RemembERR: Leveraging Microprocessor Errata for Design Testing and Validation

Flavien Solt, Patrick Jattke and Kaveh Razavi
Computer Security Group, ETH Zürich

***Abstract*—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.

I. 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 [1]. Despite advances in design testing tools and techniques [2]–[23], 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 paper, 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 [2], [16]. Once the design matures, formal verification techniques ensure the correctness of selected design parts [17]–[19], [24], [25]. Finally, many bugs can only be found in post-silicon testing under real-world conditions [26]–[31]. These design testing and validation methods, unfortunately, do not scale to the complexity of today's microprocessor designs [16]. In the testing steps, the lingering question is whether the test cases are providing a sufficient coverage [32]–[34]. 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 [15], [35]–[38]. 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.

II. Background

This section provides some brief background on existing hardware bug detection techniques (Section II-A) and errata documents (Section II-B).

A. Bug detection methods

We provide background on the three commonly used techniques for detecting hardware bugs [16]: 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 [16]. 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 [12] or inspected manually [16], [39]. Modern simulators provide a rich set of features, such as undefined values (i.e., *don't care* values), signal injection, and various coverage metrics [40]–[42].

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

Testing with simulation has two major shortcomings. First, simulation-based testing is extremely slow. Therefore, it can process only a few inputs 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 [43]. 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 [44], 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 [16], [45]–[47]. Therefore, a typical approach is to verify only selected design parts while modeling the rest [48]–[50]. Second, properties may be difficult to express formally, and there can exist many properties for complex designs [51], [52]. Third, properties related to power management or other physical effects may be difficult to reason about [44], [53]–[56]. 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 [26]–[31], [57]. Therefore, silicon testing remains a crucial part of design validation, and takes up to 50% of the testing cost for commercial designs [58]. 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.

B. 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 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*

Table I. An erratum for Intel Core 12th generation.

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 II. An erratum for AMD Zen 3 family.

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.

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 I, we show the first erratum for Intel Core 12th generation CPUs, and in Table II, the most recent erratum for AMD Zen 3 family CPUs.

III. Motivation: Learn from the Past

The number of published errata has not significantly decreased over time, as we show in Section IV. 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 [59]–[61]. 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 [39], [62]. In Section VI, 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.

IV. RememberR

In this section, we introduce RememberR, an annotated database of 2,563 errata from AMD and Intel microprocessors. In Section IV-A, 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 IV-B.

A. Methodology

Figure 1 presents an overview of our methodology. Our approach can be summarized into four steps: ❶a First, we acquired the latest errata documents from Intel and AMD, and ❶b analyzed duplicate errata. This already allows us to make general observations about errata's current state (Section IV-B). ❷ We then generalized the triggers, contexts, and observable effects to derive a universal classification scheme for errata (Section V-A). 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 V-B).

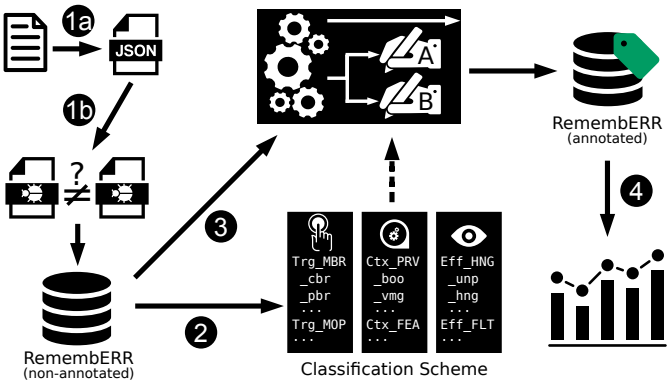

Fig. 1. Overview of our methodology.

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

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			

Examined documents. We comprehensively examined all the errata documents listed in Table III. 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 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.

B. Observations

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

1) 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.

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

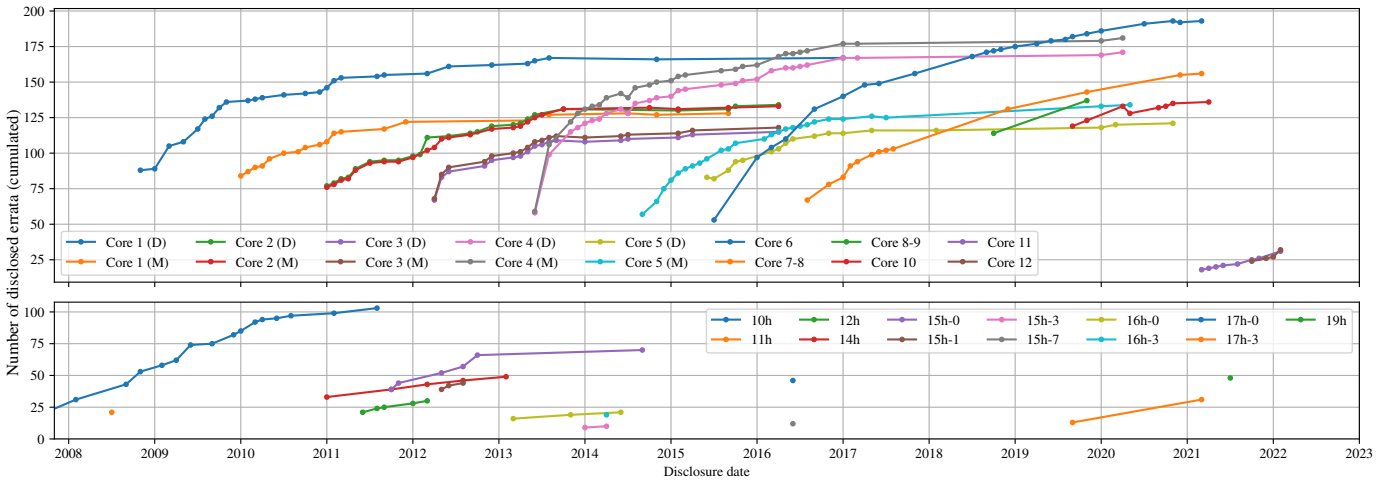

Fig. 2. Disclosure dates of Intel Core errata (top) and AMD CPU errata (bottom). The y-axis represents the cumulative number of disclosed errata. The data point represents the errata's release date.

in two consecutive revisions. In this case, we consider the date of the earlier of the two revisions as the correct one.

