2019 IEEE/ACM Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS)

# Evaluating Compiler IR-Level Selective Instruction Duplication with Realistic Hardware Errors

Chun-Kai Chang
University of Texas at Austin
chunkai@utexas.edu

Guanpeng Li
University of British Columbia
gpli@ece.ubc.ca

Mattan Erez
University of Texas at Austin
mattan.erez@utexas.edu

***Abstract*—Hardware faults (i.e., soft errors) are projected to increase in modern HPC systems. The faults often lead to error propagation in programs and result in silent data corruptions (SDCs), seriously compromising system reliability. Selective instruction duplication, a widely used software-based error detector, has been shown to be effective in detecting SDCs with low performance overhead. In the past, researchers have relied on compiler intermediate representation (IR) for program reliability analysis and code transformation in selective instruction duplication. However, they assumed that the IR-based analysis and protection are representative under realistic fault models (i.e., faults originated at lower hardware layers). Unfortunately, the assumptions have not been fully validated, leading to questions about the accuracy and efficiency of the protection since IR is a higher level of abstraction and far away from hardware layers. In this paper, we verify the assumption by injecting realistic hardware faults to programs that are guided and protected by IR-based selective instruction duplication. We find that the protection yields high SDC coverage with low performance overhead even under realistic fault models, albeit a small amount of such faults escaping the detector. Our observations confirm that IR-based selective instruction duplication is a cost-effective method to protect programs from soft errors.**

***Keywords*-Fault-tolerance; fault-injection; resilience; instruction-duplication;**

## I. Introduction

Transient hardware faults are predicted to increase in future high performance computing (HPC) systems due to growing system scale, progressive technology scaling, and lowering operating voltages [1]. These faults are also called soft errors as they do not cause any permanent damages to hardware components. In the past, HPC systems were protected through hardware-only solutions such as hardware redundancy and circuit hardening techniques. However, these techniques incur large overheads in performance and energy consumption, making them increasingly challenging to deploy in modern HPC systems [2]. As a result, researchers have postulated that future systems will expose hardware faults to the software and expect the software to tolerate them [3], [4], [5], [6].

One infamous consequence of soft error is silent data corruption (SDC), which is incorrect program output without any user awareness. SDCs are very difficult to detect and can lead to severe consequences [1], [7]. Studies have shown that only a small fraction of instructions are responsible for almost all the SDCs in a program [7], [6], [5], [8]. Therefore, developers can identify them and selectively duplicate the vulnerable instructions to protect the program from SDCs with low overhead — this is also known as selective instruction duplication which is widely used in modern HPC systems [9], [10], [8].

In the past, researchers have relied on compiler intermediate representation (IR), such as LLVM IR [11], in the ecosystem of selective instruction duplication [12], [9], [4], [13], [14]. This includes program vulnerability analysis, fault injection evaluation, and code transformation when guiding the protection and implementing the duplication. This is because LLVM is platform-independent and largely supported as open-source tools across research communities and industry, leveraging IR-based infrastructure allows developers to easily pinpoint their analysis to the protection in given programs [12], [4], [9].

Since IR is a higher level of abstraction of program and far away from lower hardware layers where soft errors originate, debates remain in the accuracy and efficiency of the IR-based protection. Therefore, researchers have made two assumptions when using IR in selective instruction duplication, which have yet to be validated: (1) A fine-grained IR-level analysis is accurate enough to project the vulnerability of its lower layer counterparts (i.e., generated binary), and (2) instruction duplication at IR-level captures realistic hardware faults. Although researchers have investigated the accuracy of IR-based fault injection [15], [12], [16], they focused on the programs without any protections and did not use realistic fault models (i.e., faults injected from lower hardware layers) in their evaluations. Therefore, the answer to the question remains unclear. Our long-term goal is to validate the aforementioned assumptions and settle the problem, which is critical as there has been a large amount of work in dependability and HPC communities relying on compiler IR in their protections [6], [13], [17], [18].

In this paper, we take the first step to validate the assumptions by evaluating SDC coverage in IR-based selective instruction duplication. We first increase the amount of IR instructions for duplication, then compare the SDC coverage measured by both IR-level and hardware-level fault injections. When injecting faults at IR level, we use popular single-bit flip fault model for the measurement, as it is widely used to estimate the SDC coverage when guiding the protection [12], [4], [8]. For the ground truth, we employ hardware-level fault injection techniques [19] at gate-level and binary-level to examine any discrepancy of SDC coverage. *To the best of our*

978-1-7281-6013-9/19/$31.00 ©2019 IEEE
DOI 10.1109/FTXS49593.2019.00010

*knowledge, we are the first ones who examine the efficiency of IR-based selective instruction duplication using realistic fault models.*

Our main contributions and findings are as follows:

- We examine the efficiency of IR-based selective instruction duplication using realistic fault models in 8 benchmarks with various amount of instructions duplicated. We find that the SDC coverage of the protection measured under the realistic fault models is mostly similar to the anticipated coverage derived by IR-based analysis and fault injection. This indicates that IR-based selective instruction duplication is able to provide reasonable trade-off between SDC coverage and performance overhead claimed in prior studies [4], [13], [8].
- However, we find that a small amount (0.25% on average) of realistic hardware errors escape the detection (even with full IR instruction duplication). This is because some assembly instructions are not present at the IR level, so duplicating all IR instructions may still fail to protect these assembly instructions.
- Our takeaway is that IR-based selective instruction duplication is a cost-effective way to protect programs when developers want to reach certain SDC coverage given a limited overhead budget. This is a typical application scenario in commodity systems where a good-enough reliability on the cheap is highly desirable. In contrast, IR-based selective instruction duplication should not be used in safety- or reliability-critical tasks where SDCs need to be completely eliminated.
- In our fault injection evaluations, we also compare the impact of different fault models. Similar to prior observations [19], injecting single-bit flip faults at the binary-level yields similar outcome distributions to that of a RTL-level fault injection in the evaluation of selective instruction duplication. The SDC coverage of the protection measured by the two fault models are highly consistent.

