Hindawi
Scientific Programming
Volume 2022, Article ID 9996593, 19 pages
https://doi.org/10.1155/2022/9996593

Research Article

Characterization of Program Behavior under Faulty Instruction Encoding

Junchi Ma, Zongtao Duan, and Lei Tang

School of Information Engineering, Chang'an University, Xi'an 710061, China

Correspondence should be addressed to Junchi Ma; bjbzmjc@126.com

Received 23 May 2021; Accepted 23 December 2021; Published 7 February 2022

Academic Editor: Daniela Briola

As process technology scales, electronic devices become more susceptible to soft errors. Soft errors can lead to silent data corruptions (SDCs), seriously compromising the reliability of a system. Researchers have explored error resilient encodings, which leverage crash patterns to detect SDCs. Despite its importance, much still remains to be determined regarding how errors propagate to cause SDCs or crashes. Understanding error propagation patterns could lead to more efficient implementation of error detection. An experimental study of program behavior in the presence of faulty instruction encoding under the IA-32 architecture is described in this study. Extensive fault injection experiments including over 70,000 faults were conducted, targeting all fields of instruction encoding. The analysis of the obtained data shows the following: (1) If the alignment of an instruction sequence is not preserved after injection, it causes crashes in a high probability (93.2%). (2) The SDC rate of an alignment-preserved category is close to that of a typical data injection. The SDC-prone fields include the opcode field, reg field, and immediate field. (3) Several crash patterns, such as violation of calling conventions, are revealed to extend the detection methods. These findings help us identify the vulnerable parts of instruction encoding, which need to be protected against soft errors. By applying the implications provided by the findings, we discuss feasible modifications, including swapping reg encodings, to reduce SDC rate, thus increasing the resilience of instruction set to soft errors.

1. Introduction

Soft error has emerged as a severe challenge in electronic system design [1]. Progressive technology scaling and lowering of operating voltages have made contemporary and future electronic systems more susceptible to soft errors [2]. Soft errors can produce a bit flip in the instruction encoding flipping a single bit from 0 to 1 or vice versa. Possible outcome types derived from this bit flip are benign, crash, hang, or silent data corruption (SDC) [3]. When SDC occurs, the program generates an erroneous output. Compared with other outcome types, SDC is more insidious since it occurs without any indications [4]. Applying the erroneous output incurred by SDC may lead to loss of properties and even casualties.

Instruction encodings are stored in memory, instruction cache, instruction queue, instruction buffer, etc. Li reveals that instruction buffer is one of the three largest contributors to the architectural FIT rate on average [5]. Researchers tackle the issue of preventing SDC using schemes such as encoder/decoder [6] and bloom filter [7] by converting many SDCs into crashes. The encoder/decoder scheme propagates the error to the most sensitive bit, which is chosen based on the probability of causing invalid encoding exception. Intrinsically, instruction set architecture (ISA) can detect the error if a changed encoding produces a crash.

Another approach changes the encodings of certain frequently used instructions to increase the likelihood of producing crashes [8]. While these schemes leverage invalid instruction exception of ISA, they are not able to detect SDC efficiently. We identify the following challenges.

(1) The major crash pattern used to design a detection scheme is invalid instruction exception. It is easy to predict an invalid instruction exception when a bit of encoding is flipped. However, invalid encodings often account for a small portion of all encodings, which restricts the range of encodings that it can protect.

(2) The propagation of faults in instruction encoding to cause SDC has not been investigated. The encoding is considered SDC-prone if the changed encoding is not an illegal instruction. Since many non-SDC-prone encodings are considered vulnerable, they are protected by the detection scheme, which is unnecessary and leads to performance loss.

These issues are vital for increasing the intrinsic error detection rate of ISA. There has been little work using fault injection to study the effects of faulty encodings. To address these issues, we inject faults into the encodings and extract features from the results of fault injection.

In this study, we target instruction encoding of IA-32 instruction set architecture [9]. Over 70,000 faults are injected into all fields of instruction encodings (instruction encodings consist of optional instruction prefixes, opcode field, an addressing form specifier consisting of the MODR/M byte and the SIB, a displacement, and an immediate field). The injection traces, including massive execution information, are also recorded. The analysis provides detailed insight into the behavior of the program. The SDC or crash rates of all fields of instruction encoding are evaluated, and the causes for high SDC or crash rates are analyzed by investigating the injection traces. Moreover, typical fault propagation patterns are revealed, which provides guidelines for developing resilient encoding. We also inject faults in the data of instruction (register file or memory) to create a baseline for instruction encoding. The injection results of instruction encodings are compared with results from data fault injection, indicating that certain fields can be more vulnerable than data. The major findings include the following:

(i) A change in the alignment of an instruction sequence significantly increases the crash rate. A realigned category experiences a substantially higher crash rate (93.2%) compared with an alignment-preserved category (59.1%).

