2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA) | 978-1-6654-3333-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/ISCA52012.2021.00075

Demystifying the System Vulnerability Stack: Transient Fault Effects Across the Layers

George Papadimitriou Dimitris Gizopoulos
Department of Informatics and Telecommunications
University of Athens, Greece
{georgepap | dgizop}@di.uoa.gr

***Abstract*—In this paper, we revisit the system vulnerability stack for transient faults. We reveal severe pitfalls in widely used vulnerability measurement approaches, which separate the hardware and the software layers. We rely on microarchitecture level fault injection to derive very tight full-system vulnerability measurements. For our architectural and microarchitectural measurements, we employ GeFIN, a state-of-the-art fault injector built on top of the gem5 simulator, while for software level measurements we employ the LLFI fault injector. Analyzing two different Arm ISAs and two different microarchitectures for each ISA, we quantify the sources and the magnitude of error of architecture and software level vulnerability evaluation methods, which aim to reproduce the effects of hardware faults. We show that widely applied methodologies for system resilience evaluation fail to capture important fault manifestation and propagation aspects and lead to misleading findings, which report *opposite* vulnerability results than a comprehensive cross-layer analysis. To justify the validity of our findings we employ a state-of-the-art software-based fault tolerance technique and evaluate its impact at all layers through a case study. Our evaluation shows that although higher-level methods can report significant vulnerability improvements (up to 3.8x vulnerability reduction), the actual cross-layer vulnerability of the protected system can be degraded (increased) by up to 30% for the selected benchmarks. Our analysis firmly suggests that only accurate methodologies for full-system vulnerability evaluation of a microprocessor can guide informed transient faults protection decisions either at the hardware or at the software layer.**

***Index Terms*—reliability; microprocessors; transient faults; system vulnerability stack; silent data corruptions; crash; microarchitecture-level fault injection**

I. Introduction

The assessment of system vulnerability to transient faults (soft errors) employs different techniques, which vary in the design stage (early or late pre-silicon as well as post-silicon), the level of hardware accuracy, the assessment throughput, and the granularity of the evaluation [1]-[4]. Fault injection is the most commonly used method for reliability evaluation and can be performed at any level of abstraction: from the gate and the Register-Transfer Level (RTL) [5]-[8] to the microarchitecture [9]-[11] and the software [12]-[19]. By injecting faults in the accessible resources of each level's model (gates, microarchitectural structures, architectural locations, or instructions), such approaches measure the probability for a fault to affect the execution of an application. Typically, software-level fault injection is fast but is practically hardware-agnostic since it can access only a very limited set of resources. Microarchitecture-level fault injection is slower but has complete access to most of the hardware resources that correspond to storage arrays (register files, buffers, caches, etc.). RTL fault injection can, in principle, access any logic gate or storage element but at this level it is impossible to perform full system vulnerability evaluation (including the user and the system software layers) because of the extremely low simulation throughput.

The most comprehensive way to measure the vulnerability of the entire system stack including the microarchitecture, the architecture, and the software layers (both user and kernel space) is to determine the *Architectural Vulnerability Factor* (AVF) of each individual microarchitectural structure during full program execution. The AVF of a hardware structure is the probability that a transient fault in it will affect the execution of the program [20] [21]; AVF can be measured either using analytical methods such as the Architecturally Correct Execution (ACE) analysis [20] or using fault injection [21]. AVF measurements provide useful insights for the vulnerability across the entire system stack at the expense of long simulation runs. In an effort to separate the effects of different layers, the concept of *system vulnerability stack* was proposed [22] [23]. Consequently, hardware designers can employ the Hardware Vulnerability Factor (HVF) to measure the microarchitecture-*dependent* portion of the AVF (the effect of hardware faults until an architecturally visible point is reached), while software designers can leverage architecture-level vulnerability evaluation (measuring the Program Vulnerability Factor – PVF) to quantify the microarchitecture-*independent* portion of the AVF [22]-[24]. Vulnerability estimation methods that operate at the software or at the architecture level of abstraction (assuming that the origin of a flipped hardware bit is a software or architecture visible location) are obviously significantly faster than detailed full-system AVF measurements which account for all the hardware bits. Because of their speed advantage, software level and architecture level vulnerability evaluation approaches have eventually become common practice. The assumption is that such approaches: (a) reasonably model the effect of hardware faults to the software layer (i.e., overall resilience evaluation), and (b) at least provide correct relative vulnerability comparisons [12]-[19], [25]-[30].

978-1-6654-3333-4/21/$31.00 ©2021 IEEE
DOI 10.1109/ISCA52012.2021.00075

In this paper we *challenge* the validity of these fundamental assumptions on which multiple recent studies [12]-[19], [25]-[30] are based and simplistically employ software or architecture level injection to assess the effects of hardware faults and the effectiveness of fault tolerance schemes. This way, software designers, for example, use estimations derived from the software-level or the architecture-level assessment methods to harden their programs against hardware faults ignoring the underlying hardware. This very common simplification has never been quantitatively justified and, as we show in this paper, it leads to severe pitfalls that have not been reported before. While, in theory, the system vulnerability stack facilitates the measurement of the fault propagation (or masking) of each layer individually, it can create the illusion that this measurement can be performed independently to the other layers, and any change in any of the system's components does not significantly affect the vulnerability of multiple layers. The vulnerability of the software may intuitively be considered independent of the hardware, but the actual hardware faults and the models that describe the propagation from the hardware to the software are not. Any software modification may alter the hardware access and utilization patterns (and thus, its vulnerability to soft errors) in such a way that the mix of fault manifestations that reach the software layer also changes dramatically. As a result, any effort to reduce the vulnerability of the software which does not consider the vulnerability of the underlying hardware may skew the analysis and lead to pitfalls in design protection.

As an example, Fig. 1 shows that while the cross-layer AVF analysis (on the right) for program *sha* reports that the vulnerability is dominated by Crashes and not Silent Data Corruptions (SDCs, i.e., output mismatches), the software-layer analysis (i.e., microarchitecture-independent) shows (on the left) exactly the *opposite* relation (SDCs are by far the largest contributors to the vulnerability). Note the different scales of the vertical axes; full-system vulnerability absolute values (right axis) are always much smaller than the software-only vulnerability ones (left axis). More importantly, while the cross-layer AVF analysis shows that *qsort* is almost two times *more vulnerable* than *sha*, the software-layer analysis shows completely the opposite: *qsort* is almost two times *less* vulnerable than *sha*.

Overall, this paper makes the following contributions:

1. We demystify, in the finest possible granularity, the impact of hardware faults all the way to the applications output. Our experimental analysis is based on two different Arm ISAs (Armv7 and Armv8), and two different out-of-order microarchitectures for each ISA (Cortex-A9 and Cortex-A15 for Armv7, and Cortex-A57 and Cortex-A72 for Armv8). This diversity of our analysis is important to demonstrate the fault propagation effects across different layers of the system stack on different ISAs and microarchitectures.
2. We present a comprehensive software-level and architecture-level (PVF) analysis of ten diverse workloads using the widely used approaches for both techniques, to quantify the measurement divergences when assessment stays at these layers compared to the full-stack evaluation (AVF).
3. We conduct a comprehensive HVF analysis of the major microprocessor structures to augment our analysis and pinpoint the sources of these divergences and the magnitude of error that higher-level vulnerability evaluation methods encompass.
4. We introduce a *refined* PVF (rPVF) analysis, which considers the underlying hardware vulnerability obtained from the HVF measurements. Our findings show that even the rPVF provides skewed results and opposite directions compared to cross-layer vulnerability.
5. To further support our claims about the pitfalls of software-level and architecture-level analysis, we employ a recent software-based fault-tolerance technique, which aims to detect hardware faults with very high probability. We demonstrate that although both the architecture and the software level measurements report significant improvement (*reduction*) of the vulnerability (by up to 3.8x and 3.3x, respectively), the full-system vulnerability of the new protected system can be actually degraded (increased) and is up to 30% higher than when the unprotected code is executed.

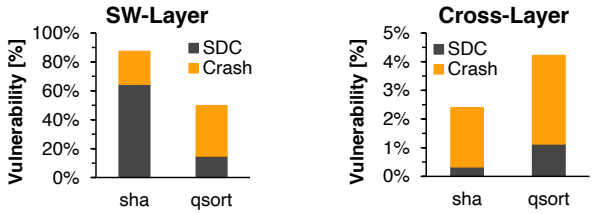

Fig. 1. Software-layer estimations (left) for two benchmarks and their corresponding cross-layer (AVF) estimations (right) on Arm Cortex-A72.

II. Vulnerability Evaluation: Concepts & Definitions

A. Concepts & Background

The Architectural Vulnerability Factor (AVF) is a reliability metric for microprocessor hardware structures that depends on the design (the microarchitecture of a microprocessor) and the workload (program structure and input data). *AVF expresses the probability that a transient fault in a hardware (microarchitectural) structure generates a program-visible error* [20]. AVF is technology independent[1] and quantifies the full-system vulnerability, which includes both the phases of the fault activation and its propagation to the program output through the hardware and the software layers [31]. For instance, assume a fault in the physical register file of an out-of-order microprocessor. The hardware may use the fault in a speculatively executed instruction and then discard the result due to a pipeline flush. In such a case, the fault *will not* affect the execution of the program. But

[1] The FIT (failures in time) rate of a hardware structure *s* can be calculated as FIT(*s*)=AVF(*s*)×FIT(bit)×#Bits(*s*); FIT(bit) is a single bit's failure probability (determined by technology and operational conditions). The FIT rate of the processor is the sum of the FIT rates of all structures.

even if the fault is activated and propagated to the software layer, the software algorithm may not, in turn, use the corrupted data (discard or overwrite the fault) and again deliver the same correct result. Therefore, faults can get masked at either the hardware or the software.