## II. Background

### A. Fault Models

We define *faults* as physical events that affect hardware components. If a fault eventually changes the architectural state, it becomes an *error*. We focus on *soft errors* in particular since they are random and transient in nature and thus hard to detect. In contrast, permanent faults that frequently lead to errors are typically detected and do not corrupt any results silently.

We assume that faults escape electrical masking and temporal masking in this paper; that is, we only consider particle strikes that carry sufficient charge and result in faulty signals that arrive on time at the next latch. However, we respect logical masking because it depends not only on the circuit but also on the application. In other words, *we are concerned with the impact of errors, but not the rate of errors*. For definitions of masking factors, see [19].

Also, we assume on-chip SRAM and system DRAM are protected by ECC. As a result, we only inject errors to instructions using arithmetic and logic units because errors occurring within those circuits directly affect application data and thus more likely lead to SDCs. The same is assumed in previous work [12], [20], [5].

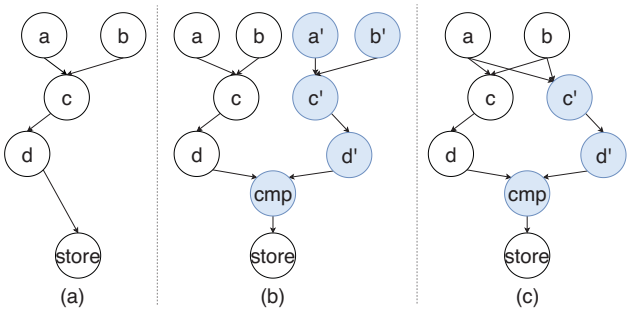


Fig. 1: An example of instruction duplication methods. (a) original dataflow graph. (b) full instruction duplication. (c) selective instruction duplication.

### B. Selective Instruction Duplication

Although hardware-based fault detection techniques provide high coverage, they are costly in terms of area and power, such that they are only adopted by mission-critical systems. As power constraints pose challenges for future HPC systems, researchers have been advocating software-based alternatives to harden systems against hardware errors. Software detectors are attractive because they are both flexible and efficient, and require no hardware modifications. In terms of flexibility, they are hardware-agnostic and can be enabled only for target applications. In terms of efficiency, software detectors can focus on critical parts of the applications to maximize error coverage given fixed cost of performance overhead. Furthermore, software detectors are shown to provide high coverage for hardware errors [7], [6], [5], [18], [17].

Specifically, we focus on detectors based on instruction duplication (Figure 1), a technique that detects errors by inserting redundant instructions and checking instructions at compile time [21], [7], [6], [17]. The naive method known as full duplication duplicates all instructions in the program but incurs significant performance overhead. On the other hand, selective instruction duplication protects a subset of instructions to maximize error coverage at the cost of reasonable performance overhead. This is because not all instructions in a program are equally likely to be vulnerable. In fact, only a small amount of instructions in a program are responsible for almost all the failures. So protecting these highly vulnerable instructions with priority gives developers a reasonable trade-off between the coverage and performance overhead.

Evaluating the cost-effectiveness of selective instruction duplication requires an appropriate fault model applied at the right abstraction layer. Both the level of injection and the fault model can affect the accuracy of the whole evaluation. Most selective instruction duplication techniques are guided, implemented, and measured at the LLVM IR level, with the single-bit flips introduced at IR instructions in their evaluations

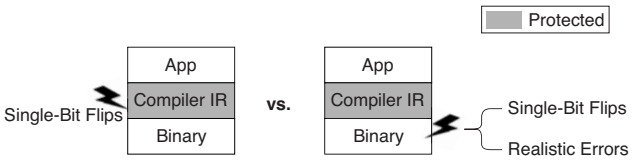


Fig. 2: Scope of this work. Shading denotes the level where protection is applied.

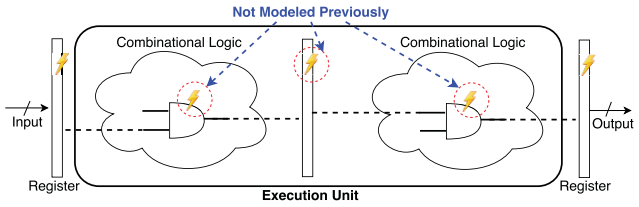


Fig. 3: Hardware faults that are not modeled by high-level error injection [19].

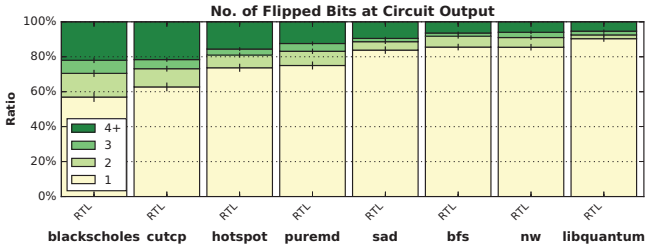


Fig. 4: Distributions of how many bits are flipped if a fault propagates to circuit output.