Figure 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 IV-B2, we study this bug *transmission effect* across design generations in more detail.

Figure 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.

2) Heredity: It is known from well-studied bugs such as Meltdown [63], Foreshadow [64], RIDL [64] and ZombieLoad [65], 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

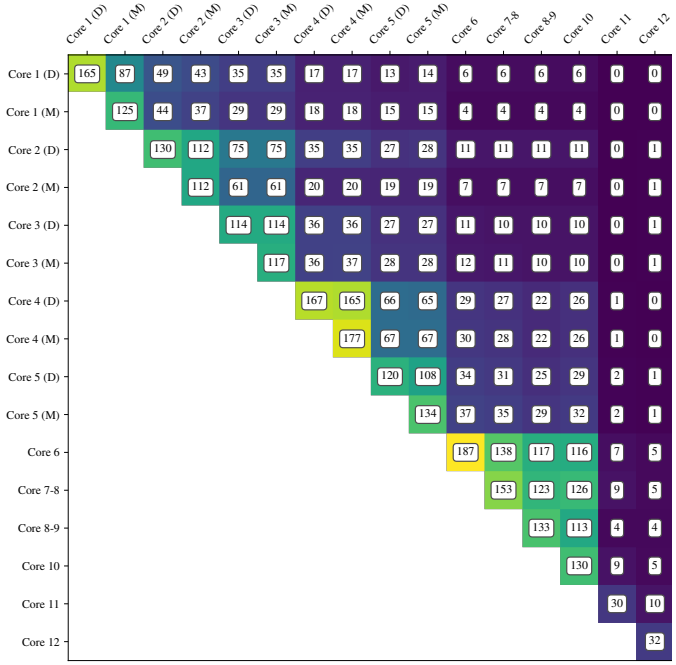

Fig. 3. Bug heredity: number of common bugs across Intel microprocessor generations.

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. Hence, we focus this part of our study on Intel errata.

Figure 3 shows the number of identical errata between pairs of Intel errata documents. We can observe that Desktop and

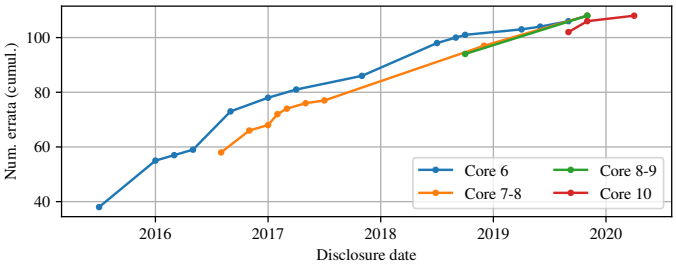

Fig. 4. Disclosure dates of Intel Core errata for bugs that are shared by all Intel Core generations from 6 to 10.

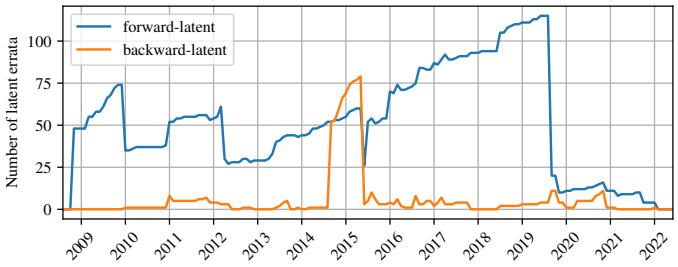

Fig. 5. Forward-latent and backward-latent errata among Intel Core generations.

mobile processors share the vast majority of bugs, matching our observation of similar curves in Section IV-B1. 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 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 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.

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 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 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 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 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 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.

3) 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 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 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

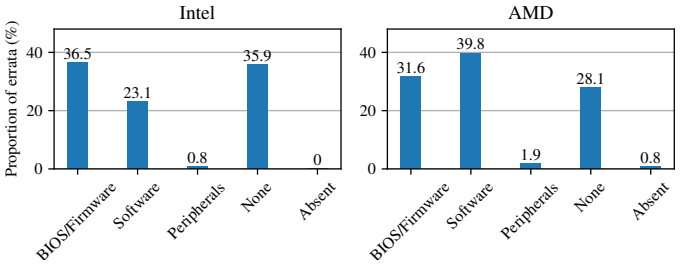

Fig. 6. Suggested workarounds of errata by category.

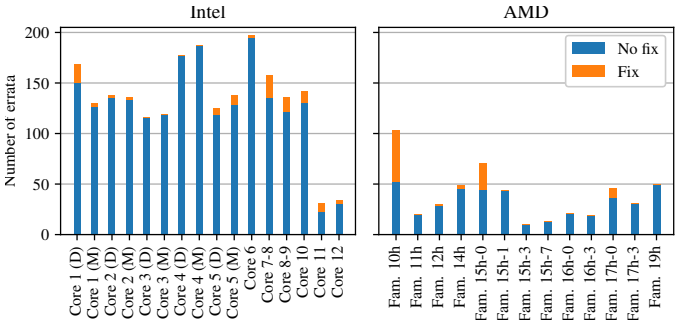

Fig. 7. Proportion of fixed vs. unfixed bugs.

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.

4) 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 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 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.

V. 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 V-A, 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 V-B.

A. 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 classification 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.

1) 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.

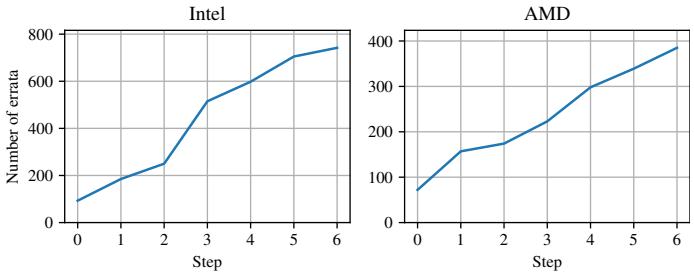

Fig. 8. Number of errata per errata classification discussion step.