(ii) The opcode field in a certain instruction (such as Jcc and MOV) and the reg and immediate field obtain higher SDC rates compared with the SDC rate of injections on data (14.4%). Flip in n-bit of the opcode field in Jcc instruction causes the wrong branch to be taken, and it obtains the highest SDC rate (46.7%).

(iii) We reveal crash patterns including realignment of instruction sequence, addressing failure, and calling convention violation, which extends the patterns that can be used to derive effective error detection mechanisms.

We discuss some feasible schemes that can be used to modify encodings of SDC-prone fields, such as the opcode and reg fields. The proposed schemes set crash-prone encodings as neighbors of certain SDC-prone encodings in the Hamming space, which prevents the error propagation that may cause SDC. We evaluate the proposed scheme by fault injections, and the results show that SDC rate can be reduced significantly.

The rest of the study is organized as follows. Related research is discussed in the next section. The experimental methodology is described in Section 3. An overview of the injection results is presented in Section 4. The injection results are classified as realigned, preserved, and invalid. We discuss results in each category separately in Sections 5~7. Some implications for designing resilient instruction encoding are presented in Section 8. The study is concluded in Section 9.

2. Related Work

A series of prior studies were concentrated on evaluating the effect of faulty instruction encoding and error detection mechanisms. We classify related work into three categories, namely (1) error detection with encoder/decoder schemes, (2) optimization of instruction encoding resilience, and (3) vulnerability assessment of instruction encoding.

2.1. Error Detection with Encoder/Decoder Scheme. Martinez presented an encoder/decoder scheme to detect errors in instruction encoding [6]. When a soft error affected an encoded instruction and produced a bit flip, the error was propagated by the decoding, causing a second bit flip on the sensitive bit. Many SDC cases were converted into crashes because an error in the sensitive bit was likely to produce a crash. The choice of sensitive bit affected the SDC rate of the encoder/decoder scheme, and thus, a methodology for profiling the vulnerability of specific ISA was proposed in their later study [10]. A fault was considered to cause SDC if it did not incur invalid instruction exception and memory access exception. There was still a significant difference (up to 27.0%) between the estimated SDC rate and the real SDC rate obtained in the fault injection. A portion of predicted SDC cases might be actually crash cases. Few crash patterns were considered, and faults that failed to match the patterns were identified to cause SDC. Crash propagation is addressed in this study, revealing more crash patterns that are needed to make a more accurate estimation.

In an encoder/decoder scheme, error detection causes a system crash. To avoid the performance loss caused by a system crash, Atamaner used a bloom filter with the encoder/decoder scheme [7]. While the instruction was being decoded, the bloom filter was queried with the fetched instruction to check whether it was in the initial original instruction set. If it was not in the set, an error occurred and the system stopped execution to avoid system crash. To reduce the number of elements in the filter, a dynamic checking of the instructions based on vector that identified the opcodes was proposed [11].

2.2. Optimization of the Instruction Encoding Resilience. Opcode swapping based on the usage frequency of instructions (OSUFI) was proposed to reduce the vulnerability of the ISA and instruction cache. OSUFI swapped the opcode bits between a frequently used instruction with more vulnerable bits and a less-frequently used instruction with less vulnerable bits [8]. The bit was considered a vulnerable bit if the corrupted opcode became another legal opcode for another instruction. The OSUFI scheme can only be used with an ISA that has a non-negligible proportion of invalid encodings because the definition of vulnerability is based on the invalid encoding. When the proportion of invalid encodings of ISA is low, all instruction bits tend to be equally vulnerable. In this study, we obtained several findings, which help reveal the sources of vulnerable bits.

The IA-32 instruction set currently uses continuous encoding of all the conditional branch instructions, in which a single-bit error can subvert the flow of control. To address this issue, an instruction set encoding scheme that increased the Hamming distance between the blocks of conditional branch instructions was proposed [12]. Any parity encoding in the new encoding scheme has a minimum Hamming distance of 2, which means at least two different bits between opposite condition encodings, preventing system from subverting the programmer's intended flow of control.

