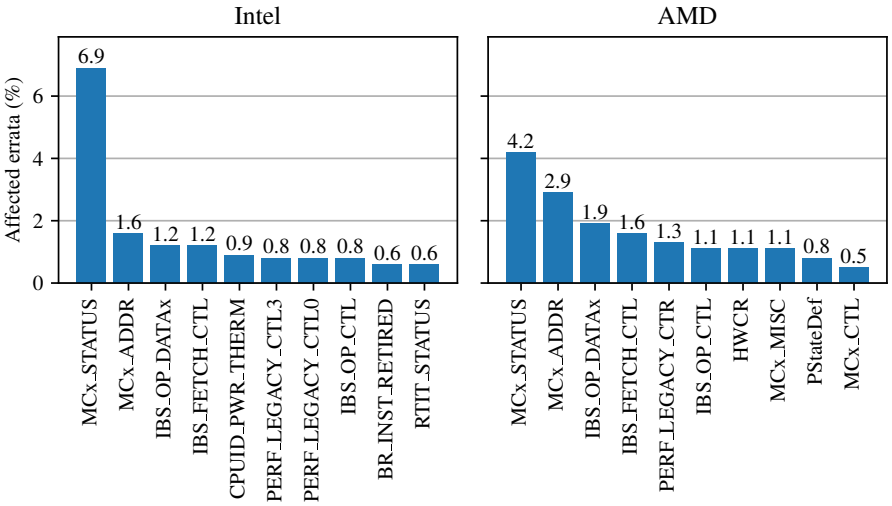

FIGURE 2.19: Most frequent MSR containing observable effects for Intel and AMD.

Figure 2.19 shows the most frequent observable effects in Intel and AMD designs. For both vendors, the machine check status registers (`MCx_STATUS` and `MCx_ADDR`) witness a bug in most cases (7.1% to 8.5% of all unique errata), followed by Instruction Based Sampling (IBS) registers and performance counters.

(O13) Observation. Among MSRs, Machine Check Status Registers most often indicate a bug's occurrence.

In summary, we designed a new hierarchical errata classification scheme that helps underlining new insights about reported bugs in complex designs. In Section 2.6, we concretely discuss how these insights may lead to improvements in design validation methodologies and toolchains.

2.6 APPLICATIONS TO DESIGN TESTING

In this section, we answer two questions. First, why do design testing and validation tools and methodologies, widely used in the industry, fail at detecting bugs reported in errata? Second, how would they benefit from RemembERR to detect past, present, and future bugs? We focus on how RemembERR can improve different families of testing and validation

methodologies rather than focusing on specific tools, given that vendors often make use of in-house tools [32].

2.6.1 *Dynamic methods*

Dynamic methods consist of applying inputs to a simulated, emulated, or physical (manufactured) design and verifying compliance of signals with a specification or expected values. While simulation [54–57, 90] and emulation [91, 92] are useful to find simple bugs early in the design process, silicon testing is necessary to find many complex bugs [41–45]. Two major challenges when looking for bugs with dynamic methods are the immense input space and the vast observation space, where state observation may interfere with attempts to trigger bugs. In both cases, RemembERR offers potential for improvement.

CHALLENGE: INPUT SPACE. CPUs usually have many pins and take sequential inputs over many cycles until a bug is eventually triggered; therefore, an exhaustive exploration is infeasible. A common response to this challenge is Constrained Random Verification (CRV) [19]. CRV applies a series of input signals that comply with a set of constraints, such as respecting a bus protocol. However, while this method can find shallow bugs in common design paths, the probability of triggering complex bugs is comparatively small.

Today's fuzzers have not settled on seed input corpora or a way to generate them. For example, RFUZZ [12] requires that "only some parameters to the fuzzer, such as the mutation technique and seed inputs to use, need to be specified by the user.". While reference [93] pretends to improve over the state of the art using an empty seed file, some successful experiments were seeded with some input sequences that stress interesting design features. DifuzzRTL [5] does not specify how it chooses its initial seed corpus, while HyperFuzzing [94] only vaguely says that it requires "an initial pool of inputs for the fuzzer seeded with a few interesting behaviors." TheHuzz [4] takes its configuration instructions statically from existing codebases, does not target specific functionalities, and samples its test instructions uniformly. Therefore, the young movement toward fuzzing hardware designs will seemingly profit from a more carefully selected initial input corpus.

Active research has targeted input generation for the pre-silicon phase [20–26, 35–38, 95–101], but it does not extend to emulation or silicon testing. Therefore, input generation for emulation or silicon testing is still an open

problem. RemembERR provides the best possible solution that does not require modifying the design in silicon. This is a significant advantage as modifying physical design is an enormous effort and ends up not testing the original design in all aspects. RemembERR precisely indicates which sets of inputs empirically interact and could trigger bugs, as shown in Figure 2.12. This knowledge can then be integrated into automatic dynamic testing of an emulated or manufactured design, taking the best of both worlds: targeted inputs and high execution speed under real-world conditions.

CHALLENGE: OBSERVATION SPACE. A too frequent and exhaustive observation can have detrimental effects. RemembERR provides empirical observation points that indicate CPU malfunctions and correlates them with the set of input types provided. This enables a much more fine-grained observation strategy, where the observation footprint is minimal.

In simulation, an excessive observation causes a longer run time. In emulation and silicon testing, the observation challenge becomes critical because almost all observations must be performed online, e.g., by reading performance counters. Excessive observations not only reduce testing performance but also hinder triggering bugs because heavy inspection may prevent timing-sensitive bugs from happening.

Besides, recent fuzzing work would benefit from enhanced observation heuristics. RFUZZ [12] is incapable of discovering any bug on its own [4] as it does not compare its state with any reference. Authors of TheHuzz [4] confirmed that directing observations is one major challenge for a synthesizable porting of their simulator-based testing system. DifuzzRTL [27] relies on a golden model implemented in software and may benefit from limiting its observation volume as well. Knowledge about which design parts to observe, in correlation with the supplied inputs, has the potential to empower such new fuzzer proposals.

RUNTIME DETECTION. Previous work proposes inserting pervasive CPU modifications for online bug detection [29, 30, 51, 52]. They are all data-driven, and in particular, each work performed an ad-hoc partial and non-systematic errata study. RemembERR provides all the necessary data to strengthen these systems and foster future work in this area without requiring researchers to conduct time-consuming errata studies repetitively.

2.6.2 *Formal methods*

Formal methods statically analyze a design along with properties that are usually specified manually [31]. For instance, Formal Property Verification (FPV) [32, 33, 39, 102, 103] ensures that a set of assertions is never violated in a given design. This technique is often used in two design testing stages [34]:

(a) after thorough CRV to validate corner cases, and

(b) when a silicon bug happens that has not been detected before.

Another instance of a formal method is Secure Path Verification (SPV) [102–106], which checks for unexpected information flows across designs, therefore allowing for design-global policies that are otherwise difficult to express with classical assertions. Formal methods are subject to state explosion and require a manual definition of the policies.

CHALLENGE: STATE EXPLOSION. A common approach for tackling the state explosion problem [31, 40, 60, 61] is to treat parts of the design as a "black box" and replace them by a model [40]. However, this limits the validation to properties specific to those parts that are not black-boxed [31]. An interesting yet unaddressed challenge is choosing the subset of the design that can be black-boxed to find a given class of bugs.

RemembERR provides a large amount of empirical data for identifying modules that typically interact with each other and could cause bugs. Our most immediate observation is that power management modules seem to have been vastly excluded from the parts of the design that are formally verified. We argue that the input correlation knowledge provided by RemembERR will substantially help to scope black-boxing more suitably.

CHALLENGE: HANDWRITTEN POLICIES. Policies and assertions are usually hand-written, for example, as SystemVerilog assertions [107] or in Property Specification Language [108]. This process is manual and error-prone, and to the best of our knowledge, no existing work proposed a way to resolve it.

RemembERR provides the necessary data to foster research in automatic data-driven policy generation for formal verification.

2.6.3 *Manual inspection*

Manual inspection is integral to design validation [31] but is challenging for complex designs and bugs of interest. Without any knowledge of common errors, manual inspection is deemed to fail. For example, an engineer responsible for testing a memory controller will benefit from the dozens of concrete bug instances that happened in the past and in other designs to ensure that they do not repeat. However, the current state of RembemberR is limited by the black-box nature of errata as they are published today. A description of the root cause associated with each erratum would provide enormous help in empirically distinguishing safe from dangerous hardware design practices.

2.7 DISCUSSION

We discuss selected observations we made while creating and interpreting RembemberR and provide further information for its future users.

PATCHABLE ERRORS. The errata documents do not show all known errors present in CPUs when they are released. Two classes of bugs are missing. First, bugs that are patchable by microcode updates. Such updates are usually not documented but can be reverse-engineered [109]. Second, bugs that are no longer valid, e.g., because a re-spin has been released and the older version is no longer officially supported. Errata of this type (about 2%) are listed in the summary of errata documents, but details (e.g., the description) remain hidden.

OTHER MICROPROCESSOR VENDORS. Intel and AMD are not the only major vendors of complex microprocessors that provide errata for their designs. ARM, for example, does so as well. We focused our work on Intel and AMD because they produce their design entirely from scratch, without relying on other vendors of major blocks (e.g., complete cores). Therefore, we expect these errata to be the most insightful as the vendors control the whole design process chain. Besides that, Intel and AMD microprocessors have a long history of designs, which gives us access to valuable long-term data.

LEARNING FROM THE PAST. The entire corpus of errata that we analyzed relies exclusively on bugs that have already been discovered, albeit some of them more recently. Therefore, our analysis cannot tell much about yet

undiscovered bug classes and trigger-context-effect correlations. However, in Section 2.4 we showed that bugs are sometimes rediscovered years apart, suggesting that industrial testing and validation methods can still gain rigor from our analysis. For example, our findings may help direct testing efforts to specific areas (i.e., contexts) that are known to be most affected by bugs. The efficiency of design testing can be improved by focusing on the most common triggers of these areas and the components where they typically have an observable effect. In addition, the growing open-source microprocessor community will learn tremendously from the errors of their more mature siblings, which will help in preventing similar mistakes.

RECOMMENDATIONS FOR ERRATA FORMATTING. Current errata documents are formatted for humans, and given the mistakes present in the documents, it is likely that vendors do not have any form of a systematic knowledge base for erratum storage and analysis. One vendor has confirmed not having such a database, and that their only source of errata knowledge is the errata documents and the employees' experience.

We propose a new format for errata descriptions, as exemplified by the transformation of Table 2.1 into Table 2.14, because the current state of the art for errata description (composed of a title, a problem description, implications, workarounds, and a status field) is unsatisfying for systematic analysis.

ROOT CAUSE. The root cause information is currently absent from almost all errata. One CPU vendor confirmed that triggers and effects are intentionally left inaccurate to avoid revealing design details, therefore there is limited hope for root cause publication but such databases may be maintained internally. With information on root causes, an errata database would go one step further than RemembERR by providing empirical data correlated with triggers, contexts and effects for identifying root causes, which is known to be a difficult problem [42, 44, 45, 48, 110]. Root cause information would additionally first underline which design parts are the most difficult to implement correctly and what are the most common mistakes.

2.8 RELATED WORK

Following, we present existing work that examines public CPU bug information (Section 2.8.1). Afterward, we introduce previous work that improves silicon testing (Section 2.8.2).

ID: [Some unique identifier shared with identical errata in other designs]

Title: x87 FDP Value May be Saved Incorrectly

Triggers:
Abstract: `Trg_FEA_fpu`
Concrete: Execution of FSAVE, FNSAVE, FSTENV, or FNSTENV

Contexts:
Abstract: `Ctx_PRV_rea`
Concrete: Operating in real-address mode or virtual-8086 mode

Effects:
Abstract: `Eff_HNG_unp`
Concrete: Incorrect value for the x87 FDP

Comments: This erratum does not apply if the last non-control x87 instruction had an unmasked exception.

Root cause: [Here, an explanation of the root cause may be provided]

Workaround: None identified.

Status: No fix.

TABLE 2.14: An erratum in the proposed format.

	Errata	Criteria	Goal
[29]	296 (OpenSPARC)	Type	Programmable module to react to errata online
[111]	280 (students)	Type	Design a testing flow capable of catching errors
[110]	Students & research	Type	Recommendations for end users

TABLE 2.15: Summary of work that examined errata in open-source CPUs.

	Errata	Criteria	Goal
RemembERR	2,563	Trg/Ctx/Eff	Provide support for data-driven design testing
[30]	301	Severity	Monitor hardware security invariants
[53]	37	Location	Find internal signals corresponding to bugs
[52]	172	Type, severity	Programmable module to react to errata online
[51]	470	Location, severity	Programmable module to react to errata online
[50]	535	Location, severity	Taxonomy for dependable systems
[112]	–	–	Recommendations for end users

TABLE 2.16: Summary of work that examined errata in commercial CPUs.

2.8.1 *Errata-based*

Table 2.15 and Table 2.16 summarize work that studied errata from open-source and commercial CPUs, respectively. Previous work provides frag-

mented information extracted from errata to provide a classification that justifies a specific approach that solves a given problem. Unlike RembembERR, none provides clear insights on representative bug triggers and effects in modern CPUs. Insights from RembembERR are exploitable for future research in design testing and validation.

2.8.2 *Directed silicon testing*

There are two directions aiming to improve inputs and observations for silicon testing.

ELIMINATING THE GOLDEN MODEL. Golden models are a bottleneck for silicon testing. Wagner et al. [47] propose to use complementary pairs of instruction blocks to ensure that the CPU's architectural state is left untouched after execution if no bug has been triggered on the CPU. Foutris et al. [49] identify equivalences between instructions from different ISAs to compare executions on largely different CPUs.

BUG OBSERVABILITY. In some cases, it is difficult to triage a bug. Lin et al. [48, 113] propose methods to transform an instruction sequence to accelerate the observability of a triggered bug. Farahmandi et al. [114] propose an observability measure consisting of test sequences.

RembembERR is complementary to and compatible with these directions by providing insights for more effective input generation and guidelines about the elements to observe for efficient testing.

2.9 CONCLUSION

We analyzed 2,563 errata from all Intel Core and AMD CPUs since 2008. Individually, these errata provide little insight other than a description of a particular bug, but collectively they provide insightful information about gaps in current design testing and validation practices. To this end, we built RembembERR, a large-scale database of annotated errata. We solved the challenge of unclear triggers by the observation that triggers are almost always conjunctive. In contrast, contexts and effects are disjunctive: observing the most convenient location is sufficient. In our analysis using RembembERR, we discovered the most common triggers, contexts, and effects in errata, which we then correlated to provide concrete guidelines for the next generation of design testing and validation tools.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their valuable feedback. The work in this chapter was supported by a Microsoft Swiss JRC grant, the Swiss State Secretariat for Education, Research and Innovation under contract number MB22.00057 (ERC-StG PROMISE), and the Swiss National Science Foundation under NCCR Automation, grant agreement 51NF40_180545.

3

CELLIFT: LEVERAGING CELLS FOR SCALABLE AND PRECISE DYNAMIC INFORMATION FLOW TRACKING IN RTL

Dynamic Information Flow Tracking (dynamic IFT) is a well-known technique with many security applications such as analyzing the behavior of a system given an input and detecting security violations. While there are many widely used open dynamic IFT solutions that scale to large software, the same level of support is unfortunately lacking for hardware. This gap is becoming more pronounced with the increasing complexity of open-source hardware and the plethora of recent hardware attacks.

We introduce CellIFT, a new design point in the space of dynamic IFT for hardware. CellIFT leverages the logical macrocell abstraction (e.g., an adder) to achieve scalability, precision and completeness when instrumenting a given Register Transfer Level (RTL) hardware design. Cell-level dynamic IFT does not suffer from the scalability problems that are inherent to lower levels of abstraction such as gates, yet it achieves completeness given the limited number of cell types. We show the versatility of CellIFT by instrumenting five distinct RISC-V designs, one of which is a complete SoC. The only existing complete solution already fails to instrument two of these designs. Our extensive evaluation using microbenchmarks and standard RISC-V benchmarks on the instrumented designs shows that CellIFT is 21× to 61× faster than the state of the art in terms of simulation runtime without losing precision. We further show-case concrete applications of CellIFT in four scenarios by detecting: 1) sources of microarchitectural information leakage, 2) microarchitectural bugs such as Meltdown, 3) speculative vulnerabilities such as Spectre-BCB, and 4) SoC-wide architectural design flaws. We release CellIFT as open source to enable RTL-level security research for the wider community.

3.1 INTRODUCTION

Despite substantial design verification efforts, there is a continual discovery of security-critical hardware design flaws that are exploitable from software [2, 3, 77, 78, 115–126]. These vulnerabilities are often difficult to mitigate post-silicon, leaving systems exposed for long periods of time. While critical, current tools and techniques are unfortunately incapable of capturing these issues in large designs. There is hence an urgent need for empowering Electronic Design Automation (EDA) tools for detecting security vulnerabilities during hardware design.

DYNAMIC INFORMATION FLOW TRACKING. A promising approach for analyzing the state of a system given an input and verifying certain security properties is Information Flow Tracking (IFT). Static IFT either needs to consider all possible states which does not scale beyond very simple designs [31, 60, 61] or it over-approximates, leading to precision problems [127]. In contrast, dynamic IFT only considers changes to the state made in a single execution, allowing it to potentially scale to larger hardware designs for checking design-wide security properties. As an example, many of the recently discovered security flaws [2, 3, 77, 117, 123–126] can be formulated as a dynamic IFT constraint. While there are many popular dynamic IFT tools that scale to large software [128–130], the same level of support is currently lacking for hardware. With the increasing popularity of open-source hardware, an open-source dynamic IFT solution that scales to larger designs has the potential to significantly improve security testing and enable new security applications.

CELLIFT. It is possible to instrument a given design at one of the two extreme abstraction levels for dynamic IFT: either by considering its low-level gate netlist [1]; or by considering the high-level Hardware Description Language (HDL) [131]. Unfortunately, both abstraction levels come with significant shortcomings: instrumenting the gates has severe *scalability* issues while instrumenting the HDL requires managing complex language constructs, making it challenging to achieve *completeness*. We make a key observation that instrumenting the *macrocells*, which are at a slightly lower abstraction level than HDL, preserves the benefits of both extremes without their shortcomings. There are a manageable number of cell[1] types (e.g., adder, shifter, multiplexer, etc.) for which we can create *shadow cell types* that precisely track the information flows and scale comfortably to larger designs.

1 We use the terms cell and macrocell interchangeably.

Unlike gates, these shadow cells map efficiently to large units available in commodity CPUs (e.g., registers or arithmetic and logic units), significantly improving the performance of dynamic IFT during simulation. To design these shadow cells, we introduce a generic precise information flow tracking logic scheme called *m-replica* based on cell replication. To make this scheme scale efficiently with increasing cell widths while preserving precision, we exploit the cells' mathematical properties of monotonicity, transportability and Translability that we formally define and leverage for the first time. We then prototype CellIFT, our dynamic IFT solution that instruments a given Register Transfer Level (RTL) design by leveraging the design of our shadow cell types.

To show the versatility of CellIFT, we use it to instrument five RISC-V cores, one of which integrated in a System on a Chip (SoC) which we also instrument to show that CellIFT can successfully be applied on various complex and heterogeneous designs. The only existing (gate-level) solution that can handle generic designs [1] already fails at instrumenting and simulating two of these five designs. Our evaluation shows that compared to instrumenting gates, CellIFT is more precise, more than 5× faster with instrumentation and synthesis while requiring 5× less memory, and more than 21× faster during simulations. We show-case the benefits of CellIFT in four different scenarios using our instrumented designs: first, we show how CellIFT can be used to measure changes to the microarchitectural state as a result of executing an instruction or a memory access. This information can be used to detect various sources of timing-based information leakage [132–134]. Second and third, we show how CellIFT can enable the detection of microarchitectural vulnerabilities such as Meltdown [2] or MDS [77, 78], or speculative execution attacks such as Spectre [3] respectively. Finally, we use CellIFT to detect design flaws that can lead to architectural security vulnerabilities using various scenarios from a hardware hacking challenge [31].

In summary, we make the following contributions:

- We present a scalable, precise and complete cell-level dynamic IFT design.
- We implement CellIFT based on this new design as new passes into the Yosys open-source synthesizer [135].
- We evaluate CellIFT on five RISC-V designs: Ibex [136], Ariane [74], Rocket [137], BOOM [75] and the PULPissimo SoC from the Hack@DAC'18 competition [31].

- We use CellIFT in several scenarios to show the benefits of scalable and precise dynamic IFT support as part of the hardware design toolchain.

OPEN SOURCING. To enable reproducibility and to let researchers and practitioners benefit from CellIFT, we publish the source code of CellIFT, the experiments and the instrumented designs at this URL: `https://comsec.ethz.ch/cellift`.

3.2 BACKGROUND

In this section, we provide a brief background on existing techniques for detecting architectural vulnerabilities, their (in)effectiveness against recent microarchitectural vulnerabilities, and discuss how hardware dynamic IFT mechanisms can provide a better alternative.

3.2.1 *Detecting architectural vulnerabilities*

Hardware designers employ various methods in an attempt to detect flaws in the RTL representation. These methods are manual or automatic; automatic methods are local or global.

MANUAL INSPECTION. Dessouky et al. [31] recently showed that existing verification methods do not effectively cover many of the cross-layer bugs resulting from subtle hardware-software interactions. Hardware designers often resort to manual inspection of the RTL and simulation of hand-crafted input to detect these complex cases. Unfortunately, this approach is cumbersome, error-prone, and incompatible with any form of continuous integration that can catch subtle vulnerabilities introduced after the initial hardware design. Errata of recent complex designs such as 12th generation Intel® Core™ processors contain an overwhelming proportion of such bugs [138].

LOCAL METHODS. Hardware designers often rely on SystemVerilog assertions (SVA) [139] to ensure correct behavior and capture unintended behavior that can lead to bugs at all design stages. These assertions express local properties such as compliance to a given bus protocol, or compliance of a state machine with some expected properties. Assertion-based verification formal methods such as Formal Property Verification (FPV) [32, 33] can provably check whether these assertions can be triggered in a given

RTL design and provide examples when they do. These formal methods unfortunately suffer from state explosion due to their static nature, limiting their scalability. Therefore, hardware designers and verification engineers often deploy Constrained Random Verification (CRV) extensively [19]. CRV tries out a series of signals while respecting some constraints, such as complying with some input bus protocol, to empirically find out whether local assertions can be triggered.

GLOBAL METHODS. Many security properties are not expressible in the form of local assertions. For instance, a local assertion cannot infer the origin or destination of some data transfer and therefore cannot verify confidentiality properties in the general case. Formal methods such as Security Path Verification (SPV) [140] and Formal Security Verification (FSV) [102] are designed to statically catch unauthorized information flows. However, these methods are reported to suffer from the same scalability issues as local formal methods and typically requires black-boxing parts of the design [40], hampering their ability to detect information flows across the entire design.

3.2.2 *Microarchitectural vulnerabilities*

While architectural security vulnerabilities already pose a challenge for existing tools and techniques, microarchitectural security vulnerabilities that do not explicitly collide with architectural specifications pose an even greater challenge. These security vulnerabilities are not only legion [2, 3, 77, 78, 115–126], but industry leaders still struggle to mitigate them, sometimes requiring multiple generations of attempts before reaching an effective mitigation as seen in the recent MDS class of vulnerabilities [77, 78, 118, 141]. Such vulnerabilities can lead to confidentiality breaches, effectively enabling arbitrary read primitives. Recent work [142] attempts to detect microarchitectural vulnerabilities by exploiting the RTL description, but requires tailoring to a specific design and vulnerability, without providing exploitability insights. We hence urgently need a design-agnostic solution that can detect different classes of vulnerabilities with little effort.

3.2.3 *Dynamic hardware IFT*

Hardware dynamic IFT provides the possibility of following how information flows propagate in a design [1, 131]. Confidentiality, integrity, isolation, constant time and design integrity properties are canonical properties covered by dynamic IFT [143]. As opposed to static methods, dynamic IFT does not consider the entire set of possible states in a given design, but instead allows to dynamically prove properties in a specific context. Consequently, it is immune to the state explosion problem.

Dynamic IFT requires: a *mechanism* for tracking information flows and *policies* expressed on top of this mechanism. According to certain policies, signals are temporarily or permanently labeled as *taint sources* or *taint sinks* during runtime, and an alarm is triggered when an information flow from a taint source to a taint sink is detected through the mechanism. Confidentiality policies inspect the data flow from secret data locations (taint sources) to unauthorized entities (taint sinks). Conversely, integrity policies inspect data flows from unauthorized entities (taint sources) to sensitive locations (taint sinks). As an example, Meltdown-type [144] class of vulnerabilities [2, 77, 78, 117, 118, 123–126] can be expressed as a policy that disallows memory loads from a different domain.

Two hardware dynamic IFT mechanisms have been proposed so far: GLIFT [1] and RTLIFT [131]. They add new elements to the design to support dynamic IFT, but at different levels: GLIFT *instruments* the design at the level of elementary logic gates (AND, NOT, OR and multiplexers), and RTLIFT proposes to instrument the HDL directly. We will show that GLIFT has critical scalability problems, and (to the best of our knowledge) a complete RTLIFT has never been implemented due to the tremendous engineering effort required.

Table 3.1 summarizes all the verification techniques discussed in this section. While dynamic IFT is an attractive alternative for detecting hardware vulnerabilities, existing solutions such as GLIFT [1] and RTLIFT [131] have severe limitations that hamper their adoption. In the following section, we analyze these limitations and discuss how our new hardware dynamic IFT design addresses them.

	Local	Global
Static	FPV [32, 33]	SPV [140], FSV [102]
Dynamic	CRV [19]	GLIFT [1], RTLIFT [131]

TABLE 3.1: Classification of design verification methods.

3.3 DYNAMIC HARDWARE IFT USING CELLS

There are three properties that are significant for any hardware dynamic IFT mechanism to see adoption: first, it should be able to operate on any given (valid) digital design (i.e., the *completeness* property). Second, it should scale to large designs with high instrumentation performance and usable simulation overhead (i.e., the *scalability* property). Finally, it should faithfully propagate tainted signals while minimizing the taint spilled to additional signals in the design (i.e., the *precision* property). Unfortunately, none of the existing techniques cover all these important properties together.

The most common strategy is instrumenting designs with IFT logic at the gate level [1, 145–148]. While achieving completeness due to the limited number of different gates, it suffers from scalability problems: since IFT logic construction occurs after logic synthesis (i.e., once the design is expressed as a list of elementary logic gates), it incurs an exponential worst-case time complexity at instrumentation time [146]. As an example, instrumenting Ibex [136], a small RISC-V design, requires 72× more elements when instrumenting the design with gates, in comparison with the non-instrumented design. Perhaps more importantly, the gate-level approach also incurs high overhead at simulation time, as it forces simulators to simulate the design and the shadow logic gate by gate, while a significant speed-up would result from simulating higher-level constructs such as additions or comparisons.

Another issue with gate-level IFT logic is its precision. While precise IFT for a given design is an undecidable problem [149], doing so at the (low) level of gates exacerbates the problem. While there exists precise IFT logic for individual gates, the interaction of IFT logic from different gates leads to imprecision (i.e., overtainting).

To improve this situation, it is possible to generate the IFT logic at a higher level of abstraction. Previous work on elevating the abstraction level uses

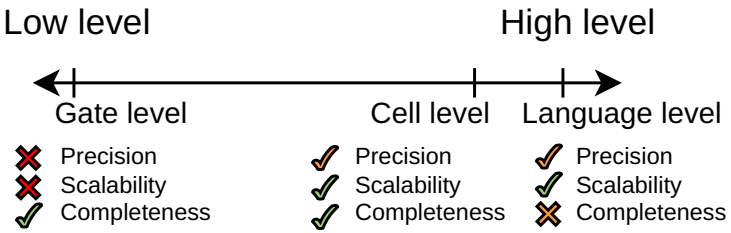

FIGURE 3.1: Different levels of instrumentation and their characteristics.

HDL [131, 150]. They either introduce new language features that require porting by a human expert at significant time and backward-compatibility cost [150], or it is challenging to make them complete given the many possible language constructs to be supported [131].

IFT LOGIC BASED ON MACROCELLS. We argue that to gain the benefits of both strategies, we need to operate at a different level of abstraction than gates or HDL. This level should be high enough to achieve high performance and precision, while generic enough to achieve completeness and backward-compatibility. Macrocells, to which we refer as *cells*, denote higher-level intermediate representations of synthesizable hardware primitives (such as an adder, comparator, etc.). Cells are limited in types but are parametrizable in widths and in other important properties such as signedness. Whereas working at gate-level requires breaking these primitives, cells are close to HDL and often map directly to HDL constructs. Therefore, cells form typical intermediate representations in hardware tools, explicitly in LLHD [151] and Yosys [135], and in Verilator which maps similar constructs to the simulating machine's ISA [90]. This makes cells an ideal candidate for generating the IFT logic as shown in Figure 3.1. Because all cells correspond to HDL constructs, CellIFT's shadow logic design is also a *necessary* basis for any HDL instrumentation that would strive to achieve completeness in the future.

Designing a scalable and precise IFT logic for general-purpose digital designs, however, poses certain challenges. First, it is unclear whether we can follow a generic approach for designing a precise IFT logic for any given cell, leading us to our first research question:

RQ1. Is there a generic IFT logic pattern that could precisely instrument any combinational cell?

Second, while a generic approach will enable us to achieve completeness as we will soon discuss, it will come at a high cost. To reduce this cost, we can perhaps make use of the structure of the cells themselves, leading to our second research question:

RQ2. Do cells have certain logical properties that we can exploit to scale the resulting IFT logic?

Section 3.4 answers the first question by introducing a novel m-replica architecture which can be adapted to implement the IFT logic with perfect precision for any given combinational cell. This architecture, however, scales exponentially with the cell's width. Section 3.5 answers the second question by introducing three fundamental logical properties of different cells, namely monotonicity, transportability, and Translability that can be leveraged to adapt the m-replica architecture for creating efficient per-cell IFT logic.

3.4 THE CANONICAL M-REPLICA ARCHITECTURE

We introduce a replication-based architecture that can be used to generate IFT logic for any given cell with perfect precision.

3.4.1 *Precise information propagation*

Let C be a combinational cell, and C^t its IFT logic. C has some input bits $(I_j)_{0 \leq j < N_i}$ and output bits $C(I) := (Y_j)_{0 \leq j < N_o}$[2]. We define $(I_j^t)_{0 \leq j < N_i}$ and $C^t(I, I^t) := (Y_j^t)_{0 \leq j < N_o}$ to be the taint signals corresponding to C's input and outputs, respectively. The terms N_i and N_o represent the number of input bits and the number of output bits of the cell. This means that $I_j^t = 1$ if the input signal I_j is tainted, and similarly $Y_j^t = 1$ if the output signal Y_j is tainted.

2 We use the := notation when defining new terms.

Because some cells, such as adders, have two inputs A and B of identical width, we define $A_j := I_j$ and $B_j := I_{j+\frac{N_i}{2}}$. We refer to the index j in A_j or B_j as *operand position*, as opposed to the position in the aggregated input I.

PRECISE INFORMATION PROPAGATION RULE. Information propagates from the set of tainted input signals $\{I_j \mid I_j^t = 1\}$ to some output signal Y_j if there exists another input vector $\tilde{I}$ for C, which differs from I only on tainted input bits, and such that the two input vectors I and $\tilde{I}$ cause distinct values for Y_j. Equation 3.1 formalizes this rule. $\oplus$ denotes exclusive or.

$$\begin{aligned} C^t(I, I^t)_j = 1 \iff \\ \exists \tilde{I} \mid (I \oplus \tilde{I}) \wedge \overline{I^t} = 0 \text{ and } C(\tilde{I})_j = \overline{C(I)_j} \end{aligned} \tag{3.1}$$

Equivalently, for a given input I with taint vector I^t, and for a given output bit Y_j, Y_j is tainted if the value of Y_j can be changed by only changing the value of I at some tainted indices. We use this insight in the design of IFT logic using cell replication.

3.4.2 *Replication-based design*

Any digital circuit can be represented as an interconnection of state-holding cells (flip-flops and latches) and purely combinational (stateless) cells. We discuss how we can instrument these different cell types using replication.

STATE-HOLDING CELLS. A simple replication suffices to instrument state-holding cells: when a signal is delayed by entering such a cell, the same delay affects the information carried by this signal, as shown in Figure 3.2a. This means that for every state-holding cell, we simply need an additional shadow cell that stores the taint information for that cell.

COMBINATIONAL CELLS. Combinational blocks should be instrumented with combinational logic as illustrated in Figure 3.2b. As shown in Figure 3.2c, dividing combinational blocks results in simulating more elements individually to generate intermediate signals and limits performance and precision.

Based on Equation 3.1, precise information flow tracking can be achieved by trying each possible input $\tilde{I}$, filtered by the input taints I^t. This can be achieved by replicating 2^{N_i} instances of C. This *canonical* design can create precise IFT logic for any cell, given that it uses copies of the cells.

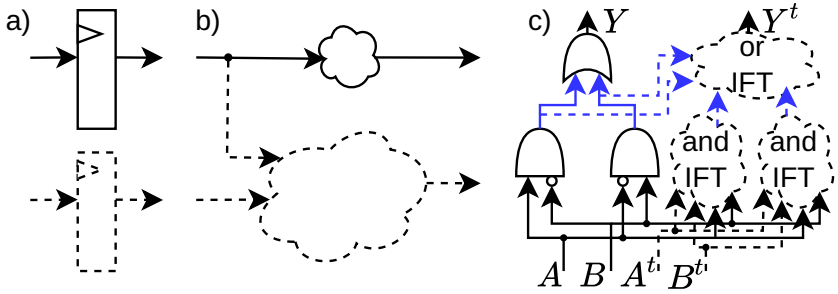

FIGURE 3.2: Instrumentation of a) a state-holding cell, b) a combinational block and c) an exclusive-or cell with single input bit at gate level. Signals generated to feed the gate-level IFT logic but absent at cell level are drawn in blue. The t exponent indicates taint signals.

Figure 3.3 shows the design of such an IFT logic for a combinational cell C with 2 inputs and 3 outputs. The four instances C^{00}, C^{01}, C^{10}, C^{11} are identical instances of C supplied with distinct inputs depending on the input taint assignment. Equation 3.2 formalizes these instances as $C^v(I, I^t)$, with v in unsigned binary representation. In Equation 3.2, I^t has the role of selector in multiplexers between I and v.

$$C^v := C^v(I, I^t) := C((I \wedge \overline{I^t}) \vee (v \wedge I^t)) \tag{3.2}$$

We call such IFT logic architectures m-replica, where m is the number of instances of the original cell present in the IFT logic. The canonical m-replica architecture requires an exponential number of copies of C in the input size N_i. In the following sections, we specialize this generic structure using cells' mathematical properties to improve its scalability.

3.5 EXPLOITING THE LOGICAL PROPERTIES OF CELLS

We exploit monotonicity (Section 3.5.1), transportability (Section 3.5.2) and translatability (Section 3.5.3) properties to enhance the performance of replication-based cell IFT logic by reducing its size without loss of precision.

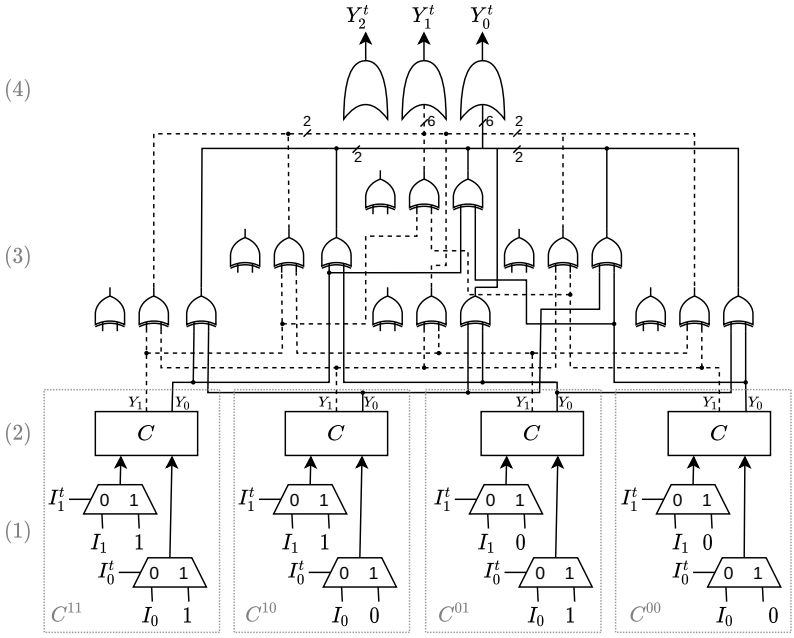

FIGURE 3.3: IFT logic for a cell C with 2 input bits and 3 output bits. The wires corresponding to the highest order output bit are omitted, and those of the second output bit are dashed. Stage (1) replaces the tainted inputs with all possible value combinations. In stage (2), C is replicated 2^{N_i} times to take all input combinations. Stage (3) compares the outputs of all replicas. Finally, for each output bit index, a taint is set if two replicas have different output bits at the corresponding index.

3.5.1 *Monotonic cells*

Prevalent cells such as comparators and some logical reductions (e.g., multi-bit OR cells) feature a property which we call monotonicity. Monotonicity allows pruning of the canonical replica-based IFT logic to obtain a constant-complexity 2-replica IFT logic. We define three monotonicity properties:

1. *bitwise non-decreasing.* A cell is bitwise non-decreasing in input offset j if no output bit of the cell can fall from 1 to 0 when I_j is raised from 0 to 1. For example, an OR cell is bitwise non-decreasing in all its input bits.

2. *bitwise non-increasing.* A cell is bitwise non-increasing in input offset j if no output bit of the cell can raise from 0 to 1 when I_j is raised from 0 to 1. For example, an inverter cell is bitwise non-increasing in all its input bits.
3. *bitwise monotonic.* A cell is monotonic if with respect to each of its input bits, the cell is non-increasing or non-decreasing.