It is conceptually useful to separate the full system into layers and calculate the AVF based on individual vulnerability factors (also called derating or masking factors) of each layer, as shown in Fig. 2(a); this approach can potentially guide error protection approaches at individual layers, and was initially proposed by Sridharan and Kaeli as the *System Vulnerability Stack* [22]. A fault occurring at the hardware layer has a probability to reach the software layer (turn visible). A fault that is visible at the software layer has a probability of reaching the program output. In the full system stack, as shown in Fig. 2(a), a bit is assigned a vulnerability value at every layer to which it becomes visible. Therefore, the Hardware Vulnerability Factor (HVF) of a hardware structure is the fraction of faults in the structure bits that are either activated within the hardware layer or exposed to a higher layer. A hardware-visible fault is exposed to the user program once it reaches a software resource. The Program Vulnerability Factor (PVF) (which assumes that the origin of a fault is an architectural and not a microarchitectural structure, and thus, it is microarchitecture-independent) quantifies the architecture-level fault masking in a program [23] [24]. Thus, the PVF separates the microarchitecture-dependent portion of AVF from the software. A subsequent study of Fang *et al.* proposed an enhanced PVF analysis (ePVF) [32] suggesting that PVF can be also used to explain the error resilience behavior of a program independently of the microprocessor. Moreover, the authors in [33] propose an extension to PVF, which considers the shared resources between multiple threads. These studies employ ACE analysis [20] for the AVF and PVF calculations, which aims to profile the data lifetime inside the hardware structures and quantify their exposure to estimate the vulnerability. In contrast to fault injection, ACE does not provide fine-grained insights on the fault effects and is known to be pessimistic (i.e., it overestimates the vulnerability of a microprocessor structure) [34]; therefore, we employ fault injection at all levels of abstraction to deliver statistically significant measurements.

Along the same lines and at one level higher, software-level reliability measurement approaches inject faults (bit flips) either at the source or intermediate code, or at the assembly or machine code (and not at any microarchitectural or architectural structure). Such approaches are widely used

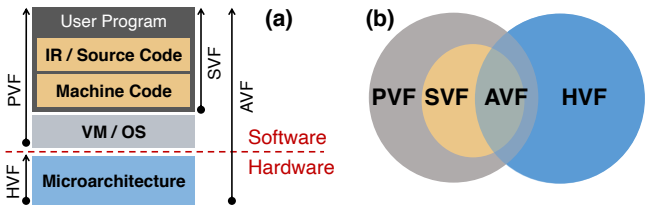

Fig. 2. System vulnerability stack and the vulnerability factors considered in this paper. The vertical arrows represent the fault propagation from its origin to where it becomes visible.

to explain the error resilience behavior of a program independently of the microprocessor. Thus, in software-level injection approaches, the origin (starting point) of the fault is an instruction (operand or operation). Software-level vulnerability evaluation methods report what we will refer to as the *Software Vulnerability Factor (SVF)* to distinguish it from the PVF. This means that the PVF considers that the fault exists in the architectural structure (for example an architectural register) until it is overwritten, while the SVF considers instantaneous faults only when a single instruction is executed, and under no circumstances SVF can take into consideration the kernel instructions that can be executed during program execution. Overall, if a fault eventually becomes visible at every layer of the system (microarchitecture, architecture, software, and the output of the executed program), it contributes to the full system stack vulnerability (AVF in Fig. 2(a) and Fig. 2 (b)). This end-to-end probability expresses the cross-layer vulnerability that is considered the ground truth.

By separating the full-system stack vulnerability into layers, software designers attempt to pinpoint the vulnerability of different segments of the program by employing fast PVF or SVF estimation techniques (since both are supposed to be microarchitecture independent) in order to gain insights for the implementation of application-specific fault-tolerance mechanisms [35]-[38]. However, a modification in the software will certainly alter the hardware access and utilization patterns (and thus, its vulnerability) in such a way that the mix of fault manifestations that reach the software layer also changes dramatically. Therefore, as we show in the rest of the paper, any effort to reduce the vulnerability of a system that is guided by a PVF-only or SVF-only analysis on a fixed hardware design, may lead to the *opposite result*: a system that is eventually more vulnerable to soft errors than the original. In the next sections, we show that in many cases both the PVF and the SVF provide the *opposite* vulnerability measurements than the correct full system AVF and even when a program is hardened to tolerate soft errors, the real vulnerability of the program is also very likely to be increased.

B. Definitions & Hardware Fault Modeling

Program Vulnerability Factor (PVF) quantifies the architecture-level fault masking inherent in a program. According to [23], a software resource is defined as any independently addressable architectural structure. Thus, a flipped bit can occur in any architectural resource, which may or may not be used by the program. However, this concept, as it has been initially defined and as subsequent PVF-based studies consider (e.g., [27]-[30]), raises several issues that cannot not be clearly defined, especially for memory structures. The existence of virtual memory, for example, makes PVF definition even more complicated. Memory is indirectly accessed by the software, typically involving a hardware translation mechanism and operating system support. A question that comes up is: *which portion of the virtual address space should be considered as an architectural resource*? The mapped one or the entire virtual address space (which is the one directly addressable)? For the

mapped part, multiple addresses can correspond to the same hardware structure, and others can be temporarily mapped to peripheral devices. From the HVF point of view, any valid cache line is an architectural visible resource as long as the same line is not valid on a higher cache level. However, an eviction caused by the microarchitecture can immediately change this condition, and a fault that was characterized as software visible can turn software invisible. This is a major concern that, by definition, higher-level vulnerability evaluation methods cannot handle. To this end, in an effort to clearly define and evaluate the PVF-based and SVF-based experiments, we consider the following definitions in the context of this study.

For the PVF measurements, we consider the concept of dynamic instruction flow, which we will refer to as the *program flow*. Program flow includes a subset of the software resources limited to those resources that are being used by the program, including any kernel operations which are executed along with the user program. This applies to both registers and memory. The program flow consists of: (i) the decoded instruction and its operands, (ii) the data transactions in both the registers and the memory, (iii) the program instruction order, and (iv) the execution time of each instruction (allows the monitoring of performance deviations). Therefore, an architecturally mapped register which is not used by the specific program and it does not participate in the program flow is not considered software visible, although it is part of the architectural state. Software masking can then be defined as the probability that a fault which was involved in the program flow is eventually masked by the program. For the rest of this document, we will consider as Program Vulnerability Factor (PVF) the following formula: *PVF* = 1 – *Software Masking*.

For the SVF measurements, we consider the same concept of the program flow, as it is described before, but without considering any kernel operations executed along with the user program, since SVF methods do not take this into account (see Fig. 2). Note that in Fig. 2(b) the SVF is shown to be a subset of PVF, in the sense that SVF lacks the ability to take into account the kernel operations and considers instantaneous faults only when a single instruction is executed. This scenario is completely equivalent to all SVF-based studies (e.g., [12]-[18]). For our SVF measurements we use the LLFI fault injection tool [16]. LLFI is a fault injector which reports SVF measurements and has been widely used in SVF-based and fault tolerance studies (e.g., [15], [39]-[42]). On the complementary hardware side, we define as HVF the probability that a fault on a hardware structure reaches a visible point of the architectural visible resource, i.e., the fault is either activated within the hardware layer or is exposed to a higher layer. For both PVF and HVF calculations (as well as the full system AVF calculations), we use the exact same infrastructure (see details in Section III.B).

III. Experimental Methodology & Setup

A. Fault Propagation Models & Fault Effect Classification

Reliability evaluation through fault injection at either the hardware or the software layers, is based on injecting faults (bit flips to model transient faults) and experimentally observing their effect on program execution. It is, therefore, important to model the way that faults propagate from the hardware layer and manifest to the software layer. For this purpose, we employ a set of *mutually exclusive* Fault Propagation Models (FPMs) listed in Table I. The FPMs practically define the interface between the hardware and the software in fault propagation and can potentially isolate the software layer from the knowledge of the underling hardware. We adopt the fault propagation models from [43] [44], but reduced to four groups, shown in Table I. We merge the corruptions in the immediate and the operand fields in the same FPM class (Wrong Operand or Immediate – WOI), since both are faults in the operand bits of an instruction [43]. Similarly, instruction replacements and control flow errors are merged in the same FPM (Wrong Instruction – WI), as they both are faults that can lead to the execution of a different/wrong instruction. Since hardware and software layers are complementary in the system vulnerability stack, the FPMs, which correspond to the input of software layer are also the fault effect classes of the hardware layer. A hardware fault that reaches the software will belong to one and only one of these classes. In practice SVF-based and PVF-based evaluation methods in the literature consider only the Wrong Data (WD) fault propagation model (e.g., [12]-[19], [27]-[30]), although WI and WOI can be also modeled in PVF-based evaluations. Our analysis reveals, for the first time, a critical conceptual gap in the system vulnerability stack concept which is related to these FPMs.

As Table I shows, there is a separate FPM class "*Escaped"* (ESC). The name of the class is intentionally used to represent the effect of all faults that hit (a part of) the application output that is exposed in a hardware structure (a cache for example) but will not pass *any more* through the microprocessor, i.e., the program flow. Such faults *most certainly corrupt the output* (there is no further masking opportunity) and *can never be considered* at the SVF or PVF analysis. This is a major pitfall of SVF-based and PVF-based techniques which can also lead to inaccurate measurements, because it is not possible for these techniques to take into consideration the class of ESC faults which, as we show, is of significant magnitude. As an example, assume that a program writes a 16KB memory region which gets corrupted (a bit flip occurs) at a cache level,

TABLE I. Fault Propagation Models and Description.

Acronym	FPM	Description
WD	*Wrong Data*	The correct resource was used, but the content of the resource (register or memory word) is corrupted.
WI	*Wrong Instruction*	A different instruction was executed compared to the original program flow. This includes both corrupted Opcode as well as incorrect instruction fetching (PC corruption).
WOI	*Wrong Operand or Immediate*	One or more instruction operand fields were corrupted. This includes register pointers or immediate values.
ESC	*Escaped*	Faults that corrupt the program output without ever reaching the software layer.

and then this region (with the corrupted data) is mapped and accessed through a DMA (Direct Access Memory) device. When a peripheral device accesses this chunk, the corrupted data will not pass *again* through the program trace (i.e., the pipeline) but will certainly corrupt the output. This effect neither will be captured nor can be modeled by SVF and PVF (and corresponds to the ESC model in our analysis). The system vulnerability stack concept fails to capture this scenario (where a fault that is exposed at a lower layer can directly corrupt the output without the interference of the higher software layer). Our experimental results show that the ESC FPM accounts for up to 62% of the overall effects.

Any fault injection campaign, regardless of the abstraction layer, assumes that the origin of faults may have an effect on the eventual output of the program. We classify the *effect on the program output* into the following fault effect classes (typically used in all fault injection studies at any layer):

Silent Data Corruption (SDC): Simulation finished normally, but the program output was different than the fault-free simulation, without any observable indication.