2.3. Vulnerability Assessment of Instruction Encoding. An architectural vulnerability factor for instruction queue and execution unit was calculated in a previous study [13]. All bits in the opcode field were categorized as architecturally correct execution (ACE) bits, even for the NOP instruction. The vulnerability of these bits was considered in a conservative manner that these bits may affect execution of a program. However, whether these bits incur SDC, crash, or hang was not discussed further. Hamming-distance-one analysis [8] was found to reduce the number of ACE bits in the frequently used encodings. If a corrupted bit made the opcode illegal, the bit was considered an ACE bit because the error could be detected during instruction decoding.

Several earlier studies on the duplicated instruction technique discussed the capability of detecting faulty instruction encodings [14]. Error detection by duplicated instructions (EDDI) deployed duplication of instruction for detecting errors during usual system operation. The cases that cannot be covered by EDDI in the presence of faulty instruction encodings were given by theoretical analysis. The study only intended to prove that technique of instruction duplication, which was used to protect data, was not able to protect instruction encodings. However, it did not give the vulnerable encodings that need to be protected. This issue is addressed in this study by analyzing the fault injections on encodings, and the SDC-prone fields that need to be protected are obtained.

Another series of studies concentrated on triggering bit flips from software to craft powerful attacks and completely subvert a system. The Rowhammer hardware bug allows an attacker to modify memory simply by repeatedly accessing a given physical memory location until a bit in an adjacent location flips [15]. Notable examples include escaping Google's Native Client (NaCl) sandboxing feature and gaining kernel privileges from an unprivileged process [16]. Olesen showed that a single-bit flip in jump instruction encoding is enough to allow any user to access a system on most UNIX platforms [17]. Our approach can be used to find the vulnerable portion of the code segment for memory protection or attack surface reduction.

3. Experimental Methods

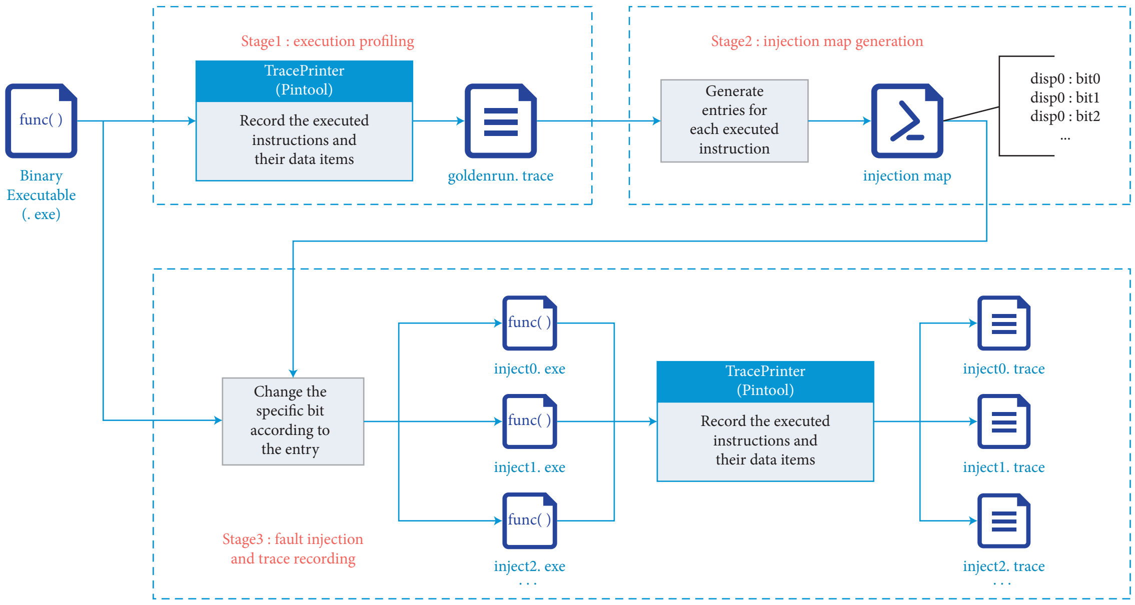
The fault model, experimental infrastructure, and application workload are described in this section. The fault model we assume is a single-bit flip within the instruction encoding. Fault can occur in any places, which store instruction encoding, such as memory without ECC protection, instruction queue, instruction cache, instruction buffer, or pipeline. We do not consider the ECC protection. ECC only protects SRAM structures on-chip and does not protect pipeline logic or other processor structures [18]. Certain components may not use ECC to acquire a high performance. Moreover, it is more scalable to increase the resilience of ISA than depending on reliable hardware since all systems applying this ISA can be resilient to faulty instruction encodings. Applying this fault model can reduce limits on hardware and serve the purpose of designing resilient instruction encoding.