IFT LOGIC FOR BITWISE NON-DECREASING CELLS. Let C be bitwise non-decreasing in all its input bits. We build a 2-replica IFT logic using *polarization* as described in Equation 3.3.

$$Y^t = C^{0\ldots0} \oplus C^{1\ldots1} \tag{3.3}$$

Proof. Let C be a bitwise non-decreasing cell in all its input bits. Let $0 \leq j < N_o$ (N_o is the output width of the cell), and consider a fixed input taint vector I^t. If the output bit Y_j is zero for some input I, then Y_j is also zero for the input $\tilde{I^0} := I \wedge \overline{I^t}$ given the non-decreasing property. Conversely, if Y_j is one for some input value $\tilde{I}$ such that $I \wedge \overline{I^t} = \tilde{I} \wedge \overline{I^t}$ (i.e., I and $\tilde{I}$ differ only on tainted bits), then Y_j is also one for the input $\tilde{I^1} := I \vee I^t$, again given the non-decreasing property. It follows that all output bits that can be toggled by applying two inputs I and $\tilde{I}$ of tainted bits are also toggled between applying $\tilde{I^0}$ and $\tilde{I^1}$. This allows us to reduce m-replica to 2-replica using polarization: $Y^t = C(\tilde{I^0}) \oplus C(\tilde{I^1}) = C^{0\ldots0} \oplus C^{1\ldots1}$. □

IFT LOGIC FOR BITWISE MONOTONIC CELLS. We now extend polarization to all bitwise monotonic cells. If C is non-increasing with respect to an input bit at index i, then it is non-decreasing in the negation of this input bit. Because bit inversion does not affect taint propagation, any monotonic cell can similarly be instrumented according to Equation 3.4, where $d_i := 1$ if C is non-decreasing in input bit i, and $d_i := 0$ if C is non-increasing in input bit i.

$$Y^t = C^{d_{N_i-1}\ldots d_0} \oplus C^{\overline{d_{N_i-1}\ldots d_0}} \tag{3.4}$$

SUMMARY. Table 3.2 shows the 2-replicas used to instrument logic reductions and comparisons. Unsigned comparisons, and or-/and-reductions are bitwise non-decreasing cells in all input bits. Signed comparison cells are bitwise non-decreasing on all bits except the most significant bit of the operands, which is non-increasing.

Cell	Replica 1		Replica 2
Reductions	$C^{0\ldots0}$	$\oplus$	$C^{1\ldots1}$
Unsigned comparisons	$C^{00.0;11.1}$	$\oplus$	$C^{11.1;00.0}$
Signed comparisons	$C^{10.0;01.1}$	$\oplus$	$C^{01.1;10.0}$

TABLE 3.2: 2-replica-based instrumentation of monotonic cells. Logical reductions represent multi-input OR or AND cells with single output. Comparisons represent $<$, $\leq$, $\geq$ and $>$.

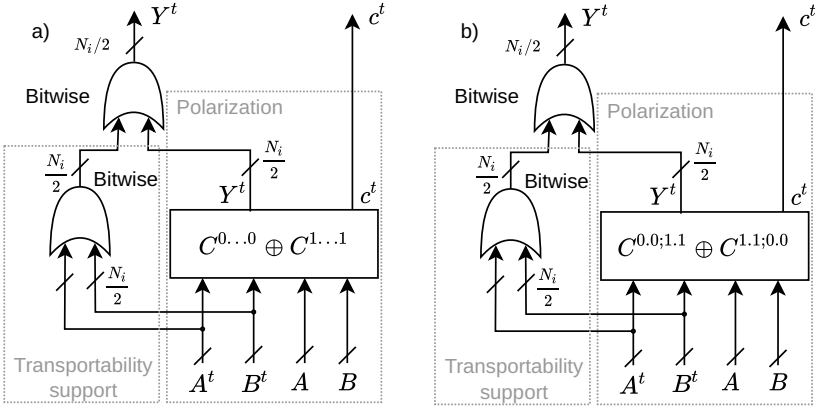

FIGURE 3.4: 2-replica-based IFTL for a) an adder cell and b) a subtractor cell.

3.5.2 *Transportable cells*

Abundant arithmetic cells such as addition and subtraction cells can be instrumented in constant complexity thanks to their transportability property. Transportable cells have the same width for each operand, and information from an operand bit always flows to the corresponding output bit. We can instrument these cells using polarization (similar to monotonic cells) complemented with transportability-supporting IFT logic that implements a conjunction of the input bits.

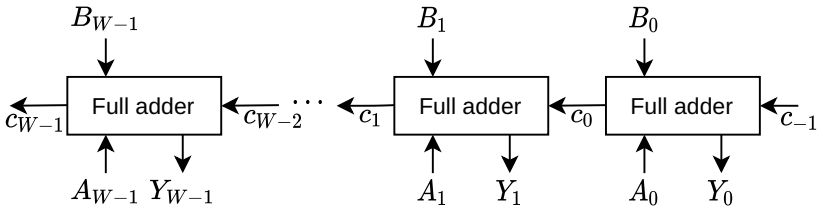

FIGURE 3.5: Notations for the ripple carry adder.

Adder cells

We design the IFT logic of the adder cell as described by Equation 3.5 and illustrated in Figure 3.4-a using polarization conjuncted with the transportability term $A^t \vee B^t$. We prove the correctness and precision of the adder IFT logic illustrated in Figure 3.4 and equivalently given by Equation 3.5. We conduct a proof by induction on a ripple carry adder. Because any adder implementation fulfills the same mathematical function, the proof holds for any adder implementation.

NOTATIONS. We denote by $W := \frac{N_i}{2}$ the width of each input A and B. As illustrated in Figure 3.5 we adopt the following notations: for the jth full adder inputs: A_j, B_j and c_{j-1}, and outputs Y_j and c_j, with the carry bit sequence $c := \{c_W, ..., c_0, c_{-1}\}$ where $c_{-1} := 0$. We define $\tilde{I^0} := I \wedge \overline{I^t}$ (tainted inputs deasserted) and $\tilde{I^1} := I \vee I^t$ (tainted inputs asserted). The generic notation X represents Y or c.

PROOF BY INDUCTION. We consider the following induction property H_{X_j}, for $0 \leq j < W$, for a fixed input I with taints I^t:

«[$\exists\ \tilde{I}$ such that $\overline{(I \oplus \tilde{I}) \wedge \overline{I^t}}$ (match_on_nontaint property) and $X_j(I) \neq X_j(\tilde{I})$ (toggled$_{X_j}$)] $\iff$ [$X_j(\tilde{I^0}) \neq X_j(\tilde{I^1})$ (polarization$_{X_j}$) or $A_j^t \vee B_j^t$ (transport$_j$)]»: H_{X_j}.

Intuitively, we want to prove, for each output bit Y_j and each carry bit c_j (including the intermediate carry bits), that if I and some other input $\tilde{I}$ match on non-tainted bits and this bit Y_j or c_j is toggled between inputs I and $\tilde{I}$, then polarization and transportability taint the information flow (*correctness*), and conversely (*precision*).

Proof. Let us first prove H_{c_0} and H_{Y_0}. Because $Y_0 = A_0 \oplus B_0$ and $c_0 = A_0 \wedge B_0$, then for Y_0 or c_0 to be tainted, at least one of A_0 and B_0 must be tainted, and conversely. Therefore H_{Y_0} and H_{c_0} hold by (transport$_0$).

Let us now prove H_{c_j} and H_{Y_j} for a given $1 \leq j < W$, assuming that $H_{c_{j-1}}$ holds. We start by showing the implication: $\exists \tilde{I}$ such that (match_on_nontaint) and (toggled$_{X_j}$) $\implies$ (polarization$_{X_j}$) or (transport$_j$), corresponding to correctness.

If A_j^t or B_j^t, then H_{c_j} and H_{Y_j} hold by (transport$_j$). Let us suppose from now that $A_j^t = B_j^t = 0$ and suppose the existence of $\tilde{I}$ that satisfies the conditions (match_on_nontaint) and (toggled$_{c_j}$). Because A_j and B_j are not tainted and therefore identical in I and $\tilde{I}$ by (match_on_nontaint), and because $c_j = \left[A_j + B_j + c_{j-1} \geq 2\right]$, it results that c_{j-1} is toggled when applying $\tilde{I}$ instead of I. By $H_{c_{j-1}}$, (a) Either c_{j-1} is toggled between $\tilde{I}^0$ and $\tilde{I}^1$ (polarization$_{c_{j-1}}$), (b) Or $A_{j-1}^t \vee B_{j-1}^t$ (transport$_{j-1}$). Hence, in both cases (a) and (b), c_j is also toggled between $\tilde{I}^0$ and $\tilde{I}^1$, i.e., (polarization$_{c_j}$) holds.

Let us now prove: $\exists \tilde{I}$ such that (match_on_nontaint) and (toggled$_{X_j}$) $\impliedby$ (polarization$_{X_j}$) or (transport$_j$), corresponding to precision. If (polarization$_{X_j}$), then $\tilde{I} = \tilde{I}^0$ or $\tilde{I} = \tilde{I}^1$. If (transport$_j$), then because $c_j = \left[A_j + B_j + c_{j-1} \geq 2\right]$ and Y_j is $\left[A_j + B_j + c_{j-1}\right] \mod 2$, $\tilde{I} : I \oplus (1 \ll j)$ is a candidate.

Therefore, H_{Y_j} and H_{c_j} hold. We have proved correctness and precision of the IFT logic described by Equation 3.5. □

This proof generalizes to any adder implementation, since different architectures provide the same mathematical function for addition.

$$Y^t = \left[C^{0\ldots0} \oplus C^{1\ldots1}\right] \vee A^t \vee B^t \tag{3.5}$$

Subtractor cells

Similarly to the adder cell, we base the IFT logic of the subtractor cell on a 2-replica polarized architecture supplemented with transportability logic to form the IFT logic given in Figure 3.4-b. The first polarization term takes operand A with tainted bits set to zero and operand B with tainted bits set to one, and conversely for the second polarization term. A proof by induction in all aspects is similar to the proof given for the adder since

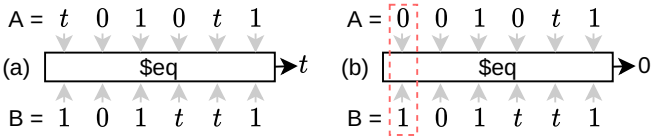

FIGURE 3.6: Examples of an equality (a) with tainted output, and (b) without a tainted output. The equality cells compare the top input bits with the corresponding bottom input bits.

subtraction can be formulated for this architecture based on the identity $A - B = A + \overline{B} + 1$.

Negation cells

We define I as the data concatenation of the inputs (i.e., $\{A, B\}$), and I^t as the taint input concatenation (i.e., $\{A^t, B^t\}$) The negation cell defined by $Y = \overline{I} + 1$ exposes the same property as an adder, except that there is a single operand and no carry bit in the negation cell. Therefore, we instrument it in constant complexity using the same polarization replicas, conjuncted with the transparency term I^t that corresponds to a single operand.

(In)equality cells

(In)equality cells are transportable under the condition that all non-tainted bit pairs are equal. More precisely, the output bit of an (in)equality cell is tainted if and only if the two following conditions are fulfilled:

1. At least one input bit is tainted.
2. For each operand position where no input bit is tainted, the two input bits are equal.

Figure 3.6 shows two equality cells. The equality cell (a) has its output tainted because it fulfills the two conditions. However, cell (b) has its output non-tainted, because the leftmost bits do not match and are not tainted. Note that the value of tainted input bits never matters. We design the IFT logic of the equality cell as in Equation 3.6, where $T_{AB} := \overline{A^t \vee B^t}$

Cell	Polarization	Junction	Transportability
Add	$C^{0\ldots0} \oplus C^{1\ldots1}$	$\vee$	$A^t \vee B^t$
Sub	$C^{0.0;1.1} \oplus C^{1.1;0.0}$	$\vee$	$A^t \vee B^t$
Neg	$C^{0\ldots0} \oplus C^{1\ldots1}$	$\vee$	I^t
Eq/Neq	$C(A \wedge T_{AB}; B \wedge T_{AB})$	$\wedge$	$\bigvee I^t$

TABLE 3.3: 2-replica-based instrumentation of transportable (addition, subtraction, negation) or conditionally transportable (equality, inequality) cells.

represents which bits are neither tainted in A nor in B. The term $\bigvee I^t$, where $I^t := \{A^t, B^t\}$, is one if and only if any of the input bits is tainted.

$$Y^t = C(A \wedge T_{AB}; B \wedge T_{AB}) \wedge \bigvee I^t \tag{3.6}$$

Summary

We summarize the IFT logic of transportable and conditionally transportable cells in Table 3.3.

3.5.3 *Translatable cells*

Some cells do not present powerful properties such as monotonicity or transportability for generating efficient and precise IFT logic. In this section, we take a different approach to tackle the problem: instead of relying on the replication mechanism immediately, we consider each output bit, and examine which input condition results in tainting output bits.

Input decomposition

Examining each output bit of a cell and its relationship with input taints and values results in complex formulas that are not efficient to implement for wide-input cells that can be present in real digital designs. It is often convenient to make simplifying assumptions such as some operand being non-tainted. For instance, for the left shift operator $A \ll B$, if we assume that $B^t = 0$, then $Y^t = A^t \ll B$. We show that the combination of two

properties, namely *translatability* and *taint combination surjectivity*, allows us to construct new architectures that support such simplifying assumptions.

TRANSLATABILITY. A two-input cell C is said to be *right side translatable* (over addition) if it satisfies Equation 3.7 for all inputs A and B.

$$C(A, B' + B'') = C(C(A, B'), B'') \tag{3.7}$$

As an example, a left shift by an unsigned offset provides this property: $A \ll (B' + B'') = (A \ll B') \ll B''$.

Additionally, the following decomposition holds: $B = B^0 + B \wedge B^t$, where $B^0 := B \wedge \overline{B^t}$ (i.e., B where all the tainted input bits are zeros). This leads to Equation 3.8, which has two instances of C on the right-hand side: one instance with a non-tainted second operand B^0, and one instance where all non-tainted bits of the second operand are zero.

$$C(A, B^0 + B \wedge B^t) = C(C(A, B^0), B \wedge B^t) \tag{3.8}$$

TAINT DECOMPOSITION SURJECTIVITY. Equation 3.7 relies on cell composition. However, it is known that in the general case, cell composition does not preserve precision of the IFT logic [1, 131]. We introduce the *taint decomposition surjectivity* property: a cell's IFT logic L is taint decomposition surjective if and only if precision is unaffected when performing IFT through this cell. Assuming k output bits are tainted, if L can generate the 2^k outputs by changing the data at tainted input bits, then L is taint decomposition surjective.

Let us formalize this property and use it to prove the precision of the IFT logic for unsigned-offset logical shift cells. Let C be a cell with IFT logic $L := C^t$. L is said to be *taint combination surjective* if for all inputs I and all taint vectors I^t, it satisfies Equation 3.9. Informally, such an IFT logic does not create any new interdependencies between taint signals, because all potential outputs $\tilde{Y}$ that match with Y on non-tainted output bits Y^t are obtainable from inputs $\tilde{I}$ that match with I on non-tainted input bits I^t.

$$\forall \tilde{Y} \exists \tilde{I} \mid C^t((I \wedge \overline{I^t}) \vee (\tilde{I} \wedge I^t)) = (Y \wedge \overline{Y^t}) \vee (\tilde{Y} \wedge Y^t) \tag{3.9}$$

A composition of cells $C'(C(I))$, each with precise IFT logic L and L', can be precisely instrumented with $L' \circ L$ if L is taint combination surjective. A composition of taint combination surjective IFT logics is also taint combi-

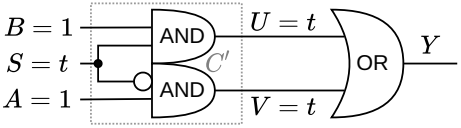

FIGURE 3.7: Multiplexer as the succession of a cell C' and an OR gate. $A = B = 1$, $A^t = B^t = 0$ and $S^t = 1$, therefore the value of S does not matter for the IFT logic.

nation surjective: the first IFT logic allows to generate all the intermediate combinations for the second by surjectivity.

INTUITION. We introduce a counterexample, where C^t is not taint combination surjective. The multiplexer is known to be imprecise [1] if it is instrumented as the composition of one OR gate after two AND gates. Figure 3.7 shows a multiplexer made of two cells: C' and an OR gate. The outputs U and V of C' are tainted, given the precise taint propagation rule. C'^t is not taint combination surjective, because for $A = B = 1$, $A^t = B^t = 0$ and $S^t = 1$, there is no input $\tilde{I} = \{\tilde{A}, \tilde{B}, \tilde{S}\}$ which matches with I on non-tainted bits (i.e., on A and B), that produces the output $\tilde{Y} := \{U = 0, V = 0\}$.

LOGICAL SHIFTS. All logical shift cells with untainted offset are taint combination surjective. We provide a proof for the left shift. The proof for the right shift is similar.

Proof. Let B be the untainted offset (i.e., $B^t = 0$). Let A be the shifted input, with taints A^t. Then, $Y = A \ll B$ and $Y^t = A^t \ll B$. Let $\tilde{Y}$ be a vector of same width as Y, with $Y \wedge \overline{Y^t} = \tilde{Y} \wedge \overline{Y^t}$. Then, $\tilde{A} := \tilde{Y} \gg B$ is a preimage of $\tilde{Y}$. □

Logical unsigned-offset shift cells

We show that logical shifts by an unsigned offset $C(A, B) := A \ll B$, or $C(A, B) := A \gg B$ can be decomposed in two successive shifts, each instrumented separately, without losing precision. We proceed according to Equation 3.8, which implies decomposing C into two cells copies, instrumented separately: the first cell in the decomposition is $C(A, B^0)$, where

the offset B^0 is independent of the tainted value assignments. The IFT logic for this replica is given by $C(A^t, B^0)$, i.e. $A^t \ll B^0$ and $A^t \gg B^0$ for logical left and right shifts respectively.

The second cell in the decomposition is $C(A', B \wedge B^t)$, where A' is the output of the first cell. Let us discuss a precise bitwise IFT logic for this second cell.

We compute an IFT logic for the right shift cell by an unsigned offset by introducing a pivoting technique. This computation relies on the assumption that the shift offset B and its taint vector B^t verify $B \wedge \overline{B^t} = 0$ (i.e., B is zero at all non-tainted indices). This property is provided in the second shift cell obtained from the translatability property.

LOGICAL SHIFTS. We introduce the notation $\overset{t}{=}$, which denotes that two vectors match on non-tainted bits, as defined in Equation 3.10, and its counterpart $\overset{t}{\neq}$ in Equation 3.11. The decomposition by translability guarantees that all the non-tainted bits in the shift offset $B \wedge B^t$ of this cell are zero, which substantially simplifies the IFT logic computation. We prove that an IFT logic for this second cell can be expressed by Equation 3.12 for right shifts and by Equation 3.13 for left shifts.

$$U \overset{t}{=} V :\iff U^t \vee V^t \vee \overline{[U \oplus V]} \tag{3.10}$$

$$U \overset{t}{\neq} V :\iff U^t \vee V^t \vee [U \oplus V] \tag{3.11}$$

$$Y_j^t = \bigvee_{k=0}^{2^{N_B}-1} \left(\left[B \overset{t}{=} k \right] \wedge \left[A_j \overset{t}{\neq} A_{j+k} \right] \right) \tag{3.12}$$

$$Y_j^t = \bigvee_{k=0}^{2^{N_B}-1} \left(\left[B \overset{t}{=} k \right] \wedge \left[A_j \overset{t}{\neq} A_{j-k} \right] \right) \tag{3.13}$$

Focusing on a logical right shift, the jth output bit Y_j is tainted if either (*explicit tainting*) the taint results from shifting some tainted bit A_k to the right to the index j, or (*implicit tainting*) some offset $\tilde{B}$ that matches with B on non-tainted bits gives a different value for Y_j, than the offset B.

Because we know that the full-zero vector satisfies the matching condition of $\tilde{B}$, there is always some *pivot* $\tilde{B}^0 = 0$ that maps A_j to Y_j. We iterate with

some integer k through all the 2^{N_B} values of $\tilde{B}$, assuming the worst case where the taint vector B^t is full of ones. We complete A with zeroes beyond the most significant bit to simplify the equations without loss of generality. If there is any integer k in this range such that (*implicit tainting*) there is a $\tilde{B}$ that can bring A_{j+k} to Y_j, then Y_j is tainted if $A_j \neq A_{j+k}$ (because there are two $\tilde{B}$ instances that result in different values for Y_j), or (*explicit tainting*) if A_j or A_{j+k} is tainted, because then there is some $\tilde{B}$ that shifts a tainted value to the output bit Y_j, leading to Equation 3.12.

ARITHMETICAL RIGHT SHIFT. The IFT logic of the second cell in the translatability decomposition of an arithmetical right shift can be expressed by Equation 3.14.

$$Y_j^t = \bigvee_{k=0}^{2^{N_B}-1} \left(\left[B \stackrel{t}{=} k \right] \wedge \left[A_j \stackrel{t}{\neq} A_{min(j+k,N_A-1)} \right] \right) \tag{3.14}$$

The only difference with logical right shift tainting relies in the *implicit tainting* part. Instead of completing A with zeroes, A must be completed with values equal to its most significant bit A_{N_A-1}. From there, a calculation similar to the logical counterpart leads to the IFT logic described in Equation 3.14.

We now sketch a proof of the precision of the composition of the two IFT logic instances resulting from the decomposition of the arithmetical right shift.

The $N_A - B^0$ most significant bits of $C(A, B^0)$'s output benefit from the taint combination surjectivity of the first cell's IFT logic, because these bits are shifted by a non-tainted offset. Therefore, the second cell's output taints corresponding to these inputs in step 2 are precise. The only tainted signals that cannot be surjectively enumerated are the $N_A - B^0$ most significant bits, which take A's most significant bit's value. However, although a comparison between these could lead to an erroneous intermediate result, they will be eventually tainted by the $\stackrel{t}{=}$ operator.

Therefore, the composition of the two instances is precise. Because logic unsigned-offset shift cells are taint combination surjective, the sequence of the two IFT logics is precise.

Cell	Comp. 1	Comp. 2 (bitwise)
$\ll$	$C(A, B^0)$	$\bigvee_{k=0}^{2^{N_B}-1} \left(\left[B \stackrel{t}{=} k \right] \wedge \left[A_j \stackrel{t}{\neq} A_{j-k} \right] \right)$
$\gg$ (logic)	$C(A, B^0)$	$\bigvee_{k=0}^{2^{N_B}-1} \left(\left[B \stackrel{t}{=} k \right] \wedge \left[A_j \stackrel{t}{\neq} A_{j+k} \right] \right)$
$\gg$ (arith)	$C(A, B^0)$	$\bigvee_{k=0}^{2^{N_B}-1} \left(\left[B \stackrel{t}{=} k \right] \wedge \left[A_j \stackrel{t}{\neq} A_{min(j+k, N_A-1)} \right] \right)$

TABLE 3.4: 1-replica-based instrumentation of translatable cells. The IFT logic is the sequence of two components. Shift cells in this table have an unsigned interpretation of B.

Arithmetical unsigned-offset shift cells

The arithmetical unsigned-offset shift cell benefits from the same right-side translatability as the logical shifts. We decompose them in two cells before instrumentation, similarly to their logical shift counterparts. The only difference is the propagation of the most significant bit. The IFT logic of the first cell is identical to the logical counterpart.

Summary

In this section, we used the translatability property to decompose some cells into two identical cells with simplifying assumptions on operands as summarized in Table 3.4, while preserving perfect precision.

3.6 IMPLEMENTATION

In this section, we provide additional details and describe the implementation of CellIFT.

3.6.1 *CellIFT flow implementation*

We integrated CellIFT into the Yosys synthesizer as a synthesis pass over the design's internal representation. Since Yosys does not have a complete understanding of all the SystemVerilog constructs, we first pre-process the designs using the open-source sv2v tool that converts SystemVerilog (IEEE

1800-2017) to Verilog (IEEE 1364-2005) [152]. We refer to the processing by Yosys as the *instrumentation* step.

For simulation, we use the open-source Verilator simulator [90]. For emulation, we use Xilinx Vivado. We refer to the processing by Verilator to produce a simulation binary, or the processing by Vivado until the end of the FPGA implementation, as the *synthesis* step.

The CellIFT Yosys pass is made of 4575 lines of C++ code. CellIFT supports a total of 181 Yosys cell types, of which many are variations of state-holding elements with reset, enable and clear signals. 8 unsupported cell types are memories and their ports, because they are substituted in a previous stage and do not reach the CellIFT pass. The 33 remaining cell types are not supported because they have never been encountered when experimenting on various heterogeneous designs. Next, we provide more information on how we support memories in CellIFT and describe simple techniques for instrumenting the remaining cells other than the ones described in Section 3.5.

3.6.2 *Instrumenting other cells*

MEMORY MODELS. In digital designs, memories are typically substituted with specific components depending on the design's target: simulation model (simulation), block RAM (FPGA), or SRAM (ASIC). CellIFT implements memory models with *conservative* IFT rules as follows: (a) Taints are written and read along with the corresponding data. (b) Memory read from a tainted address also returns a tainted value. (c) Memories are marked fully tainted as soon as the inputs and their taints authorize a write operation to a tainted address. We also implemented memory models that only perform *explicit* tainting, i.e., when only rule (a) applies. This allows us to learn which microarchitectural components are dependent on the tainted address of a load instruction.

LOWER-LEVEL CELLS. Exclusive ORs propagate taint as soon as at least one input bit is tainted. Logic gates ((N)ANDs, (N)ORs and inverters) are parametrizable in width to profit from wider instructions on the simulating machine. Multiplexers with multiple selector bits are decomposed into a tree of single-bit selector multiplexers without losing precision. Then, for a multiplexer of formula $Y = S?B : A$, we design the IFT logic expressed in Equation 3.15, where S, $\overline{S}$ and S^t are replicated to have the same width as A and B.

$$Y^t = (A^t \wedge \overline{S}) \vee (B^t \wedge S) \vee ([A \oplus B] \wedge S^t) \tag{3.15}$$

3.6.3 *Imprecise cell instrumentations*

IMPRECISE SHIFT CELLS. As opposed to their unsigned counterpart, signed-offset shift cells are not right-side translatable. Because a precise implementation of these shift cells is expensive, CellIFT implements a replication-based approximate propagation policy: taint Y completely whenever B is tainted, else shift the taints of A by B.

Additionally, on all the designs that we considered, we noted that right shifts are never larger than 10 bits, whereas left shifts are often larger in five reference open-source designs. Therefore, for unsigned shift offsets, in CellIFT we instrument right shifts precisely and left shifts imprecisely.

IMPRECISE MULTIPLIER CELL. As proposed in [131], we decompose the multiplier into a sequence of adders before instrumenting it. This results is an imprecise IFT logic, but this does not affect overall precision much since multipliers are mostly present only on data paths. For emulation targets, to decrease the critical paths we used a shadow logic made of a single OR reduction, without practical decrease in precision.

3.6.4 *Summary*

CellIFT instruments all the 22 combinational cell types that we encountered in our diverse experiments. From these 22 cell types, 3 are similar to GLIFT when they have a single bit per input. Section 3.7 provides the cell composition of non-instrumented, and instrumented designs[3].

3 One year after publication of this article, RTLIFT authors finally shared their code with us. For comparisons and equality testing, RTLIFT taints whenever one input bit is tainted. It treats logic operators AND, OR and NOT operators similarly to CellIFT. It instruments AND and OR reductions using basic gates. It instruments the shift in the same way as CellIFT's conservative version. RTLIFT is not able to instrument any of the designs in this study because of the small number of supported Verilog contructs.

3.7 EVALUATION

In this section, we evaluate CellIFT in terms of performance, precision and completeness by instrumenting heterogeneous and complex designs. We use microbenchmarks to show how CellIFT compares in terms of precision and scalability with previous work [1, 131] (Section 3.7.1). To show the scalability of the instrumentation phase, we instrument five heterogeneous designs of various complexities (Section 3.7.2). To show the performance and precision of CellIFT-instrumented designs, we simulate standard RISC-V benchmarks (Section 3.7.3). Finally, we show FPGA portability of the CellIFT-instrumented designs (Section 3.7.4).

EVALUATION SETTING. The performance results were obtained on a machine equipped with an AMD EPYC 7H12 processor at 2.6 GHz equipped with 256 logical cores and 1 TB of DRAM. We used Verilator 4.212 and g++11.2 with the -O0 compiler flag similar to what is used for certain OpenTitan designs [153, 154]. Using compiler optimizations causes spurious segmentation faults in the Verilator simulation binaries.

BASELINES. To the best of our knowledge, no mature open-source or transparent-enough commercial implementations of gate-level or language-level hardware IFT is available. Therefore, we implemented GLIFT as described in the original paper [1] for comparison purposes. As the authors of RTLIFT [131] were reluctant to share their implementation or provide additional details, we re-implemented the few operators described in [131]. Our implementation reproduces their operator-level performance and precision results.

3.7.1 *Microbenchmarks*

To evaluate the performance and precision of CellIFT, we instrument and simulate individual combinational cells with various widths to evaluate scalability. We apply one million random inputs to each cell and randomize the taint bits. We measure precision by counting the tainted output bits.

Figure 3.8 shows the performance and precision results of cell microbenchmarks: CellIFT provides a massive speedup in simulation over existing mechanisms. RTLIFT and CellIFT-instrumented adders and multipliers have the same precision, whereas CellIFT-instrumented cells are faster. By design, left shifts suffer from poor precision in this benchmark, which

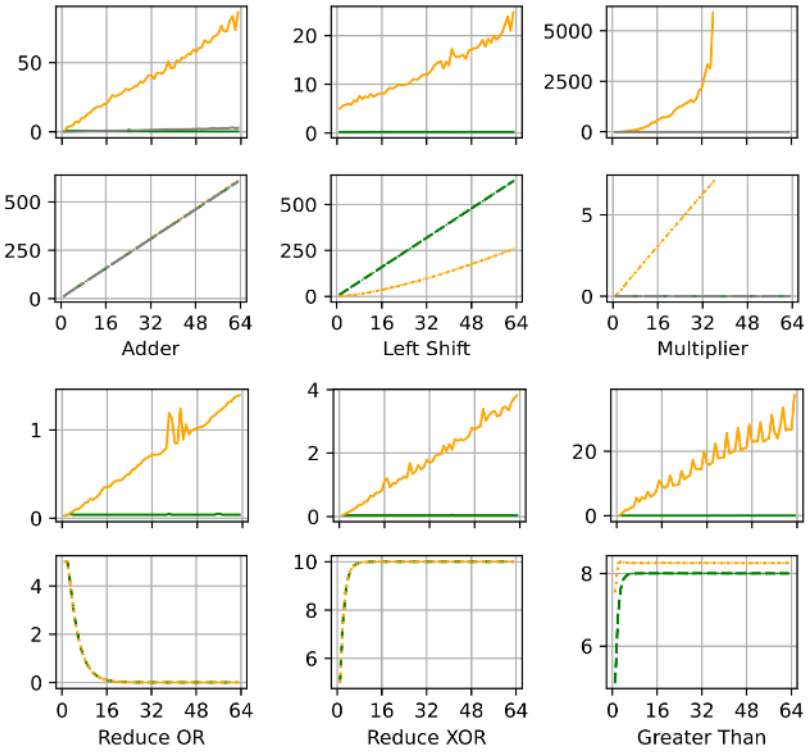

FIGURE 3.8: Cell microbenchmarks: runtime performance (top) and precision (bottom) depending on the cell input width, measured from 10 parallel identical cells to reduce time measurement noise. *Reduce* cells are multi-bit input, single-bit output logic cells (e.g., AND). Precision numbers represent the cumulative number of tainted output bits. The left shift cell has an 8 bits wide offset operand, which is common in the designs that we examined. RTLIFT only instruments *add* and *mul*. Solid lines represent performance and dashed lines represent precision of CellIFT (green), RTLIFT (gray) and GLIFT (orange).

is based on randomly tainting all input bits: we traded off some empirically superfluous precision in this cell for speed. This design point is not fundamental as we showed a perfectly precise shift implementation in Section 3.5. All the other cells are instrumented at least as precisely as GLIFT

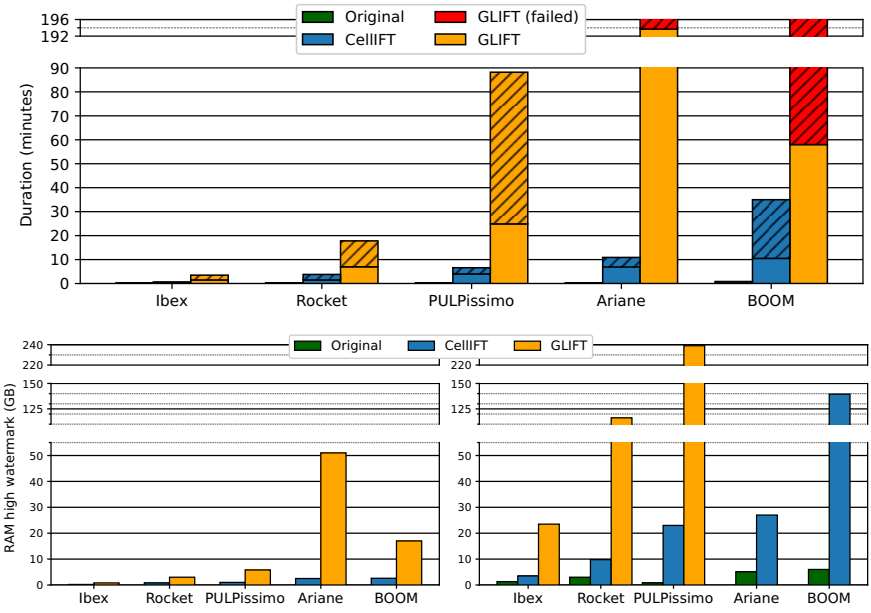

FIGURE 3.9: Top: duration of Yosys instrumentation and Verilator synthesis. Hashed rectangles represent synthesis. Bottom: RAM usage of each mechanism on each design. Left: instrumentation. Right: synthesis. Note: vertical axes are broken.

and RTLIFT. Comparison cells are more precise with CellIFT compared to GLIFT. Since these cells can have large taint fanouts, this precision improves overall taint results significantly, which is crucial in practical scenarios as we will show in Section 3.8. Superior scalability of CellIFT is made evident by its steadily low runtime for any cell width.

3.7.2 *Instrumentation*

CONFIGURATIONS. We built simple SoCs for Ibex [136] (in its default *Small* configuration) and Ariane [74] (with 4-way associative 8 kB instruction and data L1 caches) by adding memory models and protocol adapters at the design top levels (caches remain untouched), and inserted memory models in PULPissimo (Hack@DAC'18 version) in place of the L2 SRAM.

We used the Rocket chip generator to create a SoC to interface with the Rocket core [137] and the BOOM core [155] with reduced cache sizes. These configurations allow us to run standard software on these designs. We additionally replaced PULPissimo's technology-dependent oscillator with a model. Code and data are preloaded into the memory models prior to any measurement.

PERFORMANCE. We attempt to instrument and synthesize each design with both CellIFT and GLIFT, and report the wall clock durations and the resident memory consumption in Figure 3.9. Verilator failed to synthesize Ariane (out of memory) and BOOM (timeout after 96 hours) instrumented with GLIFT due to the excessive complexity of the GLIFT instrumentation. While piecewise instrumentation of Ariane by GLIFT at a greater engineering cost could lead to an eventually successful gate-level instrumentation, these results make the limitations of gate-level instrumentation apparent.

3.7.3 *Benchmarks*

We run a standard series of RISC-V benchmarks [156] on each design to assess the performance and precision of CellIFT. We run the single-threaded benchmarks, where we skip the *pmp* benchmark, which is not supported by Ibex in its default configuration. We taint some relevant part of each benchmark: the first data element in *median*, *mm*, *multiply*, *qsort*, *rsort* and *spmv*, the first instruction in *dhrystone*, and the number of discs in *towers*. Because running benchmarks on a simulated RTL takes very long, we resized the benchmarks' inputs so that each experiment point finishes under 30 minutes. This translated to simulating 4 M cycles for Ibex, 200 k cycles for PULPissimo, and 40 k cycles for Ariane, Rocket and BOOM. As we will show in Section 3.8 simulating this number of cycles is enough to enable many interesting scenarios.

Figure 3.10 provides details on the cell composition of the evaluated designs before instrumentation, and after instrumentation by each mechanism. For readability, the size of the cells is not indicated.

Figure 3.11 shows the performance results on all designs and the number of tainted stateful elements (i.e., precision) on Ibex. Since the performance variation between the benchmarks is small, we only indicated the average and standard deviation between the benchmarks. CellIFT is significantly faster than GLIFT in simulation. Regarding precision, some benchmarks such as *multiply* taint the control flow in Ibex, resulting in abundant taint-

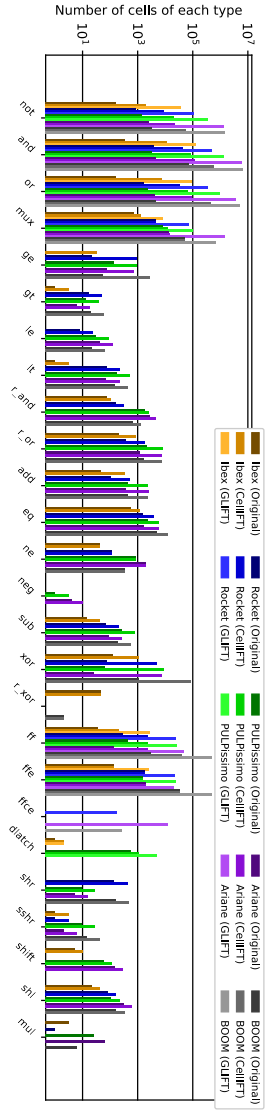

FIGURE 3.10: Cell composition for each design. The *r_* prefix denotes reduction cells (logic cells with multiple input bits but a single output bit. Note that the Y axis is logarithmic.)

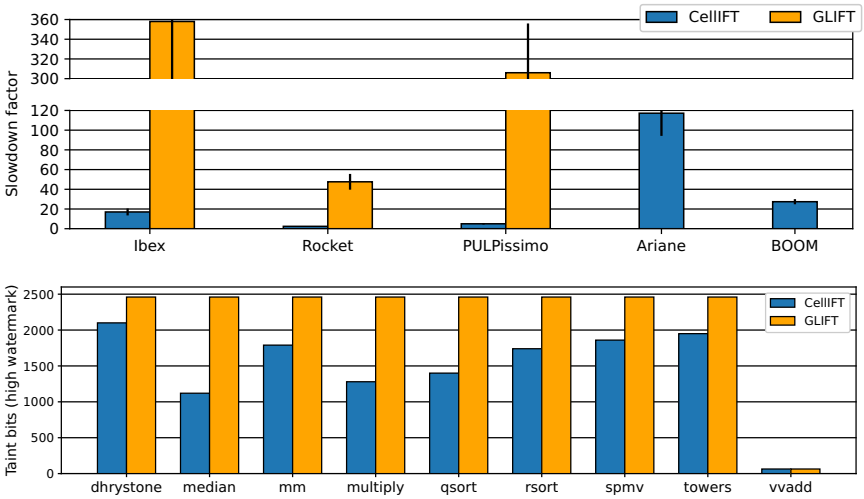

FIGURE 3.11: Top: average wall clock slowdown of the RISC-V benchmarks of each design compared with the original designs. Bottom: number of tainted stateful elements on Ibex.

ing [1]. Manual investigation shows that some ALU operations influence subsequent branch predictions, which CellIFT legitimately revealed. CellIFT's precision allows exploration of different scenarios in Section 3.8 without observing any false positive.