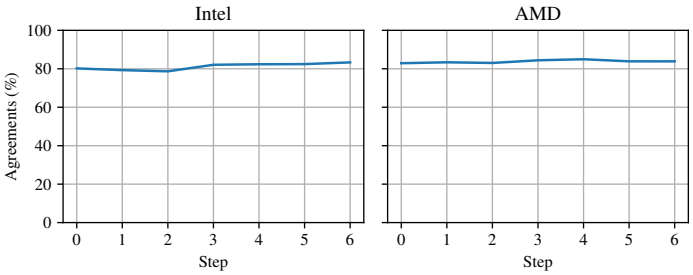

Fig. 9. Percentage of human-classified errata-category pairs classified identically by both humans before the discussion.

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 Tables IV to VI.

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 work 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 8 shows the cumulative number of errata in each classification step. Figure 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 9 is chronological. There are multiple reasons for mismatches, notably (i) human errors as classification is a tedious, long, and difficult process; (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 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.

2) 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 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.

Table IV. Classification of triggers.

Trg_MBR	*a data operation on a...*
– _cbr	cache line boundary.
– _pbr	page boundary.
– _mbr	memory map boundary such as canonical.
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.
Trg_FLT	***related to exceptions and faults***
– _ovf	a counter overflow.
– _tmr	a timer event.
– _mca	a machine check exception.
– _ill	an illegal instruction.
Trg_PRV	***related to privilege transitions***
– _ret	a resume from System Management or OS mode.
– _vmt	a transition between hypervisor and guest.
Trg_CFG	***related to dynamic configuration***
– _pag	a paging mechanism interaction.
– _vmc	a virtual machine configuration interaction.
– _wrg	a configuration register interaction.
Trg_POW	***related to power states***
– _pwc	a transition between power states.
– _tht	a change in thermal or power supply conditions, or throttling.
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.).
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.).

In Table IV, 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. 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.

Table V. Classification of contexts.

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 VI. Classification of observable effects.

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.

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

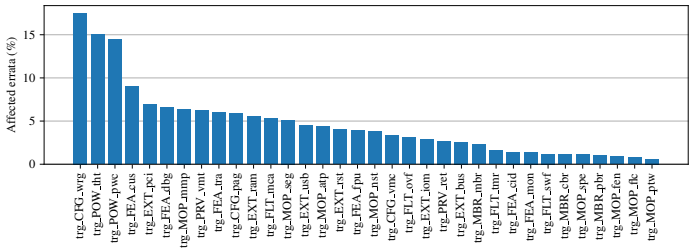

Fig. 10. Most frequent triggers of all errata.

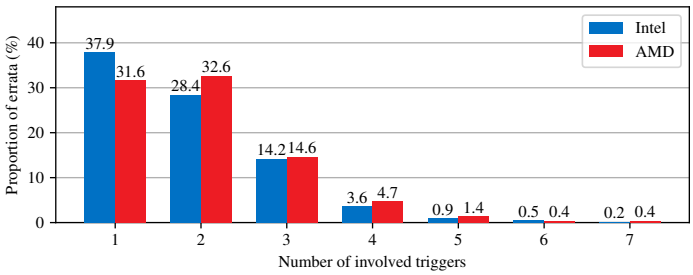

Fig. 11. Number of errata by the number of triggers.

4) 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 VI, we describe all the *abstract* and *class* categories for observable effects that we derived from the errata under study. Similar to 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 [66], 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 [67]–[76]. Wrong performance counter values open exploitable breaches in these defense systems.

B. 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.

Triggers. Our analysis starts by studying the most frequent triggers for Intel and AMD designs. We present the results in Figure 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

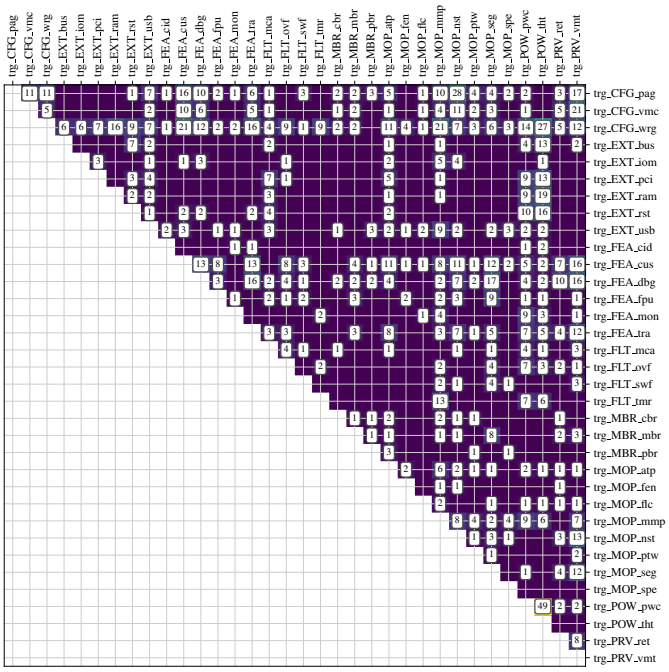

Fig. 12. Pairwise cross-correlation between distinct abstract triggers. The values represent the number of errata documents that require *at least* these two triggers.

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 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%

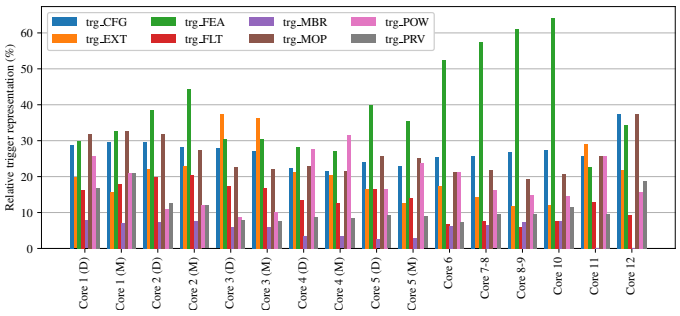

Fig. 13. Trigger classes over Intel Core generations.

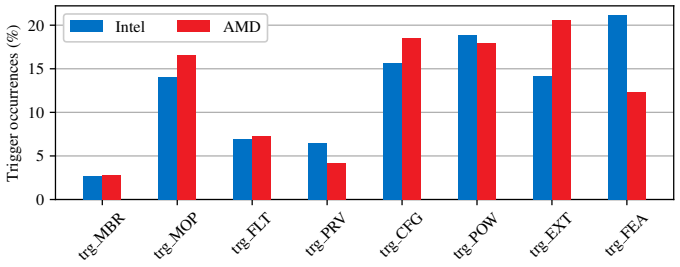

Fig. 14. Relative representation of trigger classes between Intel and AMD.