Crash: A simulation that neither reached the end of the workload nor finished within a certain amount of time, because it was disturbed by a catastrophic event. As a result, no program output was produced. The crash may refer to a process crash, a system crash (kernel panic), deadlock or livelock situations.

Finally, when the simulation was executed with no deviations from a fault-free execution, the injected fault is categorized as a **Masked** fault. This means that the fault did not affect the system or the application in any observable way.

B. Fault Injection Frameworks

For our AVF, HVF and PVF measurements we resort on microarchitecture-level simulation using gem5 [45]. Unlike lower-level simulation models (e.g., gate and RTL), gem5 allows deterministic end-to-end execution of large workloads on top of an operating system, i.e., full system analysis which is impossible at lower levels. Further, injection on RTL models [4] would marginally augment our analysis with combinational logic vulnerability, since logic has very low raw failure rates compared to storage elements, which are the basis of this study [46]. We, therefore, employ GeFIN [9] [34], which has been developed and extended to support fault injection campaigns for AVF, HVF, and PVF calculations. GeFIN is a state-of-the-art microarchitecture-level fault injection framework built on top of the gem5 [45]; the most widely used cycle-accurate full-system simulator. The fault-injection framework consists of a modified gem5 version that allows fault injection along with instrumentation for running and controlling simulation campaigns on full-system setup [9] [34] [47].

Fig. 3 shows the simulation timeline on GeFIN for the HVF and PVF measurements. The fault-free simulation period represents the interval from the beginning of the application until the injection of the fault. After the fault is injected, the simulation continues until the fault becomes visible in the software execution. This corresponds to the cycle

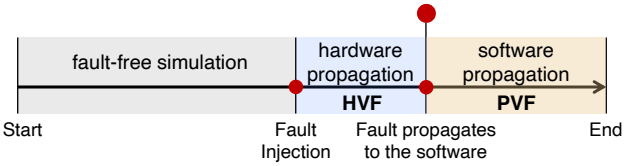

Fig. 3. Illustration of a fault injection simulation, fault propagation and HVF / PVF estimation.

in which the first instruction affected by the fault commits to the architectural state. After that moment, the fault can be considered to have propagated to the software. On the other hand, for the SVF measurements we need to resort on a software-level fault-injection tool. For our SVF experiments we use the popular LLFI fault injector [16], which allows fault injection directly into program variables or statements at the LLVM compiler's intermediate representation (IR) level (it has been shown that IR-level and assembly-level injections deliver very similar results [57]).

C. Experimental Setup

For the needs of this study, we resembled four different out-of-order Arm microprocessors: (i) Cortex-A9, (ii) Cortex-A15, (iii) Cortex-A57, and (iv) Cortex-A72. The first two models implement the Armv7 ISA, while the other two implement the Armv8 ISA. The most important simulated hardware parameters of each considered microprocessor model are shown in Table II. Of course, there are several more configuration parameters we model in the different simulated microprocessors such as: cache configurations (associativity, miss buffers, latencies), branch prediction unit configurations, functional units, latencies, etc. All these constitute different microarchitecture configurations and resemble the actual differences between the mentioned configurations. They affect performance and utilization/access patterns of the injection target components, and thus, can affect vulnerability. The reason of having multiple configurations (different ISAs and microarchitectures) is to showcase that although we are using the exact same source workloads, which would suggest having exactly the same PVF and SVF among different microarchitectures of the same architecture (according to their definitions), the actual full-stack vulnerability is different and more than often not in an obvious way.

For the purposes of this study, we present results targeting five important hardware components: integer physical register file (RF), load and store queue (LSQ), L1 instruction cache (L1I), L1 data cache (L1D) and L2 unified cache (L2). Note that, there is no L3 cache in any Arm-like microprocessor of this study. We estimate that these memory array structures occupy more than 93% of the chip's SRAM area and are, therefore, the largest contributors to the vulnerability of the entire chip [48]. The SVF experiments using the LLFI were performed natively (not in simulation) on top of an Arm Cortex-A72-like machine (Ampere® eMAG® 8180 [49]) since LLFI supports only 64-bit ISAs. Therefore, the comparisons among AVF, PVF and SVF will be only on the 64-bit Armv8, while the comparisons between AVF and PVF will be on both Armv7 and Armv8.

TABLE II. BASIC SIMULATED HARDWARE ARCHITECTURE PARAMETERS OF EACH CONSIDERED MICROPROCESSOR MODEL.

	A9	A15	A57	A72
ISA	Armv7	Armv7	Armv8	Armv8
Pipeline Stages	8	15	15	15
L1 I/D	32 / 32 KB	32 / 32 KB	48 / 48 KB	48 / 32 KB
L2	512 KB	1 MB	1 MB	2 MB
Phys. Reg. File	66	128	128	192
ROB	40	40	128	128
LSQ	8 x 32bit	16 x 32bit	16 x 64bit	16 x 64bit
IQ	32 x 32bit	32 x 32bit	32 x 64bit	64 x 64bit
Fetch/Execute/ Writeback width	2 / 4 / 4	3 / 6 / 8	3 / 6 / 8	3 / 6 / 8

We employ a diverse set of 10 workloads from the MiBench benchmarks suite [50]. The suite is commonly used in reliability studies [9] [34] [47] [51]-[55], as it combines realistic benchmarks with reasonable execution (thus simulation) time and facilitates complete end-to-end executions for the thousands of fault injections required in such a comprehensive analysis. The suite also includes programs from diverse application domains that have similar data and control flow characteristics with SPEC benchmark suite [56]. For each of the five components, 2,000 single-bit faults were randomly generated following the uniform distribution as defined in [21], resulting in 400,000 faults for all 10 benchmarks and the 4 different microarchitectures. For the SVF experiments using the LLFI we also performed 2,000 fault injections for each program. We follow the widely adopted formulation of [21] for the statistical fault sampling calculations; our 2,000 fault samples correspond to 2.88% error margin with 99% confidence level.

IV. QUANTIFYING VULNERABILITY AT ALL LAYERS

In this section, we first pinpoint the coarse-grained inaccuracies that higher-level vulnerability evaluation methods introduce. We present architecture-level vulnerability results (PVF) by employing GeFIN and we further expand our analysis by also providing SVF estimations using the LLFI injection tool and present a comprehensive comparison among AVF, PVF and SVF to show the wider view of the pitfalls that PVF-based and SVF-based analyses can lead to. We finally show that even considering the underlying hardware vulnerability, higher-level methods fail to capture the actual vulnerability trends.

A. AVF, PVF, and SVF Vulnerability Evaluation

Fig. 4 shows the PVF and SVF estimations for Armv8 and the correct full system stack AVF values (for Arm Cortex-A72 in this case for a fair comparison of all three cases) for the two fault effect classes and for each benchmark. To comprehensively aggregate the detailed results of AVF and compare them to the SVF and PVF estimations, since AVF is calculated per hardware structure, we weight the AVF of each benchmark across the five structures. To account for the different hardware structure sizes, instead of calculating the straightforward arithmetic mean of the AVFs of the benchmark for the different structures, we weight the AVFs with the size (number of bits) of each structure (this AVF weighting is equivalent to the FIT rate calculation of the entire microprocessor by considering the different component sizes – see footnote in the second page). Thus, smaller hardware structures will have a smaller impact on the benchmark's AVF compared to larger ones. SVF results for *cjpeg* and *djpeg* are not shown because the instrumentation of these codes in LLFI was failing, and thus, we could not run these workloads on LLFI. Note that the PVF/SVF and AVF values (y-axes) are at different scales because it is important to see the correlation trends and not to compare the actual vulnerabilities. As shown in Fig. 4, in several cases the vulnerability estimation results obtained through PVF and SVF lead to *opposite* relative results between benchmarks both in terms of the total vulnerability and in terms of the magnitude of each individual fault effect class.

First, there are 13 benchmark pairs (out of the 45 total pairs among the 10 benchmarks) whose PVF and SVF show the *opposite* vulnerability relation between the benchmarks of a pair than the AVF (the actual cross-layer vulnerability). See, for example, the *fft* and *qsort* benchmarks (the leftmost green dotted rectangle on each graph). While the full system AVF shows that *qsort* is almost two times more vulnerable than the *fft*, the SVF and PVF show exactly the opposite: *fft* is significantly more vulnerable than *qsort*. Similar observation between *rijndael* and *sha*, between *sha* and *corner*, and so on. Table III shows the frequency of these *opposite* relative vulnerability comparison regarding the total vulnerability of programs ("Total" column), between PVF and AVF, SVF and AVF, and between SVF and PVF. Note that, as we discussed in section II.A, PVF and SVF evaluations are assumed to be microarchitecture independent, and thus, different microarchitectures (of the same architecture) provide similar results. However, we also conduct the PVF experiments for Armv7 and we found that also between similar architectures (e.g., Armv7 and Armv8) their PVF is very close to each other. As we discussed, LLFI does not support 32-bit architectures, and thus, we cannot conduct SVF evaluations on Armv7.

Second, in Fig. 4 we can also observe that there are more than a few cases in which PVF and SVF shows that SDC

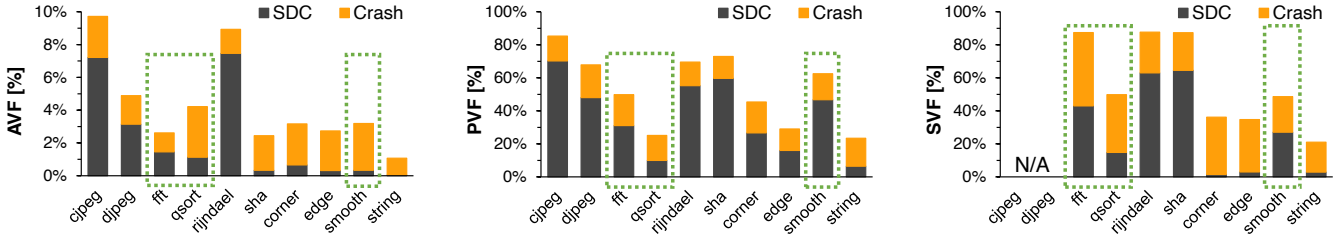

Fig. 4. PVF and SVF estimations and the full-system stack AVF probabilities (for Arm Cortex-A72) for each fault effect class.