(Figure 2 left) [8], [17], [4], [13]. It is not clear whether existing techniques are still cost-effective if realistic hardware errors are injected at a lower hardware layers (Figure 2 right). In this paper, we re-examine the cost-effectiveness of IR-based selective instruction duplication by injecting realistic hardware errors at the binary level and RTL level.

To quantify cost-effectiveness, we examine the SDC coverage as we increase the amount of instructions being duplicated [5], [22]. By plotting this, we derive a protection curve for a given program and fault model. The protection curve is a graphical representation that helps researchers trade-off SDC coverage and performance overhead. The x-axis is the protection level (i.e., the amount of dynamic instruction duplicated). For instance, a protection level of 50% means that at most half of the total dynamic instructions are duplicated for protection. The y-axis is the SDC coverage, the reduction of SDC probability after protection divided by the SDC probability before protection. For example, if the initial SDC probability is 20% and the resultant SDC probability is 10% for some protection level, then the SDC coverage at the protection level is 50%. See Figure 5 for examples of protection curves.

### C. LLVM Compiler

LLVM-based tools have been widely used in the literature for resilience studies and selective instruction duplication [6], [20], [12], [4], [9], [15], [8]. The reasons are threefold: (1) LLVM is an open-source compiler infrastructure and has strong support from both academia and industry communities, providing an abundant third-party compiler passes and tools for code analysis and transformation. (2) LLVM IR is platform-independent and thereby portable to different architectures, allowing developers to focus on program-level error resilience characteristics. (3) LLVM IR is close to assembly code yet preserves high-level code structure for the ease of program analysis, allowing researchers to target specific code for the evaluation and protection. In the past, researchers have relied on LLVM to analyze programs, guide the selection of vulnerable instructions, implement instruction duplication, and estimate protection coverage [6], [20], [12], [4], [9], [15], [8]. Implementation details of selective instruction duplication using LLVM in this work are presented in Section III-A.

### D. Need of a Realistic Fault Model

We use an RTL fault model as a ground truth because it is not clear whether other fault models (i.e., single-bit flip at IR level) can accurately model soft errors. Consider a fault that occurs at a random location within a circuit module consisting of an input buffer, combinational logic, internal pipeline buffers, and an output buffer (Figure 3). Note that soft errors can be grouped based on their initial fault site within the circuit.

First, consider the faults that occur at either the input buffer or the output buffer. When a fault happens at the output buffer, it directly manifests as an error. On the other hand, when a fault occurs in the input buffer, it can be masked by the operation (e.g., erroneous bits are shifted out). Such errors can be modeled by bit flips because they either directly affect the output or logical masking can be modeled by running the operation with erroneous inputs. These soft errors are already modeled in previous work via injecting errors into instruction operands [15], [4], [23], [24], [25], [12].

Next consider the case where the fault site is at either a logic gate within combinational logic or internal pipeline buffers. Here we assume the fault induces a pulse that flips the output of the affected unit. Although this faulty signal may be masked logically before propagating to output buffer, it is possible that it leads to a soft error that corrupts multiple bits of the output buffer. Because the exact impact of the soft error on the output buffer depends on the initial fault site, the circuit, and the input data vector, there is no corresponding simple architecture-level modeling for soft errors originating from these internal circuit nodes.

To study characteristics of these errors, we perform gate-level fault injection. Figure 4 shows the distributions of how many bits in the circuit output buffer are flipped if a fault from a circuit internal node is not logically-masked for applications studied in this paper. On average, 77% of errors manifest as single-bit flips. Hence, the single-bit flip model does not accurately reflect 23% of errors, and these errors potentially cause more severe data corruption. Note that the distributions

vary across applications since logical masking depends on circuit input. Such complex and data-dependent error patterns are not modeled by existing random bit-flipping models.

In this paper, we adopt the following 3 fault models in our experiments:

- **Single-bit flip at LLVM IR level (RB1-LLVM)**: randomly flips a single bit in the destination register of a random dynamic LLVM IR instruction. This fault model is inline with the ones used in most of prior studies to guide and evaluate IR-based selective instruction duplication [15], [4], [12], [8].
- **Single-bit flip at binary level (RB1-BIN)**: randomly flips a single bit in the destination operand of an random dynamic assembly instruction. We include this fault model because it is commonly used in prior work in HPC communities [4], [12], [20], [5].
- **RTL-level model (RTL)**: adopts the method in [19] to obtain a fault that is injected into RTL representation and corrupts the architectural state. This is shown to be the most accurate software-based fault injection methodology for simulating soft errors; hence, it is our ground truth in this work [19].

## III. Experimental Setup

### A. Implementation of Selective Instruction Duplication

In this paper, we select which instructions to duplicate for protection based on the fault injection results using *RB1-LLVM* fault model (Section II-D). The selection process is mapped to the 0-1 knapsack problem, same as the methodology used in [5], [17].

*1) The 0-1 Knapsack Problem:* Given a knapsack of limited capacity and a set of items with an associated value and volume, the goal is to select a subset of items that fit into the knapsack such that the total value is maximized.

*2) Mapping Instruction Selection to 0-1 Knapsack:* Before presenting the mapping, we define the following terms for each static instruction $I$:

- *SDC contribution*: the fraction of SDCs that can be detected by protecting instruction $I$.
- *Dynamic ratio*: the number of dynamic instances of $I$ divided by the number of all instruction instances in the program.

With these definitions, we can treat each static instruction as an item in the 0-1 knapsack problem. For each instruction, its value is the SDC contribution and its volume is the dynamic ratio, both of which are the characteristics of the application. The knapsack's capacity is a user-specified parameter that bounds the cost of selective instruction duplication. We use protection levels to quantify the knapsack's capacity (i.e., the amount of dynamic instruction duplicated).