In this study, we attempt to gain insight into program behavior when instruction encoding is corrupted. The fault was injected into the binary code in the system's memory. We injected faults exhaustively so that faults on every instruction and every bit could be tested. All bits in the executed instructions of the studied program were injected into the experiment, providing a complete view of fault impacts. For example, the instruction TEST EAX, EAX (encoding 0x85C0) has two bytes (16 bits). Sixteen injections for this instruction were run, each running with 1 of the 16 bits corrupted. In total, over 70,000 faults were injected into the binary executables.

3.1. Workflow of Experiment. The experiment was divided into three stages, as shown in Figure 1:

(i) Execution profiling: We took an instruction-level profile of execution, recorded the executed instructions, and located the executed instructions in the binary executable. This step is to make effective injection because injection on an unexecuted instruction does not make a real bit flip, definitely causing benign.

(ii) Injection map generation: Each entry of the injection map has two fields: the displacement from the beginning of the executable file and the bit to be



Stage1 : execution profiling
func( )
Binary Executable (. exe)
TracePrinter (Pintool)
Record the executed instructions and their data items
goldenrun. trace
Stage2 : injection map generation
Generate entries for each executed instruction
injection map
disp0 : bit0
disp0 : bit1
disp0 : bit2
...
Change the specific bit according to the entry
func( )
inject0. exe
inject1. exe
inject2. exe
...
TracePrinter (Pintool)
Record the executed instructions and their data items
inject0. trace
inject1. trace
inject2. trace
...
Stage3 : fault injection and trace recording

Figure 1: Workflow of fault injection experiment.

injected. An entry was generated for each bit of the executed instructions.

(iii) Fault injection and trace recording: One entry from the injection map was used in each injection run. According to the displacement item, we can find the address of the instruction and change the specific bit from 0 to 1 or vice versa, generating a new binary executable. The binary executable was run, and the program output, error code, and data trace were recorded.

Data traces of instructions were extracted and used to analyze error propagation. We developed an instrumentation tool, called TracePrinter, based on Pin [19]. Pin is a dynamic binary instrumentation framework for the IA-32 and x86-64 instruction set architectures, and it enables the creation of dynamic program analysis tools. The tool was used to record executed instructions in stage 1 and record data items in stage 3. The example data trace from the trace file of *print_tokens* is shown in Figure 2. The data trace contains the sequence number of the dynamic instruction, instruction address, name of the operand, and the value of the operand before/after execution. In the example, ESP is written and its value is recorded before and after execution. Because it is a SUB instruction and EFLAGS is affected, the value of EFLAGS is also recorded. TracePrinter is also embedded in a run-time probe that probes the "/proc" system of LINUX to record the segment boundaries.

The error code was recorded after execution to determine the causes of the crash. A successful execution returns 0, while an unsuccessful execution returns a nonzero value. For example, error code 139 denotes segmentation error.

3.2. Infrastructure and Benchmark. The experiment was conducted on an IPASON P18 with an Intel i5 processor running Ubuntu 10.04. GCC 4.4.3 was used for compilation. The benchmarks studied here are from the Siemens suite [20] and MiBenchmark suite [21]. Siemens benchmark and MiBenchmark cover basic operations such as mathematic calculations, task scheduling, and word processing, which are common in an embedded environment. We choose the benchmark because it contains abundant data flow, control flow, and addressing operations, which are beneficial to the analysis. These benchmarks were widely accepted by the related research work [4, 22, 23]. The programs considered are *replace* (which performs string matching and replacement), *schedule*, *schedule2* (which are priority schedulers), *print_tokens* (which perform lexical analysis), and *qsort_small* (which performs quick sort). These are C programs consisting of a few hundred lines of C code. The characteristics of benchmarks are shown in Table 1. It takes 28 hours to finish the whole fault injection experiment. For a single run in fault injection, it takes a factor of about 18 larger run-time overhead than the native run.

```
###########no.1543   ip 0x8048a75
//sequence number=1543, instruction address=0x8048a75
assemble= sub esp, 0x4
in rtnin_pat_set// the procedure name
Reg ESP bfbaaeb8 before // the value of ESP before execution
Reg EFLAGS 286 before // the value of EFLAGS before execution
Reg ESP bfbaaeb4 after // the value of ESP after execution
Reg EFLAGS 286 after // the value of EFLAGS after execution
```

Figure 2: Example data trace recorded by TracePrinter.