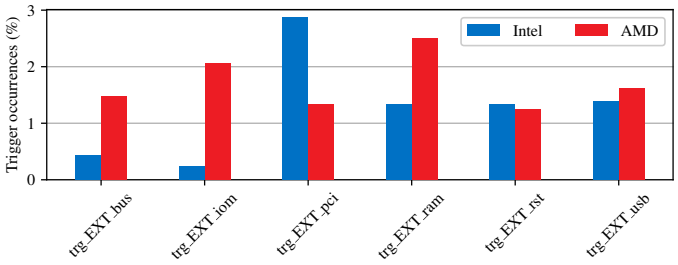

Fig. 15. Relative representation of triggers related to external stimuli between Intel and AMD.

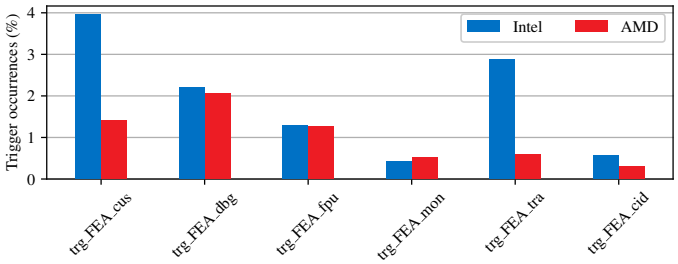

Fig. 16. Relative representation of triggers related to specific features between Intel and AMD.

of the errata require at least two combined triggers to cause a faulty behavior. However, we cannot derive whether bugs involving 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 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 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 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 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 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.

Figures 15 and 16 show a more specific analysis of the two latter trigger classes and clearly indicate the more specific

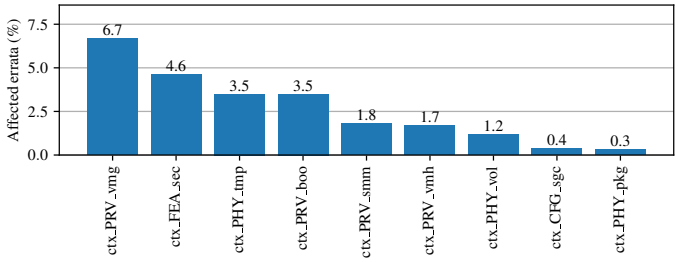

Fig. 17. Most frequent contexts of all errata.

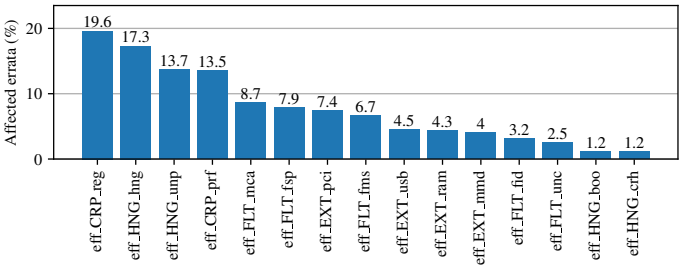

Fig. 18. Most frequent effects for all errata.

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 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 18 shows the 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.

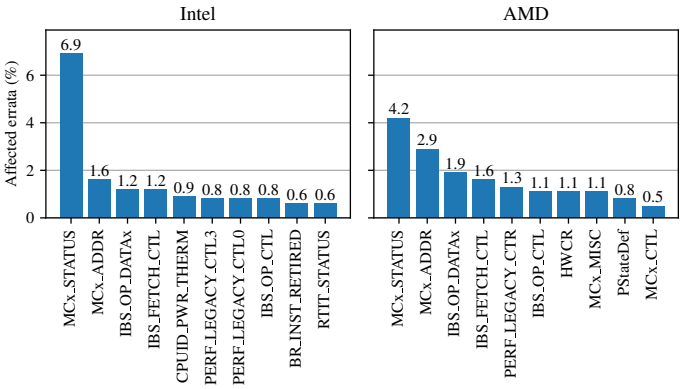

Fig. 19. Most frequent MSR containing observable effects for Intel and AMD.

(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 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 VI, we concretely discuss how these insights may lead to improvements in design validation methodologies and toolchains.

VI. 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 [17].

A. 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 [39]–[42], [77] and emulation [78], [79] are useful to find simple bugs early in the design process, silicon testing is necessary to find many complex bugs [26]–[30]. 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) [2]. 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 [13] 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 [80] 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 [81] does not specify how it chooses its initial seed corpus, while HyperFuzzing [82] only vaguely says that it requires "an initial pool of inputs for the fuzzer seeded with a few interesting behaviors." TheHuzz [12] 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 [3]–[9], [20]–[23], [83]–[89], 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 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 [13] is incapable of discovering any bug on its own [12] as it does not compare its state with any reference. Authors of TheHuzz [12] confirmed that directing observations is one major challenge for a synthesizable porting of their simulator-based testing system. DifuzzRTL [10] 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 [14], [15], [36], [37]. 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.

B. Formal methods

Formal methods statically analyze a design along with properties that are usually specified manually [16]. For instance, Formal Property Verification (FPV) [17], [18], [24], [90], [91] ensures that a set of assertions is never violated in a given design. This technique is often used in two design testing stages [19]: (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) [90]–[94], 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 [16], [45]–[47] is to treat parts of the design as a "black box" and replace them by a model [25]. However, this limits the validation to properties specific to those parts that are not black-boxed [16]. 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 [95] or in Property Specification Language [96]. 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.

C. Manual inspection

Manual inspection is integral to design validation [16] 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 RemembERR 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.

VII. Discussion

We discuss selected observations we made while creating and interpreting RemembERR 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 [97]. 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 IV 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.

Table VII. An erratum in the proposed format.

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.

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 I into Table VII, 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 [27], [29], [30], [33], [98]. Root cause information would additionally first underline which design parts are the most difficult to implement correctly and what are the most common mistakes.

VIII. Related Work

Following, we present existing work that examines public CPU bug information (Section VIII-A). Afterward, we introduce previous work that improves silicon testing (Section VIII-B).