3.7.4 *FPGA emulation*

While we optimized CellIFT's shadow logic for simulation, we show-case its flexibility by porting the five instrumented designs to an FPGA. For the FPGA flow, we use Vivado-2019-03 on a machine equipped with a Intel Xeon Gold 6146 CPU with 48 logical cores at 3.20 GHz with 196 GB of DRAM, for licensing reasons. We target a xcvu440-flga2892-3-e FPGA at 100MHz. We set a time limit of 48 hours for the synthesis.

RESULTS. We successfully port the CellIFT-instrumented designs to the target FPGA. The synthesis of GLIFT-instrumented BOOM requires 27.57 hours and Ariane times out still at an early stage. With CellIFT, it lasts respectively 5.93 and 4.57 hours to synthesize BOOM and Ariane. Figure 3.12

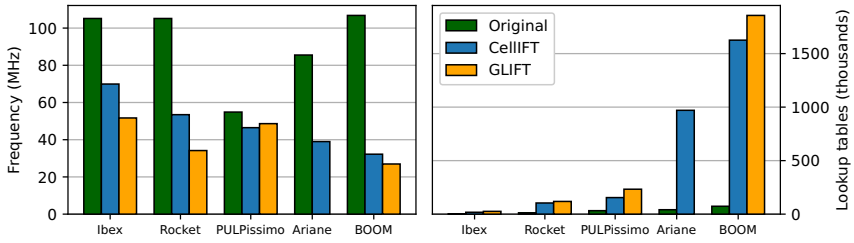

FIGURE 3.12: FPGA results. Left: Achieved FPGA frequency. Right: 6-input lookup table usage.

summarizes the achieved frequencies and resource requirements for each design. CellIFT usually shortens the critical path compared to GLIFT, providing a frequency increase of up to 39% over the state of the art. Only PULPissimo shows a slight frequency decrease of less than 5%. We obtain an FPGA acceleration over simulation of 256.2 k× and 94.1 k× for Ariane and BOOM respectively. Since CellIFT is the only dynamic IFT mechanism to instrument Ariane and to port it to an FPGA, and improves frequency and utilization compared to the state of the art, these results advocate for a cell-level (or higher) instrumentation for FPGA emulation.

3.7.5 *Summary*

We showed that CellIFT is at least as precise and significantly faster than the state of the art. It can instrument designs that were so far inaccessible to the existing solutions, without aggressive and imprecise approximations [148]. In the next section, we show some of the new opportunities offered by CellIFT thanks to its completeness, correctness, performance, and precision.

3.8 SCENARIOS

We now show how CellIFT could be used to enable new applications. While there are many applications possible with hardware dynamic IFT, here we focus on detecting different classes of hardware vulnerabilities with CellIFT.

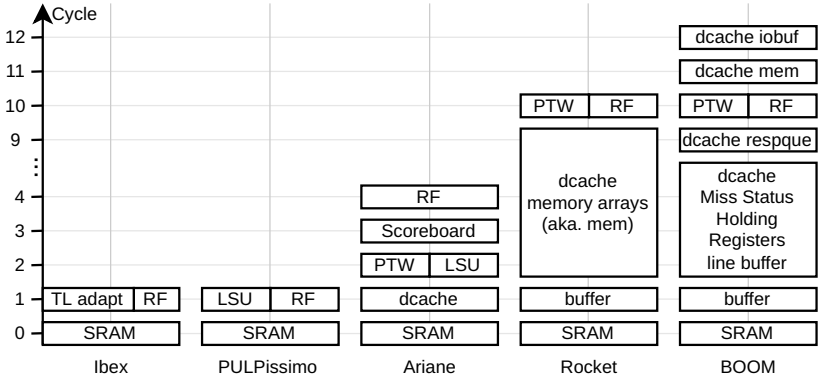

FIGURE 3.13: Affected microcomponents for a load with tainted address or tainted data. RF: Register File, TL: TileLink, LSU: Load-Store Unit, PTW: Page Table Walker.

3.8.1 *Discovering microarchitectural leakage*

Changes made to the microarchitectural state from secret-dependent data or control paths can be exploited to leak secret data [132–134]. Cache partitioning schemes attempt to mitigate such attacks [157–159], but there are other components that can leak information, such as Translation Lookaside Buffers (TLBs) [160] or branch prediction schemes [161] to name a few. CellIFT can be used to detect microarchitectural components that can leak information. This information is valuable for both attackers looking to discover a new source of leakage and for the defenders that want to protect the components that might leak sensitive information. We execute a memory load of some tainted data in memory. The taint propagation in the design gives us a cycle-accurate knowledge of when, and which bits of the design are tainted.

Figure 3.13 shows a chronological list of the components that become tainted in designs that we have instrumented with CellIFT. We make several observations. First, taints make it very easy to see when any buffer in the design contains sensitive information. Second, taints provide a convenient way of measuring latencies in a system. We detected that PULPissimo and Ibex load data in a single cycle, while the other designs require respectively 4, 10 and 10 cycles. Third, privilege level and page misses do not affect

which components are tainted in any of the considered designs. Finally, CellIFT is precise in this scenario since the control flow was legitimately never tainted.

3.8.2 *Detecting Meltdown-type vulnerabilities*

Meltdown-type vulnerabilities have been haunting CPUs since their introduction in 2018 [2, 77, 78, 117–119, 123–126] At the core, this class of vulnerabilities allows an invalid load to transiently access data from a different (higher) privilege level. CellIFT can detect such loads by checking that that they do not access data from a higher privilege.

ARIANE. Ariane features an MMU and may suffer from Meltdown, Foreshadow or MDS [2, 77, 78, 117, 118]. We craft the following test cases to trigger these three potential issues:

1. An unprivileged load of a supervisor page (triggering Meltdown [2]).
2. An unprivileged load of a supervisor page with the present bit unset (triggering Foreshadow [117]).
3. A load from an invalid address without the address bits in the page table entry (triggering MDS [77, 78, 118]).

To see which of these cases trigger the relevant problem, we taint a target (supervisor) memory page and configure the page table entry according to one of the above scenarios. Prior to measuring the malicious load, we access the tainted page to ensure it is in the cache, and possibly in the microarchitectural buffers that also have been used for it. The results from these different cases show that in none of the cases signals get tainted as a result of the malicious access, showing that Ariane does not suffer from Meltdown-type vulnerabilities in any of these specific cases. We contacted the authors who confirmed our observations that Ariane indeed does not suffer from this class of vulnerabilities.

BOOM. BOOM v2.2.3 was reported to be susceptible to Meltdown-type vulnerabilities [142]. We instrument the exact same version, and use CellIFT for detecting Meltdown-type leakages. Contrary to previous work [142], we do not use any knowledge of the design's microarchitecture. We run a simple Meltdown experiment by transiently loading privileged data in unprivileged mode and using it as an address for a subsequent load. We ensure that the privileged page table entry is present in the TLB, and we taint the privileged word. Given that the load address will be tainted, a

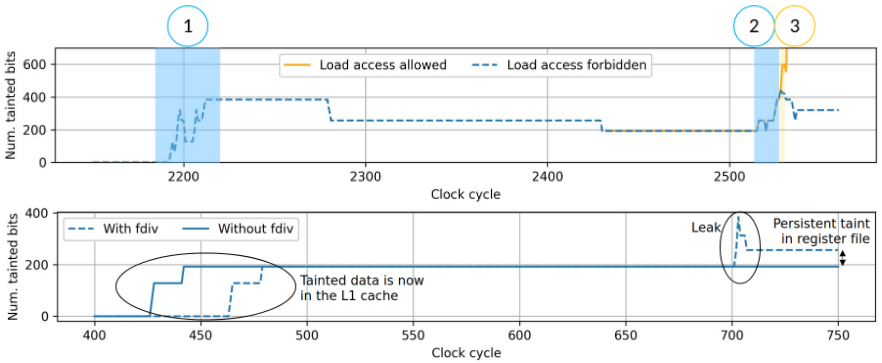

FIGURE 3.14: Tainted bits in a BOOM SoC. Top: Meltdown-type leak: (1) Tainted privileged data is brought into the L1 cache. (2) The user loads the privileged data into a physical register. (3) The user immediately attempts to load using the tainted data as an address. This load is either allowed (orange, realigned) or forbidden (blue). Bottom: Spectre-type leak (at cycle 700) occurs if the speculation window is large enough.

load will spread taint to all sets of the L1 cache among other elements. Conversely, if we see no taint in the cache, we know that the load fails, and any leakage from this load cannot be observed using a cache covert channel in a later stage.

In our experiments, we observe that when the user tries to load privileged data, (a) If the data is not in the L1 data cache, then the data is fetched into the cache's load buffer regardless of any privilege mismatch, and (b) If the data resides in the L1 data cache already, then the data is loaded into the physical register file. According to [142], these constitute a Meltdown-type leaks of L and R classes, respectively.

To assess exploitability, we first establish a benign baseline. We first do a legal load of a tainted address (orange curve in Figure 3.14, top). As predicted, this taints a large number of elements. Next, we repeat this experiment with a secret-dependent load, using the privileged (tainted) data as an address. Figure 3.14 summarizes the taint propagation during the attack, aligned on the load event. We observe that the taint propagation is blocked when the page table entry shows a privileged page (blue curve), and the level of tainting is immediately restored as it was before the malicious

load. While exploitability remained an open question in [142], this result shows that in this specific case (R1 [142]), the leakage is not exploitable because it does not leave any observable change. We repeated the exact same experiments on the newest version of BOOM (v3.0), and draw the exact same conclusions.

3.8.3 *Detecting Spectre-type vulnerabilities*

To show how CellIFT can detect Spectre in complex designs, we consider a Spectre-BCB exploit on BOOM where we taint the secret data. Our exploit consists of two steps, visible in Figure 3.14 (bottom). First, the secret is brought to the L1 data cache. Second, the data is speculatively loaded into the physical register file. We run two experiments. In the first case, the mispredicted branch relies on a simple condition (solid line in figure). In the second case, this branch condition is made more complex by adding four dependent floating-point divisions, as in a reference exploit [162], which enlarges the period of speculative execution (dashed line). Intuitively, the first experiment may not reveal Spectre-type leakage because the speculation window may be too short. We observe that the Spectre-type leakage only happens in the second experiment, consistent with this intuition. As opposed to classical techniques such as [162], CellIFT does not require a cache attack to assess whether data leaks to microarchitectural elements.

3.8.4 *Detecting architectural vulnerabilities*

We show how simple policies built on top of CellIFT can detect a large number of bugs in the PULPissimo-based faulty design used in the Hack@DAC'18 contest, some of which are not detectable by common verification flows. Detailed bug descriptions are provided in the corresponding paper [31].

ADDRESS SPACE VIOLATIONS. We build two policies that check for correct behavior in the interfaces of the memory-mapped components: (a) All components in the address map must comply to the specified boundaries. (b) No aliasing must occur; may it be in the specification or because of an implementation bug. To force these policy checks in the CellIFT-instrumented design, we perform store operations of tainted data from the CPU to the addresses before and after each address space boundary.

These policies reveal bugs 1, 2, 6, 8 and 22 which could not be expressed by either SPV or FPV [31].

REACHABILITY VIOLATIONS. Our reachability policy checks that certain instructions do not affect certain components by executing a tainted instruction. Then, after N cycles, all the components that *can* be affected by the CPU are tainted. This allows us to check the integrity and reachability of components against a specification, revealing bugs 4 and 27. We did not find bug 24 with this method, although we had expected it. Manual code inspection and simulation showed that the bug was not present in the open-source version of the faulty design [163]. Similarly, we discovered that bug 7 was inserted in a module which is never instantiated in the design.

RESET VIOLATIONS. The reset policy checks that the reset signal clears the state in the design. We check for the violations of this policy by tainting state holding elements in the design, and then applying a reset signal. This reveals which registers and buffers are not cleared during reset, and which ones are accidentally cleared. This reveals bug 5, but not bug 12 (the corresponding register has been optimized out because it was not used and not connected to a clock or reset signal) and bug 16, which lacks a clear specification.

PRIVILEGE VIOLATIONS. Our privilege policy checks that instructions execute at the right privilege level according to the specification. We check for violations by executing a tainted instruction. Because an unprivileged user requires to trap to supervisor mode to access privileged locations, these locations will be tainted some cycles after unprivileged locations. We did not see such an expected timing difference in tainting of the CSRs (Control and Status Registers), which hinted to the existence of bug 25.

SUMMARY. We showed that the ability of CellIFT to instrument a complete design efficiently and precisely enables the implementation of novel techniques to detect bugs; some of these bugs are known to be difficult or impossible to express in SPV [140] and FPV [32, 33]. For completeness, we like to mention that hardware dynamic IFT is not suitable for detecting hardcoded design parameters (e.g., bugs 15 and 19) or checking functionality (e.g., bugs 9 and 15). Some bugs such as 11, 13, 14 and 20, 30 or 31 require more advanced policies. We leave the design of systematic methods to detect such bugs as future research.

3.9 DISCUSSION

We discuss some observations we made while developing CellIFT and provide more information for its future users.

MATURITY OF THE OPEN SOURCE FLOW. During the course of the work on this Chapter, we provided feedback to the developers of Verilator [90] and sv2v [152]. None of these two tools was originally mature enough for the dynamic IFT flow to complete for all the designs under study. These tools have been improved according to our feedback and have reached a sufficient maturity to instrument the complex designs that we evaluated.

APPLICATIONS. CellIFT aims to provide scalable hardware dynamic IFT. Alone, this mechanism does not aim at discovering new microachitectural vulnerabilities: an important additional element is the scenarios running on top of CellIFT. These scenarios go beyond the verification of handcrafted information flow policies. As an example, CellIFT can be leveraged to provide a new coverage metric, enabling the development of new hardware fuzzers, which are still in their infancy [93]. Another example is enabling system-wide confidentiality and integrity policies for generic processors. We leave the exploration of these directions to future work.

INSTRUMENTATION EFFORT. CellIFT can instrument any digital design without modification, after parsing by the Yosys SystemVerilog parser [135] into intermediate cells. However, it is common not to synthesize the memories, and to instead replace them with a model. Typically, the option of ignoring memories during synthesis is available, as this is common practice when mapping a design to an FPGA or to an ASIC flow, where memories are mapped to specific components.

3.10 RELATED WORK

We briefly discuss work related to CellIFT in five areas.

HARDWARE DYNAMIC IFT. GLIFT [1] is the first hardware-level dynamic IFT mechanism and operates at gate level. Previous work discusses the scalability problems of GLIFT [146] and our results indeed show that it does not scale to the state-of-the-art RISC-V processors. Various techniques try to address the scalability issues of GLIFT by using dedicated hardware [164], static analysis [165], policy-specific IFT logic [166, 167], and trading precision with simpler IFT logic [148]. Orthogonally, CellIFT solves

the scalability problems of GLIFT by generating the IFT logic at a higher level of abstraction. It would be interesting to see how previous techniques that scale GLIFT, can be used to scale CellIFT even further.

RTLIFT [131] aims to address scalability and precision problems of GLIFT by instrumenting the HDL code directly. Unfortunately, existing HDLs are complex, and it is challenging to achieve completeness by instrumenting in HDL directly. In comparison, CellIFT achieves completeness by choosing a slightly lower-level yet generic cell abstraction, and outperforms RTLIFT as shown in Section 3.7.1. Because all cells correspond to simple HDL constructs, CellIFT provides a strong basis for any future HDL-level instrumentation.

Non-synthesizable hardware fuzzing of simulation binaries was recently proposed [93]. However, because the simulation binary's control flow depends on values in the design, software dynamic IFT will lead to critical overtainting. Another major drawback of this approach is its restriction to Verilator.

SUPPORT FOR SOFTWARE DYNAMIC IFT. Previous work proposes to provide hardware support for software dynamic IFT [168, 169]. These solutions are more lightweight but are only suited to find issues in software, not in hardware.

STATIC ANALYSIS OF HARDWARE. Static analysis in combination with model checking or verification of security properties is a common technique for improving hardware design security [32, 33, 102, 140, 170]. These techniques, however, have scalability issues due to the state explosion problem. To make static analysis tractable in certain cases, previous work introduces a new type system to an existing HDL [171] or a new HDL [150] for checking security properties. These techniques are not backward compatible to existing designs. In comparison, CellIFT provides a scalable alternative for checking security properties in unmodified RTL designs.

MODEL-BASED MELTDOWN DETECTION. IntroSpectre [142] also detected Meltdown-type leakages on BOOM. It uses a Gadget Fuzzer to generate fuzzing rounds, uses known values instead of taints, and augments BOOM with a Leakage Analyzer, an ad-hoc IFT mechanism. Whereas this mechanism may have a faster simulation runtime, CellIFT proposes a trade-off with several advantages. First, CellIFT is not built into a simulator. It is therefore compatible with any tool flow. Second, CellIFT is design-agnostic: it does not require the engineering effort and precise knowledge about the design to be deployed; CellIFT even reveals the relevant microar-

chitectural elements traversed by tainted signals. Third, CellIFT supports processed secrets (for instance, the result of an addition with a secret) and taints in the control paths, whereas the Leakage Analyzer in IntroSpectre only monitors unchanged secret data in the data path. This last feature is essential in analyzing exploitability: because no taint reached back caches or control path, we concluded that R1 [142] is not exploitable, whereas it remained an open question with IntroSpectre.

HARDWARE TESTING. Test cases are an effective tool used by practitioners to combat hardware bugs. To increase the coverage of test cases, random testing or more guided methods can be used to increase the coverage [5, 19, 93, 172]. CellIFT can complement these approaches by providing a mechanism to detect when vulnerabilities are triggered.

SPECS [30] dynamically checks security-critical state using policies derived from the ISA to combat post-silicon vulnerabilities. These policies could be leveraged by CellIFT to check the entire state during pre-silicon testing. Post-silicon, CellIFT can be used as a low-overhead alternative to existing hardware dynamic IFT techniques for enforcing security properties in the entire system [173].

3.11 CONCLUSION

We presented CellIFT, the first hardware dynamic IFT mechanism that can scale to complex state-of-the-art open-source RISC-V processors. CellIFT instruments the RTL using the cell abstraction, which is high-level enough for high performance and precision, yet generic enough to handle generic and heterogeneous digital designs without modification. To achieve this, CellIFT leverages the logical properties of cells such as monotonicity, transportability, and Translability. Our evaluation using five real RISC-V designs with various complexities shows the superior scalability, precision and performance of CellIFT. We further used CellIFT in different scenarios to show-case its effectiveness in detecting various classes of flaws and vulnerabilities. CellIFT is the first open-source dynamic IFT solution, enabling hardware security research for the wider community.

ACKNOWLEDGEMENTS

The authors would like to thank the anonymous reviewers for their valuable feedback. The work in this chapter was supported in part by a Microsoft Swiss JRC grant and by the Swiss State Secretariat for Education, Research and Innovation under contract number MB22.00057 (ERC-StG PROMISE). As to the opinions and positions in this document that the authors express or to which the authors contributed, they are those of the authors and do not represent the views of any current or previous employer, including Intel Corporation or its affiliates.

4

CASCADE: CPU FUZZING VIA NTRICATE PROGRAM GENERATION

Generating interesting test cases for CPU fuzzing is akin to generating programs that exercise unusual states inside the CPU. The performance of CPU fuzzing is heavily influenced by the quality of these programs and by the overhead of bug detection. Our analysis of existing state-of-the-art CPU fuzzers shows that they generate programs that are either overly simple or execute a small fraction of their instructions due to invalid control flows. Combined with expensive instruction-granular bug detection mechanisms, this leads to inefficient fuzzing campaigns. We present Cascade, a new approach for generating *valid* RISC-V programs of arbitrary length with highly *randomized* and *interdependent* control and data flows. Cascade relies on a new technique called asymmetric ISA pre-simulation for entangling control flows with data flows when generating programs. This entanglement results in non-termination when a program triggers a bug in the target CPU, enabling Cascade to detect a CPU bug at program granularity without introducing any runtime overhead. Our evaluation shows that long Cascade programs are more effective in exercising the CPU's internal design. Cascade achieves 28.2x to 97x more coverage than the state-of-the-art CPU fuzzers and uncovers 37 new bugs (28 new CVEs) in 5 RISC-V CPUs with varying degrees of complexity. The programs that trigger these bugs are long and intricate, impeding triaging. To address this challenge, Cascade features an automated pruning method that reduces a program to a minimal number of instructions that trigger the bug.

4.1 INTRODUCTION

With the increasing popularity of open-source RISC-V CPUs and the high cost of formal verification, CPU fuzzing is gaining momentum [4–10]. The effectiveness of CPU fuzzing strongly depends on the quality of the test cases and efficient bug detection. State-of-the-art CPU fuzzers fail at both: they execute only a small fraction of the intended instructions per test case,

and rely on incomplete or expensive instruction-granular bug detection mechanisms at runtime.

This Chapter presents Cascade, a new CPU fuzzer that explicitly generates *valid* RISC-V programs of arbitrary length with highly randomized and interdependent data and control flows. Executing many instructions per program enables Cascade to efficiently trigger bugs. When a bug is triggered, the interdependence of the data and control flows results in program non-termination, enabling Cascade to detect bugs at program granularity without any runtime overhead. Our evaluation shows that Cascade achieves significantly more coverage than the state-of-the-art fuzzers and discovers a large number of new bugs in RISC-V CPUs of various complexities.

PROGRAMS FOR CPU FUZZING. To investigate properties of the programs generated by state-of-the-art CPU fuzzers [4, 5, 12], we defined two metrics measuring 1) *completion*: the proportion of the instructions in a program that actually execute, 2) *prevalence*: the overhead due to non-randomized initial and final sequences. We find that for each generated program, only a small fraction of the instructions gets a chance to actually execute on the target CPU and that most executed instructions perform non-randomized initialization. These findings hint at low instruction throughput in existing state-of-the-art CPU fuzzers. Furthermore, detecting a bug triggered by these programs introduces its own set of challenges.

BUG DETECTION. Generally, the most obvious way to check whether a program triggered a bug is by checking the CPU's architectural state (i.e., registers and memory) instruction by instruction against a golden model [4, 6–9]. Apart from performance implications, this approach has practical limitations; as an example, the registers are not always easy to identify in a given CPU's RTL design. In particular, out-of-order CPUs may keep multiple versions of the registers at the same time and identifying the correct ones for monitoring introduces a non-trivial effort when porting a fuzzer to a new CPU [75]. It is also possible to force termination by handling uncontrolled exceptions in the programs [5]. While this technique reduces the problem to only checking the architectural state at the end, it can miss bugs that happen in the middle of the program. An ideal solution generates programs that execute to completion without the need for checking the state during program execution

CASCADE. We rely on the idea that finding a bug-triggering program in a CPU is equivalent to finding a program for which the CPU behaves incorrectly. We propose a new fuzzer called Cascade that constructs complex

and random RISC-V programs that exert the CPU's architectural and microarchitectural features without requiring time-consuming and error-prone expert effort specific to each CPU. The programs generated by Cascade mix a highly randomized data flow with the control flow, while steering the control flow at all times. To efficiently predict some necessary register values, we introduce the novel notion of asymmetric ISA pre-simulation, where instead of using the Instruction Set Architecture (ISA) simulator to compare the CPU under test with a golden reference model, we use it to construct programs with valid and predictable architectural control flows. Constructing valid programs with highly interdependent control and data flows provides us with three interesting properties. First, the programs can be long, which significantly boost fuzzing performance. Second, with highly randomized data and control flows, we explore unusual operand values and control flows. Lastly, the highly entangled data and control flows enables the non-pervasive detection of bugs amidst long programs by transforming bug expressions into program non-terminations, enabling Cascade to detect bugs without any runtime overhead.

The programs generated by Cascade exercise a rich set of functionalities provided by the RISC-V ISA; they support exceptions without (necessarily) causing termination, data flow-dependent privilege transitions, complex FPU (Floating-Point Unit) operations, and operations under randomized Control and Status Registers (CSRs) exploring different operational states of the CPU. We evaluate Cascade on 6 real-world RISC-V CPUs of different complexities and ISA extensions: VexRiscv, PicoRV32, Kronos, CVA6, Rocket and BOOM. Compared to the state-of-the-art fuzzers such as TheHuzz and DifuzzRTL, Cascade achieves the same coverage 28.2 and 97 times faster, respectively. Cascade discovers 37 new bugs (29 new CVEs) in 5 of these 6 designs which is more than all the state-of-the-art CPU fuzzers combined [4, 5, 10, 12, 13]. We additionally found a critical bug in the popular Yosys synthesizer that results in a wrong netlist.

AUTOMATED PROGRAM REDUCTION. Cascade-generated programs that trigger these new bugs can be long and highly complex, making the analysis of the non-termination intractable by humans. To tackle this challenge, Cascade relies on a new automated program reduction technique that creates minor in-place modifications to a bug-triggering program. These modifications iteratively reduce the program to a minimal bug-triggering form while preserving the sufficient bug-triggering CPU state. Using this technique, we find that these bugs result in hangs, wrong values and exceptions, decode issues and quantifiable performance counter inaccuracies. Furthermore,

we find that Cascade's program reduction is significantly helpful when reporting the bugs to the respective CPU maintainers.

CONTRIBUTIONS. Our contributions are as follows:

- We design and implement Cascade for generating valid RISC-V programs with highly complex and entangled data and control flows for finding CPU bugs. Cascade correct-by-construction programs achieve high fuzzing performance without introducing any overhead for bug detection.
- We design and implement a new analysis method to efficiently reduce Cascade-generated programs to a minimal form, while preserving the bug-triggering behavior.
- We evaluate the performance of Cascade in terms of speed and coverage and compare it with state-of-the-art CPU fuzzers [4, 5, 12]. We report a total of 37 new bugs found in 5 of the 6 considered RISC-V CPU designs. We further report a bug in the popular open-source Yosys synthesizer.

OPEN SOURCING. For the benefit of the research and CPU design and testing communities, we publish the source code and experiments of Cascade at this URL:
`https://comsec.ethz.ch/cascade`.

4.2 BACKGROUND

In this section, we provide background on formal verification, fuzzing software and hardware, and finally on RISC-V.

4.2.1 *Formal verification of hardware*

Assertion-based formal verification aims to prove that a hardware design satisfies certain properties, for all possible input values that it may receive and for infinite depth in time. Usually, formal verification engineers manually write properties that target specific verification goals, often taking design-specific knowledge into account. Tools for automated property generation either generate a certain kind of properties, like information flow properties [174], require a (semi-)formal model of the specification [175, 176], or use novel languages [177]. Furthermore, exhaustively verifying

complex properties on real world designs does not always scale and requires semi-manual abstraction techniques such as black-boxing or initial value abstraction [178, 179]. Given the high computational and manual cost of formal verification, alternative fuzzing techniques are starting to gain popularity.

4.2.2 *Software fuzzing*

Fuzzing consists in applying random inputs to a unit under test and observing whether the unit behaves as expected and whether the output is correct. The goal is to find unknown bugs by covering as much state space as possible by iteratively mutating the input data. While fuzzing usually does not provide formal guarantees of correctness, it has been shown to be a very effective technique to find bugs [180].

Every fuzzer needs a strategy to *generate test cases* and to *detect bugs* when they happen. Software fuzzers may rely on some form of coverage feedback [181–184], static or dynamic taint analysis [185–188] or grammars [189–194] to incrementally find test cases that better exercise the software's functionality. Software fuzzers mainly rely on two techniques to detect when bugs are triggered: crashes [181] and sanitizers [195]. Sanitziers provide, for example, address related checks like buffer overflows [195], checks for undefined behavior such as division by zero [196] or floating-point numerical issues [197].

4.2.3 *Hardware fuzzing*

Due to the high cost of formal verification, CPUs are an interesting target for fuzzing. Hardware fuzzers for CPUs differ from software fuzzers on both aspects of test-case generation and bug detection. While software fuzzers often generate random input data streams with little or no input format considerations, hardware fuzzers need to prioritize generation of inputs that follow certain protocols, like bus or ISA specifications, in order to be effective [93]. Inspired by software fuzzing, state-of-the-art hardware fuzzers generate new test cases by generating random instruction sequences and mutating the test case [4–10, 12, 13]. Every new CPU fuzzer finds new bugs, but often fewer [4–10].

Since processors hang only in rare cases, crash detection is not sufficient for bug detection, and sanitizers are currently limited to handwritten SystemVerilog assertions [12]. Therefore, hardware fuzzing needs new methods for bug detection. Most hardware fuzzers apply differential fuzzing by comparing register values of the CPU under test with the results from a purely software-based Instruction Set Simulator (ISS) that serves as a golden model [5]. Comparing the result of every instruction is expensive in terms of runtime, and only feasible for simple CPUs where a direct mapping from RTL design signals to registers is possible. Some CPU fuzzers [5] dump the register values via storage instructions at the end of a test case and compare the results with the ones from the ISS. Such a comparison is only possible if a test case completes and intermediate deviations between the ISS and the CPU under test are propagated through time until the end of the test case.

4.2.4 *RISC-V*

RISC-V [198] is a free and open ISA, consisting of an unprivileged and a privileged specification. RISC-V targets a large diversity of CPUs, and therefore provides a set of options. First, CPUs may comply with the 32-bit or the 64-bit specification, which share most features. Second, CPUs may implement ISA extensions. In addition to the base ISA, common extensions are F (floating-point), D (double-precision support), M (integer multiplication and division), A (atomic operations), and C (compressed instructions). Compared with other established ISAs, RISC-V ISA features a small number of instructions. In particular, RISC-V requires two instructions to load an immediate 32-bit value into a register (`lui` followed `addi`). Furthermore, the only operations that influence the program control flow are `jal` (direct jump), `jalr` (indirect jump), branches, exceptions and privilege level changes. Note that in RISC-V, the targets of branches are immediates. In the case of self-modifying code, the `fence.i` instruction must be executed before executing newly-stored instructions.

RISC-V, in its common implementation in open source CPUs, supports up to three privilege levels which are machine mode (M), supervisor mode (S) and user mode (U), in decreasing order of privilege. The M mode is the only mandatory privilege level. Upward privilege transitions are done through interrupts or exceptions, and downward transitions happen through specific instructions (`mret` and `sret`). Depending on the target privilege level, the architectural fetch address following an exception is

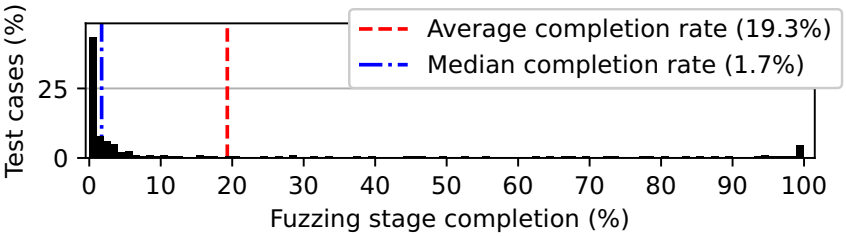

FIGURE 4.1: Completion rates of DifuzzRTL executions.

pre-set in the `mtvec` or `stvec` Control and Status Registers (CSRs). CSRs are further used to configure how the CPU should operate in certain conditions, such as whether exceptions should be delegated to another privilege level, or if the floating-point unit is enabled. Spike [199] is a widely used open-source ISS for RISC-V.

4.3 MOTIVATION AND CHALLENGES

In this section, we first analyze important aspects of the recently-published CPU fuzzers DifuzzRTL [5] and TheHuzz [4]. From these observations, we describe challenges which will guide the design of Cascade.

4.3.1 *Observations*

We collected 500 CPU inputs generated by DifuzzRTL [5] for the Rocket core to understand key aspects of test cases generated by a state-of-the-art CPU fuzzer. We also analyze the test cases generated by TheHuzz [4] based on the description in their paper since TheHuzz is not open source at the time of this writing.

COMPLETION. Figure 4.1 shows the percentage of the fuzzing instructions that execute at least once in each test case produced by DifuzzRTL. We measure completion on the ISS, assumed bug-free. We observe that these programs do not generally complete the execution of their fuzzing sections, mostly because of the difficulty to predict intermediate values that affect the control flow, such as operands of branch instructions. Simi-

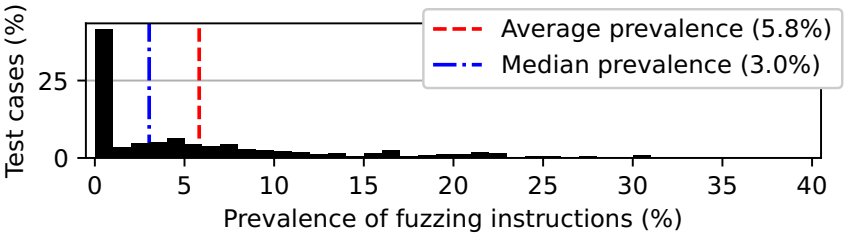

FIGURE 4.2: Prevalence of fuzzing instructions in DifuzzRTL.

larly, TheHuzz limits its test cases to 20 instructions, (10 of the first being non-control-flow instructions), again, because it is difficult to control the behavior of control-flow instructions when fuzzing.

Observation 1. Completion: CPU fuzzers struggle with fully executing their test cases.

PREVALENCE. In the programs generated by DifuzzRTL, we separate the instructions in two categories: *overhead* and *fuzzing* instructions. *Overhead* instructions are generic, non-randomized instructions such as setup routines or hard-coded exception handlers, while *fuzzing* instructions are the ones actually randomized. We define *prevalence* as the proportion of executed instructions that are *fuzzing* instructions. Figure 4.2 shows the prevalence in the programs generated by DifuzzRTL. Strikingly, only a small proportion of the executed instructions are fuzzing instructions. Similarly, the very short fuzzing sequences of TheHuzz are preceded by an overwhelming amount of configuration (overhead) instructions [4, 156].

Observation 2. Prevalence: most executed instructions correspond to overhead and are not fuzzing.

CONCLUSION. The existing coverage-guided approaches are insufficient. They produce malformed non-completing programs dominated by overhead instructions.

4.3.2 *Overview of challenges*

Our analysis suggests that we might significantly improve fuzzing results by constructing programs that address these limitations. We provide an overview of the challenges based on our previous observations and how we address them in the rest of this Chapter. The first challenge concerns completion and prevalence of the generated programs.

> **Challenge 1.** How to generate programs that complete and have a high fuzzing prevalence?

Section 4.4 discusses a new design for program construction. Longer programs have a higher prevalence, but only if they complete. We propose to build *valid* programs that are expected to complete, by pre-defining a control flow ahead of execution. This is in contrast with existing CPU fuzzers that rely on mutation-based test-case generation strategies. The result of this step is a set of *intermediate* programs that have instructions with complex data flows, but control flows that are not dependent on the (complex) data flows. Our next challenge is to make the control flows dependent on the complex data flows while ensuring completion. Entangling data flows into control flows has two major benefits. First, it helps finding bugs related to (speculative) control flows. Second, it enables transforming a data-flow bug symptom into a program non-termination, effectively providing a non-pervasive design-agnostic way of detecting data-flow bugs in arbitrarily long and complex programs without any runtime overhead.

> **Challenge 2.** How to generate valid programs with a high degree of dependence between data and control flows?

Section 4.5 proposes a novel method for efficiently constructing a complex data-dependent control flow for test cases, called asymmetric ISA pre-simulation. Asymmetric ISA pre-simulation mixes a highly randomized data flow with the control flow of the intermediate programs. This method is based on the new insight that instead of using an ISS for differential fuzzing, we can use it for generating a *valid* program. Repeated usage of the ISS for the ultimate program generation, however, would introduce a large performance penalty. We design a new scheme that allows us to reduce the number of ISS calls to only one per generated program.

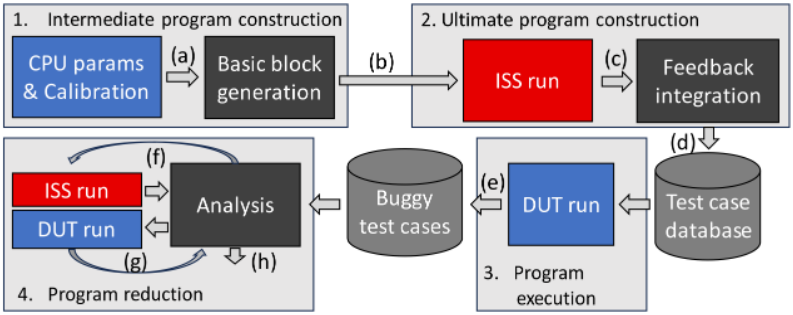

FIGURE 4.3: Overview of Cascade.

Our prototype fuzzer implementation of the ideas presented in Sections 4.4 and 4.5, called Cascade, generates programs that trigger a large number of new bugs, more than any CPU fuzzer so far. The programs leading to these bugs, however, may be long and complex by construction, to an extent that makes it inconceivable to interpret them manually from logic waveforms, like it may have been done so far. This leads us to our last challenge.

Challenge 3. How to extract and interpret the bug from a complex program?

Section 4.6 discusses a new algorithm for iterative program reduction to find a minimal number of instructions inside program triggering CPU bugs. We successfully applied it for all the bugs found by Cascade, which significantly helped us when reporting the issues to the respective CPU maintainers.

4.4 DESIGN

We outline Cascade, before explaining the intermediate program construction. In Section 4.5, we explain how Cascade uses ISS feedback to entangle data and control flow.

4.4.1 *Cascade overview*

Figure 5.5 provides an overview of the fuzzing process and its components. Cascade proceeds in four steps: (1) intermediate program construction, (2) ultimate program construction, (3) program execution and (4) program reduction. The program construction (steps 1 and 2) is decoupled from the other steps, so the same program can be executed on multiple versions of the same CPU, or on CPUs with compatible extensions.