Once we know which instructions to protect by solving the 0-1 knapsack problem, we duplicate those instructions and place a checker right before next branch, store, function call or function return using a LLVM pass. At runtime, if a checker detects an error, it prints an error message and terminates the application (Section II-B). Ideally, a protection level of 100% (i.e., full duplication) should detect all SDCs. Note that, in this paper, we guide and implement the selective instruction duplication using LLVM IR. So an instruction to be duplicated is an LLVM IR instruction of the program. For the benchmarks studied in this paper, the LLVM duplication pass takes 38 seconds at the most.

### B. Tools and Platforms

We use LLVM Fault Injector (LLFI) [12] for LLVM-level fault injection to guide the selection of instructions and estimate the SDC coverage of the protection under *RB1-LLVM* fault model. We adopt Hamartia [19] for binary-level injection (*RB1-BIN* fault model) and RTL-level injection (*RTL* fault model). For each protection level and each fault model, we perform at least 2,000 injection trial for each benchmark, one fault injected in each trial. This ensures a margin of sampling error about 2% for a confidence level of 95% [26]. Since we focus on evaluating selective instruction duplication for application code, we do not inject instructions within libraries (e.g., `libc` and `libm`). Note that checking instructions are subject to injection as well.

For the *RTL* fault model, we use the same set of circuits as in [19], which we briefly explain as follows. We synthesize gate-level netlists of integer and floating-point execution units using Synopsys tools (Design Compiler and DesignWare Library) with the 45nm Nangate Open Cell Library, optimized for performance. Because the DesignWare Library does not include pipelined floating-point units, we use the register retiming feature of Design Compiler to pipeline the circuits. The pipeline stages are tuned to mimic those used by Intel Broadwell processors based on the latency data from [27]. Note that the synthesized circuits can be different from those designed and optimized for commodity processors.

### C. Benchmarks

We evaluate the serial version of seven benchmark programs from common benchmark suites (including Parboil [28], Rodinia [29], Parsec [30], and SPEC [31]) and a molecular dynamics application, `puremd`, from Purdue University [32]. All programs are compiled with LLVM with -O2 (standard optimization). Note that the selection of benchmarks are based on the time taken to complete all fault injection experiments (as we have to balance our experiment time) and the capability of our fault injection infrastructures. Table I summarizes the benchmarks and their input.

### D. Metrics

*1) Outcome Classification:* Similar to prior work [5], [17], [8], fault injection outcomes are classified into these primary categories:

- **Masked**: the injected error is masked by application, with output identical to the error-free run.
- **Detected Uncorrectable Error (DUE)**: errors that crash the program or those detected are categorized as $DUE_{crsh}$. Errors that result in obviously erroneous

TABLE I: Benchmarks and their input.

| | Suite/Author | Input |
|---|---|---|
| bfs | Parboil | graph4096.txt |
| blackscholes | Parsec | 1 in_4.txt |
| cutcp | Parboil | watbox.sl40.pqr |
| hotspot | Rodinia | 64 64 1 1 temp_64 power_64 |
| libquantum | SPEC | 33 5 |
| nw | Rodinia | 2048 10 1 |
| puremd | Purdue University | geo ffield control |
| sad | Parboil | reference.bin frame.bin |

application output (e.g., output is not finite or mismatch in matrix size) are also in this category and denoted as $DUE_{test}$.

- **Silent Data Corruption (SDC)**: the program ends normally with a different program output compared with the error-free run.

*2) Protection Levels:* In this paper, we evaluate 7 different protection levels (specifically, 0%, 3%, 20%, 40%, 60%, 80%, and 100% of dynamic instructions for duplication) to form protection curves for each benchmark and each fault model.

### E. Research Questions

Three research questions are proposed in this paper and answered by analyzing fault injection results and protection curves.

**RQ1:** How effective is LLVM-based selective instruction duplication at reducing SDCs under a realistic fault model?
**RQ2:** Does full IR instruction duplication detect all SDCs under the realistic fault model?
**RQ3:** What is the impact of different fault models on evaluating the efficiency in selective instruction duplication?

## IV. Experimental Results

Figure 5 shows the protection curves measured by fault injections under different fault models. The selection of IR instructions to duplicate is based on the evaluation using *RB1-LLVM* fault model.

### A. Answer to RQ1

First, let's focus on the ground truth, the fault injection results under *RTL* fault model (red triangles). We observe that most SDCs can be detected by duplicating a fraction of IR instructions in a program. In most cases, protecting 20% of instructions is able to cover more than 50% of SDCs — this is a typical trade-off developers want to leverage to provide a good-enough reliability with low overhead. Note that the knee of the protection curves (i.e., the protection level after which the protection curve plateaus out) varies across applications, indicating that the protect curves are intrinsically application-specific. Prior work [5], [22], [8] has observed similar results.

Next, we compare the evaluation using *RTL* fault models (red triangles in Figure 5) and *RB1-LLVM* (dashed curves). In each benchmark, we observe that IR-based evaluation usually leads to pessimistic results at low protection levels but optimistic results at high protection levels. However, both curves rather follow similar trends across benchmark. This indicates that IR-based selective instruction duplication can be used to approach reasonable trade-offs between SDC coverage and overhead in the protection.

The exception is `nw` where the LLVM curve is consistently higher than that of *RTL*. Since the SDC ratio is already low (1.58%) even without any protection, the difference may not be statistically meaningful — we will further investigate this in our future work.