A. Errata-based

Tables VIII and IX summarize work that studied errata from open-source and commercial CPUs, respectively. Previous work provides fragmented information extracted from errata to provide a classification that justifies a specific approach that solves a given problem. Unlike RemembERR, none provides

Table VIII. Summary of work that examined errata in open-source CPUs.

	Year	Errata	Criteria	Goal
Constantinides et al. [14]	2008	296 (OpenSPARC)	Type	Programmable module to react to errata online
Miroslav et al. [99]	2003	280 (students)	Type	Design a testing flow capable of catching errors
Van Campenhout et al. [98]	2000	Students & research	Type	Recommendations for end users

Table IX. Summary of work that examined errata in commercial CPUs.

	Year	Errata	Criteria	Goal
RemembERR	2022	2,563	Triggers, contexts, effects	Provide support for data-driven design testing
Hicks et al. [15]	2015	301	Severity	Monitor hardware security invariants
Wagner et al. [38]	2008	37	Location	Find internal signals corresponding to bugs
Narayanasamy et al. [37]	2006	172	Type, severity	Programmable module to react to errata online
Sarangi et al. [36]	2006	470	Location, severity	Programmable module to react to errata online
Avižienis et al. [35]	1999	535	Location, severity	Taxonomy for dependable systems
Wichmann [100]	1993	–	–	Recommendations for end users

clear insights on representative bug triggers and effects in modern CPUs. Insights from RemembERR are exploitable for future research in design testing and validation.

B. 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. [32] 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. [34] 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. [33], [101] propose methods to transform an instruction sequence to accelerate the observability of a triggered bug. Farahmandi et al. [102] propose an observability measure consisting of test sequences.

RemembERR 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.

IX. 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 RemembERR, 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 RemembERR, 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. This work 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.

X. Artifact Appendix

A. Abstract

Our artifacts include the annotated RemembERR database along with the source code used to build and annotate it. We also provide code of all experiments described in this paper, and provide a Docker image to make reproducing the results easier. Further, we added an example script to encourage readers to write their own queries. Note that generating RemembERR from scratch is a lot of work: parsing and annotating involved tens of hours of high-focus work for two humans. Reproducing the experiments is quick (<1 h).

The `Readme.md` file in our repository contains detailed instructions.

B. Artifact check-list (meta-information)

- **Data set**: The RemembERR database
- **Run-time environment**: Python3
- **Hardware**: Any Linux machine
- **Output**: Figures and numbers
- **Experiments**: All data shown in the paper
- **How much disk space is required?** ≈3 GB (including the software dependencies)
- **How much time is needed to prepare workflow**: <1h
- **How much time is needed to complete experiments**: <1h
- **Publicly available?**: Yes
- **Code licenses** (if publicly available)?: GPLv3
- **Data licenses** (if publicly available)?: GPLv3
- **Workflow framework used**?: Luigi (Python-based, pip package)
- **Archived**?: https://doi.org/10.5281/zenodo.7011959

C. Description

1) How to access: https://github.com/comsec-group/remember

2) Hardware dependencies: None.

3) Software dependencies: We provide instructions for Ubuntu. The apt dependencies are the following:

build-essential
libpoppler-cpp-dev
software-properties-common
python3.8-dev
libgl1
libglib2.0-0
software-properties-common
git
cm-super
dvipng
texlive-latex-extra
texlive-fonts-recommended
python3.8
python3-pip
python3-distutils
python3-apt

The pip dependencies are the following:

camelot-py==0.10.1
colorama==0.4.4
luigi==3.0.3
numpy==1.22.3
openpyxl==3.0.9
pandas==1.4.2
pdftotext==2.2.2
pikepdf==5.1.1
readchar==3.0.5
matplotlib==3.5.1
opencv-python==4.5.5.64

See `Readme.md` or `requirements.txt` in our repository for details.

4) Data sets: RemembERR (provided as part of our artifacts).

5) Models: None.

D. Installation

Clone the repository and install the apt and Python dependencies. You may use a Python virtual environment:

```
python3 -m venv /.venv/rememberr
source /.venv/rememberr/bin/activate
```

Alternatively, you may find all tools preinstalled in the Docker image that we provide.

E. Experiment workflow

Follow the instructions in `Readme.md`.

F. Evaluation and expected results

Numbers will be provided in stdout and figures in the directory specified in `Readme.md`.

G. Experiment customization

We provide an example custom script that bootstrap the learning process of how to use the database for custom analyses. Please refer to `Readme.md`.

H. Notes

None.

I. Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html

References

[1] A. Danowitz, K. Kelley, J. Mao, J. P. Stevenson, and M. Horowitz, "Cpu db: Recording microprocessor history: With this open database, you can mine microprocessor trends over the past 40 years." *Queue*, vol. 10, no. 4, pp. 10–27, 2012.

[2] A. B. Mehta, *Constrained Random Verification (CRV)*. Springer, 2018.

[3] 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," *IEEE Design & Test of Computers*, vol. 21, no. 2, pp. 84–93, 2004.

[4] 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, pp. 185–192.

[5] A. Ahmed, F. Farahmandi, and P. Mishra, "Directed test generation using concolic testing on rtl models," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 1538–1543.

[6] M. Chen, P. Mishra, and D. Kalita, "Automatic rtl test generation from systemc tlm specifications," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 11, no. 2, pp. 1–25, 2012.

[7] M. Chen, X. Qin, H.-M. Koo, and P. Mishra, *System-level validation: high-level modeling and directed test generation techniques*. Springer Science & Business Media, 2012.

[8] L. Liu and S. Vasudevan, "Efficient validation input generation in rtl by hybridized source code analysis," in *2011 Design, Automation & Test in Europe*. IEEE, 2011, pp. 1–6.

[9] E. Sadredini, R. Rahimi, P. Foroutan, M. Fathy, and Z. Navabi, "An improved scheme for pre-computed patterns in core-based soc architecture," in *2016 IEEE East-West Design & Test Symposium (EWDTS)*. IEEE, 2016, pp. 1–6.

[10] S. Nilizadeh, Y. Noller, and C. S. Pasareanu, "Diffuzz: differential fuzzing for side-channel analysis," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 176–187.