INTERMEDIATE PROGRAM CONSTRUCTION. Cascade takes as an input the supported ISA extensions, addresses for dumping registers and stopping the RTL simulation, and descriptions of known bugs to circumvent, and starts with a brief calibration stage (taking less than a second) to evaluate some CPU parameters, such as supported CSR bits (a). Generating programs is then a parallel task. Cascade first generates the *intermediate* program as a sequence of basic blocks, where control flow is isolated from the data flow (b).

ULTIMATE PROGRAM CONSTRUCTION. An ISS then executes the whole *intermediate* program once to collect the data-flow dependent values of registers (c). Cascade uses this information to entangle the data flow with the control flow, as explained in Section 4.5 to produce the *ultimate* program (d). It is defined as a triple (ELF file, descriptor, and the expected final register values (optional)), where the descriptor is a short identifier that permits re-generating the same program.

PROGRAM EXECUTION. The program execution is also a parallel task. Each ultimate program is executed independently on a simulated design under test. The output (e) is a pair (descriptor, success). The descriptor is the same as in (d). The success flag is raised if the program terminated successfully, and optionally if the dumped register values match.

PROGRAM REDUCTION. The program reduction phase takes as input a test descriptor that leads to an execution failure and produces a reduced program that preserves the buggy behavior while reducing the program complexity as described in Section 4.6. The analysis consists in iteratively reducing the program (f) and re-running it to check if the bug is still triggered (g), and to produce a minimal program that is easy to understand (h), as explained in Section 4.6.

```
1    xor       x3,x4,x9
2    csrrwi    x9,mcause,15
3    beq       x9,x4,0x8000098e
4    fld       f8,(x3)
5    feq.s     x4,f9, f8
6    jalr      x9,(x7)
```

Listing 4.1: Example basic block.

4.4.2 *Intermediate program construction*

We first explain the structure of the programs generated by Cascade. We then show how Cascade selects instructions to form basic blocks. We finally discuss the memory management mechanism for ensuring sound program construction.

High-level program structure

BASIC BLOCKS. A program is a sequence of instructions within basic blocks. Basic blocks are made of zero or more instructions that do not affect the control flow, followed by a single instruction that affects the control flow. Accordingly, Cascade constructs programs as sequences of basic blocks, where the last instruction of a basic block steers to the next basic block. Each basic block is placed at a random location in memory. The order of basic blocks' execution is not necessarily identical with their placement in memory. The memory outside of basic blocks and of their data is left uninitialized, defaulting to zeros in simulation.

INITIAL AND FINAL BASIC BLOCKS. All programs start with an initial basic block that sets up an initial state and random register values before jumping to the first fuzzing basic block, which eventually jumps to the next, and so on. The program ends with a final basic block that optionally dumps register values, and sends a signal indicating that the program's end was reached. No overhead instruction, as defined in Section 4.3, is executed between the initial and final basic blocks.

Basic block generation

CONTROL-FLOW BEHAVIORS. We call instructions that do not change the control flow of a given program *still* instructions, and others *hopping* instructions. For example, a branch can be taken (hopping) or non-taken (still). Similarly, a memory load instruction such as `lw` can succeed (still) or raise an exception (hopping). We define the instructions that may be still or hopping depending on register values, or that are unconditionally hopping but whose destination depends on register values, as *cf-ambiguous*. For example, `beq` and `lw` are *cf-ambiguous*, but `add` and illegal instructions are not. Listing 4.1 shows an example basic block, where cf-ambiguous instructions are represented in red.

PICKING INSTRUCTIONS. Instructions are grouped in categories. Some groups of instructions will behave in the same context-dependent way, for example all floating-point instructions will be conditioned by whether an FPU is present and activated. Hence, Cascade picks instructions hierarchically, by first choosing a category, and then a specific instruction. Both are chosen randomly with certain probabilities, which are varied between programs. When picking a cf-ambiguous instruction, Cascade chooses immediately whether it must be still or hopping. Cascade biases the choice of operands by granting higher probabilities to registers recently used as outputs.

In particular, Cascade supports complex FPU operations and CSR interactions. It additionally supports exceptions and privilege switches as simple hopping instructions, which can only be picked under certain architectural conditions that are generated by the mechanism explained in Section 4.5. Ultimately, the complex data flow originates from combining initial static data with a diverse stream of random instructions.

The instruction categories used in Cascade are the following: REGFSM (Register lifecycle), FPUFSM (Update the FPU state), ALU (rv32 ALU operations), ALU64 (rv64 ALU operations), MULDIV (rv32 multiplications and divisions), MULDIV64 (rv64 multiplications and divisions), AMO (rv32 atomics), AMO64 (rv64 atomics), JAL (Direct jumps), JALR (Indirect jumps), BRANCH (Branches), MEM (rv32 integer memory operations), MEM64 (rv64 integer memory operations), MEMFPU (rv32 floating memory operations), FPU (rv32 floating-point operations), FPU64 (rv64 floating-point operations), MEMFPUD (rv32 double memory operations), FPUD (rv32 double operations), FPUD64 (rv64 double operations), TVECFSM (Update trap vector), PPFSM (Update previous privileges), EPCFSM (Update trap previous PC), MEDELEG (Update exception delegation), EXCEPTION

(Trigger an exception), RDWRCSR (Read/write some CSRs), DWNPRV (Transition privileges downward), FENCES (Fences or `wfi`).

CIRCUMVENTING KNOWN BUGS. Ideally, discovered bugs should be fixed immediately. In reality, however, this may take time and effort. Due to limited human resources, some bugs may remain unfixed for a long time, polluting bug reports and potentially restrain test case continuation. Known bugs may be ignored after triaging the causes of bug reports. However, known CPU bugs can influence the control flow of Cascade programs early, shadowing the rest of the program. Furthermore, triaging is an expensive operation that must then be repeated many times for the instances of the same bug. Instead, Cascade allows circumventing known bugs by constructing programs that will not trigger them. By increasing the completion rate, Cascade is able to progress and find more bugs faster. As an example, in the Rocket core, the retired instruction performance counter ignores `ecall` and `ebreak` instructions (R1) [200]. The circumvention configures Cascade to avoid that generated programs read this counter after these instructions until it has been written again, hence it covers exactly the bug.

Note that Cascade's approach of circumventing certain bugs may result in the under-exploration of the components in which the bugs exist. Automatically circumventing bugs via automated RTL patching is an interesting future direction that can address this issue altogether.

Memory management

To produce sound programs, Cascade imposes some constraints on the memory layout of the generated programs. In particular, Cascade ensures that (a) instructions do not overlap, (b) store operations do not overwrite instructions, and (c) later transformations of the program (discussed in Sections 4.5 and 4.6) do not have unintended effects. Intuitively, the constraint (b) would prevent from detecting some potential bugs related to self-modifying code. RISC-V requires self-modifying code to use `fence.i`, which means that such bugs are limited in scope. We leave the exploration of such bugs as future work and instead focus on non-self-modifying code.

PROGRESSIVE MEMORY ALLOCATION. Cascade allocates memory on the fly for each new instruction. Whenever a hopping instruction is generated, space is allocated for the first instruction in the next basic block, at a reachable random new address that offers enough space for a new basic

block. Initially, space is allocated (at random locations) for the initial and final basic blocks, which have known upper bounds in length. Additionally, to anticipate program reduction, Cascade allocates space for a *context setter* block used for program reduction and leaves it empty (details are discussed in Section 4.6).

STRONG MEMORY ALLOCATION. The memory allocator can *strongly* allocate memory areas, i.e., forbid loads from there. Reading from a memory location could entangle its data with the data and control flows. Strong allocations prevent reading from a memory location that stores very specific parts of the program where some instructions differ between the *intermediate* and *ultimate* programs as described in Section 4.5.

MEMORY OPERATIONS. Memory stores, randomized in number and size, can only target some specific memory areas, allocated at the start of the program's creation. Memory loads can target any address, except for the strong allocations. Heuristically, we bias memory loads to target more often memory areas that have been recently written, although so far, this specific heuristic has not been critical in finding bugs.

4.5 ULTIMATE PROGRAM CONSTRUCTION

The fundamental idea behind asymmetric ISA pre-simulation is to execute an *intermediate* program on the ISS to collect feedback for constructing the *ultimate* program with a control flow that is identical but dependent on the data flow.

STEERING THE CONTROL FLOW. There are three schemes for generating an arbitrary, but controlled, control flow. The first scheme is not to control the values used by cf-ambiguous instructions such as `jalr` but to place the next basic block accordingly. This scheme is not viable because most addresses are inaccessible or already allocated. The second scheme is not to involve the random data flow and rely on direct branches or registers loaded with fixed values. This is how the control flow of the *intermediate* program is constructed. The third scheme entangles data and control flow by observing the data flow and applying an offset to a register used by a cf-ambiguous instruction. We rely on the third scheme to construct the control flow of the *ultimate* program.

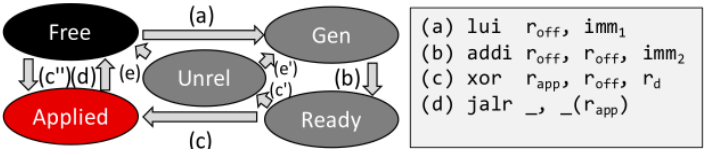

FIGURE 4.4: Life cycle of registers regarding offset construction. The instructions can be separated by other (unrelated) instructions.

4.5.1 *Offset construction and register lifecycle*

Some cf-ambiguous instructions such as indirect jumps (`jalr`) require a specific operand value `val` that Cascade intends to impose. Following the principle of dependency preservation, we let `val` depend on a randomly picked *dependent register* r_d. Since we do not want to constrain r_d's value, r_d is generated by the random data flow and its value is unknown when we pick the cf-ambiguous instruction. Hence, to calculate `val` we propose to generate an *offset register* value to be eventually combined with r_d's value. The combination of r_d and r_{off} is performed by an *offset applier* instruction, for instance `xor`, whose output is defined as the *applied register* r_{app} and holds the intended value `val`.

BRANCHES. Because *applied registers* are a somehow precious resource, Cascade does not use this method for branches. Instead, it uses the ISS feedback to obtain the operand values and selects a suitable branch opcode, depending on whether the branch should be still or hopping.

REGISTERS INVOLVED. In total, this construction involves two registers (r_d, whose value is randomized by the program's data flow, and r_{off} to offset its value). Since there is no reason to specifically reuse r_d or r_{off} as an output of the offset applier, a third register could be used as r_{app}, in accordance with the principle of maximizing the degrees of freedom.

OFFSET STATE MACHINES. This scheme implies that 1) r_{app} must be available for use when the cf-ambiguous instruction is picked, 2) r_{app}'s calculation requires that r_{off} is ready before the offset applier instruction, and 3) r_d must be available when the offset applier instruction is executed. To comply with these requirements, we maintain a simple state machine for each architectural integer register. The state machine is composed of five states: *free*, *under generation (gen)*, *ready*, *unreliable (unrel)* and *applied*, as illustrated in Figure 4.4. All registers initially start in the *free* state. Cascade

can create instruction (a) to move a *free* register to state *gen*, or (b) to move a register from *gen* to *ready*. When there is at least one register, r_{off}, in the *ready* state, then Cascade can pick an offset applier instruction. If Cascade chooses to define $r_{app} = r_{off}$, then r_{off} is moved to the *applied* state (c). Else, r_{off} is moved to the *unrel* state (c′) and r_{app} is moved to the *applied* state (c″). Once r_{app} is used by a cf-ambiguous instruction (d), it returns to the *free* state. An *unrel* register must not be used as an input to any instruction and may be overwritten by an instruction to become *gen* (e′), or by an ordinary instruction to become *free* (e).

PRE-SIMULATION. Cascade relies on an ISS to determine the value of dependent registers when encountering a cf-ambiguous instruction. So far, the state of the art [4–10] always submits the same program that is given to the CPU to the ISS. If we imitated the state of the art, the ISS should be called every time a cf-ambiguous instruction is encountered because r_{app} would be random until the correct r_{off} can be computed from r_d. The ISS could not proceed to the next basic block (or complete a correct memory operation) without running at least once per cf-ambiguous instruction.

We introduce the *asymmetric* ISA pre-simulation method to address this critical performance bottleneck by requiring a single ISS execution. The fundamental insight is to provide to the ISS not the *ultimate* program that the CPU under test will execute, but an *intermediate* program such that (a) the values of all *dependent registers* (in the constructions of *applied registers*) are identical between the two programs and (b) the programs' control flows are identical and (c) in the *intermediate* program, no applied register depends on the randomized data flow.

Concretely, the *intermediate* program differs from the intended *ultimate* program in three aspects. When generating the *intermediate* program: 1) we choose the immediates `imm1` and `imm2` to set the value of r_{off} to `val`, 2) we substitute of the offset-applying instruction with a `mv` instruction, which copies r_{off} into r_{app}, and 3) we transform non-taken branches into NOPs, because the value of the operands are not yet known. Given these transformations, the *intermediate* program complies with the aforementioned requirements.

INTERMEDIATE REGISTER STATES. This scheme for offset construction guarantees by design that at any point in the program, all *free* and *applied* registers have the same value in the *intermediate* and *ultimate* program. This invariant does not hold for registers in the *gen*, *ready* and *unrel* states. Therefore, they should never be used as the input for any instruction other

than the next instruction in the offset construction cycle, respectively steps (b) and (c) in Figure 4.4. Note that state machine transitions are a specific instruction category in Cascade.

4.5.2 *Privilege transitions*

Exceptions are a particular case of hopping instructions. They require some basic bookkeeping in the fuzzer to maintain the privilege state and delegation flags, which indicate the target privilege level active when some exception occurs.

We use one to two offset/dependent/applied register triples per exception. The first is used to populate the trap vector, either `mtvec` or `stvec`, depending on the current privilege state and whether the exception will be delegated. The second is used when an exception depends on a register value, e.g., in a misaligned memory load exception. It is populated similarly as an indirect jump or a load would be. Note that since basic blocks are generated on the fly, the value to be inserted into the trap vector is only known when the exception-triggering instruction is generated, which may be many basic blocks later; while this adds some necessary complexity to the fuzzer, it does not negatively affect the program's degrees of freedom in any way. The implementation of downward privilege transitions is in all respects comparable with exceptions.

4.6 PROGRAM REDUCTION

Cascade generates potentially long test cases, and CPU bugs are revealed by programs not terminating, thanks to the entanglement of the data and control flows. To understand the underlying CPU bug, it is necessary to reduce the programs to a minimal form, while preserving the instructions and states that trigger the CPU bug. We first show how to find the last instruction (*tail*) involved in triggering the bug, then the first (*head*). Figure 4.5 illustrates the program reduction process.

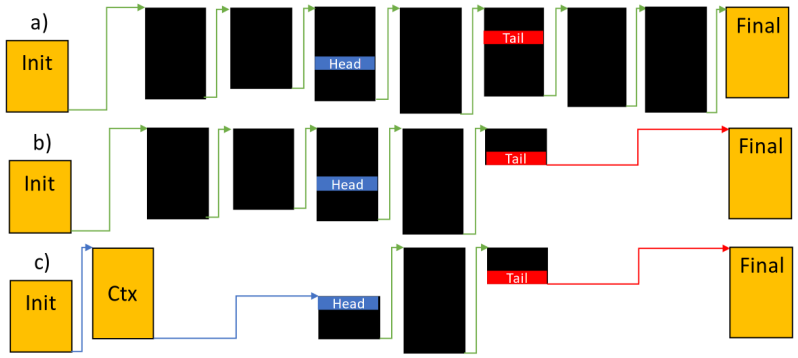

FIGURE 4.5: (a) Original, (b) tail-reduced and (c) fully reduced program. Black rectangles and arrows represent basic blocks and control flow. Ctx is the context setter block.

4.6.1 *Identifying the bug's tail*

We propose to reduce the program progressively by transforming some instructions into direct jumps that skip some of the last basic blocks and observing whether the bug is still triggered. The result is illustrated in Figure 4.5 (b).

IDENTIFYING THE TAIL BASIC BLOCK AND INSTRUCTION. Cascade finds via a binary search the last basic block which, when omitted along with its successors, erases the buggy behavior. To remove such a final sequence, Cascade replaces its predecessor's hopping instruction with a direct jump toward the final block. Cascade then searches the bug's tail instruction in the converse way.

FAILING CONTROL-FLOW INSTRUCTIONS. If the tail instruction is a hopping instruction, the algorithm above will find a tail basic block B_n, but no tail instruction. This is because for skipping the basic block B_n but not B_{n-1}, a direct jump instruction toward the final block will replace the (bug-triggering) hopping instruction of B_{n-1}, hence removing B_n erases the buggy behavior. The tail instruction being a hopping instruction is hence the necessary and sufficient condition for failing.

4.6.2 *Identifying the bug's head*

Most often, a single instruction, provided with the correct architectural context, is sufficient to trigger the bug reliably. In such cases, finding the tail instruction is enough to understand the bug. However, some bugs required a sequence of instructions (up to 2), possibly far apart, to be triggered, such as the bugs V1-V9 and V14 that we describe in Section 4.7, because they rely on a specific microarchitectural context. Hence, for these bugs, identifying the head is necessary. The result of this step is illustrated in Figure 4.5 (c).

MAINTAINING THE ARCHITECTURAL CONTEXT. Identifying the tail instruction can be done by exclusively inserting direct jump instructions in the right places iteratively, but identifying the head instruction is more challenging because removing predecessor instructions ahead may influence the architectural state. We leverage the following insight to identify the bug's head: we find the head by preserving the architectural state but simplifying the microarchitectural state. Concretely, only for this step, a *context setter* basic block is inserted, by replacing the initial block's hopping instruction. Once a candidate head basic block and instruction is chosen, the context setter uses the ISS to infer the architectural context, including for instance register values, privilege level and some performance counter values. It then loads the architectural state of the CPU, using simple instructions such as wide loads for register values, and basic instruction sequences for populating CSRs and setting the proper privilege. The detection of the bug's head is then performed similarly to the tail, following the converse algorithm where the head instruction is the first instruction that, when omitted, erases the buggy behavior.

SANDWICH INSTRUCTIONS. For all bugs found by Cascade so far, identifying the tail and head instructions has always been sufficient for understanding and reproducing the bugs. However, Cascade includes facilities to flatten the remaining instructions (in black in Figure 4.5 (c)) and removing them iteratively. Such transformations are not guaranteed to work on a given specific program, for example if the hopping instruction was an exception and some following instruction checks the exception cause. In practice, finding another program that reveals the same bug in case of failure of advanced transformations is sufficient, notably because finding new programs that trigger the same bug is fast, as we show in Section 4.7.6.

4.7 EVALUATION

In this section, we evaluate Cascade in terms of performance, program metrics, coverage, and the discovered bugs. We use microbenchmarks to quantify the performance of program construction and compare it with previous work (Section 4.7.1). We evaluate the impact of program lengths on fuzzing throughput (Section 4.7.2). We then evaluate the program metrics for Cascade (Section 4.7.3) as we did for DifuzzRTL in Section 4.3.1. We compare the coverage achieved by Cascade according to multiple coverage metrics and compare it with the state-of-the-art fuzzers that specifically target these metrics (Section 4.7.4). We then show the bug discovery efficacy of programs of different lengths (Section 4.7.5). We also describe the 37 new bugs found in 5 of the 6 evaluated RISC-V CPUs and in Yosys, evaluate the time to detection (Section 4.7.6) . Finally, we evaluate the performance of program reduction (Section 4.7.7).

EVALUATION SETTING. The performance results were obtained on a machine equipped with two AMD EPYC 7H12 processors at 2.6 GHz containing 256 logical cores and 1 TB of DRAM. We use Verilator 5.005 to simulate the CPUs' RTL. As an ISS, we use spike (version 1.1.1-dev, commit `fcbdbe79`). We use a recent version of a widely-used commercial simulator to collect simulator-based coverage similar to previous work [4]. We use the most recent versions of each CPU, where bugs are fixed or circumvented by Cascade. Notably, we tested Cascade on a variety of CPU complexities, from a simple minimal 32-bit integer core (PicoRV32) to an application-class Linux-capable out-of-order core (BOOM). We implemented Cascade as 6 k lines of Python code.

Our testbench consists of 6 RISC-V CPUs of varying complexities and ISA extensions. VexRiscv [201] (`e142e12`, Linux, AHB-L, FPU, rv32imfd) is a highly parametrable CPU written in SpinalHDL, supporting Linux. PicoRV32 [202] (`f00a88c`, Default, rv32im) is a size-optimized CPU written in Verilog (IEEE 1364-2005). Kronos [203] (`13678d4`, Default, rv32i) is a CPU optimized for FPGA applications, written in SystemVerilog (IEEE 1800-2017). CVA6 [74] (`109f9e9`, 4-way 8 kB caches, rv64imafd) is an application-class CPU with Linux support supported by the OpenHW Group organization, and written in SystemVerilog (IEEE 1800-2017). Rocket [137] (`004297b`, BigCore, rv64imafd) is a reference application-class CPU with Linux support, maintained by the Chips Alliance, written in Chisel. BOOM [75] (`004297b`, MediumBoom, rv64imafd) is an out-of-order application-class CPU with Linux support, maintained by the Chips Alliance, written in Chisel.

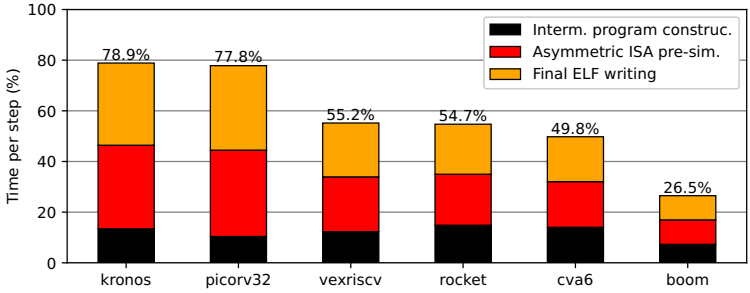

FIGURE 4.6: Performance of program construction. The rest of the time is spent in the RTL simulation.

BASELINES. We compare with the existing open-source generic CPU fuzzers, which are RFUZZ [12] and DifuzzRTL [5]. Despite the claim made in the original paper [4], TheHuzz is not open source at the time of this writing, and its authors were reluctant to answer any question or share their code over a period of a year, hence we rely on the results reported in their paper [4]. We re-implemented RFUZZ to support Verilog, and relied on the Docker image provided by DifuzzRTL [204].

4.7.1 *Program generation performance*

To quantify the performance of program construction, we measure the amount of time spent in intermediate program construction, asymmetric ISA pre-simulation and RTL simulation, for each CPU under test, over 24 hours of fuzzing. Figure 4.6 shows the results.

RESULTS. While by construction, the duration of the instruction generation and asymmetric ISA pre-simulation is identical across designs, the RTL simulation time increases with the complexity of the design, hence the proportion of time spent generating programs largely decreases with the complexity of the designs. When generating the programs in real time, the program generation takes between 26.5% and 78.9% of the total fuzzing time.

INPUT REUSE. To make Cascade's evaluation as pessimistic as possible, we systematically dynamically generate new inputs by default. Note that

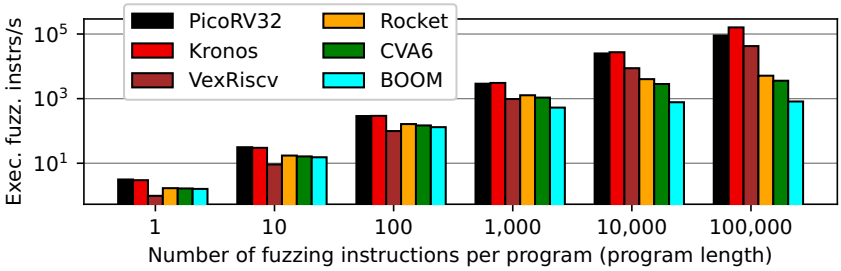

FIGURE 4.7: CPU-under-test execution throughput given program length. Note the logarithmic Y axis.

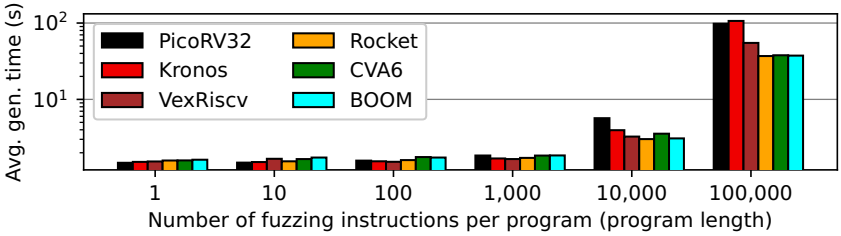

FIGURE 4.8: Program generation performance given program length. Note the logarithmic Y axis.

by construction, Cascade's inputs are reusable across designs that share compatible ISA extensions, and across CPU generations. Hence the input generation is, in fact, a one-time cost that can further be amortized.

RUNTIME OVERHEAD. DifuzzRTL reports a runtime overhead of 6.1% to 6.9% for control register coverage, and 97% for multiplexer select coverage. TheHuzz reports a runtime overhead of 71% These slowdowns do not include input generation. Cascade incurs no runtime overhead by design.

4.7.2 *Throughput of long programs*

To understand the throughput boost provided by long programs, we measure the number of fuzzing instructions executed per second when con-

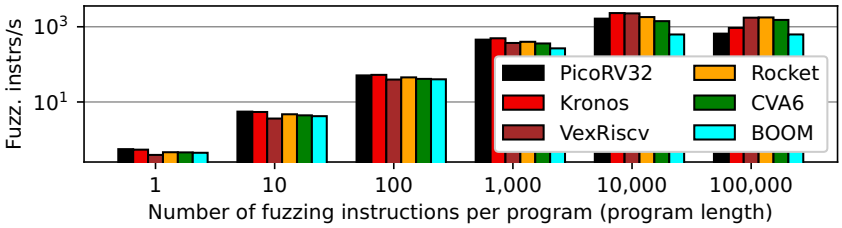

FIGURE 4.9: Effective fuzzing throughput given program length. Note the logarithmic Y axis.

trolling the number of instructions per program generated by Cascade, including program generation time. The experiment was run for 1 core-hour per bar. Figure 4.7 shows the throughputs of program execution, while Figure 4.8 details the average duration of a single program generation. Constructing very long programs requires managing a wider memory range, resulting in longer program generation times. Consequently, the overhead of generating longer programs counteracts the improvements in terms of fuzzing throughput when programs become too large.

To find the sweet spot for the length of programs, Figure 4.9 shows that the effective fuzzing throughput when considering both the raw fuzzing throughput and the overhead of program generation at the same time. These results show that programs of 10 k instructions generally provide the best effective fuzzing throughput, and that the fuzzing throughput is improved by three orders of magnitude between single-instruction and 10 k-instruction programs.

4.7.3 *Program metrics*

As part of our initial observations in Section 4.3, we exposed the completion and prevalence program properties. We evaluate these metrics for Cascade. We additionally evaluate the length of dependency chains between fuzzing instructions, which matters for non-termination when a bug is triggered.

PREVALENCE AND COMPLETION FOR CASCADE. Since Cascade always completes except when finding a CPU bug, we observe the expected completion rate of 100%. Figure 4.10 shows the prevalence of fuzzing instructions

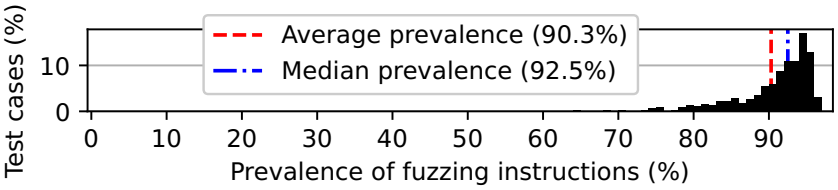

FIGURE 4.10: Prevalence of fuzzing instructions for Cascade.

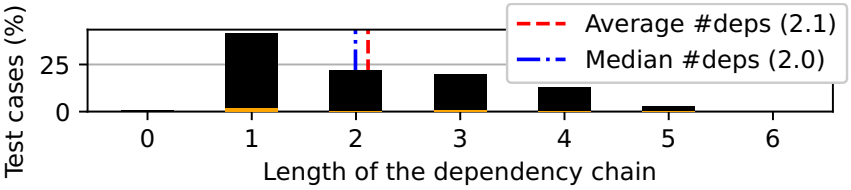

FIGURE 4.11: Length of the dependency chains for DifuzzRTL. Control-flow instructions are represented in orange.

for Cascade. The high prevalence is because programs are relatively long and fully randomized except the initial and final basic blocks.

DEPENDENCIES. We analyze dependency chains between fuzzing instructions. On top of entangling the data flow with the control flow to force non-termination when a bug is triggered, these dependencies can further exercise corner cases inside a CPU's design. For this analysis, we calculate the length of instruction dependency chains. Each register starts with a dependency number of -1, which gets reset when it is a destination of a CSR read or of an instruction that only takes immediates. We calculate an instruction's dependency number as the maximum of the dependency numbers of the source registers, plus one, which also becomes the new dependency number of the destination register.

Figure 4.11 and Figure 4.12 show the distribution of the length of the dependency chains for DifuzzRTL and Cascade, respectively. The fuzzing instructions with zero dependencies here are instructions in the form of `xor rd, rd, rd`. The programs generated by DifuzzRTL have very few interdependencies. Similarly, TheHuzz does not explicitly generate or favor programs with dependencies [4], hence we expect even lower numbers.

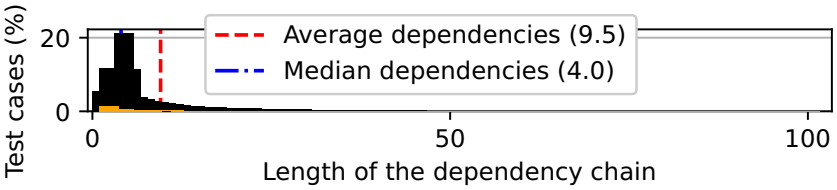

FIGURE 4.12: Length of the dependency chains for Cascade. Control-flow instructions are represented in orange. Fuzzing instructions depended on up to 270 other fuzzing instructions.

In contrast, Cascade generates programs with longer dependency chains, which could further be improved if needed by lowering the probabilities of picking dependency-resetting instructions such as CSR reads.

4.7.4 *Coverage evaluation*

We show that Cascade is faster in increasing coverage compared to the state-of-the-art fuzzers with their own coverage metrics. We consider the open-source coverage metrics used for fuzzing (i.e., multiplexer select and control register coverage). To compare with the results reported by TheHuzz [4], we additionally consider the coverage metrics provided by the commercial simulator (branches, conditions, expressions, FSM states and transitions, statements, and toggles).

Control register coverage (DifuzzRTL)

We first compare the control register coverage of Cascade with the one achieved by DifuzzRTL, which explicitly aims at maximizing this coverage. DifuzzRTL supports legacy versions of Rocket and BOOM. Running Cascade on this BOOM version leads to quasi-systematic timeouts, revealing old bugs in the obsolete BOOM RTL. Hence, we executed all test cases on the legacy Rocket core provided in the Docker image [204], which was already exempt of unexpected bugs.

RESULTS. Figure 4.13 shows the control register coverage achieved by Cascade and DifuzzRTL. Cascade achieves more coverage than DifuzzRTL, in a shorter time. In particular, Cascade achieves the same coverage in 30

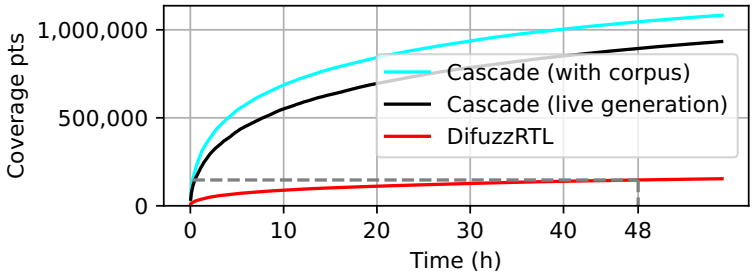

FIGURE 4.13: Achieved control register coverage.

minutes (15 minutes when using a pre-generated corpus) as DifuzzRTL in 48 hours, which is a speedup of 97x (respectively 186x).

Multiplexer select coverage (RFUZZ)

We now compare the multiplexer select coverage achieved by Cascade with the one achieved by RFUZZ. We adapted RFUZZ as a sequence of Yosys [205] passes to support Verilog, more general than FIRRTL [206], and ensured that the results match with the original open-source implementation. We will open-source the instrumentation code to foster further research. We execute Cascade on the RFUZZ-instrumented versions of the designs as well. We execute the RFUZZ experiments until completion, i.e., as implemented in the original RFUZZ fuzzer, when all input sequences in the adherence to the corpus cease to increase coverage. We had to exclude CVA6 because of the Yosys bug found by Cascade.

RESULTS. Figure 4.14 shows the multiplexer select coverage achieved by Cascade and RFUZZ. First, in terms of coverage, for the two simpler designs, RFUZZ is a bit slower than Cascade (note that in this experiment, Cascade also suffers from the runtime overhead due to the instrumentation), but eventually achieves a superior coverage. Since RFUZZ is a lower-level fuzzer, it is expected to be able to toggle some more multiplexer signals eventually, for example by fuzzing the bus protocol, whereas an ISA-level fuzzer like Cascade and many others [4–10] will not explore these state machines. However, as soon as CPUs become more complex, RFUZZ is unable to make any progress.

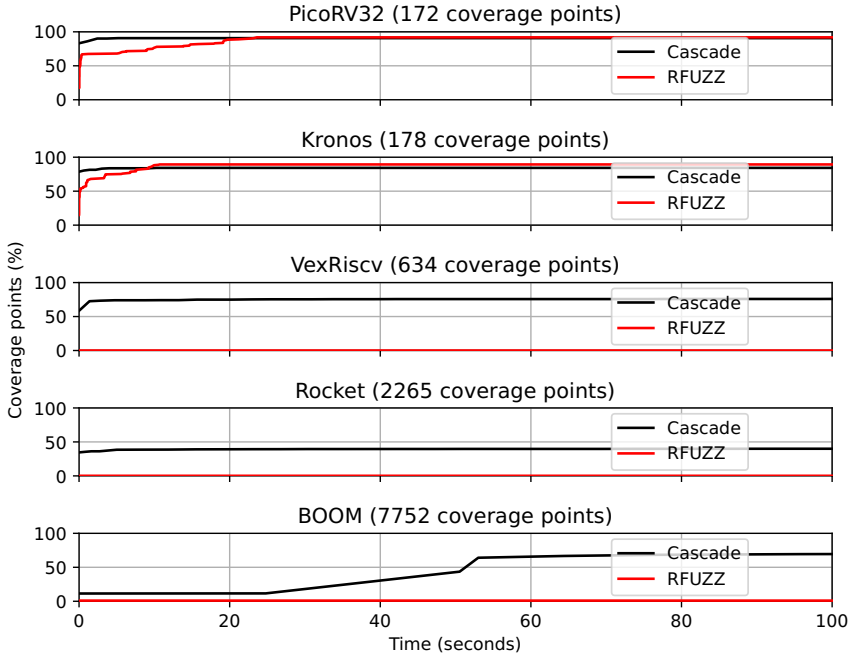

FIGURE 4.14: Multiplexer select coverage achieved by Cascade.

Second, regarding delay, the order of magnitude to saturate coverage for Cascade is approximately 100 seconds, which corresponds to its order of magnitude for finding bugs as we show later. This suggests that, although a naive RFUZZ implementation seems inadequate to finding bugs in CPUs, especially when they are complex, its coverage metric seems relevant to evaluating the quality of a fuzzing campaign.

Simulator-based coverage (TheHuzz)

We compare the simulator-based coverage achieved by Cascade with the one reported by TheHuzz. We reproduce the experiment from TheHuzz that led to their Figure 5 [4]. We compare the simulator coverages of Cascade and DifuzzRTL, and use DifuzzRTL as a pivot to compare with TheHuzz. Note that to fit the methodology of TheHuzz, both DifuzzRTL and Cascade rely on pre-generated corpuses in this experiment.

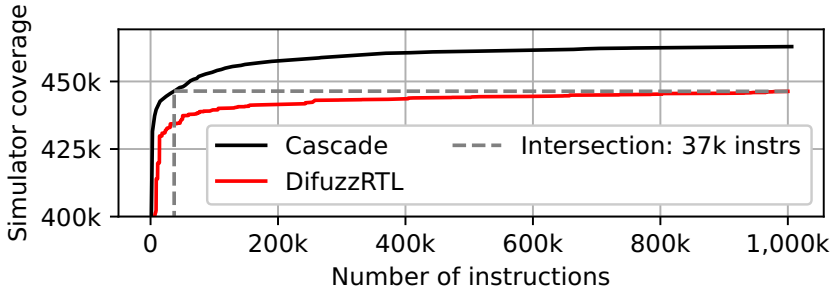

FIGURE 4.15: Simulator-collected coverage per instruction of Cascade and DifuzzRTL. Note that the y-axis starts at 400k.

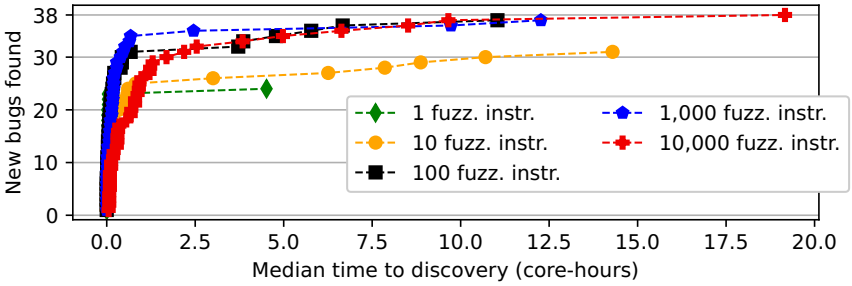

FIGURE 4.16: Time to reveal each bug when fuzzing with fixed numbers of instructions on 64 cores.

RESULTS. Figure 4.15 shows a per-instruction speedup of Cascade over DifuzzRTL of 27x for the simulator-collected coverage, while TheHuzz reports a speedup of 3.33x. Since TheHuzz reports a runtime slowdown of 71%, Cascade is 28.2x faster than TheHuzz on TheHuzz's target coverage metric.