TABLE III. FREQUENCY OF OPPOSITE TRENDS OF THE TOTAL VULNERABILITY (TOTAL) AND THE FAULT EFFECTS (EFFECT).

	Cortex-A9		Cortex-A15		Cortex-A57		Cortex-A72	
	Effect	Total	Effect	Total	Effect	Total	Effect	Total
PVF vs. AVF	40%	29%	50%	31%	50%	27%	40%	29%
SVF vs. AVF	–	–	–	–	38%	14%	38%	29%
PVF vs. SVF	–	–	–	–	38%	21%	38%	21%

fault effect is the dominant fault effect class, while AVF shows the opposite trend. Assume for example the *smooth* benchmark (the rightmost green dotted rectangle on each graph), in which we can see that although PVF and SVF show that the dominant fault effect is the SDC class, AVF shows that Crash fault class is the dominant one. Similar observations exist also on all ISAs and microarchitectures used in this study, as Table III shows in "Effect" columns.

B. Considering the Underlying Hardware Vulnerability

Although the PVF and SVF can probably deliver useful information on how faults propagate through software, it still remains unknown how and which of the FPMs constitutes the greatest hazard for the software, i.e., how likely is it for hardware faults to manifest as any of the FPMs and reach the software? In this subsection, we demonstrate how the lack of the underlying hardware information can mislead protection decisions. As the FPMs correspond to the outcome of the HVF estimation, it is not possible to estimate what is the probability of each FPM without performing a detailed HVF analysis. We cover this missing part of the fault propagation inside the full system stack by demonstrating that the FPMs magnitude can highly vary among different applications, which is a major factor that distorts PVF and SVF estimations.

Fig. 5 presents the HVF results for four of the five important components of Cortex-A9 and Cortex-A15 (similar observations in this section are valid for the other two processor models). As we can see in Fig. 5, in the Register File and the L1 Data Cache the WD FPM is the dominant one. This is in line with the typical PVF and SVF results in Fig. 4, in which it is clear that the reason that PVF and SVF provide very high SDC vulnerability, is because typical PVF and SVF analysis (which only considers WD; see Section III.A) are based on fault injection on limited architectural visible resources, such as the architectural register file and loaded/stored data in memory (e.g., [27]-[30]). However, in Fig. 5 we can also see that the other components (including the LSQ which is not shown) have very high WI, WOI and ESC FPM rates. Typical PVF-based or SVF-based estimations *do not quantify* and *do not take into consideration* any of these models, which significantly affect the total vulnerability of the program (the ESC class, by definition, cannot be considered). In practice, when only the WD FPM is considered, the fault origin is only considered to be a used (architectural) register and loaded/stored data from memory and the contribution of WI and WOI, which is significantly high as we see, is completely ignored.

This widely used approach, under no circumstances, can be assumed to reproduce the actual effect of hardware faults to the program execution, especially through an SVF-based

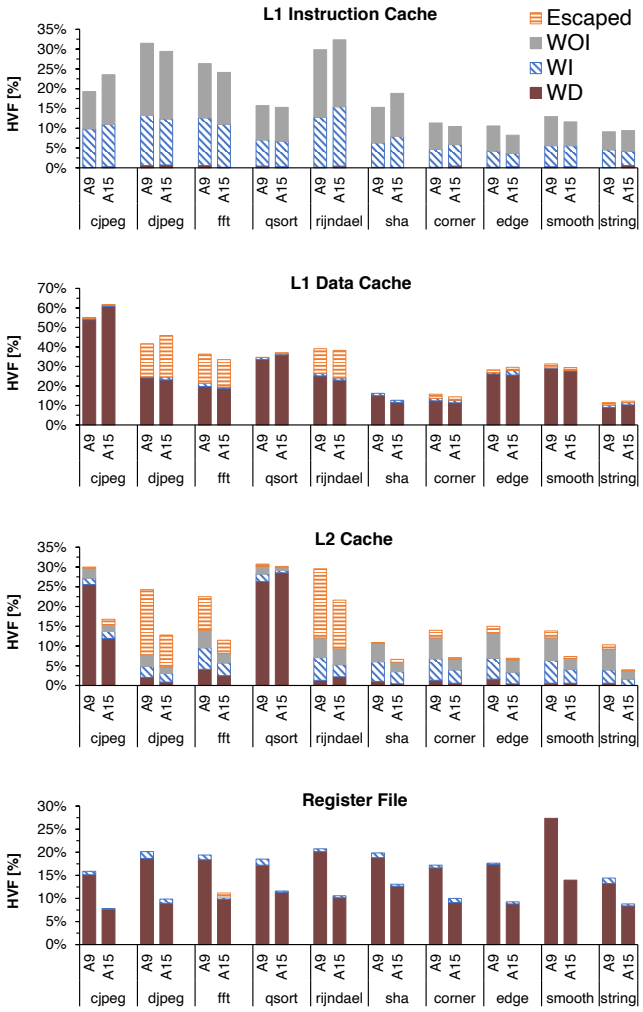

Fig. 5. HVF for four of the five microarchitectural components. The fault effect classes match the FPMs.

estimation. Moreover, the graphs show that there are significant differences between the two microarchitectures regarding the impact of each FPM on the total vulnerability. This is another important observation that PVF-based and SVF-based methodologies are not able to take into consideration for the program vulnerability evaluation. The PVF, and of course the SVF (as a subset of PVF, since SVF cannot take into consideration kernel operations during the program's execution), of the same ISA but on different microarchitecture is assumed to be the same (i.e., microarchitecture-independent); this is not true.

To further investigate this pitfall and the impact of the distribution of FPMs on the program's resiliency, we use the size of each hardware component as a weighting factor to further calculate the *weighted FPM distribution* of each microarchitecture, as shown in Fig. 6. The Escaped FPM is now included in the calculation because it is an HVF fault effect class, although it is impossible to model it as an FPM during PVF or SVF. Fig. 6 clearly shows how severe the effect of the Escaped FPM can be; it is estimated to be up to 62% at the HVF (on average 29% across the benchmarks).

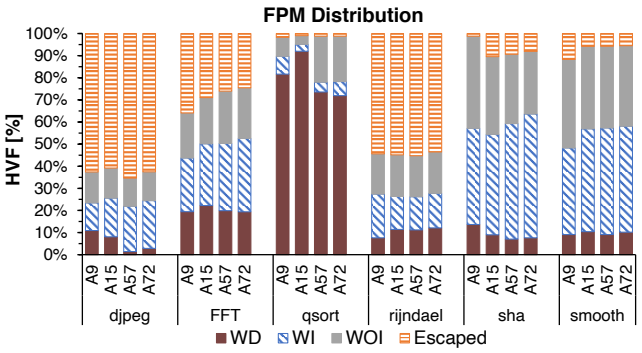

Fig. 6. Distribution of Fault Propagation Models across all studied microarchitectures.

It is important to highlight how the FPM distribution can vary depending on the workload. For instance, we can see that in *qsort*, WD is the dominant FPM that comes from the hardware layer (>70% probability) while on the other hand, for *sha* benchmark it has <12% probability to occur. Moreover, we can see that the Escaped FPM significantly varies between benchmarks. Therefore, it is clear that a *very large vulnerability measurement error* is introduced to higher-level approaches because they cannot consider the ESC FPM. Another clear conclusion from Fig. 6 is how the hardware itself (i.e., microarchitecture) can affect the FPM distribution. Therefore, it is impossible to accurately analyze the vulnerability of a program for hardware errors without having the underlying hardware information of faults propagation effects. Overall, we have measured the magnitude of the two factors that affect the FPM distribution (and consequently the accuracy of high-level measurements): the *microarchitecture* and the *workload*.

V. rPVF: A Refined PVF Analysis

In this section we first consider all FPMs that can reach the software layer (WD, WOI, WI) in an effort to explain why typical PVF analysis (which only considers the WD FPM) delivers these diverging results when comparing different programs vulnerabilities. We then combine these findings with the previous section in order calculate the *refined PVF* (rPVF) measurements using the *actual* FPM distributions as they are measured by the HVF analysis (of course ESC FPM class is not included). The purpose of calculating the rPVF is to understand whether the FPMs distributions coming from the HVF experimentation can remove the pitfalls of the PVF analysis and the opposite trends that PVF reports compared to AVF.

A. PVF per Fault Propagation Model

To calculate refined PVF values when taking into account WOI and WI FPMs (not only WD), we have extended the simulator (again using the full-system setup with OS) to inject faults (2,000 faults for every FPM) to all architecturally-visible locations to quantify the complete program vulnerability. This allows the employment of the same gem5-based infrastructure for the PVF, HVF and AVF estimations, for accurate correlation. Fig. 7 shows the calculated PVF for the three FPMs categorized by each fault effect class. Among the three FPMs, WD has the largest variability among the workloads, while the WOI and especially the WI are more uniform and have a narrower vulnerability range. As we discussed, WD is the most commonly used in PVF-based estimation methodologies (e.g., [27]-[30]). However, we can observe that WOI and WI provide a high rate of Crash effect (unlike WD which mostly leads to SDCs), which could potentially contribute to the typical PVF estimations (in which only WD model is considered), and thus, to improve the *opposite* trends for the dominant fault effect class that typical PVF provides.

B. Consolidated rPVF for all ISAs and Microarchitectures

In order to quantify the level of inaccuracy in the results that do not consider either all the FPMs or their distributions, we provide a refined PVF analysis (rPVF) for all ISAs and microarchitectural configurations, which considers the actual FPM distribution (Fig. 6). In Fig. 8 we can see the rPVF results (left diagram) which uses the output of the HVF estimation (weighted by the size of each hardware structure) of each microarchitecture, and also the cross-layer AVF results (right diagram). Unfortunately, even this refined PVF calculation (which is weighted by the FPM distribution, and thus, takes into account the different probabilities of fault effects that are propagated to the software layer) provides very similar values for each fault effect class and the total vulnerability for all microarchitectures and all but one benchmark, while the cross-layer AVF results show that (apparently) the vulnerability estimation is significantly different on different microarchitectures (benchmarks which are not shown have exactly the same trend). Only *djpeg* shows a marginal difference for rPVF between different microarchitectures. However, some benchmarks, such as *fft*, present virtually no difference among different microarchitectures (this is in line with typical PVF).