[11] T. Li, H. Zou, D. Luo, and W. Qu, "Symbolic simulation enhanced coverage-directed fuzz testing of rtl design," in *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*. IEEE, 2021, pp. 1–5.

[12] A. Tyagi, A. Crump, A.-R. Sadeghi, G. Persyn, J. Rajendran, P. Jauernig, and R. Kande, "Thehuzz: Instruction fuzzing of processors using golden-reference models for finding software-exploitable vulnerabilities," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022.

[13] K. Laeufer, J. Koenig, D. Kim, J. Bachrach, and K. Sen, "Rfuzz: Coverage-directed fuzz testing of rtl on fpgas," in *ICCAD)*. IEEE, 2018, pp. 1–8.

[14] K. Constantinides, O. Mutlu, and T. Austin, "Online design bug detection: Rtl analysis, flexible mechanisms, and evaluation," in *2008 41st IEEE/ACM International Symposium on Microarchitecture*. IEEE, 2008, pp. 282–293.

[15] M. Hicks, C. Sturton, S. T. King, and J. 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, pp. 517–529.

[16] G. Dessouky, D. Gens, P. Haney, G. Persyn, A. Kanuparthi, H. Khattri, J. M. Fung, A.-R. Sadeghi, and J. Rajendran, "{HardFails}: Insights into {Software-Exploitable} hardware bugs," in *28th USENIX Security Symposium (USENIX Security 19)*, 2019, pp. 213–230.

[17] L. Fix, "Fifteen years of formal property verification in intel," *25 Years of Model Checking*, pp. 139–144, 2008.

[18] L. Fix and K. McMillan, "Formal property verification," in *EDA for IC System Design, Verification, and Testing*. CRC Press, 2018, pp. 20–1.

[19] D. S. 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.

[20] M. R. Fadiheh, J. Urdahl, S. S. Nuthakki, S. Mitra, C. Barrett, D. Stoffel, and W. Kunz, "Symbolic quick error detection using symbolic initial state for pre-silicon verification," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2018, pp. 55–60.

[21] E. Singh, K. Devarajegowda, S. Simon, R. Schnieder, K. Ganesan, M. Fadiheh, D. Stoffel, W. Kunz, C. Barrett, W. 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)*. IEEE, 2019, pp. 1000–1005.

[22] K. Ganesan, F. Lonsing, S. S. Nuthakki, E. Singh, M. R. Fadiheh, W. Kunz, D. Stoffel, C. Barrett, and S. Mitra, "Effective pre-silicon verification of processor cores by breaking the bounds of symbolic quick error detection," *arXiv preprint arXiv:2106.10392*, 2021.

[23] V. M. Suryasarman, S. Biswas, and A. Sahu, "Automation of test program synthesis for processor post-silicon validation," *Journal of Electronic Testing*, vol. 34, no. 1, pp. 83–103, 2018.

[24] 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.

[25] O. Demir, W. Xiong, F. Zaghloul, and J. Szefer, "Survey of approaches for security verification of hardware/software systems." *IACR Cryptol. ePrint Arch.*, vol. 2016, p. 846, 2016.

[26] A. Adir, M. Golubev, S. Landa, A. Nahir, G. Shurek, V. Sokhin, and A. Ziv, "Threadmill: A post-silicon exerciser for multi-threaded processors," in *Proceedings of the 48th Design Automation Conference*, 2011, pp. 860–865.

[27] O. Friedler, W. Kadry, A. Morgenshtein, A. Nahir, and V. Sokhin, "Effective post-silicon failure localization using dynamic program slicing," in *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2014, pp. 1–6.

[28] H. D. Foster, "Trends in functional verification: A 2014 industry study," in *Proceedings of the 52nd Annual Design Automation Conference*, 2015, pp. 1–6.

[29] J. Keshava, N. Hakim, and C. Prudvi, "Post-silicon validation challenges: How eda and academia can help," in *Design Automation Conference*. IEEE, 2010, pp. 3–7.

[30] D. Lin, E. Singh, C. Barrett, and S. Mitra, "A structured approach to post-silicon validation and debug using symbolic quick error detection," in *2015 IEEE International Test Conference (ITC)*. IEEE, 2015, pp. 1–10.

[31] D. Josephson, "The good, the bad, and the ugly of silicon debug," in *Proceedings of the 43rd annual Design Automation Conference*, 2006, pp. 3–6.

[32] I. Wagner and V. Bertacco, "Reversi: Post-silicon validation system for modern microprocessors," in *2008 IEEE International Conference on Computer Design*. IEEE, 2008, pp. 307–314.

[33] D. Lin, T. Hong, F. Fallah, N. Hakim, and S. Mitra, "Quick detection of difficult bugs for effective post-silicon validation," in *DAC Design Automation Conference 2012*. IEEE, 2012, pp. 561–566.

[34] N. Foutris, D. Gizopoulos, M. Psarakis, X. Vera, and A. Gonzalez, "Accelerating microprocessor silicon validation by exposing isa diversity," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-44. New York, NY, USA: Association for Computing Machinery, 2011, p. 386–397. [Online]. Available: https://doi.org/10.1145/2155620.2155666

[35] A. Avizienis and Y. He, "Microprocessor entomology: a taxonomy of design faults in cots microprocessors," in *Dependable Computing for Critical Applications 7*, 1999, pp. 3–23.

[36] S. R. Sarangi, A. Tiwari, and J. 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)*. IEEE, 2006, pp. 26–37.

[37] S. Narayanasamy, B. Carneal, and B. Calder, "Patching processor design errors," in *2006 International Conference on Computer Design*. IEEE, 2006, pp. 491–498.

[38] I. Wagner, V. Bertacco, and T. Austin, "Using field-repairable control logic to correct design errors in microprocessors," *IEEE Transactions on computer-aided design of integrated circuits and systems*, vol. 27, no. 2, pp. 380–393, 2008.

[39] F. Solt, B. Gras, and K. Razavi, "Cellift: Leveraging cells for scalable and precise dynamic information flow tracking in rtl," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 2549–2566.

[40] Siemens, "Modelsim," https://eda.sw.siemens.com/en-US/ic/modelsim/, accessed: 2022-06-21.

[41] Synopsys, "Vcs," https://www.synopsys.com/verification/simulation/vcs.html, accessed: 2022-06-21.