TABLE II: Percentages of faults injected that escape full IR instruction duplication.

| | RTL | RB1-BIN | RB1-LLVM |
|---|---|---|---|
| bfs | 0.21% | 0.13% | 0.00% |
| blackscholes | 0.10% | 0.10% | 0.00% |
| cutcp | 0.38% | 0.71% | 0.00% |
| hotspot | 0.33% | 0.62% | 0.00% |
| libquantum | 0.05% | 0.05% | 0.00% |
| nw | 0.92% | 0.46% | 0.00% |
| puremd | 0.08% | 0.08% | 0.00% |
| sad | 1.52% | 0.84% | 0.00% |
| gmean | 0.25% | 0.23% | 0.00% |

**A1:** IR-based selective instruction duplication is able to provide cost-effective protection to mitigate SDCs under realistic hardware faults.

### B. Answer to RQ2

Table II shows that under *RB1-BIN* and *RTL* fault models, there are a small number of faults that escape full duplication of IR instructions and eventually lead to SDCs. To account for potential outliers, we compute the geometric means. On average, the SDC ratios of *RTL* and *RB1-BIN* are 0.25% and 0.23%, respectively. Since the duplication is at IR level, fault injection using *RB1-LLVM* provides full coverage. The reason why we see faults injected at lower layers escape IR instruction duplication is because there are additional instructions inserted during code generation process and they are not visible to IR (e.g., instructions that set up or tear down stack frames). Hence, duplicating IR instructions may skip duplicating some of those instructions.

**A2:** Only a small amount (0.25%) of faults injected at lower layers escape full IR instruction duplication and lead to SDCs. Vast majority of realistic hardware faults injected can be captured by the protection.

### C. Answer to RQ3

*1) Protection Curves:* We compare the protection curves in Figure 5 measured using *RTL* (red triangles) and *RB1-BIN* (blue solid curves). We investigate this comparison because prior work [19] has shown that *RB1-BIN* yields similar measurement to that of *RTL* and can be potentially used as a proxy to the ground truth since *RB1-BIN* allows much

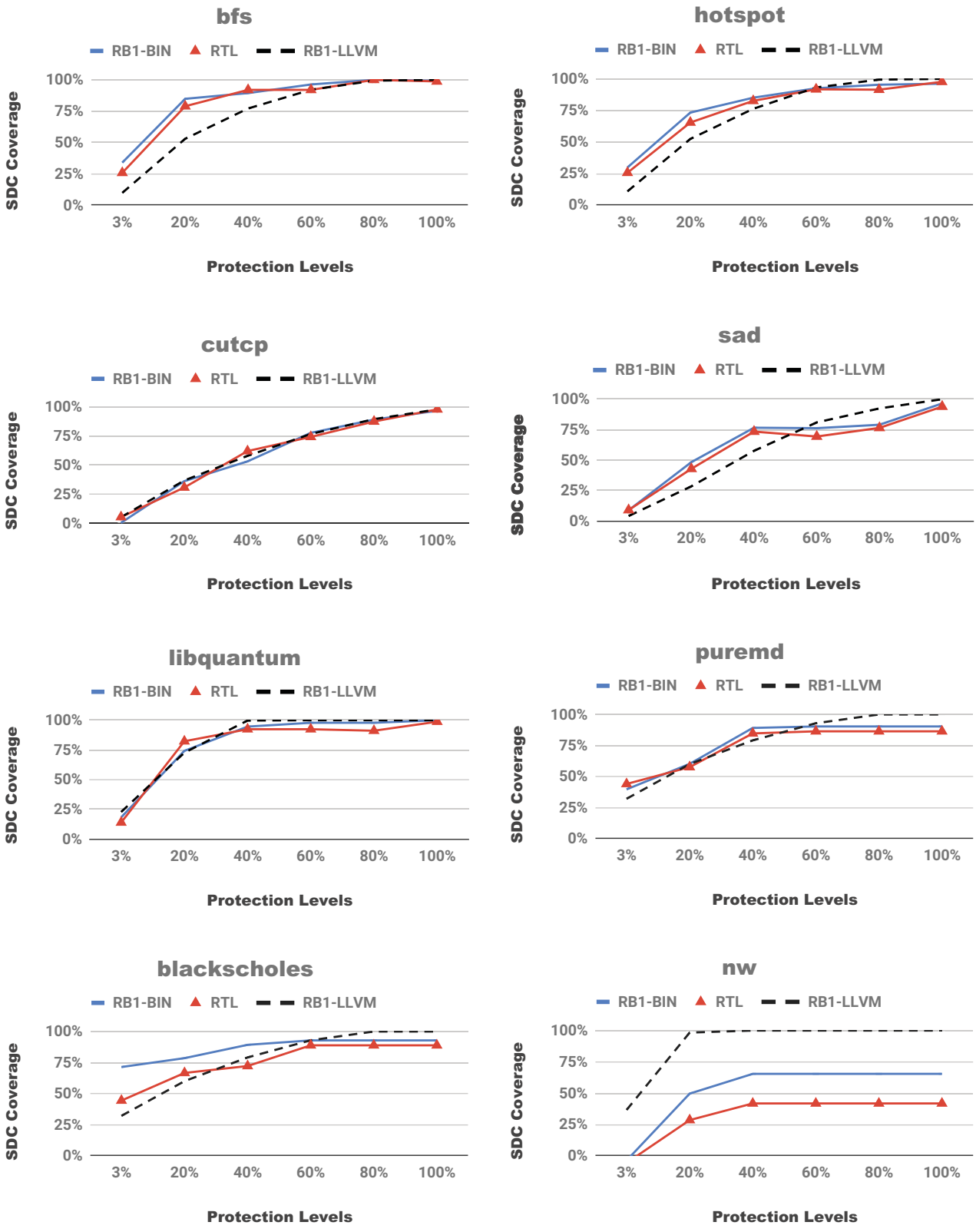


Fig. 5: Protection curves. X-axis is the protection level and y-axis is the SDC coverage.

faster fault injection campaigns. However, the study focused on programs without any protections. Hence, it is not clear if the conclusion is still true when programs are protected by selective instruction duplication.