In addition to the FPM distribution, the HVF estimation that precedes the PVF estimation for rPVF can be used to further refine the results towards the actual AVF estimation and it is clear from Fig. 8 that the balance between SDC and

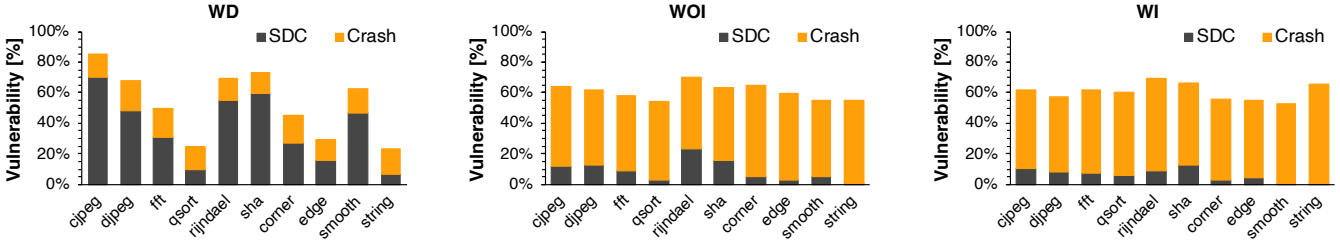

Fig. 7. PVF estimation for Wrong Data (WD), Wrong Operand or Immediate (WOI) and Wrong Instruction (WI) FPMs.

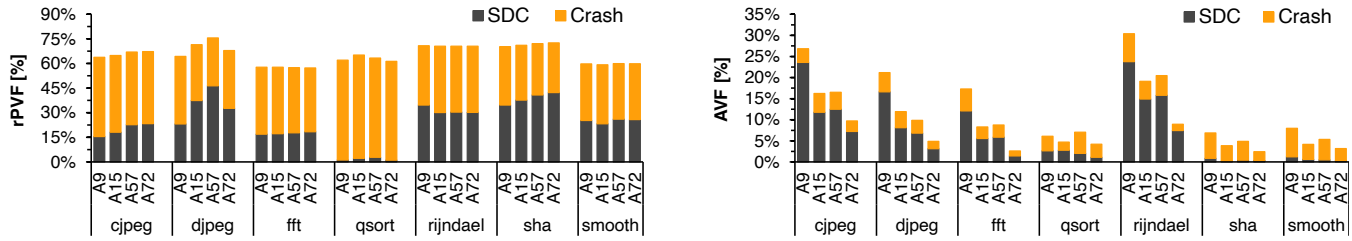

Fig. 8. Refined PVF analysis (rPVF) using actual FPM distributions from hardware layer for all ISAs and microarchitectures (left), and the cross-layer AVF evaluation for all ISAs and microarchitectures (right).

Crash fault effects is significantly changed compared to typical PVF estimations (see Fig. 4). A significant portion of the difference between PVF and rPVF is attributed to the different microarchitectural configurations. Even though most of the workloads report similar results, it is evident that the ISA and microarchitecture can affect the PVF estimation results as the balance of the FPMs can vary. It should be highlighted that the attributes that can affect the actual architecture layer (PVF) estimation include both the hardware configuration and the ISA. Any methodology that does not consider one of these factors (along with ESC fault that, by definition, cannot be considered) holds a source of inaccuracy that can lead to pitfalls.

VI. Hardware-Agnostic Estimation and Decisions About Fault-Tolerant Applications

Several software level fault injection techniques, which aim at even more abstract layer than the architecture layer (i.e., PVF), allow developers to reason about error resilience (e.g., [12]-[18]). To this end, there are numerous works that propose the use of a universal architecture layer, such as LLFI [15] [16] [57]. Unlike architecture-level fault injection, where the estimation can consider a subset of the software resources (limited to those that are actually being used by the program, including any kernel operations executed along with the user program), software-level fault injection frameworks, such as LLFI, target an intermediate abstract layer without considering kernel operations, in order to assess the hardware fault effects (as described in Section II.B). In this section, we employ a recent software-based fault tolerance technique, which aims to detect hardware faults with very high probability (in particular those that lead to SDCs) to conduct a case study about the vulnerability differences. Our case study shows that although the fault-tolerant implementation can significantly reduce the vulnerability of the application when it is evaluated through software-level (SVF) or architecture-level (PVF) techniques, the true cross-layer (AVF) vulnerability may *increase* along with the execution time of the fault-tolerant application which is apparently increased as well.

A. Hardware Vulnerability Dependence

In the previous sections we demonstrated how the vulnerability measurements of different applications strongly depend on the underlying microarchitecture which cannot be ignored. However, the hardware vulnerability analysis (HVF) or cross-layer (AVF) vulnerability analysis are time-consuming processes, which (at least) the software developers are not willing to integrate in their assessment. Especially when considering that the application of software-based fault-tolerance techniques would require several iterations and re-evaluations of the vulnerability and that the hardware vulnerability must be performed in multiple hardware structures. Changes in the software, however, significantly affect the cross-layer vulnerability (AVF), and consequently the program vulnerability, primarily because the HVF will be affected, and thus, the FPM distribution that is delivered from the hardware. However, when a software designer considers to apply a software-based fault-tolerance technique, finer-grained results are necessary to pinpoint what fault effect is the most severe for an application (SDC or Crash), and thus, to apply the most suitable software-based fault-tolerance technique, since usually software-level error protection against SDC or Crash require completely different code modifications.

To pinpoint the pitfall of PVF and SVF in this respect in a more readable way, in Fig. 9 we show the AVF, PVF, and SVF results, *separately* for the two fault effect classes: Crashes and SDCs. As we can see in Fig. 9, Crashes and SDCs also present opposite trends between AVF and PVF/SVF, as in the total vulnerability we presented previously in Fig. 4. Consider for example the *sha* and *smooth* benchmarks. While the AVF shows that these benchmarks primarily suffer from Crashes, the PVF and SVF show the *opposite* trend, i.e., it primarily suffers from SDCs. Therefore, leading by PVF or SVF results, we would incorrectly conclude that if a fault hits the microprocessor, there is a high possibility for the application to produce an SDC rather than a Crash. Apparently, a software designer would *mistakenly* decide to protect these applications against SDCs. Similarly, based on the PVF and SVF results, the SDC probability of *sha* and *smooth* is the highest one compared to all other benchmarks that also provide opposite trends. Therefore, a designer would decide to apply a software-based fault-tolerance technique to protect these benchmarks from hardware faults leading to SDCs, although the actual AVF shows that they are among the less vulnerable ones, especially regarding the SDCs. To this end, *sha* and *smooth* benchmarks are perfect candidates to conduct our case study, to demonstrate the dramatic impact of these pitfalls in the next subsection.

B. Case Study: Software-Based Fault Tolerance

We conduct a case study using the *sha* (Fig. 10) and *smooth* (Fig. 11) benchmarks in order to demonstrate that if

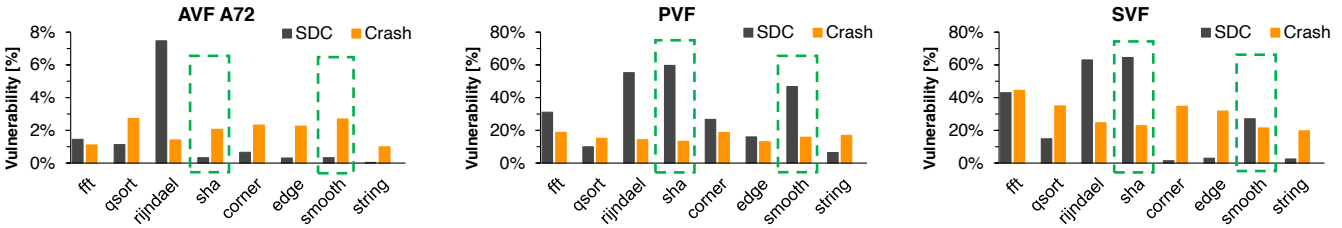

Fig. 9. Fine-grained Crash and SDC vulnerability across software-layer (SVF), architectural-layer (PVF), and cross-layer (AVF).

we were based on the results provided either from PVF or from SVF analysis, not only we would protect the wrong application (a robust one), but also, we would apply a protection scheme for the wrong fault effect (SDC instead of Crash). Our case study resembles as much as possible a state-of-the-art software-based fault tolerance method, which aims to harden programs against SDCs caused by soft errors [35]. We chose this fault-tolerance technique, because (1) it combines two popular approaches: the AN-encoding [58] [59] and duplicated instructions [37], (2) it provides very high probability for SDCs detection, and (3) it contains sufficient detail for reproduction with a reasonable accuracy (the source implementation of the method is not available).

By applying this technique, as it is expected, the execution time of the software-based fault tolerance implementation of *sha* has been increased by 2.1x and for *smooth* has been increased by 2.5x, i.e., a significant performance and energy penalty is paid (depending on the benchmark the technique increases the execution time by 2x to 4x [35]). In Fig. 10 and Fig. 11 we present our experimental results for *sha* and *smooth*, respectively, and for both the unprotected ("w/o" label) and protected ("w/" label) application. Note that the applied technique aims to detect hardware faults at the software level, and not to recover from them. Since our focus is not on proposing or evaluating fault-tolerance and recovery methods, the faults that are detected by the technique (shown with gray color in Fig. 10 and Fig. 11) are not considered in the total vulnerability of the fault-tolerant implementation, in which only the SDC and Crash fault effects are considered. The reason is that since a fault is detected, it can be also corrected by applying a recovery technique (such as a re-execution).

As shown in Fig. 10(c) and Fig. 10(d), when *sha* is hardened to detect faults, the PVF and SVF measurements are significantly reduced by up to 3.8x (excluding the detected faults and compared to the unprotected code), which is of course expected because the fault-tolerance technique aims at reducing the increased SDC vulnerability of the initial program implementation, since it exploits information redundancy and data duplication. Similarly, for Fig. 11(c) and Fig. 11(d), the PVF and SVF of *smooth* are reduced by up to 3.4x. On the contrary, we see in Fig. 10(b) and Fig. 11(b) that when the same application code (with and without the fault-tolerance technique) is evaluated through the microarchitecture-level fault injection providing the cross-layer AVF, the fault-tolerant version shows 30% and 10% increased vulnerability, respectively; i.e., the *opposite* result. This happens primarily because of the increased number of Crashes, which can be safely attributed to the prolonged execution time of the hardened code. This fact is supported by the data in Fig. 10(a) and Fig. 11(a) which show the detailed AVF analysis of the hardware components: the fault-tolerant version of the code *increases the vulnerability of all structures*, primarily because the Crashes have been increased. Note that the L2 cache holds the dominant role in the weighted AVF because it is the largest component (see discussed in section IV.A and Table II).