[42] 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.

[43] J. Balkind, K. Lim, F. Gao, J. Tu, D. Wentzlaff, M. Schaffner, F. Zaruba, and L. Benini, "Openpiton+ ariane: The first open-source, smp linux-booting risc-v system scaling from one to many cores," in *Workshop on Computer Architecture Research with RISC-V (CARRV)*, 2019, pp. 1–6.

[44] S.-H. Chen and J.-Y. Lin, "Implementation and verification practices of dvfs and power gating," in *2009 International Symposium on VLSI Design, Automation and Test*. IEEE, 2009, pp. 19–22.

[45] E. M. Clarke, W. Klieber, M. Nováček, and P. Zuliani, "Model checking and the state explosion problem," in *LASER Summer School on Software Engineering*. Springer, 2011, pp. 1–30.

[46] F. Erata, S. Deng, F. Zaghloul, W. Xiong, O. Demir, and J. Szefer, "Survey of approaches for security verification of hardware/software systems," *Cryptology ePrint Archive*, 2016.

[47] F. Farahmandi, Y. Huang, and P. Mishra, "Formal approaches to hardware trust verification," in *The Hardware Trojan War*. Springer, 2018, pp. 183–202.

[48] A. Gupta, M. KiranKumar, and R. Ghughal, "Formally verifying graphics fpu," in *International Symposium on Formal Methods*. Springer, 2014, pp. 673–687.

[49] T. Schubert, "High-level formal verification of next-generation microprocessors," in *Proceedings 2003. Design Automation Conference (IEEE Cat. No. 03CH37451)*. IEEE, 2003, pp. 1–6.

[50] M. M. Wilding, D. A. Greve, R. J. Richards, and D. S. Hardin, "Formal verification of partition management for the aamp7g microprocessor," in *Design and Verification of Microprocessor Systems for High-Assurance Applications*. Springer, 2010, pp. 175–191.

[51] A. Armstrong, T. Bauereiss, B. Campbell, A. Reid, K. E. Gray, R. Norton-Wright, P. Mundkur, M. Wassell, J. French, C. Pulte *et al.*, "Isa semantics for armv8-a, risc-v, and cheri-mips," *POPL*, 2019.

[52] T. Bourgeat, I. Clester, A. Erbsen, S. Gruetter, A. Wright, and A. Chlipala, "A multipurpose formal risc-v specification," *arXiv preprint arXiv:2104.00762*, 2021.

[53] S. Greenberg, J. Rabinowicz, and E. Manor, "Selective state retention power gating based on formal verification," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 62, no. 3, pp. 807–815, 2014.

[54] A. M. Gharehbaghi and M. Fujita, "Specification and formal verification of power gating in processors," in *Fifteenth International Symposium on Quality Electronic Design*. IEEE, 2014, pp. 604–610.

[55] H. Choi, M.-K. Yim, J.-Y. Lee, B.-W. Yun, and Y.-T. Lee, "Formal verification of an industrial system-on-a-chip," in *Proceedings 2000 International Conference on Computer Design*. IEEE, 2000, pp. 453–458.

[56] N. Reddy, S. Menon, and P. 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)*. IEEE, 2020, pp. 1–6.

[57] M. Dusanapudi, S. Fields, M. S. Floyd, G. L. Guthrie, R. Kalla, S. Kapoor, L. Leitner, C. F. Marino, J. McGill, A. Nahir *et al.*, "Debugging post-silicon fails in the ibm power8 bring-up lab," *IBM Journal of Research and Development*, vol. 59, no. 1, pp. 12–1, 2015.

[58] P. Mishra, R. Morad, A. Ziv, and S. Ray, "Post-silicon validation in the soc era: A tutorial introduction," *IEEE Design & Test*, vol. 34, no. 3, pp. 68–92, 2017.

[59] D. Petrisko, F. Gilani, M. Wyse, D. C. Jung, S. Davidson, P. Gao, C. Zhao, Z. Azad, S. Canakci, B. Veluri *et al.*, "Blackparrot: An agile