3.3. Outcome of Injection. Four outcomes after the injection are listed as follows [24], which are mutually exclusive and exhaustive:

(i) Benign, meaning the program produces the correct output. If the error code equals 0 and the output after the injection equals the one in fault-free run, the outcome is benign.

Table 1: Characteristics of benchmarks.

Program	#Dynamic instructions	#Faults injected
replace	6957	19690
schedule	6808	16679
schedule2	6650	13832
print_tokens	2903	16343
qsort_small	13456	4190

(ii) Crash, which means the error causes the program to stop execution. If the error code is a nonzero value, the outcome is crash.

(iii) SDC, which means the program continues running but generates an erroneous output. If the error code equals 0, and the output is different from that in the fault-free run, the outcome is an SDC.

(iv) Hang, which means resources are exhausted but the program still cannot finish execution. Execution is halted when the execution time exceeds a threshold (1 min).

3.4. Baseline: Data Injection Experiment. We also performed data injection on the same benchmarks for comparison. The data injection results were used as a *baseline* for evaluating the encoding injection results. The fault model for data injection is a single-bit flip within the register file or memory. Our fault model of data injection is in line with other work in the area [3, 4, 24]. Similar to encoding injection, the injection map was generated by analyzing the trace of fault-free execution. Each bit in the destination operand within the executed instruction was injected. Therefore, the number of entries in the injection map depends on the length of the destination operand. During each run, the injection was performed by altering a selected bit in the destination operand. The alteration was achieved using an injection PinTool developed based on Pin [19]. We injected a single-bit fault into the value of destination operand after the instruction was executed. In total, over 650,000 faults were injected during the data injection experiment. The number of faults in the data injection campaign is at least a factor 8 larger than that in the encoding injection campaign. The instruction encoding bit was injected once during the encoding injection campaign. The number of individual bits injected in the data injection campaign depends on the number of execution instructions, so more injections are performed than encoding injection campaign.

4. Overview of Injection Results

A single-bit flip in an instruction's encoding can affect the alignment of an instruction sequence. We determine whether the alignment of an instruction sequence is preserved and whether it significantly impacts the outcomes of the injection results. We define the following alignment statuses and show examples in Table 2.

(i) *Realigned.* A flipped instruction has a different length compared with the original instruction, which changes the alignment of the instruction sequence. In the example shown in Table 2, the length of the original first instruction is 2 bytes. After fault injection, the length of first instruction changes to 5 bytes, and the two subsequent instructions' encodings are also affected.

(ii) *Preserved.* The length of the flipped instruction is equal to the original instruction, and thus, the alignment of the instruction sequence is preserved. In this example, the length of TEST EAX, EAX is 2 bytes, which equals the length of the flipped instruction TEST AL, AL.

(iii) *Invalid.* The corrupted instruction encoding is not defined by the ISA, and it incurs an illegal instruction exception. In the example, the encoding 8D C0 cannot be decoded.

The largest proportion (71.0%) of instruction sets are in the preserved category, followed by the realigned category (27.3%) and the invalid category (1.7%). The number of cases in terms of the instruction field is shown in Figure 3. The instruction encodings often consist of the opcode field, an addressing form specifier consisting of the MODR/M byte (MOD field, R/M field, reg field) and the SIB, a displacement field, and an immediate field. Realigned cases are incurred by faults in the opcode, MOD, R/M field, or SIB. MOD, R/M field, and SIB determine the addressing mode. A bit flip in these fields may change the length of the displacement and further change the length of the instruction. A fault in the reg, immediate, or displacement fields only leads to a preserved case. Invalid cases are incurred by injections on the opcode or MOD field.

Injection results of the alignment-based category are shown in Table 3 (the percentage of hang for each category is less than 2%. Due to limited space in this article, we omit the discussion of hang cases), including the average rate and standard deviation for each outcome rate. The crash/SDC/benign rates in the realigned and preserved categories are very different, which indicates that the preservation of original alignment of instruction sequence has an important effect on the outcome. The obvious difference is that the realigned category produces a significantly higher crash rate (93.2%) than faults in the preserved category (59.1%). Furthermore, a higher percentage of the realigned category tends to produce a crash in a shorter time compared with the preserved category. The crash rate within 10 dynamic instructions is shown in Figure 4. Latency is measured in terms of the number of dynamic instructions executed by the program from fault activation to crash [25]. 0 crash latency means the program crashes right after the injected instruction is executed. The crash rate for the realigned category is a factor 2 larger than the crash rate for the preserved category within 5 dynamic instructions. The accumulated crash percentage within 5 dynamic instructions for the realigned category is 88.1%, while the preserved category has a 58.0% accumulated crash percentage. Moreover, the standard deviation in crash rates for the preserved category (0.039) is a factor 4 larger than that of the realigned category

Table 2: Examples of realigned, preserved, and invalid categories.