4.7.5 *Efficacy of long programs*

To better understand the influence of program length on the efficacy of finding bugs, we enforce the number of fuzzing instructions per program,

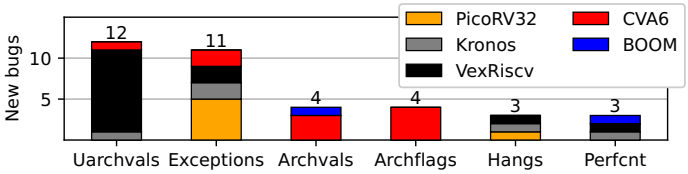

FIGURE 4.17: Discovered CPU bugs. Exceptions: missing or spurious exceptions. Uarchvals: wrong computations under microarchitectural conditions. Archvals: systematic wrong computations. Archflags: wrong status flags. Hangs: CPU hangs. Perfcnt: wrong performance counter values

fuzz on 64 cores and report the bug discovery times in Figure 4.16. Even limited to a single fuzzing instruction, Cascade discovers C8-C9 in a core-hour, undetected by TheHuzz [4] and HypFuzz [10]. Yet clearly, longer programs are more efficient at finding bugs. Also note that some bugs are hard to separate for this measurement, in particular C2-C7 and C10, and may overlap. The bug found only by the longest programs in 24 core-hours is K3.

4.7.6 *Bug discoveries*

Cascade discovered 37 new bugs in 5 CPUs and 1 bug in the Yosys synthesizer. Figure 4.17 classifies the new bugs in six categories, and Table 4.1, Table 4.2 and Table 4.3 the bugs we found, the corresponding CWEs and CVEs.

We first analyze the discovered bugs and their security implications, and then evaluate the bug detection performance of Cascade.

CONCURRENT FINDINGS. After a first bug report campaign, we submitted some new bug reports that were, in the meanwhile, concurrently found by the VexRiscv maintainer. These bugs are V10 and V11. From existing github issues, it cannot be excluded that symptoms of C1 had been noticed in the past. However, the root cause was clearly not understood, until Cascade and its analysis facilities allowed us to propose a fix, that has been approved and merged by the maintainers.

Design	Id	Bug Description	CWE	CVE	Sev.
VexRiscv	V1	Non-deterministic conversion from single-precision float to int	681	2023-34885	4.9
	V2	`fmin` with one NaN does not always return the other operand	193	2023-34885	4.9
	V3	Conversion from double to float may pollute the mantissa	681	2023-34885	4.9
	V4	Dependent arithmetic/muldiv FPU operations may yield incorrect results	193	2023-34885	4.9
	V5	Equal registers may be considered distinct by `fle.s` and `feq.s`	697	2023-34885	4.9
	V6	`flt.s` may return 1 when operands are equal	697	2023-34885	4.9
	V7	Under some microarchitectural conditions square root may be imprecise	1339	2023-34885	4.9
	V8	Single-precision muldiv followed by conversion may pollute the mantissa	681	2023-34891	4.9
	V9	Dependent arithmetic/muldiv operations may cause largely wrong output	682	2023-34891	4.9
	V10	Operations on floating-point registers are authorized when FPU is disabled	1189	2023-34885	4.9
	V11	Wrong access control to the FPU flags leaks information	1189	2023-34892	4.9
	V12	Hang on speculatively executed compressed FPU instructions	1342	2023-34896	7.7
	V13	Inaccurate instruction count when `minstret` is written by software	684	2023-40063	3.6
	V14	Some register comparisons are still incorrect despite a partial fix	697	2023-34885	4.9

TABLE 4.1: Bug Report Table (VexRiscv).

Bug descriptions

EXCEPTIONS. Cascade discovered 11 exception-related bugs in 4 designs, which we define as missing or spurious exceptions. For example, in VexRiscv, interactions with a disabled FPU are wrongly permitted (V10, V11). In Kronos, writes to a non-existent CSR fail to trigger an exception

Design	Id	Bug Description	CWE	CVE	Sev.
PicoRV32	P1	Accessing a non-implemented CSR causes the CPU to hang	1281	2023-34898	4.6
	P2	Spurious exceptions when reading mandatory CSRs	1281	2023-34897	2.6
	P3	Performance counters are not writable	284	2023-34900	2.6
	P4	Performance counters can only be read using some opcodes	284	2023-34914	2.6
	P5	Performance counter addresses are incorrect	684	2023-34913	2.6
	P6	Spurious exception when decoding `fence` instructions	705	2023-34899	5.0
Kronos	K1	RaWaW double-hazard may cause a wrong register value to be forwarded	226 226	2023-34902 2023-34902	6.6 6.6
	K2	Reading existing CSRs causes the CPU to hang in some uarch conditions	1281	2023-34901	7.1
	K3	In some uarch conditions, no exception when writing inexistent CSRs	1281	2023-42310	2.6
	K4	Inaccurate instruction count when `minstret` is written by software	684	2023-40066	3.6
	K5	Incorrect decode logic for `fence` and `fence.i`	684	2023-34903	5.0

TABLE 4.2: Bug Report Table (PicoRV32 and Kronos).

(K3). In PicoRV32, Kronos and CVA6, spurious exceptions may be triggered by some CSR accesses (P2-P5, C8-C9) or incorrectly decoded valid instructions (P6, K5).

MICROARCHITECTURAL-STATE-DEPENDENT WRONG COMPUTATIONS. Cascade discovered 12 bugs that produce wrong computations under certain microarchitectural conditions (*Uarchvals*) in Kronos and VexRiscv. In Kronos, a bug in the hazard detection unit (K1) causes, under some conditions, wrong register forwarding in a read-after-write-after-write double-hazard, where it forwards the first written value instead of the second. In VexRiscv and CVA6, wrong calculations occur under some microarchitectural FPU conditions (V1-V9, V14, C10). Such bugs are often hard to fix. Indeed, the VexRiscv maintainers proposed a fix, which solved most of the occurrences,

Design	Id	Bug Description	CWE	CVE	Sev.
CVA6	C1	Double-precision multiplications yield wrong sign when rounding down	682	2023-34904	4.4
	C2	Single-precision floating-point operations may treat NaNs as zeros	684	2023-34906	5.1
	C3	Division by NAN incorrectly sets NX and NV fflags	684	2023-34905	5.1
	C4	The inexact (NX) flag not set in case of overflow or underflow	684	2023-34907	5.1
	C5	Division of zero by zero incorrectly sets the DZ flag	684	2023-34909	5.1
	C6	+inf and -inf microarchitectural structures are inverted	1221	2023-34910	4.4
	C7	Infinities are not rounded properly and stick to infinity	1339	2023-34911	4.4
	C8	Spurious exceptions when reading some performance counters	684	2023-34911	5.0
	C9	Wrong supervisor performance counter access control	684	2023-42311	5.0
	C10	Under some uarch circumstances wrong NAN conversion	682	2023-34908	5.1
BOOM	B1	Static rounding is ignored for `fdiv.s` and `fsqrt.s`	1339	2023-34882	6.5
	B2	Inaccurate instruction count when `minstret` is written by software	684	2023-40065	3.6
Yosys	Y1	Logic synthesis of CVA6 inserts a logic bug into the FPU	682	2023-34884	6.8

TABLE 4.3: Bug Report Table (CVA6, BOOM and Yosys).

but Cascade discovered a way to tamper with the FPU's microarchitectural state again with a different approach (V8, V9, V14), which has ultimately been fixed by the maintainers.

SYSTEMATIC WRONG COMPUTATIONS. Cascade discovered 4 bugs in BOOM and CVA6 that produce wrong output values regardless of the microarchitectural state (*Archvals*). In BOOM, double-precision divisions and square roots ignore the (immediate) static rounding mode (B1). In

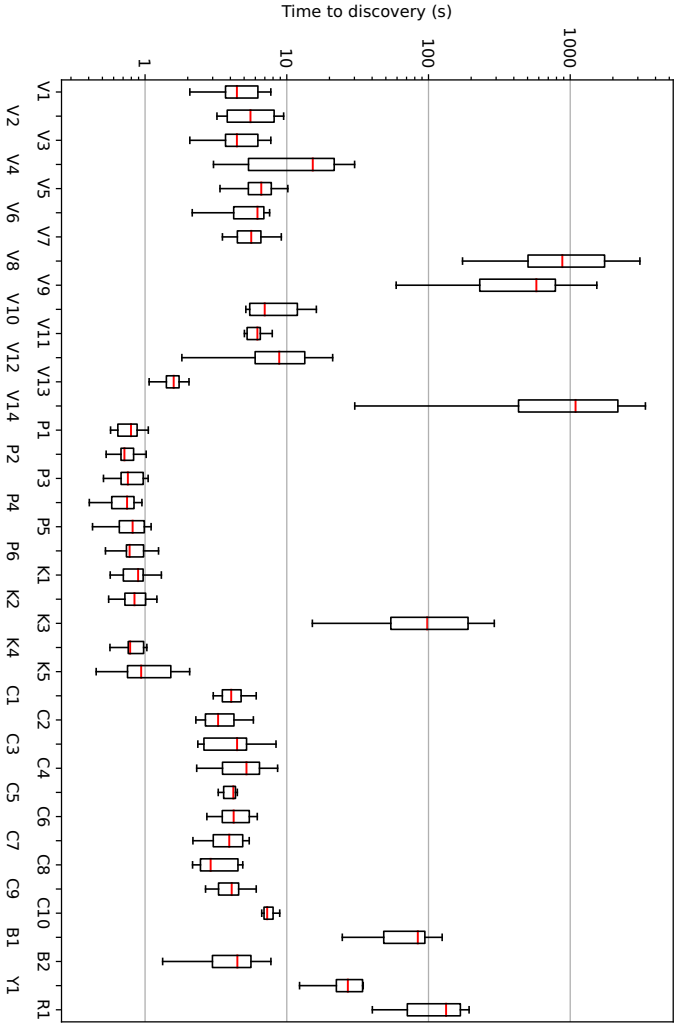

FIGURE 4.18: Time to reveal each bug when fuzzing on 64 processes. R1 is the known instruction counting bug in Rocket. Note the logarithmic scale. Bugs are labeled as follows: B (BOOM), C (CVA6), K (Kronos), P (PicoRV32), V (VexRiscv), and Y (Yosys).

CVA6, we found two wrong output sign bugs (C1, C6), and one unexpected infinity bug (C7).

SYSTEMATIC WRONG FLAGS. Cascade discovered 4 incorrect flag bugs in CVA6 (*Archflags*) (C2-C5). Since flags are typically used to guide a control flow, emitting FPU operations that will set flags incorrectly may provide an illegitimate control flow influence. Such bugs are difficult to fix. The CVA6 maintainers contacted us to test a fix, which we found to fix some bug (C2), but preserving another (C3).

HANGS. Cascade discovered 3 bugs that cause hangs in Kronos, PicoRV32 and VexRiscv (V12, P1, K2). In PicoRV32, the hang is systematic for accessing some CSR addresses. The hang in Kronos, also related to CSR access, depends on the microarchitectural state. The hang in VexRiscv is achievable when speculatively executing an illegal compressed floating-point instruction and later executing a legitimate floating-point instruction: microarchitectural resources are reserved but are never released, which results in a deadlock. The latter bug is similar to the recently-discovered Zenbleed bug [207], where an architectural bug leaves traces after speculation.

PERFORMANCE COUNTER INACCURACIES. Cascade discovered 3 inaccurate performance counter bugs (*Perfcnts*) in Kronos, VexRiscv and BOOM (K4, V13, B2). They incur an offset in the retired instruction counters when written by software.

YOSYS LOGIC SYNTHESIZER BUG. Yosys [205] is a popular open-source logic synthesizer, which is typically used for instrumenting a CPU [1, 5, 12, 54]. It may also be used to part of an emulation or ASIC flow. Cascade found a bug (Y1) that leads to incorrect logic synthesis of CVA6's FPU.

OTHER FINDINGS. Besides discovering new bugs in CPUs and in Yosys, Cascade found known bugs in CVA6, and the performance counter inaccuracy in Rocket discovered by DifuzzRTL and reported by TheHuzz and HypFuzz [4, 5, 10].

Additionally, Cascade found two issues related to X over-propagation in VexRiscv (shadowing of RTL logical signals by undefined values [206, 208]), which may happen under some conditions. First, some uninitialized instruction cache lines can enter the pipeline and pollute signals. Second, in case of FPU underflow, a hardware table is accessed with a too large index. HypFuzz [10] discusses the implications of such vulnerabilities when describing their third vulnerability [10].

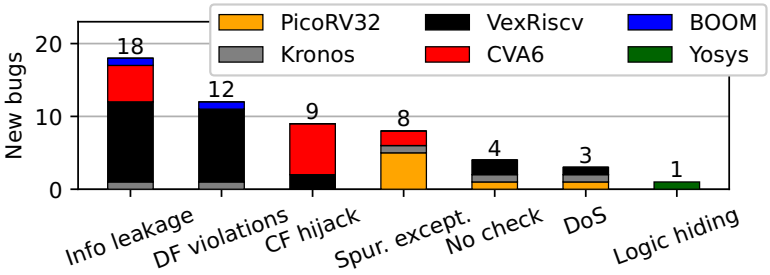

FIGURE 4.19: Security implications of discovered bugs. Note that some bugs belong to more than one category.

Security implications

We classify the security implications of the discovered bugs into six categories, as presented in Figure 4.19. The same bug may belong to more than one category.

DATA-FLOW INTEGRITY. We define data-flow integrity bugs as allowing a malicious time-shared user to cause a victim entity to execute computations with a wrong result. Cascade found 12 such bugs, where preparing the (micro)architectural state can cause wrong computations (V1-V9, V14, K1, B1). For example, exploiting the BOOM dynamic rounding bug (B1), an attacker process can force a victim process to take a different floating-point rounding mode than expected.

INFORMATION LEAKAGE. Sources of information leakage are many. For example, data-flow integrity bugs may also enable information leakage in the opposite direction, from the victim to an attacker running on the same core. So do bugs that illegally set some flags (C2-C5), and some unauthorized accesses (V10, V11), creating core-local side channels.

CONTROL-FLOW HIJACK. Cascade found 9 bugs that let an attacker process influence the control flow of a victim process (C1-C7, V5, V6). For example, CVA6 sets wrong flags in diverse situations (C2-C5) and produces wrong finitude or sign results (C1, C6, C7), which typically influence the control flow. Concretely, we found that the inexact flag is not set in some cases of underflow or overflow (C4), preventing some potential security checks. On VexRiscv, the microarchitecture can be prepared to corrupt

comparisons, for example making two equal registers be considered distinct (V5, V6). This preparation must be done by a process on the same core.

SPURIOUS EXCEPTIONS. Cascade found 8 spurious exception bugs (P2-P6, K5, C8, C9), which break isolation boundaries from higher privilege levels, for example, in the context of trusted execution environments. For example, in PicoRV32, reading some mandatory CSRs causes exceptions (P2), hence, a malicious machine could give the illusion that they are accessible, but actually the machine solely emulates the interactions, providing arbitrary values.

MISSING CHECKS. Cascade found 4 missing checks, with several implications. First, they may permit to bypass security checks (V10, V11), problematic, for example, if the FPU is time-shared with other cores. Second, they can deceive a victim to believe that a feature is supported (K3, P5). They can additionally be exploited to escape program analysis, where supposedly dead code is shadowed by an exception.

DENIAL OF SERVICE. All the hangs that we found (V12, P1, K2) can happen at any privilege level, leading to DoS attacks.

LOGIC HIDING. Using the discovered Yosys bug (Y1), a malicious contributor can inject bugs into a design while submitting an apparently innocuous RTL design, by transforming the RTL into an equivalent one that will trigger Y1 accordingly.

Bug detection performance

We fuzz on 64 processes for 24 hours and summarize the time to discover each bug in Figure 4.18, where we repeated the discovery 10 times with different seeds. Note that some bugs are hard to separate for this experiment, in particular C2-C7 and C10, and may overlap. In most runs, bugs are discovered in less than 18 core-hours (17 minutes). Design and bug complexities influence the discovery time.

Considerations of Previous Fuzzing Claims

In TheHuzz [4], it is considered as a bug (B4) that CVA6 does not throw an exception when executing self-modifying code without `fence.i`. RISC-V does not specify any behavior in this case, hence this is a feature request and not a bug. In HyPFuzz [10], their vulnerability V2 was later denied by the CVA6 maintainers [209], and V3 is not a bug [210]. Since V3 is not a bug,

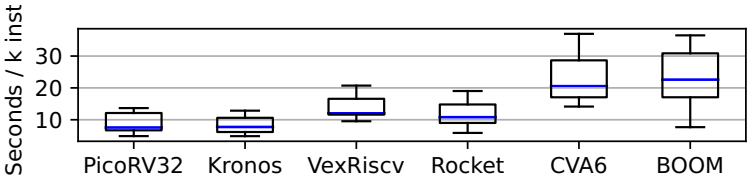

FIGURE 4.20: Program reduction performance.

we naturally did not count similar occurrences as bugs in the evaluation of Cascade (see Section 4.7.6). Conclusively, HypFuzz only describes a single new bug. In the archived version of ProcessorFuzz [211], Bug 10 is not a bug but a feature request, because the described behavior is explicitly permitted by the RISC-V specification.

4.7.7 *Program reduction performance*

We evaluate the performance of Cascade's program reduction. Since all designs still have at least one non-fixed bug (V14, P6, K4, C9, B2 and R1), we focus on them for this analysis. We run the fuzzer to collect 100 programs that trigger the bug in each design. Then, we perform the program reduction on these programs, and measure the time required to reduce each program and normalize it by the number of instructions in the program, including the initial and final blocks.

RESULTS. Some program reductions could fail, e.g., because a load instruction would target the hopping instruction of a basic block preceding a candidate tail block during the reduction phase and its result would propagate to the control flow. Experimentally, we did not observe such failures. Figure 4.20 shows the time to reduce each program, normalized by the length of the program. Program lengths varied between 193 and 62,870 instructions. The timeout upper bound, conservatively set to $30 \times n_{instructions} + 1000$ cycles, influences program reduction performance. In this experiment, the reduction always identified the head and tail instructions.

4.8 DISCUSSION

We discuss porting Cascade to other ISAs and limitations of coverage metrics used by the state-of-the-art CPU fuzzers.

ADAPTATIONS TO OTHER ISAS. While the fundamentals of Cascade's approach comply with most widespread ISAs, its current implementation is RISC-V-specific. Adaptation to RISC ISAs such as ARM and MIPS requires ISS and instruction generator adaptations. We expect porting Cascade to CISC ISAs to be more challenging due to more instructions.

COVERAGE METRICS. The coverage metrics used by the state-of-the-art CPU fuzzers such as TheHuzz [4] and DifuzzRTL [5] are dominated by ineffective terms while introducing runtime overhead. For TheHuzz, toggle coverage represents around a million points and 89% of the achievable points on Rocket. Given that TheHuzz considers the sum of all coverage points of all types, any mutation that would produce such a single new toggle would be considered as discovering new coverage. Limitations of such simple metrics are well-known [212], yet the more powerful alternative, functional coverage, requires significantly more effort [213]. This explains why HypFuzz, building on the same metric as TheHuzz, found a single new bug. DifuzzRTL defines control registers as registers which control a multiplexer inside the same HDL module. This coverage metric continues to increase when exploring registers whose fanout multiplexers already took all values. Ultimately, DifuzzRTL optimizes for exploring an immense number of combinations of values for registers that are mainly arbitrarily selected [17].

4.9 RELATED WORK

In the recent years, hardware fuzzing has flourished. We first cover generic hardware fuzzers, then CPU fuzzers.

GENERIC HARDWARE FUZZERS. RFUZZ [12] is a generic hardware fuzzer that relies on multiplexer control signals to generate inputs based on AFL [181]. DirectFuzz[13] targets RFUZZ toward specific modules. Trippel et al. [93] proposed to fuzz the hardware simulation binary itself, using software fuzzing methods. Ruep et al. [214] proposed a fuzzer for SpinalHDL designs. Li et al. [28] proposed a new coverage metric for fuzzing. Ragab et al. [215] proposed a distance-to-target feedback metric to direct fuzzers

towards specific targets. None of these publications reported the discovery of any bugs.

CPU FUZZERS. Many active open-source repositories rely on testing tools, such as the RISC-V compliance suite [156], which generates basic unit tests, and RISCV-DV, a UVM-based testing framework based on commercial simulators [216]. To address their insufficiencies, several projects proposed fuzzing CPUs. DifuzzRTL [5] generates instructions and collects control register coverage to guide the fuzzing process and discovered 16 new bugs. Its source code, while relying on obsolete languages, is available for fuzzing legacy versions of Rocket and BOOM[204]. ProcessorFuzz [17] is a concurrent work that generates instructions and collects coverage of control and status registers on the ISS and reported the discovery of 9 new bugs, including 7 on BlackParrot, maintained by certain authors of ProcessorFuzz. Kabylkas et al. present Dromajo and Logic Fuzzer [9] and report the discovery of 13 new bugs, including 4 with the help of Logic Fuzzer. Bruns et al. [8] exploited ISS coverage to find new VexRiscv bugs. Similarly, Herdt et al. [7] exploited ISS coverage to find 10 new bugs in their own core. N. Bruns et al. [6] generate an infinite instruction stream on the memory channel to find a bug in their own core. Such an infinite instruction stream is known to cause compatibility issues due to strong microarchitectural assumptions [8]. Kande et al. [4] proposed TheHuzz, based on commercial RTL simulator coverage feedback, and fond 7 new bugs. Chen et al. [10] proposed a technique to speed up the coverage obtained by TheHuzz and found one new bug. Cascade is the first fuzzer to take a constructive approach to tackle the observations and challenges exposed in Section 4.3.

4.10 CONCLUSION

We presented Cascade, a CPU fuzzer based on the explicit construction of intricate RISC-V programs. Cascade is effective: it finds 37 new bugs on 5 RISC-V CPUs with varying degrees of complexity and vastly outperforms the state-of-the-art coverage-guided CPU fuzzers. What sets Cascade apart from the state of the art is its ability to efficiently construct long complex programs that enable high-throughput CPU fuzzing while terminating by design. Any non-termination signifies the discovery of a bug in the target CPU. We described how Cascade generates its programs using a new technique, asymmetric ISA pre-simulation, which enables Cascade to

efficiently entangle an arbitrarily-complex control flow with an arbitrarily-complex data flow. Since the bug-triggering programs generated by Cascade may be long and complex, we introduced a new technique for program reduction transforming a program to the only few instructions that trigger the CPU bug.

ETHICAL CONSIDERATIONS. We reported all bugs to their respective maintainers, and proposed fixes when our understanding of the language and design was sufficient.

ACKNOWLEDGEMENTS

The authors would like to thank the anonymous reviewers for their valuable feedback, Tobias Kovats for his contribution to RFUZZ re-implementation, and the maintainers of the designs we tested for their support in understanding and fixing some of the bugs. The work in this chapter was supported in part by a Microsoft Swiss JRC grant and by the Swiss State Secretariat for Education, Research and Innovation under contract number MB22.00057 (ERC-StG PROMISE).

5

LOST IN TRANSLATION: CONFUSED DEPUTY ATTACKS ON EDA SOFTWARE

We introduce MiRTL, a new class of attacks that enables a seemingly correct RTL design to translate into synthesized malicious hardware. MiRTL is the result of the confused deputy problem in EDA software such as simulators or synthesizers that translate RTL designs to lower-level representations. This paper explores the discovery and exploitation of such vulnerabilities for realizing MiRTL attacks. Our new fuzzer, called TransFuzz, generates randomized RTL designs containing many operators with complex interconnections for triggering translation bugs. The expressiveness of RTL, however, makes the construction of a golden RTL model for detecting deviations due to translation bugs challenging. To address this challenge, TransFuzz relies on comparing signal outputs from multiple RTL simulators for detecting vulnerabilities. TransFuzz uncovers 20 translation vulnerabilities among 31 new bugs (25 new CVEs) in four popular open-source EDA applications. We show a real-world instantiation of a MiRTL attack by injecting malicious hardware using seemingly benign RTL code into the CVA6 RISC-V core using these discovered vulnerabilities.

5.1 INTRODUCTION

Community-driven open-source hardware development is on the rise [74, 75, 137, 217–224], exposing EDA software used for RTL simulation and synthesis to hardware designs from many, potentially distrusting, developers. Similar to standard software, RTL simulators and synthesizers are subject to standard vulnerabilities such as buffer overflows. The unique nature of hardware development flows, however, exposes EDA software to a more silent and sinister class of vulnerabilities that we explore in this Chapter. These vulnerabilities enable a hardware developer or a malicious intermediate EDA application to turn a seemingly benign piece of hardware design into synthesized malicious hardware.

TRANSLATION BUGS. RTL simulators are used in all stages of hardware development for testing the validity of a design by transforming the RTL, often expressed in an HDL, into a lower-level model suitable for execution on the designer's system. Once the hardware design is final in the late stages of hardware development, the HDL representation is transformed into a gate-level netlist by a synthesizer. The translation of the HDL by an RTL simulator or a synthesizer to lower-level representations involves a variety of complex optimizations, e.g., for performance or area. This complexity can cause the EDA software to mistranslate the given HDL design. These translation bugs can lead to a class of attacks that we call MiRTL (Mistranslated RTL). When targeting RTL simulators, MiRTL enables certain HDL behavior not to register during simulation, but appearing in the synthesized hardware. When targeting synthesizers, simulators correctly register all the values correctly, but the malicious behavior is injected by the synthesizer. To employ MiRTL, attackers need to discover translation bugs in the target EDA software.

MIRTL. Fuzzing is a common technique to find vulnerabilities in software [181–194] and hardware [4, 5, 10, 11, 17, 225] by generating input against the hardware or software interface. The input to EDA software instead is an arbitrary RTL design that takes arbitrary inputs. Hence, our new fuzzer, TransFuzz, generates randomized RTL designs and provides these designs with randomized inputs under different RTL simulators and synthesizers. TransFuzz must ensure that the generated RTL designs are sufficiently random to provide ample opportunities for optimizations. Generating randomized RTL at the HDL-level often results in regular representations and interconnections of macrocells after parsing by the EDA software. Instead, TransFuzz generates randomized RTL designs at the macrocell-level which provides maximum flexibility for randomizing the number and type of macrocells as well as their interconnection.

TransFuzz-generated designs may trigger translation bugs, yet it is unclear how we can detect them. Unlike software fuzzers that can rely on crashes [182–186, 188–194] or sanitizers [181, 185], and hardware fuzzers that rely on golden models [5, 10, 11, 17] or the ISA itself [225], there exists no golden model for HDL and constructing one is challenging due to the expressiveness of HDLs. Instead, TransFuzz generates randomized designs in a way that a configurable number of output signals provide a signature of the design over an arbitrary number of cycles. TransFuzz then compares these values across different EDA software to detect deviations, similar to differential fuzzing [226–228]. Given the freedom in the specification of

HDLs, however, we must slightly constrain the generation of our circuits to avoid false positives. Namely, TransFuzz avoids race conditions with respect to signals that control state-saving elements and avoids the generation of X signals inside the designs.

Our evaluation shows that TransFuzz triggers 31 new bugs (25 CVEs) in three open-source RTL simulators, namely Verilator, CXXRTL, and Icarus Verilog, and the open-source Yosys synthesizer. Of these 31 bugs, 20 are translation bugs, enabling MiRTL attacks on all of these four popular EDA applications.

EXPLOITING TRANSLATION BUGS. To build exploits using translation bugs, we introduce the MiRTL gadget, a primitive that produces a certain value under normal circumstances and the opposite value when triggering the translation bug. These gadgets serve as a basis for building various MiRTL attacks. As an example, we have built a MiRTL attack that allows the synthesis of a malicious version of CVA6 [74] (a RISC-V CPU) to enable unprivileged software to leak supervisor memory, yet this leakage is ignored by all three simulators. We also show how MiRTL attacks can bypass standard hardware-level leakage detection schemes such as information flow tracking [54, 131, 229].

CONTRIBUTIONS. The following lists our contributions:

- We explore MiRTL, a new class of confused deputy attacks exploiting RTL translation bugs on EDA software.
- We design and implement TransFuzz, a new fuzzer that uncovers translation bugs by generating complex randomized RTL designs and a differential approach for detecting these vulnerabilities.
- We apply TransFuzz on 4 popular EDA applications to discover 31 new bugs, including 20 translation bugs.
- We instantiate MiRTL attacks by introducing MiRTL gadgets, building a malicious version of a RISC-V core and evading information flow tracking using the discovered translation bugs on our target EDA applications.

RESPONSIBLE DISCLOSURE. We have reported all the discovered bugs to the maintainers of the respective EDA software.

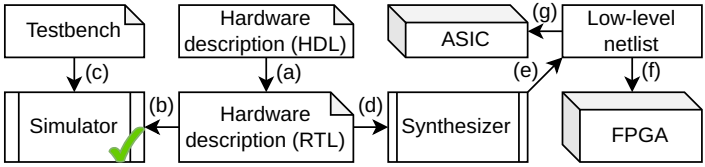

FIGURE 5.1: Typical digital hardware flow.

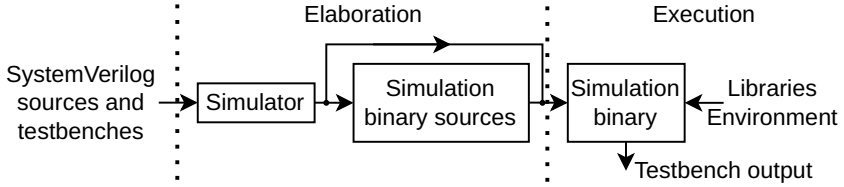

FIGURE 5.2: Elaboration and execution phases of RTL.

5.2 BACKGROUND

This section provides the necessary background about the hardware development flow, RTL simulation, and fuzzing.

5.2.1 *Digital hardware development flow*

A typical hardware development flow is illustrated in Figure 5.1. A hardware design is described at the Register-Transfer Level (RTL) (a). Designs at such a level are usually expressed in VHDL and Verilog. The latter language has gained significant traction in the last decade, to the extent that almost all significant open-source hardware projects are written in Verilog, or in a language that is typically compiled to Verilog [214, 230–232]. The hardware design is then simulated (b) along with some testbench to check that the design fulfills the intended functionality (c). Once validated, the design is then synthesized (d) to a lower-level netlist (e) that is destined for a specific FPGA family (f) or ASIC fabrication process (g). Simulation of the lower-level netlist are often orders of magnitude slower than pre-synthesis

simulations [54]. Further steps depend on the target technology and often include floorplanning, place and route.

5.2.2 *RTL simulation*

From hardware descriptions and testbenches, RTL simulators are software applications that can produce outputs in multiple forms, such as binary pass/fail or a simulation trace. Popular open-source RTL simulators include Verilator [233], Icarus Verilog [234] and CXXRTL [205]. RTL simulators generally operate in two stages, illustrated in Figure 5.2. First, they take a hardware design and produce a simulation model. We call this step *elaboration*. Verilator and CXXRTL translate the design into C++ code meant to be compiled against the testbench to produce a simulation model, while Icarus produces an intermediate simulation model that is an input to a static executable (`vvp`). Second, they take a simulation model and a set of input values, and run the simulation to produce the expected form of output. We call this step *execution*.

To simulate a design efficiently, RTL simulators perform a series of optimizations, such as constant propagation, dead code elimination, and common subexpression elimination. These optimizations are intended to preserve the design's functionality for all synthesizable parts that are not affected by undetermined "X" or high-impedance "Z" values.

5.2.3 *Fuzzing*

Fuzzers apply random inputs to a tested (hardware or software) unit and observe its behavior. They generate inputs following various strategies often classified as white box, gray box or black box, and depending on the feedback information they consider [4, 10, 11, 17, 181–194, 225]. Then, they observe whether a bug was triggered, using strategies such as crash observation [181, 225], sanitizers [195–197] or comparison with a reference implementation [5].

DISCUSSION. Existing fuzzers are not suitable for discovering deep bugs in EDA software, particularly translation bugs. The inputs must represent complex hardware and stimuli. For simulators, this even requires fuzzing the simulator and the simulation model. Furthermore, output deviations are not covered by the capabilities of traditional fuzzers.

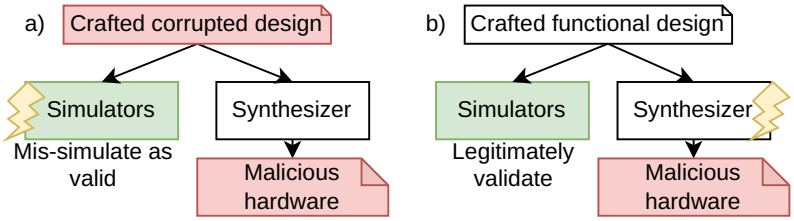

FIGURE 5.3: High-level MiRTL attacks exploiting (a) simulator bugs and (b) synthesizer bugs. Red hardware is malicious.

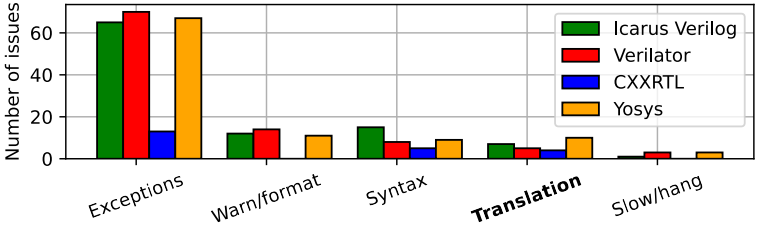

FIGURE 5.4: Distribution of recent public bug reports. Exceptions: crashes and assertion failures. Warn/format: wrong warning or text/trace formatting. Syntax: syntax-bound issues. Translation: translation bugs. Slow/hang: hang or unjustified significant slowdown.

5.3 THREAT MODEL

We consider a malicious hardware IP, an EDA software vendor or a contributor whose intention is to discretely alter a hardware design. We do not assume any property of the victim Verilog design. In the concrete exploits that we present, the victim uses some of the three popular open-source simulators Icarus Verilog [234], Verilator [233] or CXXRTL, and the state-of-practice open-source synthesis tool Yosys [205].

5.4 MIRTL ATTACKS

While the threat of hardware trojans is well-understood [235–242], we show that exploiting bugs in EDA software in a confused deputy scenario is a realistic and practical alternative for inserting vulnerabilities that are *undetectable* under normal testing conditions, as we illustrate in Figure 5.3. A translation bug in an RTL simulator enables a MiRTL attack where an attacker-crafted behavior in HDL is not detectable by the RTL simulators and gets synthesized into malicious hardware (a), and a translation bug in the synthesizer enables a MiRTL attack where the malicious behavior is not present in the HDL (hence also undetectable by simulators) and injected by the synthesizer into malicious hardware (b).

We build TransFuzz to find such translation bugs in designs that are expressed in synthesizable Verilog, which is nowadays the most popular RTL language. Interestingly, only a few of such bugs were discovered in the last years despite the high activity around open-source EDA. In Figure 5.4, we show the distribution of up to the 100 most recent public bug reports from four popular open-source EDA applications. We provide a detailed methodology in Section 5.5.1. In total, only 22 of the 322 reported bugs are translation bugs, which hints at the difficulty of looking for and discovering such bugs. We provide an overview of challenges in the design and implementation of TransFuzz and performing MiRTL attacks using the vulnerabilities that it discovers.

5.4.1 *Overview of challenges*

The first challenge in the design of TransFuzz regards the structure and abstractions of the inputs that we supply to the EDA software.

> **Challenge 1.** Design suitable abstractions and structures for fuzzing RTL simulators.

In Section 5.5, we start by analyzing the recent bug reports from popular open-source EDA software to understand the properties of inputs that may trigger translation bugs. From this analysis, we deduce beneficial characteristics of the inputs that we supply to the EDA software to exert translation bugs. To comply with these requirements, we propose a new abstraction

level: netlists of macro-cells, and expose a concrete way to generate them, along with random stimuli sequences with suitable dimensions.

The second challenge regards the detection of bugs, both for simulators and synthesizers, as they do not have a golden model or a formal specification.

Challenge 2. Detect bug occurrences.

In Section 5.6, we propose to use differential fuzzing, using different simulators or sets of parameters, to account for the absence of a formal model. Slightly constraining the input space allows TransFuzz to avoid all undefined behaviors that would occur with fully random circuits to enable sound differential fuzzing. We additionally present the mechanism used by TransFuzz to reduce the complexity of the test cases. In Section 5.7, we evaluate the performance of TransFuzz and describe the vulnerabilities that it uncovers.

The final challenge regards exploitation.

Challenge 3. Exploit the newly discovered bugs.

To implement practical exploits, Section 5.8 introduces the notion of *MiRTL gadget*, a synthesizable primitive that relies on simulator or synthesizer bugs to inject a mistaken value into the hardware design. Using these MiRTL gadgets as building blocks, we build a concrete kernel information leakage vulnerability into the Ariane RISC-V CPU [74] using either simulator or synthesizer bugs. We additionally build a shadow gadget that stops information flow tracking mechanisms from detecting MiRTL attacks.

5.4.2 *Overview of MiRTL*

Figure 5.5 summarizes the overall design of TransFuzz. TransFuzz operates in four steps. (1) It generates test case descriptions. (2) From the test case description, it generates a testbench and a standard Verilog design. (3) It then performs differential execution. (4) For each test case that triggered a mismatch, it categorizes and reduces it.

TEST CASE GENERATION. TransFuzz first generates designs as networks of macrocells with randomized types, attributes and connections, as we will

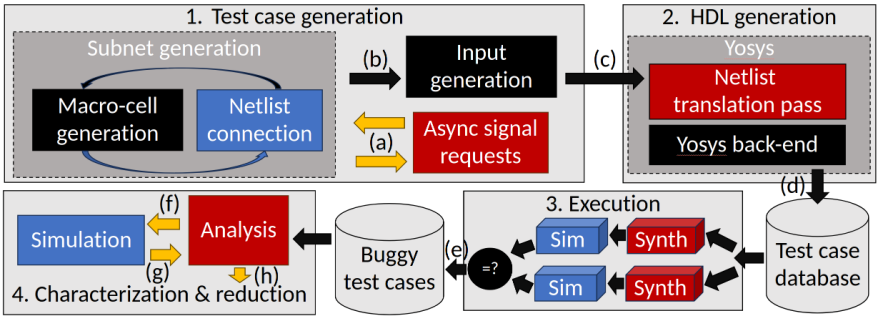

FIGURE 5.5: Overview of TransFuzz.

discuss in Section 5.5. It further populates asynchronous signals for state elements iteratively (a), and adds a stimuli component (b).

HDL GENERATION. TransFuzz then transmits a description (c) of the hardware component to an HDL generator written as a pass in the Yosys synthesizer. The output is a standard Verilog description of the final hardware component (d).

TEST CASE EXECUTION. TransFuzz submits the test case to several EDA applications for differential fuzzing and records mismatching instances (e).