open-source risc-v multicore for accelerator socs," *IEEE Micro*, vol. 40, no. 4, pp. 93–102, 2020.
[60] F. Zaruba and L. Benini, "The cost of application-class processing: Energy and performance analysis of a linux-ready 1.7-ghz 64-bit risc-v core in 22-nm fdsoi technology," *IEEE VLSI*, vol. 27, no. 11, pp. 2629–2640, 2019.
[61] C. Celio, D. A. Patterson, and K. Asanovic, "The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-167*, 2015.
[62] S. Mitra, S. A. Seshia, and N. Nicolici, "Post-silicon validation opportunities, challenges and recent advances," in *Design Automation Conference*. IEEE, 2010, pp. 12–17.
[63] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin *et al.*, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018, pp. 973–990.
[64] S. Van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "Ridl: Rogue in-flight data load," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 88–105.
[65] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "Zombieload: Cross-privilege-boundary data sampling," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 753–768.
[66] A. Carelli, A. Vallero, and S. Di Carlo, "Performance monitor counters: interplay between safety and security in complex cyber-physical systems," *IEEE Transactions on Device and Materials Reliability*, vol. 19, no. 1, pp. 73–83, 2019.
[67] T. Zhang, Y. Zhang, and R. 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, pp. 118–140.
[68] M. Chiappetta, E. Savas, and C. Yilmaz, "Real time detection of cache-based side-channel attacks using hardware performance counters," *Applied Soft Computing*, vol. 49, pp. 1162–1174, 2016.
[69] S. Ferracci, "Detecting cache-based side channel attacks using hardware performance counters," Ph.D. dissertation, Sapienza, University of Rome, 2019.
[70] M. Alam, S. Bhattacharya, D. Mukhopadhyay, and S. Bhattacharya, "Performance counters to rescue: A machine learning based safeguard against micro-architectural side-channel-attacks," *Cryptology ePrint Archive*, 2017.
[71] X. Wang, C. Konstantinou, M. Maniatakos, and R. Karri, "Confirm: Detecting firmware modifications in embedded systems using hardware performance counters," in *ICCAD)*. IEEE, 2015, pp. 544–551.
[72] X. Wang, C. Konstantinou, M. Maniatakos, R. Karri, S. Lee, P. Robison, P. Stergiou, and S. Kim, "Malicious firmware detection with hardware performance counters," *IEEE Transactions on Multi-Scale Computing Systems*, vol. 2, no. 3, pp. 160–173, 2016.
[73] R. Elnaggar, K. Chakrabarty, and M. B. Tahoori, "Run-time hardware trojan detection using performance counters," in *2017 IEEE International Test Conference (ITC)*. IEEE, 2017, pp. 1–10.
[74] Y. Xia, Y. Liu, H. Chen, and B. Zang, "Cfimon: Detecting violation of control flow integrity using performance counters," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*. IEEE, 2012, pp. 1–12.
[75] C. Li and J.-L. 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)*. IEEE, 2018, pp. 25–28.
[76] H. Sayadi, H. Wang, T. Miari, H. M. Makrani, M. Aliasgari, S. Rafatirad, and H. Homayoun, "Recent advancements in microarchitectural security: Review of machine learning countermeasures," in *2020 IEEE 63rd International Midwest Symposium on Circuits and Systems (MWSCAS)*. IEEE, 2020, pp. 949–952.
[77] Veripool, "Verilator, the fastest verilog/systemverilog simulator," https://veripool.org/verilator/, accessed: 2022-06-21.
[78] T. Feist, "Vivado design suite," *White Paper*, vol. 5, p. 30, 2012.
[79] I. K. Ganusov, M. A. Iyer, N. Cheng, and A. Meisler, "Agilex™ generation of intel® fpgas," in *2020 IEEE Hot Chips 32 Symposium (HCS)*. IEEE Computer Society, 2020, pp. 1–26.
Security Symposium (USENIX Security 22). Boston, MA: USENIX
[80] T. Trippel, K. G. Shin, A. Chernyakhovsky, G. Kelly, D. Rizzo, and M. Hicks, "Fuzzing hardware like software," in *31st USENIX* Association, Aug. 2022, pp. 3237–3254. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/trippel
[81] J. Hur, S. Song, D. Kwon, E. Baek, J. Kim, and B. Lee, "Difuzzrtl: Differential fuzz testing to find cpu bugs," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 1286–1303.
[82] S. K. Muduli, G. Takhar, and P. Subramanyan, "Hyperfuzzing for soc security validation," in *ICCAD*, 2020, pp. 1–9.
[83] F. Corno, E. Sanchez, M. Reorda, and G. Squillero, "Automatic test program generation: a case study," *IEEE Design & Test of Computers*, vol. 21, no. 2, pp. 102–109, 2004.
[84] G. Squillero, "Microgp—an evolutionary assembly program generator," *Genetic programming and evolvable machines*, vol. 6, pp. 247–263, 2005.
[85] G. Squillero and G. Squillero, "Evolving assembly programs: how games help microprocessor validation," *IEEE Transactions on Evolutionary Computation*, vol. 9, pp. 695–706, 2005.
[86] P. Bernardi, E. E. S. Sánchez, M. Schillaci, G. Squillero, and M. S. Reorda, "An effective technique for the automatic generation of diagnosis-oriented programs for processor cores," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 3, pp. 570–574, 2008.
[87] E. Sánchez, M. S. Reorda, and G. Squillero, "Efficient techniques for automatic verification-oriented test set optimization," *International Journal of Parallel Programming*, vol. 34, no. 1, pp. 93–109, 2006.
[88] D. Gizopoulos, M. Psarakis, M. Hatzimihail, M. Maniatakos, A. Paschalis, A. Raghunathan, and S. Ravi, "Systematic software-based self-test for pipelined processors," *IEEE VLSI*, vol. 16, no. 11, pp. 1441–1453, 2008.
[89] J. Hudec and E. Gramatová, "An efficient functional test generation method for processors using genetic algorithms," *Journal of Electrical Engineering*, vol. 66, no. 4, p. 185, 2015.
[90] Synopsys, "Vc formal," https://www.synopsys.com/verification/static-and-formal-verification/vc-formal.html, accessed: 2022-06-21.
[91] Siemens, "Questa formal verification apps," https://eda.sw.siemens.com/en-US/ic/questa/formal-verification/, accessed: 2022-06-21.
[92] G. Cabodi, P. Camurati, S. F. Finocchiaro, C. Loiacono, F. Savarese, and D. Vendraminetto, "Secure path verification," in *2016 1st IEEE International Verification and Security Workshop (IVSW)*. IEEE, 2016, pp. 1–6.
[93] W. Hu, X. Wang, and D. Mu, "Security path verification through joint information flow analysis," in *2018 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*. IEEE, 2018, pp. 415–418.
[94] 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.
[95] E. Cerny, S. Dudani, J. Havlicek, D. Korchemny *et al.*, *SVA: the power of assertions in systemVerilog*. Springer, 2015.
[96] R. Armoni, L. Fix, A. Flaisher, R. Gerth, B. Ginsburg, T. Kanza, A. Landver, S. Mador-Haim, E. Singerman, A. 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, pp. 296–311.
[97] P. Koppe, B. Kollenda, M. Fyrbiak, C. Kison, R. Gawlik, C. Paar, and T. Holz, "Reverse engineering x86 processor microcode," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 1163–1180.
[98] D. Van Campenhout, T. Mudge, and J. P. Hayes, "Collection and analysis of microprocessor design errors," *IEEE Design & Test of Computers*, vol. 17, no. 4, pp. 51–60, 2000.
[99] M. N. Velev, "Collection of high-level microprocessor bugs from formal verification of pipelined and superscalar designs," in *International Test Conference, 2003. Proceedings. ITC 2003.* Citeseer, 2003, pp. 138–138.
[100] B. A. Wichmann, "Microprocessor design faults," *Microprocessors and Microsystems*, vol. 17, no. 7, pp. 399–401, 1993.
[101] D. Lin, T. Hong, Y. Li, S. Eswaran, S. Kumar, F. Fallah, N. Hakim, D. S. Gardner, and S. Mitra, "Effective post-silicon validation of system-on-chips using quick error detection," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 33, no. 10, pp. 1573–1590, 2014.
[102] F. Farahmandi and P. Mishra, "Observability-aware post-silicon test generation," in *Post-Silicon Validation and Debug*. Springer, 2019, pp. 111–123.