As seen in Figure 5, in most benchmarks, the protection curves of *RB1-BIN* and *RTL* are in very similar shapes. The exceptions are `nw` and `blackscholes` whose SDC ratios are low (around 1%) even without any protection. Notice that for `blackscholes` when protection level is at 3%, there is a 27% difference between the two curves. This is because the SDC ratios without protection are 1.4% and 0.9% for *RB1-BIN* and *RTL* respectively. Although they are close in terms of absolute values, their difference is large relatively. Again, this may be due to the statistical natures of the fault injection measurement. We will further investigate this in our future work.

**A3.1:** Protection curves measured by *RTL* and *RB1-BIN* are mostly similar.

*2) Outcome Distributions:* Figure 6 shows the fault injection outcome distributions at different protection levels. We have three major findings. First, as the protection level increases, DUE ratios increase while SDC ratios and masked ratios decrease. Since SDCs are converted to DUEs if they are detected, increase in DUE ratios and decrease in SDC ratios are expected. The reason why masked ratios are also decreased is because some errors originally able to be masked by the unprotected application binary are now detected earlier when instruction duplication are added.

Secondly, *RB1-BIN* leads to outcome distributions similar to that of *RTL* in selective instruction duplication at all protection levels. The only exception is `cutcp` at low protection levels (0% and 3%) where the difference of SDC ratios is larger (about 4%) . However, as the protection level increases, the distributions measured by the two fault models converge.

We also observed that the SDC coverage mostly stays constant (i.e., a small amount of SDCs remain undetected) when the protection level is higher than 60% in most of the benchmarks. This means that, even when we duplicate more than 60% of total dynamic instructions, some SDCs are still not eliminated. We leave the detailed analysis as our future work.

**A3.2:** *RB1-BIN* leads to outcome distributions similar to that of *RTL* in selective instruction duplication at all protection levels.

Based on **A3.1** and **A3.2**, we conclude that *RB1-BIN* can be used as a fast proxy to realistic hardware faults in the evaluation of selective instruction duplication.

## V. Discussion

Based on the results, our takeaway is that IR-based selective instruction duplication is a cost-effective way to protect programs when developers want to reach certain SDC coverage given a limited overhead budget. This is a typical application scenario in commodity systems where a good-enough reliability with low overhead is highly desirable. In contrast, IR-based selective instruction duplication should not be used in mission-critical tasks where SDCs need to be completely eliminated.

In the future, we have four directions: (1) We plan to extend our evaluation on a larger set of benchmarks with more protection levels and different hardware designs; (2) We also plan to apply different statistical methods to quantitatively examine the efficiency of selective instruction duplication; (3) We want to investigate the reason why low-level hardware faults may escape IR duplication and come up with solutions; (4) We plan to refine the metric used to quantify the performance overhead of selective instruction duplication, which we explain below.

In production uses, developers may be concerned with the overhead in terms of execution time instead of the percentage of duplicated instructions. When selecting which instructions to protect, it would be better if we also take instruction types into account as different types of instructions can have different execution latency. As a result, we measure performance overhead at multiple protection levels in terms of execution time normalized to the original binary without duplication. The experiments are conducted on a machine that runs OpenSUSE 42.3 on an Intel i5-6500 CPU with 16GB DRAM. For the applications studied in this paper, the geometric means are 4%, 35%, 51%, 54%, 61%, and 61% for protection levels of 3%, 20%, 40%, 60%, 80%, 100%, respectively. Apparently, execution time increases in an non-linear manner with the protection level. Note that the numbers reported depend on the benchmark, the machine, and compiler optimizations. We leave the development of a better performance overhead metric as future work.

## VI. Related Work

Instruction duplication is widely adopted by prior work [21], [7], [6], [17]. They can be grouped into two classes: full duplication and selective duplication.

*Full duplication*: EDDI [21] applies full duplication at the binary level and evaluates it by flipping a single bit in the code segment. SWIFT [7] applies full duplication at compiler backend and evaluates it with single-bit flips at the binary level, same as our *RB1-BIN* fault model in this paper.

*Selective duplication*: Shoestring [6] proposes heuristics to selectively duplicate instructions at LLVM's backend (right after instruction selection and before register allocation), while we solve a 0-1 knapsack problem to determine which instructions to protect. The fault model in Shoestring is also different from ours. Although they perform fault injection at the micro-architectural level, they assume single-bit flips in the physical register file, which is usually protected by ECC in commodity processors. SDCTune [17] injects errors to unprotected applications at the LLVM IR level and uses the evaluation results to guide which instructions to protect. We follow the same methodology for implementing detectors in this paper but perform evaluation at a lower level.