We can also observe that while we apply a software-based technique that tolerates SDCs, the AVF shows that the SDC vulnerability is slightly affected (Fig. 11 (b)). One reason is that software-based fault-tolerance techniques (such as the one we have been based on) protect only the local (user) code of a specific application and not the code of library calls, and thus, the kernel operations, that are also executed along with the user program. As a matter of fact, in our *sha* implementation, for example, the transfer of control from user space to kernel space when a system call is invoked, constitutes the 19.5% of the total execution time. Another reason can be attributed to the escaped faults, as we discussed in sections IV and V. Taking also into consideration that a small number of fault injection experiments in all five microarchitectural structures lead to SDC fault effect (as we described in subsection VI.A), the probability of this kind of fault effect to reside in a software-visible location of

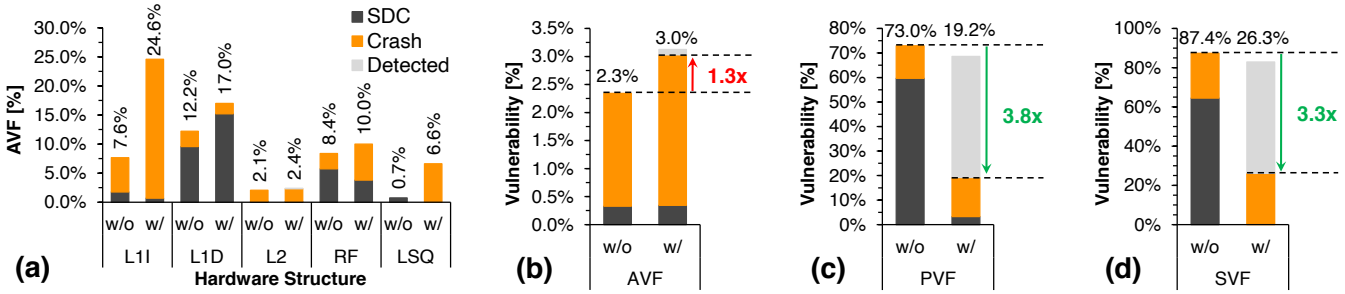

Fig. 10. Fine-grained results for PVF, SVF and AVF of Arm Cortex-A72 for all five hardware structures used in this study for the *sha* benchmark with the fault-tolerant implementation (w/) and without the fault-tolerant implementation (w/o).

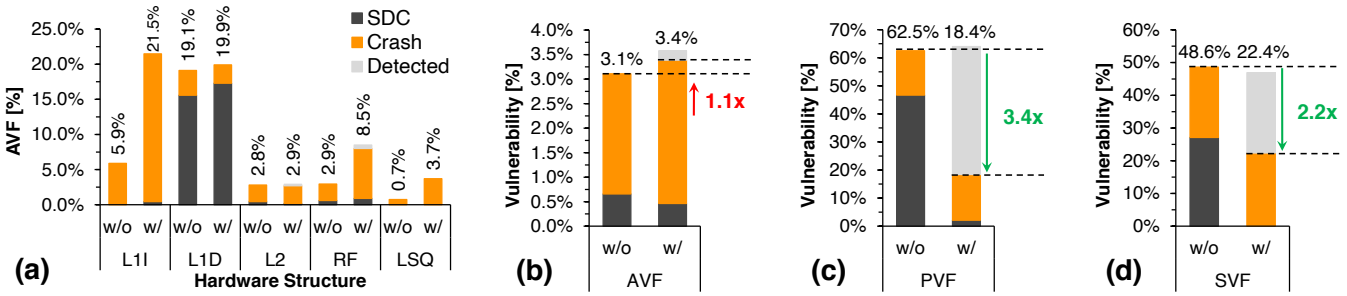

Fig. 11. Fine-grained results for PVF, SVF and AVF of Arm Cortex-A72 for all five hardware structures used in this study for the *smooth* benchmark with the fault-tolerant implementation (w/) and without the fault-tolerant implementation (w/o).

the user program, so that it would be able for the software-based fault tolerant technique to detect it, is extremely low. Therefore, a software-based technique can only protect a (likely negligible) subset of the whole execution space. This is a very important aspect, which neither software-level fault injection nor software-based fault-tolerant techniques can model and evaluate.

VII. Related Work & Discussion

Particle accelerators have been used for many years to measure and study the reliability of devices and applications [1] [60]. In [61]-[64] authors present beam experimental data on embedded Arm Cortex-A9, propose hardening solutions, and discuss the impact of the presence of an operating system in the application and device reliability. The reliability estimation of particle beaming can be considered very precise in the reported system-level FIT rates; however, it is limited to deliver coarse-grain system-level observations and no insights on the root causes of the failures. FPGAs have been also used for prototyping and reliability analysis of designs. Several fault injection frameworks have been presented for both Xilinx and Altera FPGAs [65]-[67]. The advantage of FPGA-based reliability assessment is the high throughput and level of accuracy. However, the circuit level detail requires the availability of a complete design, which can only be available on late design stages.

On the other hand, there are many methodologies in the literature that can be used to estimate, predict, or measure the reliability of a system at early design stages before the actual silicon comes in place and only models are available. These approaches vary in the level of detail (and accuracy), the required time for the estimation as well as the design time or lifetime stage at which they are available. There are two abstraction layers of the microprocessor design that can be used to evaluate its reliability at early design stages: Register Transfer Level (RTL) and microarchitecture-level models. The RTL model is the actual hardware design at almost full detail, while the microarchitecture-level model is a more abstract implementation of the design. Both allow high levels of observability and are available during the design cycle; microarchitecture-level models are available *very early* in design stages, while RTL models are available *very late* just before design signoff. These models of a microprocessor allow overcoming the limitations induced by using silicon prototypes, such as the lack of observability. But on the other hand, early design stage models suffer from other issues, which can be attributed to limited throughput and potential inaccuracies. These two drawbacks are inversely proportional: the higher the accuracy, the worse the throughput is and vice versa. Cho *et al.* [4] quantify the levels of inaccuracies between RTL and higher-level models (microarchitecture, architecture and software layer) and analyze the sources of these inaccuracies. However, this work *entirely excludes* all SRAM arrays of the microprocessor (register file, buffers, caches, etc.) which are considered protected. This is not a realistic assumption in most commercial microprocessors where several of the SRAM arrays can be left unprotected (e.g., the Samsung Exynos 5250, which is an Arm Cortex-A15 design [68], comes without any ECC protection scheme [69]). Particularly, at the microarchitecture-level, injections were only performed on logic gates, which have a very low raw failure rate (typically more than three orders of magnitude lower than the SRAM cells [46]) and random sequential elements. At the architecture-level injections were performed on architectural registers, and at the software-level injections were performed on variables. Moreover, [4] evaluates only a subset of fault effects. Our work complements the landscape of cross-layer vulnerability assessment and reveals pitfalls related to the microarchitecture, architecture, and software layers.

Multiple methodologies can be applied at different abstraction layers, during different periods of a system design or its lifetime. These methodologies are complementary to each other and all can contribute at a different level on aiding correct design decisions. Unfortunately, it remains unclear which of these techniques are closer to the ground truth and how they relate to each other; due to the high complexity, even the comparison of them is a real challenge. However, most of the research community focuses on architecture-level (PVF) and software-level (SVF) analysis to provide the full system's vulnerability because it can be performed very fast (compared to RTL or microarchitecture-level models) and can analyze very long-execution workloads. To this end, several studies have been presented that focus on detecting hardware faults by monitoring the software behavior [70]-[77]. Architecture-level and software-level evaluation tools are widely used to assess systems at native speeds [12]-[19], [25]-[30], [57]. Although these techniques allow fast assessment on long workloads, they completely fail to capture the underlying hardware vulnerability as the starting point of the experiment (origin of fault) is a corrupted instruction or the program flow (not a micro-

architectural structure). Therefore, they can deliver similar vulnerability estimations on different hardware platforms of the same ISA. In this paper we have shown that although PVF and SVF studies seem to be useful for improving the reliability on the software side, they come with pitfalls on the vulnerability estimation of a system when software engineers are based on these results to develop fault-tolerant or other mitigation methods for their software.

Per our comprehensive analysis, these pitfalls of any partial measurement of the AVF (i.e., SVF or PVF, even in refined forms) can lead to protection decisions that can degrade reliability. Since the complete full-system AVF measurement remains the only valid guide to correct and cost-effective protection approaches against transient faults, the fast and accurate delivery of complete end-to-end AVF ratings of processors and workloads is of paramount importance. To this aim different paths should be followed individually or combined: (a) microarchitecture-level analytical approaches such as ACE should significantly improve the accuracy while retaining their speed advantages along the lines of [78], (b) microarchitecture-level fault injection AVF approaches should further improve their throughput while retaining their already accurate measurements along the lines of [9] [11].

VIII. Conclusion

The system vulnerability stack is a concept that divides the system into different layers that are seemingly independent, and a soft error vulnerability evaluation can be performed separately on each of them. However, any modification in the hardware or the software will result in a change on the system setup and will consequently render previous estimations stale. As we show in this paper, and unlike common belief and practice, the vulnerability analysis at the architecture level or at the software level, delivers distorted results which can subsequently lead to design pitfalls. Our results show that when two different programs are compared through fault injection at the software or the architecture layer, their actual vulnerability is very often measured to be the *opposite* of what the correct full system analysis reports. We analyzed in detail the reasons of this phenomenon and pinpointed the critical weaknesses that vulnerability evaluation at the architectural or software layer have. We show how the PVF (or even a refined PVF supported by hardware statistics) and the SVF estimations can deliver contradicting results against the full-stack AVF and that decisions taken to reduce PVF or SVF can, in fact, degrade reliability, i.e., increase the AVF.

Acknowledgment

This research has been supported by European Union's Horizon 2020 research programme under the Tetramax project (Grant 761349), the UniServer project (Grant 688540), and the FP7 Clereco project (Grant 611404). The authors are grateful to Athanasios Chatzidimitriou for setting up the early experiments of this work, and warmly thank Vilas Sridharan and the five anonymous ISCA reviewers for valuable comments and suggestions.