CHARACTERIZATON AND REDUCTION. TransFuzz performs simulations with modified parameters to characterize the bug (f) and may attempt to reduce it further in the hardware (space) and stimuli (time) dimensions (g). Eventually, TransFuzz provides the reduced test case to be reported (h).

5.5 INPUT DESIGN

We first analyze existing bug reports to understand the properties of inputs that may trigger translation vulnerabilities in EDA software. Based on these observations, we propose a design and implementation of these potentially bug-triggering low-level inputs which we call subnets.

Report reference	Title
CXXRTL #4074	bmux does not mask the result
CXXRTL #3820	incorrect result of shl operator
CXXRTL #2780	CXXRTL [...] when routing signal via module
CXXRTL #2746	In CXXRTL edge eval is before calculating value
Verilator #4536	Shift when using streaming operator on 32 bit signal
Verilator #3824	Bit OR tree misoptimization
Verilator #3773	Seemingly incorrect terms in condition in V3Tristate
Verilator #3770	Signal skips flip-flop under some circumstances
Verilator #3509	Wire tie-off causing bad logic optimization
Verilator #3470	Wrong expression evaluation results
Verilator #3445	V3Const BitOpTree optimization is incorrect
Verilator #3409	complex assign in always_comb
Verilator #3399	Incorrect tristate enable logic
Yosys #4064	Frontend/AST: signed assign to indexed part-select
Yosys #4010	Synthesis optimization error, inconsistent simulation
Yosys #3879	LEC failed after yosys synthesis
Yosys #3867	Inconsistency Issue [...] opt_expr -fine Pass in Yosys
Yosys #3848	During synthesis [...] errors in register assignment
Yosys #3748	write_smt2: bugs caused by the '»' operator
Yosys #3680	Possible initialization issue in Xilinx DSP48E1 cell
Yosys #3431	Wrong smt-lib model behavior since yosys v0.15
Yosys #3360	synth_xilinx [...] output bit is driven 'Z'

TABLE 5.1: Previous reports of translation bugs.

5.5.1 *Analysis of past bug reports*

We analyze previously reported bugs in Verilator, Icarus Verilog and CXXRTL. For each simulator, we iterate through the bug reports, in reverse chronological order. We study the 100 most recent relevant bug reports for Verilator, Icarus and Yosys, and all the bug reports of the last three years for CXXRTL. We filter out the bug reports that have been denied by the maintainers or that are a duplicate of an earlier bug report.

RESULTS. We summarize the translation bugs from Figure 5.4 in Table 5.1. Verilator and Yosys are the most affected. On the contrary, there has been no report of a translation bug in Icarus in the past 100 relevant bug reports

(in a period of roughly three years), until TransFuzz's finding reported in Section 5.7.5. We make a number of observations about these bugs: first, a diverse set of HDL operator types such as shift, bmux and OR are required to cover these bugs.

Observation 1. Translation bugs are triggered by a diverse set of specific HDL operator types.

Second, not a single one of these bugs can be triggered with a single-operator test case. Instead, all of them require some non-trivial interconnection of multiple HDL operators, and these interconnection patterns vary from one bug to another.

Observation 2. Translation bugs require non-trivial and diverse interconnection patterns.

Third, out of these 22 bugs, up to 21 of them can be triggered with operators with a width between 2 and 4 bits.

Observation 3. Most translation bugs can be triggered with narrow cells.

These observations lead us to the following requirements for the design of TransFuzz.

REQUIREMENTS. The test cases produced for fuzzing a single RTL simulator must satisfy the following requirements:

1. *Operational diversity.* The test case generator must produce a large diversity of basic hardware operations.

2. *Relational diversity.* The test case generator must produce a large diversity of interconnection patterns.

3. *Operator size distribution.* The distribution of operator sizes must favor narrower cells, without excluding larger ones completely.

4. *Soundness.* The hardware that is produced by the test case generator must comply with the fundamental principles of digital hardware designs. In particular, being exempt from combinational loops, and multi-driven nets.

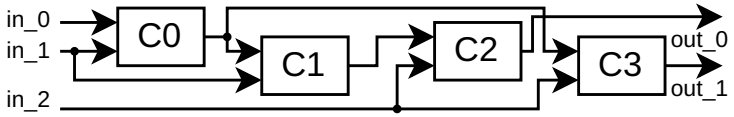

FIGURE 5.6: Subnet example.

5. *Syntactic correctness.* The hardware that is produced by the test case generator must be syntactically correct.

The last two requirements enable TransFuzz to find translation bugs in valid RTL designs.

5.5.2 *Test case generation*

The usual testing approaches systematically lack one of these requirements. Fuzzing at the text level, while finding crashes in front-ends, fails at producing any non-trivial hardware [243]. Fuzzing at the AST level guarantees syntactic correctness but hampers relational diversity [244, 245].

NETWORK OF MACROCELLS. In the light of these requirements, we propose the *network of macrocells* as a new abstraction for fuzzing EDA software. The macrocell abstraction corresponds to simple synthesizable stateful (e.g., registers with enable and clear signals) or combinational (e.g., adders or multiplexers) operators. We propose to construct a network of interconnected diverse macrocells to produce a sound relationally and operationally diverse hardware component. We will use the terms cells and macrocells interchangeably.

Subnet structure

We define subnets as hardware circuits in which the input of each cell is a design input or the output of another cell in the subnet. The *asynchronous* signals of stateful macrocells in the network, such as reset and clock signals, are supplied by dedicated inputs or by the output signals of other networks.

Figure 5.6 shows a four-cell subnet with three 32-bit input words and two 32-bit output words. The cells may individually be of any synthesizable combinational or stateful type.

CONSTRUCTING A SOUND SUBNET. To construct a sound subnet, we must ensure that there are no combinational loops and enforce all wires to have a single driver. We enforce these rules at no computational cost by enforcing two invariants during the construction of the hardware component. To enforce the single-driver rule, we make sure that a wire only takes one of the following roles: (a) a design input, (b) the output of a cell or (c) a design output.

To enforce the loop-free rule, we build subnets as sequences of cells, in which each cell can only be driven by the subnet input ports or by the output of a previous cell in the sequence. For maximizing relational diversity, TransFuzz inserts non-combinational loops through the following algorithm: for a given cell C, the algorithm colors all cells in its subnet with white (non-successor), red (combinational successor) or green (non-combinational successor). All cells start white and only green cells are eventually eligible for loop insertion. Algorithm 1 provides more detail.

WIRE CONCATENATIONS. In accordance with relational diversity, TransFuzz allows cell inputs to be concatenations of multiple wires. Wire concatenation is necessary for connecting a cell's output to a wider input of another cell.

Implementation

INPUT SELECTION. Always selecting the subnet's input words as cell inputs would impair relational diversity. So would always selecting the output of the previously generated cell in the subnet, which would underwhelm optimizations that exploit parallelism in the circuit. TransFuzz embeds an algorithm that selects the inputs of a cell as a potential concatenation of previous cell outputs and favors wires that have not been connected to a cell input yet. This algorithm is detailed as Algorithm 2.

STIMULI SPECIFICATION. Stimuli values are lists of binary words. We define stimuli as a (temporal) sequence of pairs `(subnet_id, input_values)`, where `subnet_id` identifies the subnet or asynchronous wire, and `input_values` is a (spatial) list of values to be applied to the inputs to this entity. Stimuli pairs are applied sequentially to the respective inputs without any explicit

Algorithm 1: Non-combinational loop insertion.

Data: An input port of a given cell C
Result: An eligible cell C' whose output will be connected to C's input

```
reds = ∅;
redfanout = {C} greens = ∅;
greenfanout = ∅;
whites = allCells - {C};
currcolor = red;
while redfanout ≠ ∅ or greenfanout ≠ ∅ do
    if redfanout ≠ ∅ then
        D = redfanout.pop();
        currcolor = red;
    else
        D = greenfanout.pop();
        currcolor = green;
    whites.remove(D);
    if stateful(D) or currcolor == green then
        greens.add(D);
        greenfanout.addmultiple(D.fanout ∩ whites);
    else
        reds.add(D);
        redfanout.add(D.fanout ∩ whites);
return pick(greens) if greens ≠ ∅ else ⊥
```

form of reset in between, to let the hardware component enter subsequent states.

CELL WIDTH SELECTION. Following an earlier requirement on cell widths, when generating each cell, we select the width of the cell's inputs and outputs following an offsetted geometric law with parameter $p = 1/8$: $P(W >= x) = (1-p)^{x-2}$. This leaves a 2% chance for cells of at least 32 bits, while two thirds of the widths will be below 10 bits.

VERILOG BACKEND. EDA software generally operates on Verilog sources, not on netlists of macrocells. TransFuzz relies on the Yosys Verilog backend, which is a mature and well-tested implementation of a Verilog netlist generator. Hence, TransFuzz's input generator, implemented in Python,

Algorithm 2: Input selection algorithm sketch. The function *pickcell* selects some previous cell in the circuit, with some bias toward the yet unused output bits.

```
   Data: port_width
   Result: Inputs for the next macrocell input port
1  in_offset = 0; remaining_bits = input_bits;
2  while remaining_bits > 0 do
3  |  C = pickcell(N);
4  |  out_offset = C.out.width; conn_width = min(remaining_bits,
   |    C.out.width-out_offset);
5  |  connect(port_width-remaining_bits, out_offset, conn_width);
6  return connections
```

produces a serialized netlist description. A TransFuzz pass in Yosys, written in C++, then translates this description into a Yosys's internal representation. The Yosys backend eventually produces a Verilog source.

5.6 DIFFERENTIAL FUZZING FOR BUG DETECTION

EDA software does not generally have a formal specification or a golden model. Additionally, translation bugs usually do not produce obvious signals like error messages or crashes.

One could observe all intermediate values produced by a test case in space and time, e.g., by instrumenting the test case with many probes, or by enabling tracing. However, this approach has two drawbacks. (a) Performance: monitoring all values is expensive in terms of time and memory, while fuzzing is generally performance-sensitive. (b) Intrusiveness: intermediate acquisitions may prevent some optimizations, hence may hide bugs. To address this challenge, we propose to employ differential fuzzing [226–228] to detect bugs in EDA software. Differential fuzzing requires the selection of variants for fuzzing and a way to compare these variants.

VARIANT SELECTION. We find that differential fuzzing (DF) can be applied in two ways. In *internal* DF, the test cases are executed differentially between two parameter settings of the same application, like optimizations or tracing, while in *external* DF, two distinct applications are used, as

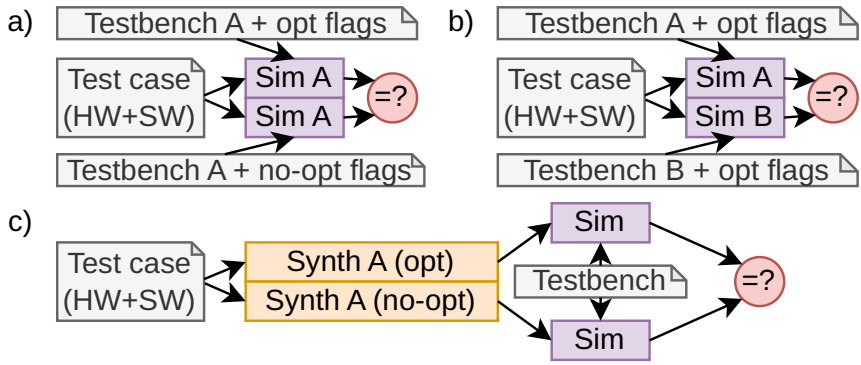

FIGURE 5.7: Differential fuzzing (DF). (a) Simulator (internal). (b) Simulator (external). (c) Synthesizer (internal).

illustrated in Figure 5.7. TransFuzz uses external DF for the simulators and internal for the synthesizer. In Section 5.6.2, we further use simulator-internal DF for characterizing and reducing simulator bugs.

Note that differential fuzzing does not, per se, specify which variant is incorrect. The most straightforward and classical solution is majority voting, yet the majority could be wrong, especially when it comes to misinterpretations of the Verilog standard. Another approach is to disable all optimizations and see how the behavior evolves. We could attribute all the bugs found by TransFuzz (Section 5.7.5) using a combination of these two methods.

CUMULATIVE SIGNATURE. To provide an effective way to compare variants, TransFuzz must increase the likelihood that an error triggered by a translation bug becomes visible by the output from these variants. To achieve this, TransFuzz relies on a few *explicit* hardware output signals that it cumulates over the required number of cycles, resulting in a *cumulative signature* that can then be compared across variants. For assisting the propagation of unexpected values, we bias the macrocell input selection towards macrocell outputs that are not yet used. This avoids trivial cases where a bug is not observed due to the absence of a path between a buggy produced value and the output. We provide further details about the section of explicit output signals in Algorithm 3.

Algorithm 3: Source cell selection.

Data: An array L of used cell ouptut ids.
P is randomly picked at the test case start.
Result: A source cell C'

```
if random() < P then
    return pickcell(L)
else
    return pickcell(allCells)
```

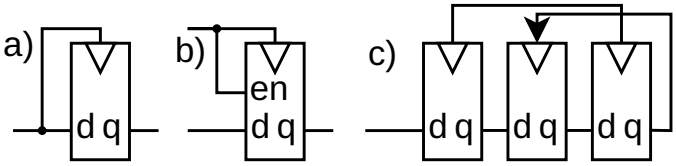

FIGURE 5.8: Examples of race conditions. (a) Incoming data and clock are connected. (b) Enable and clock signals are connected. (c) The input and clock of the central register change simultaneously.

5.6.1 *Ensuring consistency*

The Verilog standard offers slack on some aspects, such as the ordering of some events, or the precision of X propagation. Hence, even when simulating the same RTL, legitimate divergences between two simulators may exist. To avoid false positives when comparing variant outputs, we must slightly constrain test-case generation as discussed next.

AVOIDING X PROPAGATION. Verilator and CXXRTL do not explicitly support X propagation, yet Icarus does. Additionally, all simulators that support it may have multiple levels of precision [208, 210, 246–248]. While Verilog specifies 4 levels of logic (0, 1, X: "don't care", Z: "high impedance"), we use the special `bit` construct that only supports 2 levels (0, 1). However, the Verilog standard specifies divisions and modulo by zero as a special case where X propagation through `bit` signals happens, and then propagates through the design. We therefore enforce that all divisors and modulo divisors have at least one bit constantly tied to VCC. These measures ensure the absence of X propagation in the circuit.

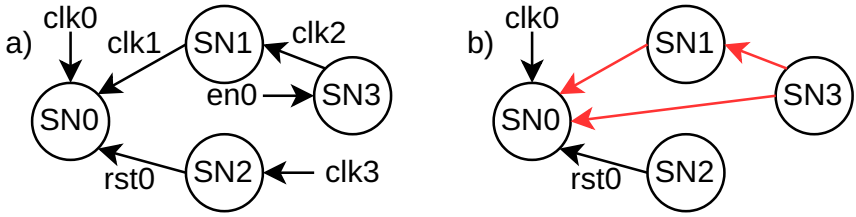

FIGURE 5.9: Examples of (a) valid and (b) invalid (race condition prone) network of subnets (SN). The input and output words of each subnet are omitted in the figure.

ASYNCHRONOUS ASPECTS. Asynchronous signals could cause race conditions that would impair cross-EDA software consistency. Figure 5.8 summarizes some examples of race conditions. They have in common that multiple input ports of some register can simultaneously toggle. In such cases, like for eventual hardware implementations, the Verilog standard does not specify the event ordering.

To overcome the race conditions in a generic way, we impose the following safety invariant on the test case hardware: two inputs of a register will never toggle at the same time. We guarantee this invariant at a subnet level, by distinguishing two wire types. *Asynchronous* wires belong to the sensitivity list of some stateful cell. The others are called *synchronous* wires. We assimilate synchronous wires to data signals produced inside the same subnet, and asynchronous wires to data signals produced outside the subnet, as illustrated in Figure 5.9 (a). Concretely, when TransFuzz generates a subnet, this subnet creates a set of requests for asynchronous wire connections (e.g., clock, reset or enable signal). Each of these requested asynchronous values is produced either by another subnet, or by a new asynchronous input wire.

To safely interconnect subnets, TransFuzz imposes on the hardware side that the network of subnets, seen as an undirected graph, must be acyclic. Furthermore, on the stimuli side, at any given point in time, the set of fanout subnets of the set of toggling asynchronous signals must be disjoint. These conditions guarantee the strict absence of race conditions. Figure 5.9 (b) illustrates a violation of this constraint.

5.6.2 *Complexity reduction*

TransFuzz generates test cases that can be complex on the hardware side, and long on the stimuli side. Bugs are revealed by divergences in the sum of output signals. To assist understanding the underlying bug, TransFuzz provides a bidimensional reduction technique that reduces both the hardware complexity and the stimuli duration of the test case.

STIMULI REDUCTION. The first step is reducing the stimuli. Given that the stimuli component of a test case can be long, diminishing the length of the stimuli one by one would be slow. Additionally, lines necessary for a bug to occur may be scattered across the stimuli. Hence, we propose an algorithm inspired from binary exponential backoff to reduce the stimuli dimension. This algorithm finds a local minimum: the removal of any additional line from the reduced stimuli leads to bugs not being observed anymore. Algorithm 4 provides additional details.

Algorithm 4: Binary exponential stimuli reduction. The function `mismatch(A)` returns whether the two simulators mismatch for the current (implicit) hardware for a list A of stimuli lines.

```
   Data: List L of stimuli lines
   Result: Lines K to keep
1  K = ∅, currlineid = 0, currslotsize = 1, growing = 1;
2  while L ≠ ∅ do
3      if mismatch(K :: L[currlineid+currslotsize:]) then
4          growing = 1; L = L[currlineid+currslotsize:];
5          currslotsize = min(2*currslotsize, len(L));
6      else if currslotsize == 1 then
7          growing = 1; K.add(L[currlineid]);
8          L = L[currlineid+1:];
9      else
10         growing = 0; currslotsize = currslotsize/2;
11 return K
```

DESIGN REDUCTION. TransFuzz recycles the cell ordering recorded when building the circuit, and reduces subnets through two binary search phases. In the first phase, TransFuzz attempts to remove cells on the output side. Reducing the cells closest to the output is done as follows. For a candidate

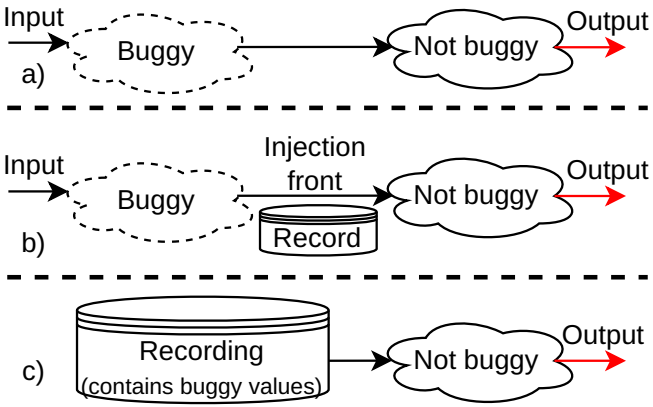

FIGURE 5.10: Removal of macrocells on the input side, where the bug is in on the input side. Dashed-line macrocells are attempted to be removed. The red arrows represent divergent outputs. (a) The original design. (b) Recording the wire values on the injection front. (c) The output may still be buggy if the buggy effect is contained in the recorded values.

number of cells on the output side, TransFuzz attempts to remove them and checks whether the divergence persists. If it does, the removal was legitimate, and the candidate cells are removed permanently. Else, the removal was excessive, so TransFuzz restores the candidate cells and reduces the number of candidate cells for removal. This process continues until no cell can be removed anymore on the output side.

In the second phase, TransFuzz removes the cells closest to the input, and must thereby preserve the values supplied to the remaining cells. We call *injection front* the set of output wires of the candidate cells to remove on the input side, that are connected to at least one cell that will be preserved, as illustrated in Figure 5.10 (b). We enable tracing in the simulator to record the subsequent valuations of the injection front, so that correct input can be provided to the macrocells that are connected to the injection front.

When removing the initial cells and replacing the injection front values with the recorded values, the divergence may either persist or disappear. If the divergence disappears, then the cell removal was excessive, so TransFuzz restores the candidate cells and reduces the number of candidate cells for

removal. Else (if the divergence persists), it is not yet clear whether the removal was legitimate. In that case, it is necessary to check whether some buggy behavior is present in the recorded injection front as illustrated in Figure 5.10 (c). To determine if such an effect occurred, we leverage external differential fuzzing to validate the removal. Once the decision is made, the input-side cell removal process continues until no cell can be removed anymore.

5.7 EVALUATION

We evaluate TransFuzz in terms of raw performance (Section 5.7.1) and cell output coverage (Section 5.7.2). From these measurements, we deduce an optimal duration for the stimuli after which the circuit must be regenerated (Section 5.7.3), and find circuit sizes for optimal differential fuzzing performance (Section 5.7.4). We finally describe the 31 new bugs found by TransFuzz in Verilator, Icarus Verilog, CXXRTL and Yosys and evaluate the time to find each one of them (Section 5.7.5).

EVALUATION SETTING. We obtain the performance results on a machine equipped with two AMD EPYC 7H12 processors at 2.6 GHz with 256 logical cores and 1 TB of DRAM. For measuring performance, we use the following tool versions: `Verilator 5.021 g6b8531f0a`, `Icarus Verilog g77d7f0b8f` and `Yosys/CXXRTL 0.37+21 3d9e44d18` with all optimizations enabled and tracing disabled by default. TransFuzz is implemented as roughly 5000 lines of Python and 1000 lines of C++ code.

5.7.1 *Raw performance*

We evaluate the performance of TransFuzz in terms of generation, elaboration and execution of test cases.

METHODOLOGY. To evaluate the performance of the test case generation and build, we generate 1000 test cases for each circuit size and average the resulting performance. To measure the execution time per stimulus, we measure the average time to execute 1000 stimuli on each of the 1000 test cases and obtain an average execution time per stimulus.

RESULTS. We summarize the results in Figure 5.11. The generation time is generally small compared to the build time. Different simulators have vastly different behaviors. Yosys and Icarus Verilog do not build an executable

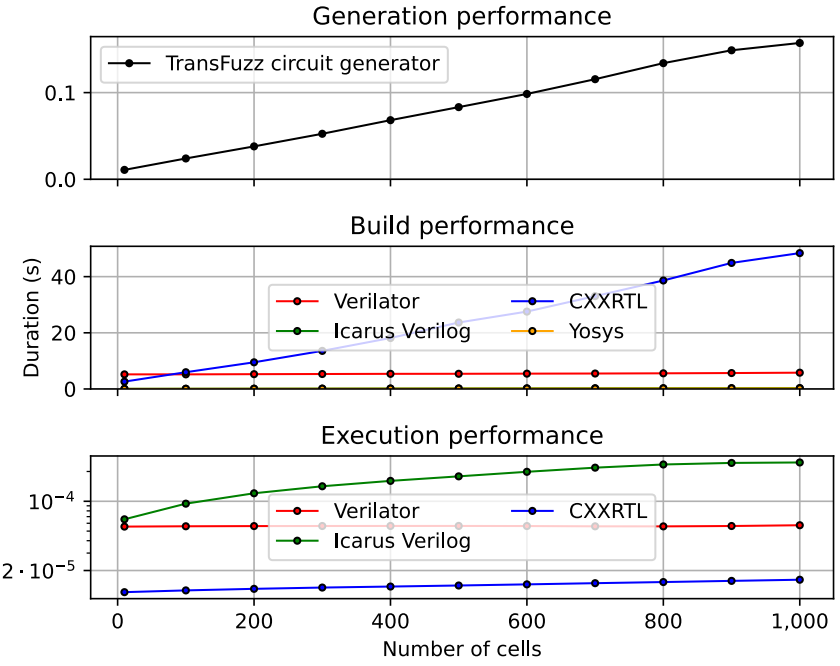

FIGURE 5.11: Raw performance evaluation. Generation represents the construction of the test case by TransFuzz. Build represents the process that transforms TransFuzz's test case representation until the simulation model. Note the logarithmic scale on the execution plot, which represents a single stimulus execution (averaged over 1000 stimuli).

simulation model (Yosys is a synthesizer and Icarus is an interpreter), hence yield faster build times. Verilator's build time is remarkably flat over circuit sizes. Note that this is not a general benchmark of the three simulators, as the inputs are not typical simulator inputs. We will show how these results impact the overall performance of TransFuzz in Section 5.7.3.

5.7.2 *Cell output toggling*

Intuitively, fuzzing the same circuit with more stimuli brings diminishing returns as increasingly less new cell behaviors will be explored over time.

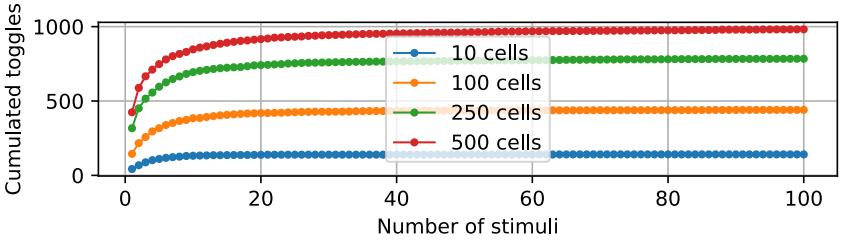

FIGURE 5.12: Cumulated toggle coverage of cell outputs.

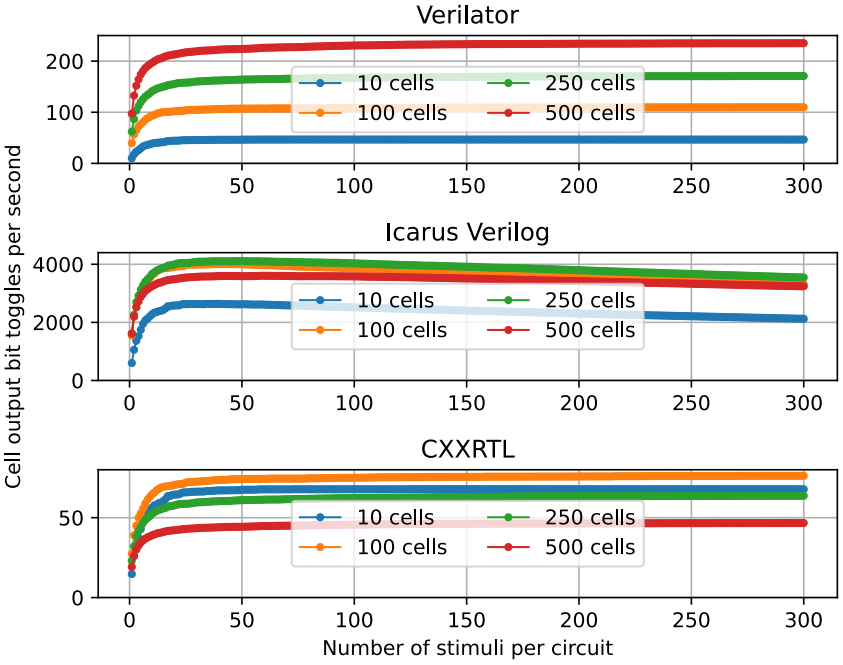

FIGURE 5.13: Toggle performance for each simulator in function of simulation length and circuit size.

METHODOLOGY. We measure the effectiveness of inputs over time by measuring the bit toggles at the cells' outputs, which reflect the different

behaviors of the cells being explored. This measurement is independent of the target EDA software. We execute 1000 test cases for each circuit size over 1000 stimuli and measure the average cell output toggle coverage by analyzing the execution traces over time.

RESULTS. We summarize the results in Figure 5.12. The toggle coverage increases rapidly for the first tens of stimuli, and then the rate of progress decreases, in accordance with the intuition of diminishing coverage returns for increasing stimuli lengths. The increase in coverage past 100 stimuli, not illustrated in the Figure, is relatively small. From 100 to 1000 stimuli, the coverage increases by less than 3.3% for all circuit sizes. Toggle coverage also increases sublinearly with the circuit size. Next, we combine this coverage with raw performance to converge on the goodput of TransFuzz over arbitrary stimuli and circuit sizes.

5.7.3 *Stimuli's length*

We intend to set the parameters of TransFuzz in a region that will maximize its efficiency. Under the hypothesis that strong toggle coverage increases indicate effective fuzzing, we want to maximize the number of cell output toggles per second to maximize the effective performance of the fuzzer. The first question that we address is how many stimuli to execute per circuit. Concretely, given the diminishing returns of the stimuli shown in Section 5.7.2, we expect that after a certain number of stimuli, it will be more advantageous to start a new circuit by paying the fixed generation and build costs again, instead of fuzzing the same circuit further.

METHODOLOGY. For each simulator, we calculate the average number of cell output toggles achievable per second for circuit size and stimuli duration. For a fixed circuit size, this value is given by Equation 5.1, where `prep_time` is the average time to produce a circuit of a given size, i.e., generation and build times, and `exec_time` is the average time to execute a stimulus for this size, as measured in Section 5.7.1, and `cumul_toggles` is illustrated in Figure 5.12. The simulation length (`simlen`) that maximizes the toggle performance is the number of stimuli after which to regenerate a fresh circuit.

$$\text{toggles}_{/sec}(\text{simlen}) = \frac{\text{cumul\_toggles}(\text{simlen})}{\text{prep\_time} + \text{exec\_time} \cdot \text{simlen}} \tag{5.1}$$

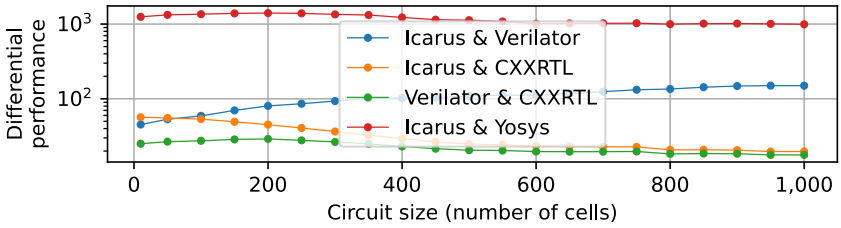

FIGURE 5.14: Differential toggle performance. Maximizing this abstract metric maximizes the performance (cell output toggles per second) for differential fuzzing.

RESULTS. We summarize the results in Figure 5.12. Icarus Verilog's performance declines after tens of cells, while the other simulators' performance stabilizes at this point to increase by less than 5% to reach peaks for simulation lengths between 124 and 2420 stimuli before decreasing. We make three observations: first, always renewing the test case after a single stimulus would be inefficient. Second, the diversity in the raw performance detailed in Section 5.7.1 translates into diversity in the toggle performance. Icarus Verilog, whose execution cost per stimuli is the highest, meets its peak throughput at 38 (10 cells) to 52 (250 cells) stimuli per circuit, and is the most efficient to fuzz in absolute numbers. Verilator is the most efficient when it comes to larger circuits because of its flat but high build cost for small circuits. Finally, after around 20 stimuli, the performance is remarkably stable in function of the stimuli duration. We choose a simulation length of 70, which is in bounds with 95% of the peak performance in all configurations. With optimal stimuli size per simulator known for different circuit sizes, we measure the optimal selection of circuit sizes for differential fuzzing.

5.7.4 *Circuit size for differential fuzzing*

When differentially fuzzing two simulators, we must choose a circuit size that maximizes the total performance given the simulators that are involved.

METHODOLOGY. We aim to maximize the metric $m = (P_a \cdot P_b)/(P_a + P_b)$, for P_a and P_b the respective simulator toggle performances. We hence

measure m for different circuit sizes and a simulation length of 70 stimuli (as measured in Section 5.7.3).

RESULTS. Figure 5.14 shows the performance of differential fuzzing for each pair of simulators. These results show that we should fuzz Verilator against Icarus Verilog for circuits of 800 to 1000 cells, Icarus against CXXRTL for 100 cells or less, and Yosys against Icarus for approximately 200 cells. Interestingly, this advocates for separate fuzzing circuit sizes for each differential pair, instead of fuzzing all simulators alike. To the best of our knowledge, in the domain of differential fuzzing, this is a unique finding.

5.7.5 *Discovered bugs*

TransFuzz discovered 31 new bugs in 4 popular EDA applications. Table 5.2 and Table 5.3 summarizes the discovered bugs and highlights the 20 discovered translation bugs. We first analyze the new bugs, then evaluate TransFuzz's performance in finding them.

Bug descriptions

TRANSLATION BUGS. TransFuzz found translation bugs in all four EDA applications. Some occur in particularly complex cases. For example, I2 is the first reported translation bug in Icarus for at least 3 years, corresponding to the last 100 relevant bug reports analyzed in Section 5.5.1. The mistranslation I2 lets some arithmetic operations mistakenly evaluate the output when supplied with inputs which are specific patterns of concatenations of specific constant bits and X (unknown) bits. A specific instance is illustrated in Figure 5.16 (b) of Section 5.8. With the multiple concatenations, this bug particularly benefits from TransFuzz's netlist-level abstraction, which eases fractioning and concatenating wires.

Verilator and CXXRTL are notably affected by translation bugs, as they seem to have more aggressive optimizations. We observe, indeed, their better runtime execution performance in Section 5.7.1. In total, TransFuzz found respectively 9 and 6 translation bugs in Verilator and CXXRTL, involving a large diversity of cell types (operational diversity) and interconnection patterns (relational diversity). Fixing these bugs has sometimes been challenging. For example, V10 and V12 are similar in their expression, as they can both be expressed through power cells. Yet they affect different internal structures, hence after fixing V10, V12 was still detected by TransFuzz,

Id	Bug Description
I2	Arithmetics deviation in specific circumstances
C1	Shifts consider signed operand in some cases
C2	Bad assignment under specific conditions
C3	Bug with modulo when operands intersect
C4	Both left shifts sometimes overflow the output signal
C5	Incorrect division
C6	Clock edges sometimes require 2 evaluations
V6	Wrong not when checking (n)eq under and/or tree
V7	Wrong simulation result with add and xor gates
V8	Optimization error for 5 optimization types
V9	Sometimes wrong conversion to 32-bit integers
V10	Under some conditions, 0 power 0 gives 0
V11	Evasive compilation-sensitive mis-simulation
V12	Power operator supplied with wide constants
V13	Incorrect bit-op-tree not optimization
V14	Bit-op-tree should not touch some subtrees
V15	Incorrect widthMin in replaceShiftOp
V16	Ignore if eq/ne is under shiftr
Y1	`opt_muxtree` wrong if twice the same mux
Y2	Misoptimization of wide shifts

TABLE 5.2: Translation bug reports for Icarus Verilog (I), CXXRTL (C), Verilator (V) and Yosys (Y).

enabling the eventual of fix this bug regarding representations of wide numbers.

The two translation bugs Y1 and Y2 found in Yosys concern uncommon cell interconnections and internal representations. Y1 arises because of redundant multiplexers in a specific configuration involving signal concatenations ahead. In that case, Y1 mistranslates one of these concatenations by replacing one input with a constant value, while the input is free to toggle. Somehow similar to V10, Y2 misrepresents wide constants in shift operations.

OTHER BUGS. TransFuzz found a segmentation fault in Icarus (I1) and 5 in Verilator (V1-V5). It additionally found exceptions in CXXRTL (C8) and

Id	Bug Description
I1	Segfault when using out-of-scope variable in slices
I3	Performance issue for a design with xor reductions
C7	Performance issue with many concatenations
C8	Elaboration fails for some stateful cells in some cases
V1	Evasive malloc failure in some instances
V2	Segfault during evaluation
V3	Segfault in traceInit
V4	Segfault with many parallel operators
V5	Segfault in Vtop___024root__trace_init_sub__TOP__0
V17	VCD corruption for 5 optimization types
V18	Compiler sees empty input due to file system races

TABLE 5.3: Non-translation bug reports for Icarus Verilog (I), CXXRTL (C), Verilator (V) and Yosys (Y).

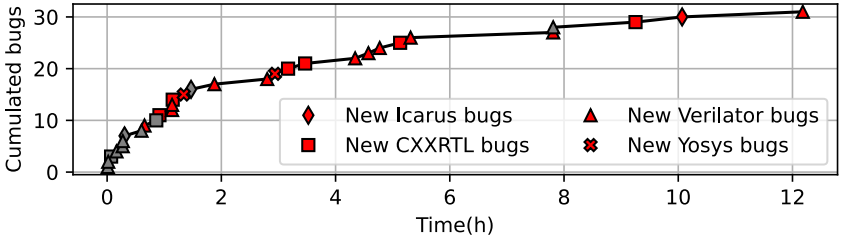

FIGURE 5.15: Cumulated time to bug. Translation bugs are colored in red.

Verilator (V18), as well an issue that eventually causes tracing issues (V17). Finally, TransFuzz found two confirmed performance bugs (C7 and I3). The former is sensitive to circuit depth and would require splitting operations into smaller pieces for speeding up. The latter happens with concatenations of many inputs, which, themselves, share bits. The evaluation time was exponential in the number of shared bits. This bug I3 has been fixed by deferring their evaluation to the end of the evaluation call.

CVES. Given the security relevance of the discovered bugs, 25 CVEs were assigned, as listed in Table 5.4 and Table 5.5.

Id	CVE	Severity
I2	CVE-2024-25471	7.1
C1	CVE-2024-26522	7.1
C2	CVE-2024-28720	7.5
C3	CVE-2024-25472	7.1
C4	-	7.1
C5	CVE-2024-28719	7.1
C6	CVE-2024-25478	7.5
V6	CVE-2024-25493	7.1
V7	CVE-2024-25485	7.1
V8	-	-
V9	CVE-2024-25486	7.1
V10	CVE-2024-25488	7.1
V11	CVE-2024-25491	7.1
V12	CVE-2024-25490	7.1
V13	CVE-2024-28721	7.1
V14	CVE-2024-25489	7.1
V15	CVE-2024-25492	7.1
V16	CVE-2024-25495	7.1
Y1	CVE-2024-25479	7.1
Y2	CVE-2024-25477	7.1

TABLE 5.4: CVEs assigned for EDA software translation bugs found by TransFuzz.

Time to bug discovery

We fuzz each pair Verilator/Icarus Verilog, Icarus Verilog/CXXRTL and Icarus Verilog/Yosys on 128 processes for 24 hours and summarize the time to discover each bug in Figure 5.15. We note that generally, translation bugs require more time to be discovered than other bugs.