Recently, several studies attempt to validate single-bit flips for evaluating the resilience of unprotected applications. Sang-

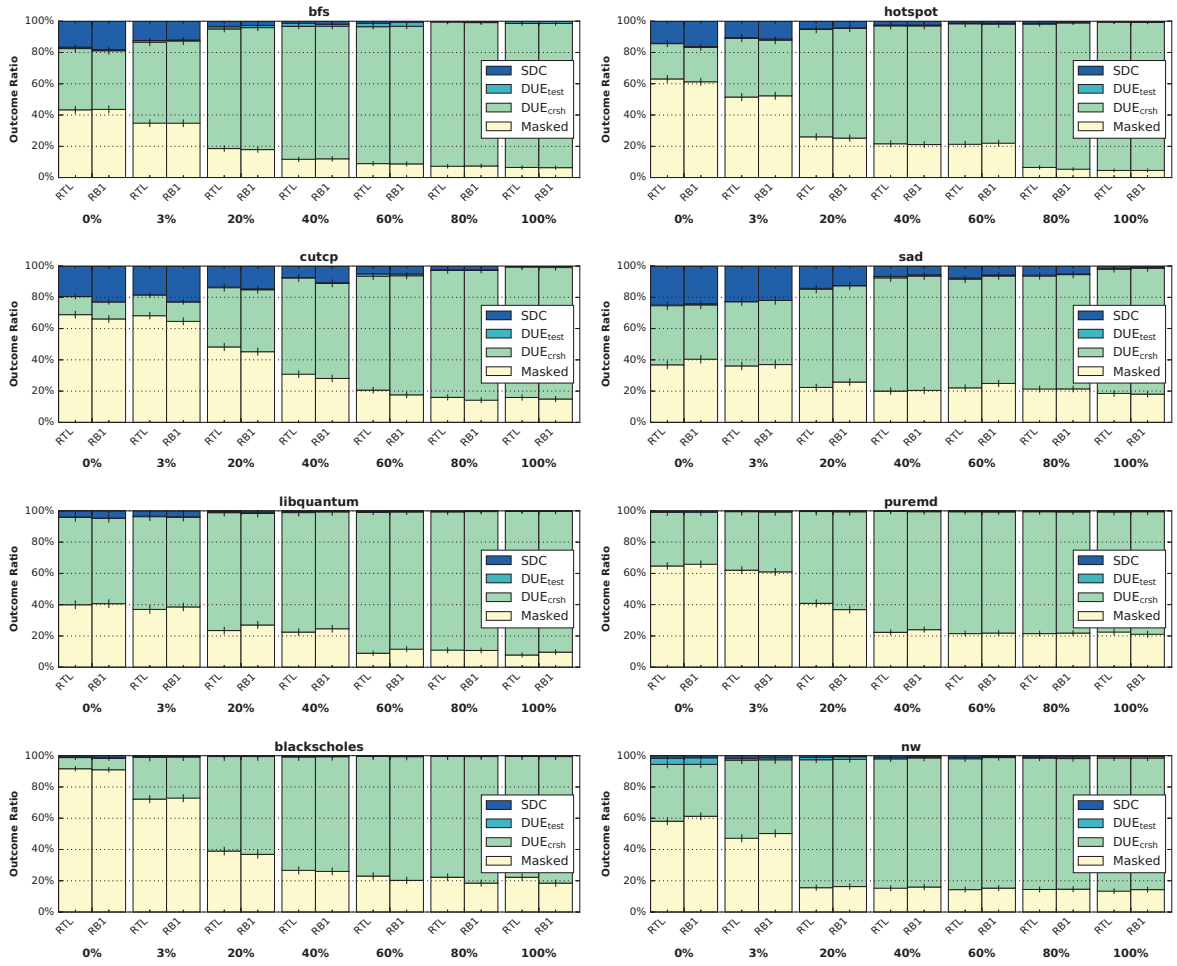


Fig. 6: Injection outcome distributions. Outer x-axis is the protection level at which X% of LLVM IR dynamic instructions are protected by selective instruction duplication. Inner x-axis is the fault models: *RTL* and *RB1-BIN*.

choolie et al. [33] find that single bit-flips result in SDC ratios reasonably close to *data-independent* multi-bit flips in many cases. Chang et al. [19] further observe that this is also true for a more realistic *data-dependent* RTL fault model. This paper further validates the cases where applications are protected by IR-based selective instruction duplication.

## VII. Conclusion

The efficacy of IR-based selective instruction duplication is validated via injecting realistic hardware faults. The results show that LLVM-level analysis and measurement can guide efficient protection to mitigate SDCs at low performance overhead. However, there is a small amount of faults originated from lower hardware layers may escape the protection, even at full IR instruction duplication. This indicates that IR-based selective instruction duplication should be only used in commodity systems where a good-enough reliability with low overhead is desirable, rather than in reliability-critical tasks where SDCs need to be completely eliminated.

## References

[1] M. Snir, R. W. Wisniewski, J. A. Abraham, S. V. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, *et al.*, "Addres[illegible] failures in exascale computing," *The International Journal of High Performance Computing Applications*, vol. 28, no. 2, pp. 129–173, 2014.

[2] C. Constantinescu, "Intermittent faults and effects on reliability of integrated circuits," in *Reliability and Maintainability Symposium*, p. 370, IEEE, 2008.

[3] V. J. Reddi, M. S. Gupta, M. D. Smith, G.-y. Wei, D. Brooks, and S. Campanoni, "Software-assisted hardware reliability: abstracting circuit-level challenges to the software stack," in *2009 46th ACM/IEEE Design Automation Conference*, pp. 788–793, IEEE, 2009.

[4] R. A. Ashraf, R. Gioiosa, G. Kestor, R. F. DeMara, C.-Y. Cher, and P. Bose, "Understanding the propagation of transient errors in hpc applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, p. 72, ACM, 2015.