References

[1] R. C. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," in IEEE Transactions on Device and Materials Reliability, vol. 5, no. 3, pp. 305-316, Sept. 2005, doi: 10.1109/TDMR.2005.853449.

[2] C. Constantinescu, "Trends and challenges in VLSI circuit reliability," in IEEE Micro, vol. 23, no. 4, pp. 14-19, July-Aug. 2003, doi: 10.1109/MM.2003.1225959.

[3] S. R. Nassif, N. Mehta, and Y. Cao, "A resilience roadmap," 2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010), Dresden, 2010, pp. 1011-1016, doi: 10.1109/DATE.2010.5456958.

[4] H. Cho, S. Mirkhani, C. Cher, J. A. Abraham and S. Mitra, "Quantitative evaluation of soft error injection techniques for robust system design," 2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC), Austin, TX, 2013, pp. 1-10, doi: 10.1145/2463209.2488859.

[5] X. Iturbe, B. Venu and E. Ozer, "Soft error vulnerability assessment of the real-time safety-related ARM Cortex-R5 CPU," 2016 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), Storrs, CT, 2016, pp. 91-96, doi: 10.1109/DFT.2016.7684076.

[6] J. Blome, S. Mahlke, D. Bradley and K. Flautner, "A Microarchitectural Analysis of Soft Error Propagation in a Production-Level Embedded Microprocessor," In Proc. First Workshop on Architectural Reliability, Nov. 2005.

[7] M. Maniatakos, N. Karimi, C. Tirumurti, A. Jas and Y. Makris, "Instruction-Level Impact Analysis of Low-Level Faults in a Modern Microprocessor Controller," in IEEE Transactions on Computers, vol. 60, no. 9, pp. 1260-1273, Sept. 2011, doi: 10.1109/TC.2010.60.

[8] N. J. Wang, A. Mahesri, and S. J. Patel, "Examining ACE Analysis Reliability Estimates using Fault-Injection," 2007 34th International Symposium on Computer Architecture (ISCA '07), San Diego, California, USA, pp. 460–469, doi: 10.1145/1250662.1250719.

[9] A. Chatzidimitriou and D. Gizopoulos, "Anatomy of microarchitecture-level reliability assessment: Throughput and accuracy," 2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Uppsala, 2016, pp. 69-78, doi: 10.1109/ISPASS.2016.7482075.

[10] G. Yalcin, O. S. Unsal, A. Cristal and M. Valero, "FIMSIM: A fault injection infrastructure for microarchitectural simulators," 2011 IEEE 29th International Conference on Computer Design (ICCD), Amherst, MA, 2011, pp. 431-432, doi: 10.1109/ICCD.2011.6081435.

[11] M. Kaliorakis, D. Gizopoulos, R. Canal and A. Gonzalez, "MeRLiN: Exploiting dynamic instruction behavior for fast and accurate microarchitecture level reliability assessment," 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA), Toronto, ON, 2017, pp. 241-254, doi: 10.1145/3079856.3080225.

[12] J. Carreira, H. Madeira and J. G. Silva, "Xception: Software Fault Injection and Monitoring in Processor Functional Units," in Conference on Dependable Computing for Critical Applications, Sept. 1995, pp. 135-149.

[13] H. Madeira, D. Costa and M. Vieira, "On the emulation of software faults by software fault injection," Proceeding International Conference on Dependable Systems and Networks. DSN 2000, New York, NY, USA, 2000, pp. 417-426, doi: 10.1109/ICDSN.2000.857571.

[14] R. Natella, D. Cotroneo, J. A. Duraes and H. S. Madeira, "On Fault Representativeness of Software Fault Injection," in IEEE Transactions on Software Engineering, vol. 39, no. 1, pp. 80-96, Jan. 2013, doi: 10.1109/TSE.2011.124.

[15] J. Wei, A. Thomas, G. Li and K. Pattabiraman, "Quantifying the Accuracy of High-Level Fault Injection Techniques for Hardware Faults," 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Atlanta, GA, 2014, pp. 375-382, doi: 10.1109/DSN.2014.2.

[16] Q. Lu, M. Farahani, J. Wei, A. Thomas and K. Pattabiraman, "LLFI: An Intermediate Code-Level Fault Injection Tool for Hardware Faults," 2015 IEEE International Conference on Software Quality,

Reliability and Security, Vancouver, BC, 2015, pp. 11-16, doi: 10.1109/QRS.2015.13.

[17] A. Jin, J. Jiang, J. Hu and J. Lou, "A PIN-Based Dynamic Software Fault Injection System," 2008 The 9th International Conference for Young Computer Scientists, Hunan, 2008, pp. 2160-2167, doi: 10.1109/ICYCS.2008.329.

[18] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, "FERRARI: a tool for the validation of system dependability properties," presented at the 1992 FTCS - The Twenty-Second International Symposium on Fault-Tolerant Computing, 1992.

[19] S. Jha et al., "ML-Based Fault Injection for Autonomous Vehicles: A Case for Bayesian Fault Injection," IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2019, doi: 10.1109/dsn.2019.00025.

[20] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36., San Diego, CA, USA, 2003, pp. 29-40, doi: 10.1109/MICRO.2003.1253181.

[21] R. Leveugle, A. Calvez, P. Maistri and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence," 2009 Design, Automation & Test in Europe Conference & Exhibition, Nice, 2009, pp. 502-506, doi: 10.1109/DATE.2009.5090716.

[22] V. Sridharan and D. R. Kaeli, "Using Hardware Vulnerability Factors to Enhance AVF Analysis," In Proceedings of the 37th annual international symposium on Computer architecture (ISCA '10), Saint-Malo, France, pp. 461–472, 2010, doi: 10.1145/1815961.1816023.

[23] V. Sridharan and D. R. Kaeli, "Eliminating microarchitectural dependency from Architectural Vulnerability," 2009 IEEE 15th International Symposium on High Performance Computer Architecture, Raleigh, NC, 2009, pp. 117-128, doi: 10.1109/HPCA.2009.4798243.

[24] V. Sridharan and D. R. Kaeli, "Quantifying Software Vulnerability," In Proceedings of the 2008 Workshop on Radiation Effects and Fault Tolerance in Nanometer Technologies (WREFT '08), Ischia, Italy, pp. 323–328, 2008, doi: 10.1145/1366224.1366225.

[25] S. Han, K. G. Shin and H. A. Rosenberg, "DOCTOR: an integrated software fault injection environment for distributed real-time systems," Proceedings of 1995 IEEE International Computer Performance and Dependability Symposium, Erlangen, Germany, 1995, pp. 204-213, doi: 10.1109/IPDS.1995.395831.

[26] T. K. Tsai, R. K. Iyer and D. Jewitt, "An approach towards benchmarking of fault-tolerant commercial systems," Proceedings of Annual Symposium on Fault Tolerant Computing, Sendai, Japan, 1996, pp. 314-323, doi: 10.1109/FTCS.1996.534616.

[27] S. K. S. Hari, S. V. Adve, H. Naeimi and P. Ramachandran, "Relyzer: Exploiting Application-level Fault Equivalence to Analyze Application Resiliency to Transient Faults," In Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII), London, England, UK, pp. 123–134, 2012, doi: 10.1145/2150976.2150990.

[28] S. K. S. Hari, R. Venkatagiri, S. V. Adve and H. Naeimi, "GangES: Gang error simulation for hardware resiliency evaluation," 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA), Minneapolis, MN, 2014, pp. 61-72, doi: 10.1109/ISCA.2014.6853212.

[29] R. Venkatagiri, A. Mahmoud, S. K. S. Hari and S. V. Adve, "Approxilyzer: Towards a systematic framework for instruction-level approximate computing and its application to hardware resiliency," 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Taipei, 2016, pp. 1-14, doi: 10.1109/MICRO.2016.7783745.

[30] R. Venkatagiri et al., "gem5-Approxilyzer: An Open-Source Tool for Application-Level Soft Error Analysis," 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Portland, OR, USA, 2019, pp. 214-221, doi: 10.1109/DSN.2019.00033.

[31] S. Mukherjee, "Architecture Design for Soft Errors," Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[32] B. Fang, Q. Lu, K. Pattabiraman, M. Ripeanu and S. Gurumurthi, "ePVF: An Enhanced Program Vulnerability Factor Methodology for Cross-Layer Resilience Analysis," 2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Toulouse, 2016, pp. 168-179, doi: 10.1109/DSN.2016.24.

[33] M. Wilkening, F. Previlon, D. R. Kaeli, S. Gurumurthi, S. Raasch, and V. Sridharan, "Evaluating the Resilience of Parallel Applications," IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2018, doi: 10.1109/dft.2018.8602987.

[34] M. Kaliorakis, S. Tselonis, A. Chatzidimitriou, N. Foutris and D. Gizopoulos, "Differential Fault Injection on Microarchitectural Simulators," 2015 IEEE International Symposium on Workload Characterization, Atlanta, GA, 2015, pp. 172-182, doi: 10.1109/IISWC.2015.28.

[35] D. Kuvaiskii and C. Fetzer, "Δ-Encoding: Practical Encoded Processing," 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Rio de Janeiro, 2015, pp. 13-24, doi: 10.1109/DSN.2015.20.

[36] M. Didehban and A. Shrivastava, "nZDC: A compiler technique for near Zero Silent Data Corruption," in 53rd ACM/EDAC/IEEE Design Automation Conference (DAC), Austin, TX, 2016, pp. 1-6, doi: 10.1145/2897937.2898054.

[37] N. Oh, P. P. Shirvani and E. J. McCluskey, "Error detection by duplicated instructions in super-scalar processors," in IEEE Transactions on Reliability, vol. 51, no. 1, pp. 63-75, March 2002, doi: 10.1109/24.994913.

[38] Y. Shen, G. Heiser, and K. Elphinstone, "Fault Tolerance Through Redundant Execution on COTS Multicores: Exploring Trade-Offs," IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), 2019, doi: 10.1109/dsn.2019.00031.

[39] B. Sangchoolie, K. Pattabiraman and J. Karlsson, "One Bit is (Not) Enough: An Empirical Study of the Impact of Single and Multiple Bit-Flip Errors," 2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Denver, CO, 2017, pp. 97-108, doi: 10.1109/DSN.2017.30.