5.8 EXPLOITATION

To enable the exploitation of translation bugs, we introduce the MiRTL gadget, a primitive that produces predictable wrong values when processed by a given EDA application. Based on such gadgets, we build an attack that

Id	CVE	Severity
I1	CVE-2024-25470	7.1
I3	-	-
C7	-	-
C8	-	-
V1	CVE-2024-25481	7.5
V2	CVE-2024-25480	7.5
V3	CVE-2024-25482	7.5
V4	-	-
V5	CVE-2024-25484	7.5
V17	CVE-2024-25494	7.5
V18	CVE-2024-25496	7.1

TABLE 5.5: CVEs assigned for EDA software non-translation bugs found by TransFuzz.

introduces a kernel data leakage vulnerability in the CVA6 RISC-V CPU, that exposes supervisor data to user-mode processes. A standard mitigation for such an attack is information flow tracking which we show that MiRTL attacks can also bypass.

5.8.1 *MiRTL gadgets*

We build MiRTL gadgets to produce `1` under normal circumstances, but produce `0` when exploiting a translation bug. We formulate the following requirements for such gadgets:

- *Non-intrusiveness.* A gadget must not alter the functionality of the design under normal circumstances.
- *Low footprint.* Minimize area, power and timing.
- *Silence.* The gadget should not trigger warnings.

INDIVIDUAL SIMULATOR GADGETS. From the bugs discovered by TransFuzz, we construct a suitable gadget for each simulator illustrated in Figure 5.16 (a)-(d), respectively building on the translation bugs V6, I2, C1.

COMPATIBILITY. To build a gadget compatible with all three simulators, we connect the three gadgets as the three inputs of a new 3-input `and`. This

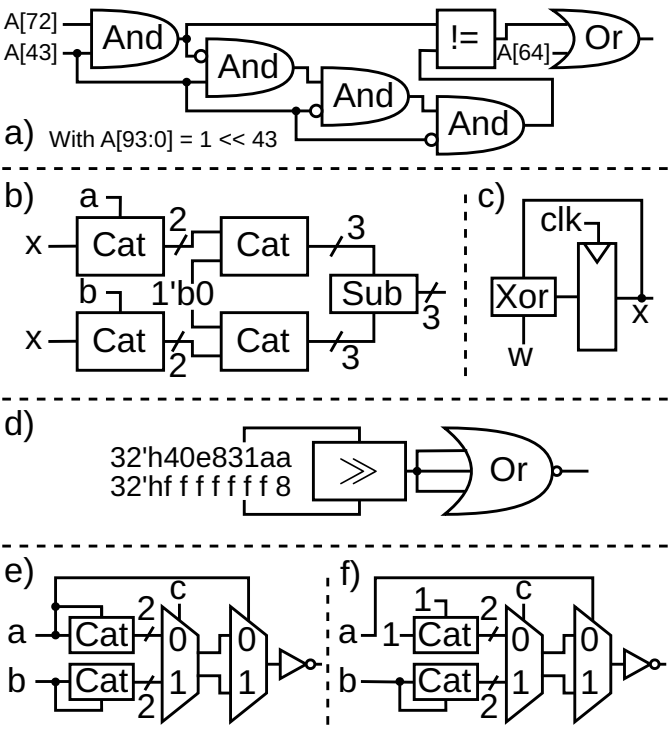

FIGURE 5.16: MiRTL gadgets. `Cat` are concatenations. (a) Verilator gadget (V6). (b) Icarus Verilog gadget (I2). (c) Primitive for X injection, useful for the Icarus gadget. (d) CXXRTL gadget (C1). (e) Yosys gadget (Y1). (f) Yosys gadget once mistranslated.

way, the gadget will produce the mistranslated `0` value if any of the three simulators is used.

SYNTHESIZER GADGET. Figure 5.16 shows a gadget for the Yosys synthesizer (e), and how it is mistranslated during the default `opt_muxtree` Yosys optimization pass (f). By setting `a=c=1'b0`, the gadget will mistakenly produce an output value of `0` in the synthesized version of the design.

HEAD REGISTER. The MiRTL gadgets take specific constants as inputs, however these constants cannot be hardcoded directly in the design, as gadgets usually require the input constants to be unknown at elaboration or

```
daccess_err = en_ld_st_translation_i && ((ld_st_priv_lvl_i == S && !sum_i
        && dtlb_pte_q.u)  (ld_st_priv_lvl_i == U && !dtlb_pte_q.u));
```

LISTING 5.1: User-accessible bit check in CVA6's MMU.

synthesis time. Instead, we find that inserting a register between the input constant and the gadget is sufficient to make the gadget work in all cases, as the EDA software does not see a fixed input value. This register takes X as the initial value by default and uses the design's global clock signal.

TAIL REGISTER. MiRTL gadgets are followed by some logic. It may happen that the end of the gadget gets optimized together with this logic before the MiRTL trait appears, which would disarm the gadget. Another issue of the following logic is that together with the gadget, it may form a critical path. Therefore, we terminate the gadget with a flip-flop to mitigate these side effects. This flip-flop is also connected to some global clock signal usually already present in the design.

5.8.2 *Kernel memory access exploit*

To show the practicality of MiRTL gadgets, we inject a security vulnerability into the CVA6 RISC-V CPU [74], also known as Ariane, that allows an attacker in user mode to read the content of kernel memory (CWE-118).

BACKGROUND. In the Sv39 virtual memory system specified by RISC-V and supported by CVA6, the bottom-most bits of a page table entry are, in increasing index order, V: valid, R: readable, W: writable, X: executable, U: user accessible. When virtual memory is enabled, user software attempting to access a page with $U = 0$ will trigger an exception.

Strategy

We aim to alter the functionality of CVA6 to allow user software to load data from a page with attribute $U = 1$.

GADGET INSERTION LOCATION. We first identify the exact location where the U bit is checked. We find that this check is, unsurprisingly, located in the `i_cva6_mmu` module instance in the load-store unit. We report

	Area	Power	Registers	Timing
Scenario 1	+0.06%	−0.07%	+0.01%	+0%
Scenario 2	+0%	+0%	+0%	+0%

TABLE 5.6: Design metrics deviations caused by gadget insertion.

the exact check operation in Listing 5.1 for data accesses. We propose to `and` the gadget output bit together with the `!dtlb_pte_q.u` signal.

SCENARIO 1: EXPLOITING SIMULATOR TRANSLATION BUGS. In this scenario, the adversary injects the combined simulator gadget into the design at the specified location. We propose to insert the gadget by inserting the following code: `&& gadget_out` after `!dtlb_pte_q.u`.

SCENARIO 2: EXPLOITING SYNTHESIZER TRANSLATION BUGS. Same as Scenario 1 except with the synthesizer gadget.

Evaluation

We first evaluate the functional correctness of the exploit. We then evaluate its impact in terms of area, power and timing.

FUNCTIONAL CORRECTNESS. To validate the correctness of both scenarios, we design a simple compliance test including a basic operating system taken from the RISC-V compliance test suite [156], where we integrate an user application attempting to read from a page with $U = 0$. Indeed, all simulators see exceptions. To ascertain that the synthesized design contains the exploit, we simulate the synthesis output with Verilator with all default optimizations off, and indeed the user access to kernel data sees no exception. By successfully executing the RISC-V compliance tests [156], we ascertain that the overall CPU functionality is unaffected.

AREA, POWER AND TIMING. To evaluate the impact of the exploit on design metrics, we synthesize Yosys's output in a popular 12nm technology using Synopsys Design Compiler 2022.03, targeting a 1 GHz clock. We summarize the deviations with the unaffected CVA6 design in Table 5.6. In particular, any variation in timing would be problematic for the attacker, as it would make the gadget obvious. We observe that all these MiRTL exploits negligibly impact all metrics.

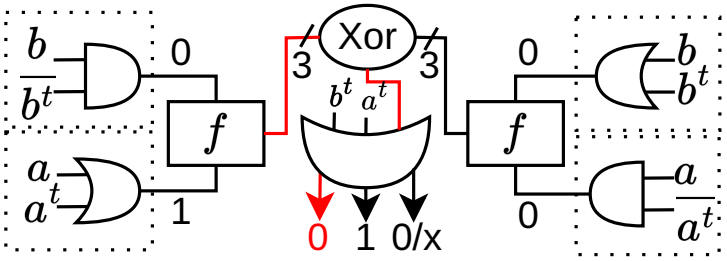

FIGURE 5.17: Cell-level shadow logic for the Icarus gadget, for $a = 1, b = 0, a^t = 1, b^t = 0$. The expected output is $Y^t =$ 0b110, but it becomes mistranslated to $Y^t =$ 0b010. We illustrate in red the propagation of the mistaken value. f is the MiRTL gadget illustrated in Figure 5.16 (b).

STEALTHINESS. It is beneficial for gadgets to be discreet in front of code inspection. Typical gadgets require around 10 lines of Verilog code. While this is few lines, they can be intertwined into the design sources in a case-by-case basis to look innocuous. If the design is programmed in a hardware construction language (HCL) [214, 230–232] (then automatically compiled into Verilog), a gadget inserted into the Verilog representation, e.g., through a malicious HCL compiler, is unlikely to be obvious in the complex Verilog output.

5.8.3 *Bypassing information flow tracking*

A common countermeasure against such leakage is hardware Dynamic Information Flow Tracking (DIFT) [54, 131, 143, 249–251]. The state-of-the-art hardware DIFT mechanism that scales to such a non-trivial CPU operates at the granularity of macrocells and uses cell replication [54]. Based on MiRTL gadgets, we build *shadow gadgets* that maliciously corrupt information flow tracking logic, concealing confidentiality breach flags that could otherwise reveal the MiRTL attack.

The challenge is to generate logic that, when instrumented, will generate a MiRTL gadget in the shadow logic. This is the converse of usual information flow tracking construction [54, 131, 229], where from a design, the goal is to generate efficient shadow logic. To the best of our knowledge, this is the first time that this converse direction of the problem is explored.

We rely on the insight that cell-level information flow tracking is fundamentally based on cell replication [54]. Consider the previously described Icarus gadget illustrated in Figure 5.16 (b). By cell replication, we expect some shadow logic to share a construct similar to this gadget. For this gadget, a cell-level shadow logic indeed contains a gadget replica, as we illustrate in Figure 5.17. We observe that for some taint inputs described in the figure, the gadget will pull down an information flow signal bit illegitimately, violating the crucial assumption of absence of false negatives, concealing confidentiality breaches.

5.9 DISCUSSION

MITIGATIONS. First, the exploitation of individual bugs can be mitigated by fixing them in the affected EDA software. While we encourage further fuzzing work against EDA software, it is yet unclear how to guarantee bug-freedom of simulators. Second, we observe that most previously known as well as current bugs require operational diversity. Hence, one mitigation for most known and unknown bugs is to simulate the design at the gate level, although this is known to be significantly slower than simulating at a higher level [54, 131], and the translation to this lower level is also error-prone, e.g., through the bugs Y1 and Y2 found by TransFuzz. Finally, the exploitation of synthesizer bugs can be mitigated by dynamically or formally checking the equivalence of the synthesized design with the original design [252]. This last mitigation is similar to the known problem of Trojan detection [235–242].

LIMITATIONS OF MIRTL. TransFuzz uses the Yosys Verilog backend to generate the final design representation. This backend does not produce behavioral blocks, and has its own, yet flexible, output structure. Hence, TransFuzz is currently unable to find some syntax-dependent bugs in simulators, where a bug is not only dependent on a design, but on the specific syntactic way of describing it. One future approach to overcome this limitation would be to design a dedicated backend that also generates behavioral Verilog and adds even more freedom in the syntax.

COMMERCIAL VS. OPEN-SOURCE EDA ENVIRONMENT. Due to licensing reasons, fuzzing and publicizing bugs in commercial EDA software is generally not accepted. However, open-source EDA software has never been more popular. For example, on the ASIC fabrication side, Yosys is used, among others, by qflow [253] and the OpenLANE flow [254], whose results

are becoming increasingly competitive with commercial flows [255]. On the FPGA side, Yosys is used by popular flows such as SymbiFlow [256] and PRGA [257]. On the simulation side, open-source simulators are nowadays used for complex SoCs [258] and diverse designs such as BLE MAC [259].

FUTURE WORK. TransFuzz is the first attempt to find bugs in simulators beyond submitting stimuli-less language-level hardware designs, and hence paves the way for more work in this direction.

5.10 RELATED WORK

We first cover testing attempts specifically built for EDA applications. We then cover other work that aims at compromising pre-silicon hardware designs. Finally, we discuss the related direction of compiler fuzzing.

FUZZING SYNTHESIZERS AND SIMULATORS. In 2013, Yosys contributors proposed VlogHammer [244], a tool for testing Verilog synthesizers by submitting complex handwritten and generated expressions. VeriFuzz [245] is a more recent proposal that supports behavioral constructs and undefined behaviors, and allows more freedom in source structure. Both approaches operate at the language level and focus on testing synthesizers by only submitting hardware designs, and no stimuli. By operating at cell level, TransFuzz is able to produce more complex valid designs. Reduction is also made easier as TransFuzz produces smaller valid designs with shorter stimuli sequences. While previous work only produced the hardware component, TransFuzz produces both hardware and stimuli, which makes testing simulators significantly more efficient, as we have shown in Section 5.7.3.

On the other hand, from github issues [243], we noted that there have been some campaigns to fuzz simulators with traditional approach of manipulating the contents of the HDL files. While this approach is able to find crashes in the simulator frontend, it is unlikely to find bugs in the simulator backend where the translation bugs may lurk.

COMPROMISING PRE-SILICON HARDWARE DESIGNS. With a similar threat model, the insertion and detection of hardware Trojans in pre-silicon hardware designs have been studied in the past [235–242]. While translation bugs can be combined with hardware Trojans, they are fundamentally different, as MiRTL attacks are invisible to the affected EDA software. Additionally, there is no notion of trigger and payload in the exploitation scenarios we presented in Section 5.8.

COMPILER FUZZING AND BUGS. Compilers share similarities with synthesizers. The confused deputy problem was initially demonstrated in a compiler and some backdoors were demonstrated using some of their bugs [260–263]. Previous work advertises multi-variant execution against such attacks [264], but we show the limitation of this proposal by compromising three simulators at a time in Section 5.8. Compiler fuzzing has been shown to be effective at finding bugs in compilers [265–270], mostly by relying on differential fuzzing [226–228]. Yet fuzzing and differential testing are vastly different in the context of EDA applications. We show in Section 5.7 that inputs of non-trivial lengths boost performance, while compiler outputs are generally executed only once [268, 269]. Additionally, fuzzing with software programs shares no clear similarity with fuzzing with hardware circuits.

5.11 CONCLUSION

MiRTL is a new class of confused deputy attacks on EDA software that relies on translation bugs. We presented TransFuzz, the first fuzzer that is dedicated to finding such translation bugs in simulators and synthesizers using pairs of hardware designs and stimuli. By creating netlists of macrocells, TransFuzz creates particularly complex hardware designs that are easy to stimulate and reduce. In addition to some new unsafe crashes, TransFuzz found 20 new translation bugs causing wrong runtime values in three major open-source RTL simulators and in the Yosys synthesizer, among the 30 new bugs that it found. We showed concrete MiRTL attacks that exploit these translation bugs by corrupting the behavior of the CVA6 CPU to leak kernel memory to user mode and bypassing the information flow tracking mitigation. Facing these newly demonstrated security risks based on yet undiscovered EDA bugs, we encourage further research in the direction of discovering and fixing bugs in EDA software.

ETHICAL CONSIDERATIONS. We reported all bugs to their respective maintainers and provided support when required.

6

CONCLUSION AND OUTLOOK

In this dissertation, we looked at the existing CPU bugs and several potential adaptations of software security techniques to find and fix bugs in hardware. To perform this work, we benefitted from the novel abundance of available information, be it microprocessor errata, open-source hardware designs, and open-source EDA software. In particular, we have shown that *software security techniques are extremely promising for finding and fixing bugs in hardware, but they require a careful adaptation to be effective*.

Over the chapters, we have explored the adaptability of some relevant software security techniques to hardware, and we have conducted adaptations that vastly outperformed the state of the art. Chapter by chapter, we will outline the contributions of the work that we conducted and opportunities for future work.

REMEMBERR. In Chapter 2, by observing thousands of contemporary CPU errata, we designed an unprecedented classification of CPU bugs based on triggers, contexts, and observable effects to provide a human-interpretable overview of the state of the art in terms of CPU bugs that plague our machines. While the errata have traditionally been expressed in natural language, we translate all these errata to a machine-readable format in compliance with this new errata classification. Benefitting from this unprecedented database, we identify some promising research directions.

> **Research Question for Future Work 1.** How do bugs in open-source hardware compare to bugs in commercial CPUs, and how do the root causes correlate with triggers, contexts, and observable effects?

This first new research question is a natural continuation of the work that we conducted in Chapter 2 and expands the usage scope of the *RembembERR* database. While commercial CPUs are often far more complex than currently available open-source hardware, this gap is expected to progressively close, with open-source designs becoming increasingly complex, and the community design and validation practices becoming increasingly mature. A comparison between bugs present in commercial CPUs and open-source

hardware would provide an interesting indicator of the maturity of the open-source hardware community, and hopefully, this database will help prevent the same errors from being reproduced again in new designs. Finding correlations between the root causes of bugs and their triggers, contexts, and observable effects has been hard so far because errata do generally not specify a root cause. Analyzing the root causes of bugs in open-source hardware would provide a unique opportunity to find such correlations.

Research Question for Future Work 2. How to use such a database to detect new CPU bugs?

The recent emergence of machine learning and its well-known applications in various domains has been relying on the abundance of data to learn from. Before the constitution of the *RemembERR* database, such structured data was not available, hampering research in this direction. Hence, the *RemembERR* database opens exciting research directions in bug detection, for instance by allowing training machine learning models to predict the presence of a bug in a CPU design or suggest triggers and contexts that are likely to trigger a bug and propose observing specific effects that would betray the presence of a bug. This direction is particularly compelling given that the open-source hardware community, which will soon produce designs that are as complex as commercial CPUs of the last decade, will strongly benefit from new techniques to find and fix bugs in their designs that rely on the knowledge of mistakes made in the past.

CELLIFT. In Chapter 3, we presented *CellIFT*, the first hardware dynamic information flow tracking mechanism that scales to complex CPUs and SoCs. *CellIFT* relies on the insight that operating at the macro-cell level enables exploiting mathematical properties that boost scalability and precision at no cost. We presented new applications for dynamic information flow tracking in hardware: identification of leakage-prone components, detection of architectural bugs that are traditionally hard to detect, and detection of occurrences of Meltdown-type bugs [2] and Spectre-type vulnerabilities [3]. We envision broader applications of *CellIFT* in the near future.

Research Question for Future Work 3. Can CellIFT be used for formal analysis of data-independent constant-time execution?

So far, we have been using *CellIFT* in a dynamic setting. A natural next step is to use *CellIFT* in a formal setting. One major property of hardware

dynamic information flow tracking is its agnosticism to the wires being data or control signals. Hence, observing the dependence of control signals on data signals is a natural extension of the work that we conducted in Chapter 3. Yet this comes with challenges, especially regarding CPUs, which are complex and tightly self-looping. Indeed, it is expected that data can influence control signals, for example in branches. To make such a formal analysis feasible, it then appears necessary to tackle this challenge by filtering out legitimate information flows, while avoiding false negatives.

Research Question for Future Work 4. Can dynamic information flow tracking be effectively used for guiding CPU fuzzers?

We have observed in the past few years, in software security, usage of dynamic information flow tracking to guide fuzzers [185, 188]. One major double-edged sword when applying dynamic information flow tracking on CPUs is the tight interconnection of CPU components and high sensitivity to timing variations. Intuitively, any tainted value that could create a timing variation that will propagate to control signals will taint the CPU almost entirely in very few cycles. This becomes a challenge when guiding fuzzers, as tainting slightly too much will, very few cycles later, taint catastrophically and hence the fuzzer will not be able to explore the program space. There is hence an immense opportunity, but also a challenge in taming this overtainting effect.

CASCADE. In Chapter 4, we presented *Cascade*, a black-box RISC-V CPU fuzzer that discovered 28 new CVEs across several significant open-source CPUs. *Cascade* operates by generating long and intricate yet valid programs, where the data flow and control flow are closely intertwined to exert maximum pressure on the CPU being tested. It then observes whether the CPU successfully completes the program, and if not, it identifies a bug and can automatically reduce the program to a minimal reproducer consisting of a few instructions. Due to its absence of reliance on instrumentation or coverage information, *Cascade* achieves fast execution and outperforms existing coverage-guided CPU fuzzers in terms of coverage, as measured on their own coverage metric. A natural question is, then, what would be an effective coverage-guided CPU fuzzer?

Research Question for Future Work 5. How to build a coverage-guided CPU fuzzer that outperforms a black-box fuzzer?

Seeing a black-box fuzzer outperforming the existing coverage-guided CPU fuzzers questions both the relevance of the existing coverage metrics and coverage-guided fuzzers that reliy on these metrics. The observation that we made is that the coverage metrics used so far seem to have too many coverage points, and these coverage points are not necessarily relevant to the detection of bugs. The first step in building a coverage-guided CPU fuzzer that outperforms a black-box fuzzer would be to identify a new coverage metric that is more correlated with bug detections, and that is not satisfied by continuously running new complex programs in an unguided way. *Cascade* may provide a way for analyzing the relevance of such coverage metrics by correlating the occurrence of some bugs with the prevalence of some candidate coverage metric. After finding a new candidate metric, one could see, for the first time, a coverage-guided CPU fuzzer that outperforms a black-box fuzzer.

Research Question for Future Work 6. How to build the remaining RISC-V functionalities into *Cascade*?

Cascade supports most of the unprivileged RISC-V specification, and a portion of the privileged specification, for instance privilege levels, exceptions and delegation. Yet there are some remaining challenges to make *Cascade* more complete. In particular, we would envision supporting compressed instructions as well as memory virtualization and physical memory protection in a heavily randomized way. Given the construction of *Cascade*, memory virtualization would require new abstractions to keep complying with the *Cascade* philosophy of constructing long, complex and valid programs. It is yet unclear, for example, how to make the predictable control flow of *Cascade* comply with varying locations in a virtual address space, and how to architect as complex virtualization operations as possible, while maintaining the validity and predictability of the generated programs.

LOST IN TRANSLATION. In Chapter 5, we presented *TransFuzz*, a new fuzzer that generates randomized RTL designs made of complex interconnections of diverse operators to trigger translation bugs in RTL simulators and synthesizers. We introduced differential fuzzing to detect translation bugs and reduced the translation bugs to small reproducer designs. Based on the many translation bugs found by *TransFuzz*, we introduced the new concept of *MiRTL* gadgets that encapsulate a translation bug and showed how to exploit these gadgets to inject malicious hardware into a seemingly benign RTL design. *TransFuzz* pioneered fuzzing using hardware descrip-

tions and stimuli for finding bugs in synthesizers and simulators. A myriad of opportunities for future research arises from this work.

Research Question for Future Work 7. How to co-generate netlists and stimuli that will maximize the chance of finding translation bugs in simulators and synthesizers?

So far, *TransFuzz* is designed to first generate an RTL netlist, and then a sequence of random stimuli of a length that is statistically optimal. Probably, a co-generation of netlists and stimuli could boost the effectiveness of *TransFuzz*. This would typically imply trading off some randomness on the circuit and stimuli to maximize measures such as output bit toggle coverage. The main challenge in this approach, besides finding properties for the circuit and stimuli that maximize some measurement, seem to be to ensure that the reduction in randomness does not exclude constructs that could be mistranslated.

Research Question for Future Work 8. How to fuzz synthesizers and simulators in a coverage-based manner?

Taking coverage feedback would be a natural extension of the work that we conducted with *TransFuzz*, which is so far a black-box fuzzer. The setting is challenging as there are two targets being fuzzed one after the other. The first target is the synthesizer or simulator, which will create an output hardware description or simulation model. The second target is this hardware description or simulation model. There are, hence, two dimensions in the coverage feedback to take into account. To the best of our knowledge, this is an unprecedented, exciting challenge.

Research Question for Future Work 9. How to formally protect against MiRTL attacks?

There seems to be two promising ways to protect against MiRTL attacks. The first is to have a formally verified synthesizer or simulator, that is guaranteed to not mistranslate any input. The second is to have a scalable way of formally comparing the input and the output of the synthesizer or simulator. While this may be feasible for synthesizers, there is not yet any representation of intermediate understanding of the simulation model that would be amenable to this equivalence check. Some simulators are capable

of dumping intermediate abstract trees, which may provide an interesting starting point for the equivalence check.

Research Question for Future Work 10. How to test more types of EDA software?

While *TransFuzz* focused on simulators and synthesizers, many other kinds of EDA software could benefit from automatic testing. We could envision, for example, testing place-and-route or static timing analysis software. Such a testing framework could be based on the same principles as *TransFuzz*: first generating a complex design, potentially suited to the known weak aspects of the software under test, and then observing whether the software under test produces a correct output by testing it under differential fuzzing, either between two configurations of the software or between two different software, or between the original netlist and the software output. Such automatic approaches are compelling because debugging a test design will be much easier than debugging an actual design, whose mistranslation may not be visible until the silicon becomes used on a large scale. Such potential bugs will certainly enable exploits similar to the *MiRTL* attacks that we presented in Chapter 5 and based on similar gadgets.

CONCLUSION. In this dissertation, we first examined thousands of CPU bug descriptions and deduced two candidate approaches borrowed from software security for finding security-relevant bugs in hardware: dynamic information flow tracking and fuzzing. We created the first dynamic information flow tracking mechanism for hardware that scales to complex designs such as CPUs and SoCs, opening unprecedented capabilities for finding information leakage, architectural and microarchitectural bugs and security vulnerabilities. We also created a black-box CPU fuzzer that found dozens of new security vulnerabilities and reaches more coverage, faster than coverage-guided fuzzers. Finally, we introduced a new class of attacks targeting EDA software, and a new fuzzer to find bugs that permit such attacks, before malicious actors do. With *RemembERR*, *CellIFT*, *Cascade*, and *TransFuzz*, we have been providing ready-to-use and effective security solutions for increasing the security of hardware designs in an era where hardware flourishes, and we have opened a myriad of opportunities for future work in this young, exciting yet critical field.

BIBLIOGRAPHY

[1] M. Tiwari, H. M. G. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood. "Complete information flow tracking from the gates up". In: *ASPLOS*. 2009.

[2] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. "Meltdown: Reading kernel memory from user space". In: *USENIX SEC*. 2018.

[3] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, et al. "Spectre attacks: Exploiting speculative execution". In: *IEEE SP*. 2019.

[4] R. Kande, A. Crump, G. Persyn, P. Jauernig, A. R. Sadeghi, A. Tyagi, and J. Rajendran. "{TheHuzz}: Instruction Fuzzing of Processors Using {Golden-Reference} Models for Finding {Software-Exploitable} Vulnerabilities". In: *USENIX SEC*. 2022.

[5] Jaewon Hur, Suhwan Song, Dongup Kwon, Eunjin Baek, Jangwoo Kim, and Byoungyoung Lee. "Difuzzrtl: Differential fuzz testing to find cpu bugs". In: *IEEE SP*. 2021.

[6] N. Bruns, V. Herdt, E. Jentzsch, and R. Drechsler. "Cross-level processor verification via endless randomized instruction stream generation with coverage-guided aging". In: *DATE*. 2022.

[7] V. Herdt, D. Große, E. Jentzsch, and R. Drechsler. "Efficient cross-level testing for processor verification: A RISC-V case-study". In: *FDL*. 2020.

[8] N. Bruns, V. Herdt, D. Große, and R. Drechsler. "Efficient Cross-Level Processor Verification using Coverage-guided Fuzzing". In: *VLSI*. 2022.

[9] N. Kabylkas, T. Thorn, S. Srinath, P. Xekalakis, and J. Renau. "Effective processor verification with logic fuzzer enhanced co-simulation". In: *MICRO*. 2021.

[10] C. Chen, R. Kande, N. Nyugen, F. Andersen, A. Tyagi, A. R. Sadeghi, and J. Rajendran. "HyPFuzz: Formal-Assisted Processor Fuzzing". In: *arXiv:2304.02485* (2023).

[11] Jinyan Xu, Yiyuan Liu, Sirui He, Haoran Lin, Yajin Zhou, and Cong Wang. "{MorFuzz}: Fuzzing Processor via Runtime Instruction Morphing enhanced Synchronizable Co-simulation". In: *USENIX SEC*. 2023.

[12] Kevin Laeufer, Jack Koenig, Donggyu Kim, Jonathan Bachrach, and Koushik Sen. "RFUZZ: Coverage-directed fuzz testing of RTL on FPGAs". In: *ICCAD*. 2018.

[13] S. Canakci, L. Delshadtehrani, F. Eris, M. B. Taylor, M. Egele, and A. Joshi. "Directfuzz: Automated test generation for rtl designs using directed graybox fuzzing". In: *DAC*. 2021.

[14] Muhammad Monir Hossain, Arash Vafaei, Kimia Zamiri Azar, Fahim Rahman, Farimah Farahmandi, and Mark Tehranipoor. "Socfuzzer: Soc vulnerability detection using cost function enabled fuzz testing". In: *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2023, 1.

[15] Vasudev Gohil, Rahul Kande, Chen Chen, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. "MABFuzz: Multi-Armed Bandit Algorithms for Fuzzing Processors". In: *arXiv preprint arXiv:2311.14594* (2023).

[16] Chen Chen, Vasudev Gohil, Rahul Kande, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. "PSOFuzz: Fuzzing Processors with Particle Swarm Optimization". In: *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. IEEE. 2023, 1.

[17] S. Canakci, C. Rajapaksha, L. Delshadtehrani, A. Nataraja, M. B. Taylor, M. Egele, and A. Joshi. "ProcessorFuzz: Processor Fuzzing with Control and Status Registers Guidance". In: *HOST*. 2023.

[18] Andrew Danowitz, Kyle Kelley, James Mao, John P Stevenson, and Mark Horowitz. "CPU DB: Recording Microprocessor History: With this open database, you can mine microprocessor trends over the past 40 years." In: *Queue* 10.4 (2012).

[19] A. B. Mehta. "Constrained Random Verification (CRV)". In: *ASIC/-SoC Functional Design Verification: A Comprehensive Guide to Technologies and Methodologies*. Springer, 2018.

[20] A. Adir, E. Almog, L. Fournier, E. Marcus, M. Rimon, M. Vinov, and A. Ziv. "Genesys-Pro: innovations in test program generation for functional processor verification". In: *IEEE Design & Test of Computers* 21.2 (2004).

[21] A Ahmed and P Mishra. "QUEBS: Qualifying Event Based Search in Concolic Testing for Validation of RTL Models". In: *2017 IEEE International Conference on Computer Design (ICCD)*. 2017.

[22] Alif Ahmed, Farimah Farahmandi, and Prabhat Mishra. "Directed test generation using concolic testing on RTL models". In: *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2018.

[23] Mingsong Chen, Prabhat Mishra, and Dhrubajyoti Kalita. "Automatic RTL test generation from SystemC TLM specifications". In: *ACM Transactions on Embedded Computing Systems (TECS)* 11.2 (2012).

[24] Mingsong Chen, Xiaoke Qin, Heon-Mo Koo, and Prabhat Mishra. *System-level validation: high-level modeling and directed test generation techniques*. Springer Science & Business Media, 2012.

[25] Lingyi Liu and Shobha Vasudevan. "Efficient validation input generation in RTL by hybridized source code analysis". In: *2011 Design, Automation & Test in Europe*. 2011.

[26] Elaheh Sadredini, Reza Rahimi, Paniz Foroutan, Mahmood Fathy, and Zainalabedin Navabi. "An improved scheme for pre-computed patterns in core-based SoC architecture". In: *2016 IEEE East-West Design & Test Symposium (EWDTS)*. 2016.

[27] Shirin Nilizadeh, Yannic Noller, and Corina S Pasareanu. "DifFuzz: differential fuzzing for side-channel analysis". In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 2019.

[28] Tun Li, Hongji Zou, Dan Luo, and Wanxia Qu. "Symbolic simulation enhanced coverage-directed fuzz testing of rtl design". In: *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2021.

[29] Kypros Constantinides, Onur Mutlu, and Todd Austin. "Online design bug detection: RTL analysis, flexible mechanisms, and evaluation". In: *2008 41st IEEE/ACM International Symposium on Microarchitecture*. 2008.

[30] Matthew Hicks, Cynthia Sturton, Samuel T King, and Jonathan M Smith. "Specs: A lightweight runtime mechanism for protecting software from security-critical processor bugs". In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2015.

[31] Ghada Dessouky, David Gens, Patrick Haney, Garrett Persyn, Arun Kanuparthi, Hareesh Khattri, Jason M Fung, Ahmad-Reza Sadeghi, and Jeyavijayan Rajendran. "{HardFails}: Insights into {Software-Exploitable} Hardware Bugs". In: *USENIX SEC*. 2019.

[32] Limor Fix. "Fifteen years of formal property verification in Intel". In: *25 Years of Model Checking* (2008).

[33] Limor Fix and Ken McMillan. "Formal Property Verification". In: *EDA for IC System Design, Verification, and Testing*. CRC Press, 2018.

[34] D (STMicroelectronics) Vincenzoni. *Formal property verification: A tale of two methods*. `https://www.edn.com/formal-property-verification-a-tale-of-two-methods/`. Accessed: 2022-06-21.

[35] Mohammad Rahmani Fadiheh, Joakim Urdahl, Srinivas Shashank Nuthakki, Subhasish Mitra, Clark Barrett, Dominik Stoffel, and Wolfgang Kunz. "Symbolic quick error detection using symbolic initial state for pre-silicon verification". In: *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2018.

[36] Eshan Singh, Keerthikumara Devarajegowda, Sebastian Simon, Ralf Schnieder, Karthik Ganesan, Mohammad Fadiheh, Dominik Stoffel, Wolfgang Kunz, Clark Barrett, Wolfgang Ecker, et al. "Symbolic QED pre-silicon verification for automotive microcontroller cores: Industrial case study". In: *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2019.

[37] Karthik Ganesan, Florian Lonsing, Srinivasa Shashank Nuthakki, Eshan Singh, Mohammad Rahmani Fadiheh, Wolfgang Kunz, Dominik Stoffel, Clark Barrett, and Subhasish Mitra. "Effective Pre-Silicon Verification of Processor Cores by Breaking the Bounds of Symbolic Quick Error Detection". In: *arXiv preprint arXiv:2106.10392* (2021).

[38] Vasudevan Madampu Suryasarman, Santosh Biswas, and Aryabartta Sahu. "Automation of test program synthesis for processor post-silicon validation". In: *Journal of Electronic Testing* 34.1 (2018).

[39] Cadence. *Jasper FPV App*. `https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform/formal-property-verification-app.html`. Accessed: 2022-06-21.

[40] Onur Demir, Wenjie Xiong, Faisal Zaghloul, and Jakub Szefer. "Survey of Approaches for Security Verification of Hardware/Software Systems." In: *IACR Cryptol. ePrint Arch.* 2016 (2016).

[41] Allon Adir, Maxim Golubev, Shimon Landa, Amir Nahir, Gil Shurek, Vitali Sokhin, and Avi Ziv. "Threadmill: A post-silicon exerciser for multi-threaded processors". In: *DAC*. 2011.

[42] Ophir Friedler, Wisam Kadry, Arkadiy Morgenshtein, Amir Nahir, and Vitali Sokhin. "Effective post-silicon failure localization using dynamic program slicing". In: *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 2014.

[43] Harry D Foster. "Trends in functional verification: A 2014 industry study". In: *DAC*. 2015.

[44] Jagannath Keshava, Nagib Hakim, and Chinna Prudvi. "Post-silicon validation challenges: How EDA and academia can help". In: *DAC*. 2010.

[45] David Lin, Eshan Singh, Clark Barrett, and Subhasish Mitra. "A structured approach to post-silicon validation and debug using symbolic quick error detection". In: *2015 IEEE International Test Conference (ITC)*. 2015.

[46] Doug Josephson. "The good, the bad, and the ugly of silicon debug". In: *DAC*. 2006.

[47] Ilya Wagner and Valeria Bertacco. "Reversi: Post-silicon validation system for modern microprocessors". In: *2008 IEEE International Conference on Computer Design*. 2008.

[48] David Lin, Ted Hong, Farzan Fallah, Nagib Hakim, and Subhasish Mitra. "Quick detection of difficult bugs for effective post-silicon validation". In: *DAC*. 2012.

[49] Nikos Foutris, Dimitris Gizopoulos, Mihalis Psarakis, Xavier Vera, and Antonio Gonzalez. "Accelerating Microprocessor Silicon Validation by Exposing ISA Diversity". In: *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-44. Porto Alegre, Brazil: Association for Computing Machinery, 2011.

[50] A. Avizienis and Yutao He. "Microprocessor entomology: a taxonomy of design faults in COTS microprocessors". In: *Dependable Computing for Critical Applications 7*. 1999.

[51] Smruti R Sarangi, Abhishek Tiwari, and Josep Torrellas. "Phoenix: Detecting and recovering from permanent processor design bugs with programmable hardware". In: *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. 2006.

[52] Satish Narayanasamy, Bruce Carneal, and Brad Calder. "Patching processor design errors". In: *2006 International Conference on Computer Design*. 2006.

[53] Ilya Wagner, Valeria Bertacco, and Todd Austin. "Using field-repairable control logic to correct design errors in microprocessors". In: *IEEE Transactions on computer-aided design of integrated circuits and systems* 27.2 (2008).

[54] Flavien Solt, Ben Gras, and Kaveh Razavi. "CellIFT: Leveraging Cells for Scalable and Precise Dynamic Information Flow Tracking in RTL". In: *USENIX SEC*. 2022.

[55] Siemens. *Modelsim*. `https://eda.sw.siemens.com/en-US/ic/modelsim/`. Accessed: 2022-06-21.

[56] Synopsys. *VCS*. `https://www.synopsys.com/verification/simulation/vcs.html`. Accessed: 2022-06-21.

[57] Cadence. *Xcelium Logic Simulation*. `https://www.cadence.com/ko_KR/home/tools/system-design-and-verification/simulation-and-testbench-verification/xcelium-simulator.html`. Accessed: 2022-06-21.

[58] Jonathan Balkind, Katie Lim, Fei Gao, Jinzheng Tu, David Wentzlaff, Michael Schaffner, Florian Zaruba, and Luca Benini. "OpenPiton+ Ariane: The first open-source, SMP Linux-booting RISC-V system scaling from one to many cores". In: *Workshop on Computer Architecture Research with RISC-V (CARRV)*. 2019.