[5] S. K. S. Hari, S. V. Adve, and H. Naeimi, "Low-cost program-level detectors for reducing silent data corruptions," in *Dependable Systems and Networks (DSN), 2012 42nd Annual IEEE/IFIP International Conference on*, pp. 1–12, IEEE, 2012.

[6] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, "Shoestring: probabilistic soft error reliability on the cheap," in *ACM SIGARCH Computer Architecture News*, vol. 38, pp. 385–396, ACM, 2010.

[7] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "Swift: Software implemented fault tolerance," in *Proceedings of the international symposium on Code generation and optimization*, pp. 243–254, IEEE Computer Society, 2005.

[8] G. Li, K. Pattabiraman, S. K. S. Hari, M. Sullivan, and T. Tsai, "Modeling soft-error propagation in programs," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 27–38, IEEE, 2018.

[9] G. Li, K. Pattabiraman, C.-Y. Cher, and P. Bose, "Understanding error propagation in gpgpu applications," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 240–251, IEEE, 2016.

[10] A. Mahmoud, S. K. S. Hari, M. B. Sullivan, T. Tsai, and S. W. Keckler, "Optimizing software-directed instruction replication for gpu error detection," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 842–853, IEEE, 2018.

[11] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, p. 75, IEEE Computer Society, 2004.

[12] J. Wei, A. Thomas, G. Li, and K. Pattabiraman, "Quantifying the accuracy of high-level fault injection techniques for hardware faults," in *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 375–382, IEEE, 2014.

[13] A. Thomas and K. Pattabiraman, "Error detector placement for soft computation," in *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 1–12, IEEE, 2013.

[14] G. Li and K. Pattabiraman, "Modeling input-dependent error propagation in programs," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 279–290, IEEE, 2018.

[15] G. Georgakoudis, I. Laguna, D. Nikolopoulos, and M. Schulz, "Refine: Realistic fault injection via compiler-based instrumentation for accuracy, portability and speed," in *Proceedings of SC17, Denver, CO, USA, November, 2017*, IEEE.

[16] 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 (per)," 2019.

[17] Q. Lu, G. Li, K. Pattabiraman, M. S. Gupta, and J. A. Rivers, "Configurable detection of sdc-causing errors in programs," in *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 16, p. 88, ACM, 2017.

[18] D. S. Khudia and S. Mahlke, "Harnessing soft computations for low-budget fault tolerance," in *Microarchitecture (MICRO), 2014 47th Annual IEEE/ACM International Symposium on*, pp. 319–330, IEEE, 2014.

[19] C.-K. Chang, S. Lym, N. Kelly, M. B. Sullivan, and M. Erez, "Evaluating and accelerating high-fidelity error injection for hpc," in *In the Proceedings of the ACM/IEEE International Conference on High-Performance Computing, Networking, Storage, and Analysis (SC18)*, (Dallas, TX), November 2018.

[20] V. C. Sharma, A. Haran, Z. Rakamaric, and G. Gopalakrishnan, "Towards formal approaches to system resilience," in *Dependable Computing (PRDC), 2013 IEEE 19th Pacific Rim International Symposium on*, pp. 41–50, IEEE, 2013.

[21] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Error detection by duplicated instructions in super-scalar processors," *IEEE Transactions on Reliability*, vol. 51, no. 1, pp. 63–75, 2002.

[22] 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," in *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, p. 42, IEEE Press, 2016.

[23] Q. Guan, N. BeBardeleben, P. Wu, S. Eidenbenz, S. Blanchard, L. Monroe, E. Baseman, and L. Tan, "Design, use and evaluation of p-fsefi: A parallel soft error fault injection framework for emulating soft errors in parallel applications," in *Proceedings of the 9th EAI International Conference on Simulation Tools and Techniques*, pp. 9–17, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2016.

[24] D. Oliveira, V. Frattin, P. Navaux, I. Koren, and P. Rech, "Carol-fi: an efficient fault-injection tool for vulnerability evaluation of modern hpc parallel accelerators," in *Proceedings of the Computing Frontiers Conference*, pp. 295–298, ACM, 2017.

[25] D. Li, J. S. Vetter, and W. Yu, "Classifying soft error vulnerabilities in extreme-scale scientific applications using a binary instrumentation tool," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, p. 57, IEEE Computer Society Press, 2012.

[26] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence," in *Proceedings of the Conference on Design, Automation and Test in Europe*, pp. 502–506, European Design and Automation Association, 2009.

[27] A. Fog, "Optimization manual 4 instruction tables," *Copenhagen University College of Engineering, Software Optimization Resources, http://www.agner.org/optimize,(1996-2017)*, pp. 215–230.

[28] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. W. Hwu, "Parboil: A revised benchmark suite for scientific and commercial throughput computing," *Center for Reliable and High-Performance Computing*, vol. 127, 2012.

[29] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE international symposium on workload characterization (IISWC)*, pp. 44–54, Ieee, 2009.

[30] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pp. 72–81, ACM, 2008.

[31] J. L. Henning, "Spec cpu2000: Measuring cpu performance in the new millennium," *Computer*, vol. 33, no. 7, pp. 28–35, 2000.

[32] H. M. Aktulga, J. C. Fogarty, S. A. Pandit, and A. Y. Grama, "Parallel reactive molecular dynamics: Numerical methods and algorithmic techniques," *Parallel Computing*, vol. 38, no. 4-5, pp. 245–259, 2012.

[33] 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," in *Dependable Systems and Networks (DSN), 2017 47th Annual IEEE/IFIP International Conference on*, pp. 97–108, IEEE, 2017.