[40] T. Tsai and J. Huang, "Source code transformation for software-based on-line error detection," 2017 IEEE Conference on Dependable and Secure Computing, Taipei, 2017, pp. 305-309, doi: 10.1109/DESEC.2017.8073852.

[41] A. Chan, S. Winter, H. Saissi, K. Pattabiraman and N. Suri, "IPA: Error Propagation Analysis of Multi-Threaded Programs Using Likely Invariants," 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST), Tokyo, 2017, pp. 184-195, doi: 10.1109/ICST.2017.24.

[42] G. Li, K. Pattabiraman, C. Cher and P. Bose, "Understanding Error Propagation in GPGPU Applications," SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, Salt Lake City, UT, 2016, pp. 240-251, doi: 10.1109/SC.2016.20.

[43] A. Vallero, A. Savino, A. Chatzidimitriou, M. Kaliorakis, M. Kooli, M. R. Villanueva, G. D. Natale, A. Bosio, R. Canal, D. Gizopoulos and S. D. Carlo, "SyRA: Early System Reliability Analysis for Cross-Layer Soft Errors Resilience in Memory Arrays of Microprocessor Systems," in IEEE Transactions on Computers, vol. 68, no. 5, pp. 765-783, May 2019, doi: 10.1109/TC.2018.2887225.

[44] A. Vallero et al., "Cross-layer reliability evaluation, moving from the hardware architecture to the system level: A CLERECO EU project overview," Microprocessors and Microsystems, vol. 39, no. 8, pp. 1204–1214, Nov. 2015, doi: 10.1016/j.micpro.2015.06.003.

[45] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 simulator," ACM SIGARCH Computer Architecture News, vol. 39, no. 2, 1-7, Aug. 2011, doi: 10.1145/2024716.2024718.

[46] S. Mitra, N. Seifert, M. Zhang, Q. Shi and K. S. Kim, "Robust system design with built-in soft-error resilience," in Computer, vol. 38, no. 2, pp. 43-52, Feb. 2005, doi: 10.1109/MC.2005.70.

[47] A. Chatzidimitriou, G. Papadimitriou, C. Gavanas, G. Katsoridas and D. Gizopoulos, "Multi-Bit Upsets Vulnerability Analysis of Modern Microprocessors," 2019 IEEE International Symposium on Workload Characterization (IISWC), Orlando, FL, USA, 2019, pp. 119-130, doi: 10.1109/IISWC47752.2019.9042036.

[48] R. Baumann, "Soft errors in advanced computer systems," in IEEE Design & Test of Computers, vol. 22, no. 3, pp. 258-266, May-June 2005, doi: 10.1109/MDT.2005.69.

[49] Ampere Computing, Ampere® eMAG® 8180 Product Brief, 2020, https://bit.ly/35DoKXT [Accessed online at Nov. 18, 2020].

[50] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538), Austin, TX, USA, 2001, pp. 3-14, doi: 10.1109/WWC.2001.990739.

[51] A. Chatzidimitriou, P. Bodmann, G. Papadimitriou, D. Gizopoulos and P. Rech, "Demystifying Soft Error Assessment Strategies on ARM CPUs: Microarchitectural Fault Injection vs. Neutron Beam Experiments," 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Portland, OR, USA, 2019, pp. 26-38, doi: 10.1109/DSN.2019.00018.

[52] N. J. George, C. R. Elks, B. W. Johnson and J. Lach, "Transient fault models and AVF estimation revisited," 2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN), Chicago, IL, 2010, pp. 477-486, doi: 10.1109/DSN.2010.5544276.

[53] D. S. Khudia and S. Mahlke, "Harnessing Soft Computations for Low-Budget Fault Tolerance," 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture, Cambridge, 2014, pp. 319-330, doi: 10.1109/MICRO.2014.33.

[54] A. A. Nair, L. K. John and L. Eeckhout, "AVF Stressmark: Towards an Automated Methodology for Bounding the Worst-Case Vulnerability to Soft Errors," 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture, Atlanta, GA, 2010, pp. 125-136, doi: 10.1109/MICRO.2010.34.

[55] Z. Zhao, D. Lee, A. Gerstlauer, L. K. John, “Host-compiled reliability modeling for fast estimation of architectural vulnerabilities”, SELSE 2015.

[56] J. L. Henning, “SPEC CPU2006 benchmark descriptions,” ACM SIGARCH Computer Architecture News, vol. 34, no. 4, pp. 1–17, Sept. 2006, doi: 10.1145/1186736.1186737.

[57] L. Palazzi, G. Li, B. Fang and K. Pattabiraman, "A Tale of Two Injectors: End-to-End Comparison of IR-Level and Assembly-Level Fault Injection," 2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE), Berlin, Germany, 2019, pp. 151-162, doi: 10.1109/ISSRE.2019.00024.

[58] U. Schiffel, “Hardware error detection using AN-codes,” Ph.D. dissertation, Technische Universitat Dresden, 2011.

[59] D. T. Brown, "Error Detecting and Correcting Binary Codes for Arithmetic Operations," in IRE Transactions on Electronic Computers, vol. EC-9, no. 3, pp. 333-337, Sept. 1960, doi: 10.1109/TEC.1960.5219855.

[60] J. F. Ziegler and H. Puchner, “SER--history, Trends and Challenges: A Guide for Designing with Memory ICs”, Cypress, 2004.

[61] T. Santini, L. Carro, F. Rech Wagner and P. Rech, "Reliability Analysis of Operating Systems and Software Stack for Embedded Systems," in IEEE Transactions on Nuclear Science, vol. 63, no. 4, pp. 2225-2232, Aug. 2016, doi: 10.1109/TNS.2015.2513384.

[62] A. B. de Oliveira, G. S. Rodrigues and F. L. Kastensmidt, "Analyzing lockstep dual-core ARM cortex-A9 soft error mitigation in FreeRTOS applications," 2017 30th Symposium on Integrated Circuits and Systems Design (SBCCI), Fortaleza, 2017, pp. 84-89, doi: 10.1145/3109984.3110008.

[63] A. Martínez-Álvarez, F. Restrepo-Calle, S. Cuenca-Asensi, L. M. Reyneri, A. Lindoso and L. Entrena, "A Hardware-Software Approach for On-Line Soft Error Mitigation in Interrupt-Driven Applications," in IEEE Transactions on Dependable and Secure Computing, vol. 13, no. 4, pp. 502-508, 1 July-Aug. 2016, doi: 10.1109/TDSC.2014.2382593.

[64] V. Fratin, D. Oliveira, C. Lunardi, F. Santos, G. Rodrigues and P. Rech, "Code-Dependent and Architecture-Dependent Reliability Behaviors," 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Luxembourg City, 2018, pp. 13-26, doi: 10.1109/DSN.2018.00015.

[65] A. Sari and M. Psarakis, "A fault injection platform for the analysis of soft error effects in FPGA soft processors," 2016 IEEE 19th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS), Kosice, 2016, pp. 1-6, doi: 10.1109/DDECS.2016.7482459.

[66] S. Di Carlo, P. Prinetto, D. Rolfo and P. Trotta, "A fault injection methodology and infrastructure for fast single event upsets emulation on Xilinx SRAM-based FPGAs," 2014 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), Amsterdam, 2014, pp. 159-164, doi: 10.1109/DFT.2014.6962073.

[67] G. L. Nazar and L. Carro, "Fast single-FPGA fault injection platform," 2012 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), Austin, TX, 2012, pp. 152-157, doi: 10.1109/DFT.2012.6378216.

[68] ARM® Cortex®-A15 MPCore™ Processor, Revision: r4p0, Technical Reference Manual, 2013.

[69] Samsung Exynos 5 dual (Exynos 5250) user’s manual, Oct. 2012, https://bit.ly/3fjhR18 [Accessed online at Nov. 18, 2020].

[70] N. J. Wang and S. J. Patel, "ReStore: symptom based soft error detection in microprocessors," 2005 International Conference on Dependable Systems and Networks (DSN'05), Yokohama, Japan, 2005, pp. 30-39, doi: 10.1109/DSN.2005.82.

[71] N. Nakka, G. P. Saggese, Z. Kalbarczyk, and R. K. Iyer, “An Architectural Framework for Detecting Process Hangs/Crashes,” in Dependable Computing (EDCC), Springer Berlin Heidelberg, 2005, pp. 103–121, doi: 10.1007/11408901_8.

[72] K. Pattabiraman, G. P. Saggese, D. Chen, Z. Kalbarczyk and R. K. Iyer, "Dynamic Derivation of Application-Specific Error Detectors and their Implementation in Hardware," 2006 Sixth European Dependable Computing Conference, Coimbra, 2006, pp. 97-108, doi: 10.1109/EDCC.2006.9.

[73] P. Racunas, K. Constantinides, S. Manne and S. S. Mukherjee, "Perturbation-based Fault Screening," 2007 IEEE 13th International Symposium on High Performance Computer Architecture, Scottsdale, AZ, 2007, pp. 169-180, doi: 10.1109/HPCA.2007.346195.

[74] V. K. Reddy, A. S. Al-Zawawi and E. Rotenberg, "Assertion-Based Microarchitecture Design for Improved Fault Tolerance," 2006 International Conference on Computer Design, San Jose, CA, 2006, pp. 362-369, doi: 10.1109/ICCD.2006.4380842.

[75] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, D. I. August, and S. S. Mukherjee, “Software-controlled fault tolerance,” ACM Transactions on Architecture and Code Optimization (TACO), vol. 2, no. 4, pp. 366–396, Dec. 2005, doi: 10.1145/1113841.1113843.

[76] R. Venkatasubramanian, J. P. Hayes and B. T. Murray, "Low-cost on-line fault detection using control flow assertions," 9th IEEE On-Line Testing Symposium, (IOLTS), Kos Island, Greece, 2003, pp. 137-143, doi: 10.1109/OLT.2003.1214380.

[77] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou, “Understanding the propagation of hard errors to software and implications for resilient system design,” in ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Seattle, WA, USA, pp. 265–276, 2008, doi: 10.1145/1346281.1346315.

[78] S. Raasch, A. Biswas, J. Stephan, P. Racunas and J. Emer, "A fast and accurate analytical technique to compute the AVF of sequential bits in a processor," 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Waikiki, HI, 2015, pp. 738-749, doi: 10.1145/2830772.2830829.