[59] Shi-Hao Chen and Jiing-Yuan Lin. "Implementation and verification practices of DVFS and power gating". In: *2009 International Symposium on VLSI Design, Automation and Test*. 2009.

[60] Edmund M Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. "Model checking and the state explosion problem". In: *LASER Summer School on Software Engineering*. Springer. 2011.

[61] F. Farahmandi, Y. Huang, and P. Mishra. "Formal Approaches to Hardware Trust Verification". In: *The Hardware Trojan War*. Springer, 2018.

[62] Aarti Gupta, MV KiranKumar, and Rajnish Ghughal. "Formally verifying graphics FPU". In: *International Symposium on Formal Methods*. Springer. 2014.

[63] Tom Schubert. "High-level formal verification of next-generation microprocessors". In: *Proceedings 2003. Design Automation Conference (IEEE Cat. No. 03CH37451)*. 2003.

[64] Matthew M Wilding, David A Greve, Raymond J Richards, and David S Hardin. "Formal verification of partition management for the AAMP7G microprocessor". In: *Design and Verification of Microprocessor Systems for High-Assurance Applications*. Springer, 2010.

[65] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E Gray, Robert Norton-Wright, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, et al. "ISA semantics for ARMv8-a, RISC-v, and CHERI-MIPS". In: *POPL* (2019).

[66] Thomas Bourgeat, Ian Clester, Andres Erbsen, Samuel Gruetter, Andrew Wright, and Adam Chlipala. "A Multipurpose Formal RISC-V Specification". In: *arXiv preprint arXiv:2104.00762* (2021).

[67] Shlomo Greenberg, Joseph Rabinowicz, and Erez Manor. "Selective state retention power gating based on formal verification". In: *IEEE Transactions on Circuits and Systems I: Regular Papers* 62.3 (2014).

[68] Amir Masoud Gharehbaghi and Masahiro Fujita. "Specification and formal verification of power gating in processors". In: *Fifteenth International Symposium on Quality Electronic Design*. 2014.

[69] Hoon Choi, Myung-Kyoon Yim, Jae-Young Lee, Byeong-Whee Yun, and Yun-Tae Lee. "Formal Verification of an Industrial System-on-a-chip". In: *Proceedings 2000 International Conference on Computer Design*. 2000.

[70] Nagabhushan Reddy, Sankaran Menon, and Prashant D Joshi. "Validation Challenges in Recent Trends of Power Management in Microprocessors". In: *2020 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*. 2020.

[71] Manoj Dusanapudi, S Fields, Michael S Floyd, Guy L Guthrie, R Kalla, Shakti Kapoor, LS Leitner, Charles F Marino, JJ McGill, Amir Nahir, et al. "Debugging post-silicon fails in the IBM POWER8 bring-up lab". In: *IBM Journal of Research and Development* 59.1 (2015).

[72] Prabhat Mishra, Ronny Morad, Avi Ziv, and Sandip Ray. "Post-silicon validation in the SoC era: A tutorial introduction". In: *IEEE Design & Test* 34.3 (2017).

[73] Daniel Petrisko, Farzam Gilani, Mark Wyse, Dai Cheol Jung, Scott Davidson, Paul Gao, Chun Zhao, Zahra Azad, Sadullah Canakci, Bandhav Veluri, et al. "BlackParrot: An agile open-source RISC-V multicore for accelerator SoCs". In: *IEEE Micro* 40.4 (2020).

[74] 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". In: *VLSI*. 2019.

[75] Christopher Celio, David A Patterson, and Krste Asanovic. "The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor". In: *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-167* (2015).

[76] Subhasish Mitra, Sanjit A Seshia, and Nicola Nicolici. "Post-silicon validation opportunities, challenges and recent advances". In: *DAC*. 2010.

[77] Stephan Van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. "RIDL: Rogue in-flight data load". In: *IEEE SP*. 2019.

[78] 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*. 2019.

[79] Alberto Carelli, Alessandro Vallero, and Stefano Di Carlo. "Performance Monitor Counters: interplay between safety and security in complex Cyber-Physical Systems". In: *IEEE Transactions on Device and Materials Reliability* 19.1 (2019).

[80] Tianwei Zhang, Yinqian Zhang, and Ruby B Lee. "Cloudradar: A real-time side-channel attack detection system in clouds". In: *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer. 2016.

[81] Marco Chiappetta, Erkay Savas, and Cemal Yilmaz. "Real time detection of cache-based side-channel attacks using hardware performance counters". In: *Applied Soft Computing* 49 (2016).

[82] Serena Ferracci. "Detecting Cache-based Side Channel Attacks using Hardware Performance Counters". PhD thesis. Sapienza, University of Rome, 2019.

[83] Manaar Alam, Sarani Bhattacharya, Debdeep Mukhopadhyay, and Sourangshu Bhattacharya. "Performance counters to rescue: A machine learning based safeguard against micro-architectural side-channel-attacks". In: *Cryptology ePrint Archive* (2017).

[84] Xueyang Wang, Charalambos Konstantinou, Michail Maniatakos, and Ramesh Karri. "Confirm: Detecting firmware modifications in embedded systems using hardware performance counters". In: *ICCAD*. 2015.

[85] Xueyang Wang, Charalambos Konstantinou, Michail Maniatakos, Ramesh Karri, Serena Lee, Patricia Robison, Paul Stergiou, and Steve Kim. "Malicious firmware detection with hardware performance counters". In: *IEEE Transactions on Multi-Scale Computing Systems* 2.3 (2016).

[86] Rana Elnaggar, Krishnendu Chakrabarty, and Mehdi B Tahoori. "Run-time hardware trojan detection using performance counters". In: *2017 IEEE International Test Conference (ITC)*. 2017.

[87] Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. "CFIMon: Detecting violation of control flow integrity using performance counters". In: *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*. 2012.

[88] Congmiao Li and Jean-Luc Gaudiot. "Online detection of spectre attacks using microarchitectural traces from performance counters". In: *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. 2018.

[89] Hossein Sayadi, Han Wang, Tahereh Miari, Hosein Mohammadi Makrani, Mehrdad Aliasgari, Setareh Rafatirad, and Houman Homayoun. "Recent advancements in microarchitectural security: Review of machine learning countermeasures". In: *2020 IEEE 63rd International Midwest Symposium on Circuits and Systems (MWSCAS)*. 2020.

[90] Veripool. *Verilator, the fastest Verilog/SystemVerilog simulator*. `https://veripool.org/verilator/`. Accessed: 2022-06-21.

[91] Tom Feist. "Vivado design suite". In: *White Paper* 5 (2012).

[92] Ilya K Ganusov, Mahesh A Iyer, Ning Cheng, and Alon Meisler. "Agilex™ generation of intel® fpgas". In: *2020 IEEE Hot Chips 32 Symposium (HCS)*. 2020.

[93] Timothy Trippel, Kang G. Shin, Alex Chernyakhovsky, Garret Kelly, Dominic Rizzo, and Matthew Hicks. "Fuzzing Hardware Like Software". In: *USENIX SEC*. Boston, MA: USENIX Association, 2022.

[94] Sujit Kumar Muduli, Gourav Takhar, and Pramod Subramanyan. "Hyperfuzzing for soc security validation". In: *Proceedings of the 39th International Conference on Computer-Aided Design*. 2020.

[95] F. Corno, E. Sanchez, M.S. Reorda, and G. Squillero. "Automatic test program generation: a case study". In: *IEEE Design & Test of Computers* 21.2 (2004).

[96] Giovanni Squillero. "Microgp—an evolutionary assembly program generator". In: *Genetic programming and evolvable machines* 6 (2005).

[97] Fulvio Corno, Ernesto Sánchez, and Giovanni Squillero. "Evolving assembly programs: how games help microprocessor validation". In: *IEEE Transactions on Evolutionary Computation* 9.6 (2005), 695.

[98] Paolo Bernardi, Edgar Ernesto Sánchez Sánchez, Massimiliano Schillaci, Giovanni Squillero, and Matteo Sonza Reorda. "An effective technique for the automatic generation of diagnosis-oriented programs for processor cores". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27.3 (2008).

[99] Ernesto Sánchez, Matteo Sonza Reorda, and Giovanni Squillero. "Efficient techniques for automatic verification-oriented test set optimization". In: *International Journal of Parallel Programming* 34.1 (2006).

[100] Dimitris Gizopoulos, Mihalis Psarakis, Miltiadis Hatzimihail, Michail Maniatakos, Antonis Paschalis, Anand Raghunathan, and Srivaths Ravi. "Systematic software-based self-test for pipelined processors". In: *VLSI* 16.11 (2008).

[101] Ján Hudec and Elena Gramatová. "An efficient functional test generation method for processors using genetic algorithms". In: *Journal of Electrical Engineering* 66.4 (2015).

[102] Synopsys. *VC Formal*. `https://www.synopsys.com/verification/static-and-formal-verification/vc-formal.html`. Accessed: 2022-06-21.

[103] Siemens. *Questa Formal Verification Apps*. `https://eda.sw.siemens.com/en-US/ic/questa/formal-verification/`. Accessed: 2022-06-21.

[104] Gianpiero Cabodi, Paolo Camurati, Sebastiano F Finocchiaro, Carmelo Loiacono, Francesco Savarese, and Danilo Vendraminetto. "Secure path verification". In: *2016 1st IEEE International Verification and Security Workshop (IVSW)*. 2016.

[105] Wei Hu, Xinmu Wang, and Dejun Mu. "Security path verification through joint information flow analysis". In: *2018 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*. 2018.

[106] Cadence. *Jasper SPV App*. `https://www.cadence.com/ko_KR/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform/security-path-verification-app.html`. Accessed: 2022-06-21.

[107] Eduard Cerny, Surrendra Dudani, John Havlicek, Dmitry Korchemny, et al. *SVA: the power of assertions in systemVerilog*. Springer, 2015.

[108] Roy Armoni, Limor Fix, Alon Flaisher, Rob Gerth, Boris Ginsburg, Tomer Kanza, Avner Landver, Sela Mador-Haim, Eli Singerman, Andreas Tiemeyer, et al. "The ForSpec temporal logic: A new temporal property-specification language". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2002.

[109] Philipp Koppe, Benjamin Kollenda, Marc Fyrbiak, Christian Kison, Robert Gawlik, Christof Paar, and Thorsten Holz. "Reverse engineering x86 processor microcode". In: *USENIX SEC*. 2017.

[110] David Van Campenhout, Trevor Mudge, and John P Hayes. "Collection and analysis of microprocessor design errors". In: *IEEE Design & Test of Computers* 17.4 (2000).

[111] Miroslav N Velev. "Collection of high-level microprocessor bugs from formal verification of pipelined and superscalar designs". In: *International Test Conference, 2003. Proceedings. ITC 2003.* 2003.

[112] Brian A Wichmann. "Microprocessor design faults". In: *Microprocessors and Microsystems* 17.7 (1993).

[113] David Lin, Ted Hong, Yanjing Li, S Eswaran, Sharad Kumar, Farzan Fallah, Nagib Hakim, Donald S Gardner, and Subhasish Mitra. "Effective post-silicon validation of system-on-chips using quick error detection". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 33.10 (2014).

[114] Farimah Farahmandi and Prabhat Mishra. "Observability-Aware Post-Silicon Test Generation". In: *Post-Silicon Validation and Debug*. Springer, 2019.

[115] Arm Ltd. *Speculative Processor Vulnerability*. `https://developer.arm.com/Arm%20Security%20Center/Speculative%20Processor%20Vulnerability`. [Online; accessed 16-May-2022]. 2022.

[116] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus. "SMoTherSpectre: exploiting speculative execution through port contention". In: *ACM SIGSAC*. 2019.

[117] 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: *USENIX SEC*. 2018.

[118] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, et al. "Fallout: Leaking data on meltdown-resistant cpus". In: *ACM SIGSAC*. 2019.

[119] *CVE-2018-3639*. Available from MITRE, CVE-ID CVE-2018-3639. 2018.

[120] G Maisuradze and Christian Rossow. "ret2spec: Speculative execution using return stack buffers". In: *ACM SIGSAC*. 2018.

[121] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss. "Netspectre: Read arbitrary memory over network". In: *ESORICS*. 2019.

[122] J. Stecklina and T. Prescher. "Lazyfp: Leaking fpu register state using microarchitectural side-channels". In: *arXiv preprint arXiv:1806.07480* (2018).

[123] H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida. "Crosstalk: Speculative data leaks across cores are real". In: *IEEE SP*. 2021.

[124] Jo Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yarom, B. Sunar, D. Gruss, and F. Piessens. "LVI: Hijacking transient execution through microarchitectural load value injection". In: *IEEE SP*. 2020.

[125] S. Van Schaik, M. Minkin, A. Kwong, D. Genkin, and Y. Yarom. "CacheOut: Leaking data on Intel CPUs via cache evictions". In: *IEEE SP*. 2021.

[126] H. Ragab, E. Barberis, H. Bos, and C. Giuffrida. "Rage Against the Machine Clear: A Systematic Analysis of Machine Clears and Their Implications for Transient Execution Attacks". In: *USENIX SEC*. 2021.

[127] J. Taneja, Z. Liu, and J. Regehr. "Testing static analyses for precision and soundness". In: *CGO*. 2020.

[128] L. Team. *Dataflowsanitizer design document*. `https://clang.llvm.org/docs/DataFlowSanitizerDesign.html`. [Online; accessed 16-May-2022]. 2019.

[129] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. "Pin: building customized program analysis tools with dynamic instrumentation". In: *ACM SIGPLAN* (2005).

[130] V. P Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis. "libdft: Practical dynamic data flow tracking for commodity systems". In: *ACM SIGPLAN/SIGOPS*. 2012.

[131] A. Ardeshiricham, W. Hu, J. Marxen, and R. Kastner. "Register transfer level information flow tracking for provably secure hardware design". In: *DATE*. 2017.

[132] Y. Yarom and K. Falkner. "FLUSH+ RELOAD: A high resolution, low noise, L3 cache side-channel attack". In: *USENIX SEC*. 2014.

[133] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B Lee. "Last-level cache side-channel attacks are practical". In: *IEEE SP*. 2015.

[134] D. A. Osvik, A. Shamir, and E. Tromer. "Cache attacks and countermeasures: the case of AES". In: *Cryptographers' track at the RSA conference*. Springer. 2006.

[135] C. Wolf. *Yosys open synthesis suite*. 2016.

[136] lowRISC C.I.C. *Ibex: An embedded 32 bit RISC-V CPU core*. `https://github.com/lowRISC/ibex`. [Online; accessed 16-May-2022].

[137] K. Asanovic, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, et al. "The rocket chip generator". In: *Tech. Rep. UCB/EECS-2016-17* (2016).

[138] Intel Corporation. *12th Generation Intel® Core™ Processor, Document Number: 682436-006*.

[139] Ashok B Mehta. *SystemVerilog Assertions and Functional Coverage*. Springer, 2020.

[140] Cadence. *JasperGold Security Path Verification App*. https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform/security-path-verification-app.html. [Online; accessed 16-May-2022].

[141] S. Van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida. *Addendum to RIDL: Rogue in-flight data load*. https://mdsattacks.com/files/ridl-addendum.pdf. 2019.

[142] M. Ghaniyoun, K. Barber, Y. Zhang, and R. Teodorescu. "INTROSPECTRE: A Pre-Silicon Framework for Discovery and Analysis of Transient Execution Vulnerabilities". In: *ACM ISCA*. 2021.

[143] Wei Hu, Armaiti Ardeshiricham, and Ryan Kastner. "Hardware information flow tracking". In: *ACM Computing Surveys (CSUR)* 54.4 (2021).

[144] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss. "A Systematic Evaluation of Transient Execution Attacks and Defenses". In: *USENIX SEC*. 2019.

[145] W. Hu, J. Oberg, A. Irturk, M. Tiwari, T. Sherwood, D. Mu, and R. Kastner. "Theoretical fundamentals of gate level information flow tracking". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2011).

[146] W. Hu, J. Oberg, A. Irturk, M. Tiwari, T. Sherwood, D. Mu, and R. Kastner. "On the complexity of generating gate level information flow tracking logic". In: *IEEE Transactions on Information Forensics and Security* (2012).

[147] W. Hu, J. Oberg, A. Irturk, M. Tiwari, T. Sherwood, D. Mu, and R. Kastner. "An improved encoding technique for gate level information flow tracking". In: *IWLS*. 2011.

[148] W. Hu, A. Becker, A. Ardeshiricham, Y. Tai, P. Ienne, D. Mu, and R. Kastner. "Imprecise security: quality and complexity tradeoffs for hardware information flow tracking". In: *ICCAD*. 2016.

[149] D. E. Denning and P. J. Denning. "Certification of Programs for Secure Information Flow". In: *Communications ACM* (1977).

[150] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers. "A hardware design language for timing-sensitive information-flow security". In: *Acm SIGPLAN* (2015).

[151] F. Schuiki, A. Kurth, T. Grosser, and L. Benini. "LLHD: A multi-level intermediate representation for hardware description languages". In: *ACM PLDI*. 2020.

[152] Zachary Snow. *sv2v: SystemVerilog to Verilog*. https://github.com/zachjs/sv2v/. [Online; accessed 16-May-2022].

[153] lowRISC contributors. *REQ/ACK Syncronizer Verilator Testbench*. https://github.com/lowRISC/opentitan/tree/master/hw/ip/prim/pre_dv/prim_sync_reqack. [Online; accessed 16-May-2022].

[154] lowRISC contributors. *AES S-Box Verilator Testbench*. https://github.com/lowRISC/opentitan/tree/master/hw/ip/aes/pre_dv/aes_sbox_tb. [Online; accessed 16-May-2022].

[155] C. Celio, P.-F. Chiu, B. Nikolic, D. A. Patterson, and K. Asanovic. "BOOMv2: an open-source out-of-order RISC-V core". In: *CARRV*. 2017.

[156] RISC-V. *Architectural Testing Framework*. https://github.com/riscv-non-isa/riscv-arch-test. [Online; accessed 16-May-2022].

[157] Z. Zhou, M. K. Reiter, and Y. Zhang. "A Software Approach to Defeating Side Channels in Last-Level Caches". In: *ACM SIGSAC*. 2016.

[158] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa. "Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory". In: *USENIX SEC*. 2017.

[159] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee. "CATalyst: Defeating last-level cache side channel attacks in cloud computing". In: *IEEE HPCA*. 2016.

[160] B. Gras, K. Razavi, H. Bos, and C. Giuffrida. "Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks". In: *USENIX SEC*. 2018.

[161] D. Evtyushkin, R. Riley, N. CSE Abu-Ghazaleh, ECE, and D. Ponomarev. "BranchScope: A New Side-Channel Attack on Directional Branch Predictor". In: *ASPLOS*. 2018.

[162] RISC-V BOOM. *BOOM Speculative Attacks*. https://github.com/riscv-boom/boom-attacks. [Online; accessed 16-May-2022].

[163] Hack@DAC. *Phase 2 Buggy SoC*. https://github.com/hackdac/hackdac_2018_beta. [Online; accessed 16-May-2022]. 2018.

[164] H. Kannan, M. Dalton, and C. Kozyrakis. "Decoupling Dynamic Information Flow Tracking with a dedicated coprocessor". In: *DSN*. 2009.

[165] S. Banerjee, D. Devecsery, P. M. Chen, and S. Narayanasamy. "Iodine: fast dynamic taint tracking using rollback-free optimistic hybrid analysis". In: *IEEE SP*. 2019.

[166] Tortuga Logic. *Simulation Based Security Verification*. `https://tortugalogic.com/radix-s/`. [Online; accessed 16-May-2022].

[167] Tortuga Logic. *Emulation Based Security Verification*. `https://tortugalogic.com/radix-m/`. [Online; accessed 16-May-2022].

[168] C Palmiero, G. Di Guglielmo, L. Lavagno, and L.P. Carloni. "Design and implementation of a dynamic information flow tracking architecture to secure a RISC-V core for IoT applications". In: *IEEE HPEC*. 2018.

[169] C. Pilato, K. Wu, S. Garg, R. Karri, and F. Regazzoni. "TaintHLS: High-level synthesis for dynamic information flow tracking". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38.5 (2018).

[170] M. R. Fadiheh, D. Stoffel, C. Barrett, S. Mitra, and W. Kunz. "Processor Hardware Security Vulnerabilities and their Detection by Unique Program Execution Checking". In: *DATE*. 2019.

[171] A. Ferraiuolo, R. Xu, D. Zhang, A. C. Myers, and G. E. Suh. "Verification of a Practical Hardware Security Architecture Through Static Information Flow Analysis". In: *ASPLOS*. 2017.

[172] X. Meng, S. Kundu, A. K. Kanuparthi, and K. Basu. "RTL-ConTest: Concolic Testing on RTL for Detecting Security Vulnerabilities". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2021).

[173] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. "Secure program execution via dynamic information flow tracking". In: *ACM SIGPLAN* (2004).

[174] C. Deutschbein, A. Meza, F. Restuccia, R. Kastner, and C. Sturton. "Isadora: Automated Information Flow Property Generation for Hardware Designs". In: *ASHES*. 2021.

[175] K. Devarajegowda, E. Kaja, S. Prebeck, and W. Ecker. "ISA Modeling with Trace Notation for Context Free Property Generation". In: *DAC*. 2021.

[176] T. Ludwig, J. Urdahl, D. Stoffel, and W. Kunz. "Properties First—Correct-By-Construction RTL Design in System-Level Design Flows". In: *IEEE TCAD*. 2020.

[177] S. Deng, D. Gümüşoğlu, W. Xiong, S. Sari, Y. S. Gener, C. Lu, O. Demir, and J. Szefer. "SecChisel Framework for Security Verification of Secure Processor Architectures". In: *HASP*. 2019.

[178] K. Devarajegowda, M. R. Fadiheh, E. Singh, C. Barrett, S. Mitra, W. Ecker, D. Stoffel, and W. Kunz. "Gap-free Processor Verification by S2QED and Property Generation". In: *DATE*. 2020.

[179] M. R. Fadiheh, J. Müller, R. Brinkmann, S. Mitra, D. Stoffel, and W. Kunz. "A Formal Approach for Detecting Vulnerabilities to Transient Execution Attacks in Out-of-Order Processors". In: *DAC*. 2020.

[180] Z. Y. Ding and C. Le Goues. "An empirical study of oss-fuzz bugs". In: *MSR*. 2021.

[181] Google. *American fuzzy lop*. `https://github.com/google/AFL`. [Online; accessed 4-June-2023].

[182] H. Chen, Y. Li, B. Chen, Y. Xue, and Y. Liu. "Fot: A versatile, configurable, extensible fuzzing framework". In: *ACM FSE*. 2018.

[183] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse. "AFL++ combining incremental steps of fuzzing research". In: *WOOT*. 2020.

[184] P. Wang, X. Zhou, K. Lu, T. Yue, and Y. Liu. "Sok: The progress, challenges, and perspectives of directed greybox fuzzing". In: *arXiv:2005.11907* (2020).

[185] S. Österlund, K. Razavi, H. Bos, and C. Giuffrida. "Parmesan: Sanitizer-guided greybox fuzzing". In: *USENIX SEC*. 2020.

[186] V. Ganesh, T. Leek, and M. Rinard. "Taint-based directed whitebox fuzzing". In: *ICSE*. 2009.

[187] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos. "Vuzzer: Application-aware evolutionary fuzzing." In: *NDSS*. Vol. 17. 2017.

[188] P. Chen and H. Chen. "Angora: Efficient fuzzing by principled search". In: *IEEE SP*. 2018.

[189] T. Blazytko, C. Aschermann, M. Schlögel, A. Abbasi, S. Schumilo, S. Wörner, and T. Holz. "GRIMOIRE: Synthesizing Structure while Fuzzing." In: *USENIX SEC*. 2019.

[190] S. Sargsyan, S. Kurmangaleev, M. Mehrabyan, M. Mishechkin, T. Ghukasyan, and S. Asryan. “Grammar-based fuzzing”. In: *IVMEM*. 2018.

[191] P. Srivastava and M. Payer. “Gramatron: Effective grammar-aware fuzzing”. In: *ISSTA*. 2021.

[192] Martin Eberlein, Yannic Noller, Thomas Vogel, and Lars Grunske. “Evolutionary grammar-based fuzzing”. In: *SSBSE*. 2020.

[193] J. Wang, B. Chen, L. Wei, and Y. Liu. “Superion: Grammar-aware greybox fuzzing”. In: *ICSE*. 2019.

[194] P. Godefroid, A. Kiezun, and M. Y. Levin. “Grammar-based whitebox fuzzing”. In: *ASPLOS*. 2008.

[195] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. *Address-sanitizer: A fast address sanity checker*. `https://clang.llvm.org/docs/AddressSanitizer.html`. [Online; accessed 4-June-2023].

[196] LLVM Developers. *UndefinedBehaviorSanitizer*. `https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html`. [Online; accessed 4-June-2023].

[197] C. Courbet. “NSan: a floating-point numerical sanitizer”. In: *ACM SIGPLAN CC*. 2021.

[198] A. Waterman and K. Asanovic. *The RISC-V Instruction Set Manual*. `https://github.com/riscv/riscv-isa-manual`. [Online; accessed 4-June-2023].

[199] RISC-V Software. *Spike RISC-V ISA Simulator*. `https://github.com/riscv-software-src/riscv-isa-sim`. [Online; accessed 4-June-2023].

[200] Chips Alliance. *Ebreak instruction retires*. `https://github.com/chipsalliance/rocket-chip/issues/2672`. [Online; accessed 4-June-2023].

[201] SpinalHDL. *VexRiscv*. `https://github.com/SpinalHDL/VexRiscv`. [Online; accessed 2-September-2023].

[202] YosysHQ. *PicoRV32 - A Size-Optimized RISC-V CPU*. `https://github.com/YosysHQ/picorv32`. [Online; accessed 2-September-2023].

[203] SonalPinto. *Kronos RISC-V*. `https://github.com/SonalPinto/kronos`. [Online; accessed 2-September-2023].

[204] J. Hur, S. Song, D. Kwon, E. Baek, J. Kim, and B. Lee. *DifuzzRTL: Differential Fuzz Testing to Find CPU Bugs (Docker image)*. `https://github.com/compsec-snu/difuzz-rtl`. [Online; accessed 4-June-2023].

[205] C. Wolf, J. Glaser, and J. Kepler. "Yosys-a free Verilog synthesis suite". In: *Austrochip*. 2013.

[206] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, et al. "Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations". In: *ICCAD*. 2017.

[207] T. Ormandy. *Zenbleed*. `https://lock.cmpxchg8b.com/zenbleed.html`. [Online; accessed 2-September-2023].

[208] M. Turpin and P. V. Engineer. "The Dangers of Living with an X (bugs hidden in your Verilog)". In: *Synopsys Users Group Meeting*. 2003.

[209] C. Chen. *Miss illegal instruction exception when rd of MULHU is the same as rs1 or rs2*. `https://github.com/openhwgroup/cva6/issues/885/issuecomment-1469547149`. [Online; accessed 4-June-2023].

[210] S. Sutherland. "I'm Still in Love with My X!" In: *Design and Verification Conference (DVCon)*. 2013.

[211] S. Canakci, C. Rajapaksha, L. Delshadtehrani, A. Nataraja, M. B. Taylor, M. Egele, and A. Joshi. "ProcessorFuzz: Guiding Processor Fuzzing using Control and Status Registers". In: *arXiv:2209.01789* (2022).

[212] S. Tasiran and K. Keutzer. "Coverage metrics for functional validation of hardware designs". In: *IEEE Design & Test of Computers*. 2001.

[213] T. Bojan, M. A. Arreola, E. Shlomo, and T. Shachar. "Functional coverage measurements and results in post-Silicon validation of Core™ 2 duo family". In: *2007 IEEE HLDVT*. 2007.

[214] K. Ruep and D. Große. "SpinalFuzz: Coverage-guided fuzzing for SpinalHDL designs". In: *IEEE ETS*. 2022.

[215] H. Ragab, K. Koning, H. Bos, and C. Giuffrida. "BugsBunny: Hopping to RTL Targets with a Directed Hardware-Design Fuzzer". In: *SILM*. 2022.

[216] Chips Alliance. *RISC-V DV*. `https://github.com/chipsalliance/riscv-dv`. [Online; accessed 4-June-2023].

[217] Fisher Daniel K and Gould Peter J. "Open-source hardware is a low-cost alternative for scientific instrumentation and research". In: *Modern instrumentation* 2012 (2012).

[218] Jérémy Bonvoisin, Robert Mies, Jean-François Boujut, and Rainer Stark. "What is the "source" of open source hardware?" In: *Journal of Open Hardware* 1.1 (2017).

[219] Joshua M Pearce. "Building research equipment with free, open-source hardware". In: *Science* 337.6100 (2012).

[220] Joshua M Pearce. "Quantifying the value of open source hardware development". In: *Modern Economy* 6 (2015).

[221] Jérémy Bonvoisin, Jenny Molloy, Martin Häuer, and Tobias Wenzel. "Standardisation of practices in open source hardware". In: *arXiv preprint arXiv:2004.07143* (2020).

[222] Shunning Jiang, Peitian Pan, Yanghui Ou, and Christopher Batten. "PyMTL3: A Python framework for open-source hardware modeling, generation, simulation, and verification". In: *IEEE Micro* 40.4 (2020).

[223] ITS Heikkinen, Hele Savin, Jouni Partanen, Jukka Seppälä, and Joshua M Pearce. "Towards national policy for open source hardware research: The case of Finland". In: *Technological Forecasting and Social Change* 155 (2020).

[224] J Piet Hausberg and Sebastian Spaeth. "Why makers make what they make: motivations to contribute to open source hardware development". In: *R&D Management* 50.1 (2020).

[225] Flavien Solt, Katharina Ceesay-Seitz, and Kaveh Razavi. "Cascade: CPU Fuzzing via Intricate Program Generation". In: *USENIX SEC*. 2024.

[226] Rong Qu, Jiangang Huang, Long Zhang, Tianlu Qiao, and Jian Zhang. "Scope-based Compiler Differential Testing". In: *2023 IEEE 23rd International Conference on Software Quality, Reliability, and Security (QRS)*. 2023.

[227] William M McKeeman. "Differential testing for software". In: *Digital Technical Journal* 10.1 (1998).

[228] Robert B Evans and Alberto Savoia. "Differential testing: a new approach to change detection". In: *The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers*. 2007.

[229] W. Hu, D. Mu, J. Oberg, B. Mao, M. Tiwari, T. Sherwood, and R. Kastner. "Gate-level information flow tracking for security lattices". In: *ACM TODAES* (2014).

[230] Clash contributors. *Clash: A modern, functional, hardware description language*. `https://clash-lang.org/`. [Online; accessed 25-Jan-2024].

[231] Xiao-lang Yan, Long-li Yu, and Jie-bing Wang. "A front-end automation tool supporting design, verification and reuse of SOC". In: *Journal of Zhejiang University-SCIENCE A* 5 (2004).

[232] K. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avizienis, J Wawrzynek, and K. Asanovic. "Chisel: constructing hardware in a scala embedded language". In: *DAC*. 2012.

[233] Wilson Snyder. "Verilator and systemperl". In: *North American SystemC Users' Group, Design Automation Conference*. 2004.

[234] Stephen Williams and Michael Baxter. "Icarus verilog: open-source verilog more than a year later". In: *Linux Journal* 2002.99 (2002).

[235] Rajat Subhra Chakraborty, Seetharam Narasimhan, and Swarup Bhunia. "Hardware Trojan: Threats and emerging solutions". In: *2009 IEEE International high level design validation and test workshop*. 2009.

[236] Mohammad Tehranipoor and Farinaz Koushanfar. "A survey of hardware trojan taxonomy and detection". In: *IEEE design & test of computers* 27.1 (2010).

[237] Swarup Bhunia, Michael S Hsiao, Mainak Banga, and Seetharam Narasimhan. "Hardware Trojan attacks: Threat analysis and countermeasures". In: *Proceedings of the IEEE* 102.8 (2014).

[238] Swarup Bhunia and M Tehranipoor. "The hardware trojan war". In: *Cham„ Switzerland: Springer* (2018).

[239] Shivam Bhasin and Francesco Regazzoni. "A survey on hardware trojan detection techniques". In: *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2015.

[240] Zhao Huang, Quan Wang, Yin Chen, and Xiaohong Jiang. "A survey on machine learning against hardware trojan attacks: Recent advances and challenges". In: *IEEE Access* 8 (2020).

[241] He Li, Qiang Liu, and Jiliang Zhang. "A survey of hardware Trojan threat and defense". In: *Integration* 55 (2016).

[242] Hassan Salmani, Mohammad Tehranipoor, and Jim Plusquellic. "A novel technique for improving hardware trojan detection and reducing trojan activation time". In: *VLSI* 20.1 (2011).

[243] Verilator contributors. *Fuzzer-related Verilator issues*. https://github.com/verilator/verilator/issues?q=Fuzzer. [Online; accessed 30-January-2024].

[244] YosysHQ. *VlogHammer*. https://github.com/YosysHQ/VlogHammer. [Online; accessed 30-January-2024].

[245] Yann Herklotz Grave. *Fuzzing Verilog*. https://yannherklotz.com/docs/fpga2020/verismith_thesis.pdf. [Online; accessed 30-January-2024].

[246] Lisa Piper and Jin Zhang. "Don't let the X-bugs bite: Conquer elusive X-propagation issues early! Get them before they get you!" In: *2011 9th IEEE International Conference on ASIC*. 2011.

[247] Christian Krieg, Clifford Wolf, Axel Jantsch, and Tanja Zseby. "Toggle MUX: How X-optimism can lead to malicious hardware". In: *DAC*. 2017.

[248] Mike Turpin. "Solving Verilog X-Issues by Sequentially Comparing a Design with itself. You'll never trust unix diff again!" In: *Boston Synopsys Users Group (SNUG)* (2005).

[249] Wei Hu, Baolei Mao, Jason Oberg, and Ryan Kastner. "Detecting hardware trojans with gate-level information-flow tracking". In: *Computer* 49.8 (2016).

[250] Wei Hu, Chip-Hong Chang, Anirban Sengupta, Swarup Bhunia, Ryan Kastner, and Hai Li. "An overview of hardware security and trust: Threats, countermeasures, and design tools". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 40.6 (2020).

[251] Jason Oberg, Wei Hu, Ali Irturk, Mohit Tiwari, Timothy Sherwood, and Ryan Kastner. "Information flow isolation in I2C and USB". In: *DAC*. 2011.

[252] eInfochips Priyambada Mishra. *Understanding Logic Equivalence Check (LEC) Flow and Its Challenges and Proposed Solution*. https://www.design-reuse.com/articles/51622/understanding-logic-equivalence-check-lec-flow-and-its-challenges-and-proposed-solution.html. [Online; accessed 8-Feb-2024].

[253] Qflow contributors. *Qflow 1.3: An Open-Source Digital Synthesis Flow*. http://opencircuitdesign.com/qflow/. [Online; accessed 30-January-2024].

[254] Mohamed Shalan and Tim Edwards. "Building OpenLANE: a 130nm openroad-based tapeout-proven flow". In: *Proceedings of the 39th International Conference on Computer-Aided Design*. 2020.

[255] Sarah Hesham, Mohamed Shalan, M Watheq El-Kharashi, and Mohamed Dessouky. "Digital ASIC Implementation of RISC-V: OpenLane and Commercial Approaches in Comparison". In: *2021 IEEE International Midwest Symposium on Circuits and Systems (MWSCAS)*. 2021.

[256] Kevin E Murray, Mohamed A Elgammal, Vaughn Betz, Tim Ansell, Keith Rothman, and Alessandro Comodi. "SymbiFlow and VPR: An open-source design flow for commercial and novel FPGAs". In: *IEEE Micro* 40.4 (2020).

[257] Ang Li and David Wentzlaff. "PRGA: An open-source framework for building and using custom FPGAs". In: *The First Workshop on Open-Source Design Automation; Florence, Italy*. 2019.

[258] Yuan Chi, Xian Lin, and Xin Zheng. "Design of High-performance SoC Simulation Model Based on Verilator". In: *Proceedings of the 2022 5th International Conference on Algorithms, Computing and Artificial Intelligence*. 2022.

[259] Eunkyung Ham, Yujin Jeon, Jaeyun Lim, and Ji-Hoon Kim. "Verilator-based Fast Verification Methodology for BLE MAC Hardware". In: *2023 International Conference on Electronics, Information, and Communication (ICEIC)*. 2023.

[260] Norm Hardy. "The Confused Deputy: (or why capabilities might have been invented)". In: *ACM SIGOPS Operating Systems Review* 22.4 (1988).

[261] Baptiste David. "How a simple bug in ML compiler could be exploited for backdoors?" In: *arXiv preprint arXiv:1811.10851* (2018).

[262] Scott Bauer, Pascal Cuoq, and John Regehr. *Deniable Backdoors Using Compiler Bugs*. `https://mcfp.felk.cvut.cz/publicDatasets/pocorgtfo/contents/articles/08-03.pdf`. [Online; accessed 6-Feb-2024].

[263] Tim Clifford, Ilia Shumailov, Yiren Zhao, Ross Anderson, and Robert Mullins. "ImpNet: Imperceptible and blackbox-undetectable backdoors in compiled neural networks". In: *arXiv preprint arXiv:2210.00108* (2022).

[264] Cristian Cadar, Luis Pina, and John Regehr. *Multi-Version Execution Defeats a Compiler-Bug-Based Backdoor*. `https://blog.regehr.org/archives/1282`. [Online; accessed 6-Feb-2024].

[265] Michaël Marcozzi, Qiyi Tang, Alastair F Donaldson, and Cristian Cadar. "Compiler fuzzing: How much does it matter?" In: *Proceedings of the ACM on Programming Languages* 3.OOPSLA (2019).

[266] Junjie Chen and Chenyao Suo. "Boosting compiler testing via compiler optimization exploration". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31.4 (2022).

[267] Yannic Noller, Corina S Păsăreanu, Marcel Böhme, Youcheng Sun, Hoang Lam Nguyen, and Lars Grunske. "HyDiff: Hybrid differential software analysis". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 2020.

[268] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. "Finding and understanding bugs in C compilers". In: *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 2011.

[269] Vsevolod Livinskii, Dmitry Babokin, and John Regehr. "Random testing for C and C++ compilers with YARPGen". In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA (2020).

[270] Raphael Isemann, Cristiano Giuffrida, Herbert Bos, Erik van der Kouwe, and Klaus von Gleissenthall. "Don't Look UB: Exposing Sanitizer-Eliding Compiler Optimizations". In: *Proceedings of the ACM on Programming Languages* 7.PLDI (2023).