Category	Before fault injection		After fault injection		Description
	Encoding	Assembly	Encoding	Assembly	
Realigned	85 C0	TEST EAX, EAX	05 C0 74 12 B8	ADD EAX, 0xB81274C0	The most significant bit of the first instruction is changed to 0. The fault changes the length (2 bytes) of first instruction into 5 bytes. The subsequent instructions are also affected. The immediate of MOV EAX, 0x0 is interpreted as two ADD instructions.
	74 12	JZ	00 00	ADD [EAX], AL	
	B8 00 00 00 00	MOV EAX, 0x0	00 00	ADD [EAX], AL	
Preserved	85 C0	TEST EAX, EAX	84 C0	TEST AL,AL	The fault changes the operand EAX in the original instruction into AL. The length of the first instruction remains 2 bytes, and thus, the consequent JZ instruction is unaffected.
	74 12	JZ	74 12	JZ	
Invalid	85 C0	TEST EAX, EAX	8D C0	(cannot be decoded)	The encoding 8D C0 cannot be decoded. The fault incurs an illegal instruction exception.

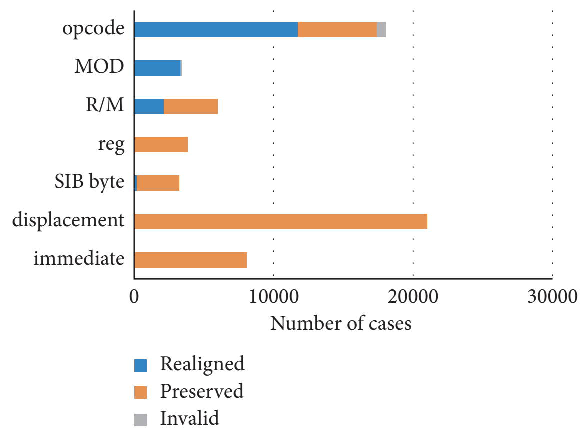


Figure 3: Number of cases grouped by instruction field.

(0.009). A lower standard deviation indicates that the crash rates fall in a narrower range; i.e., the crash rate has a weak dependence on the context of the program. Therefore, crash rate for the realigned category has a weaker dependence on the context of program than the preserved category.

The results of data injection are used as a baseline for SDC rates. The SDC rate (13.7%) for the preserved category is close to the SDC rate produced by data injection (14.4%). SDC sources cannot be eliminated by only protecting data. The fields in the instruction encoding that are likely to cause a preserved case should not be ignored when designing SDC detectors. The SDC rate of preserved category is 10.3% higher than that of the realigned category. Data injection produces the largest standard deviation in crash rate, followed by the preserved category and realigned category. Therefore, the crash rate produced by faulty instruction encoding is less weakly affected by the context of program than the crash rate produced by faulty data. This is also true for the benign and SDC rates. We present a detailed analysis for each category in the following sections.

The benchmarks cover 40.0% of instruction types of the whole ISA. Some instructions may have very low execution frequencies, so the probability that these instructions are flipped by soft error is also low. This study does not aim to target all instruction types. Since we aim to increase the resilience of ISA, the instructions with higher execution frequencies should be paid more attention. Instruction frequencies of the studied benchmarks are shown in Figure 5. These results show that data transfer instructions and arithmetic instructions represent 66.6% of the instructions used by the benchmarks. These values match results from empirical studies of instruction usage [26]. Due to the limited space, we describe the injection results of the most frequently used instructions.

5. Experimental Results for Realigned Category

In this section, we attempt to determine why faults in the realigned category produce a high crash rate (93.2%), short latency, and low standard deviation. We use *error code* in the trace file for the various crashes to identify crash causes. The distribution of crash causes for realigned cases is shown in Figure 6. The major cause of the high crash rate is segmentation error, which encompasses 90.4% of all crash cases.

To further determine the causes of segmentation error, we extract the last instruction before the crash occurs and compare the address it accesses with the segment boundaries. The address is stored in the instruction's data traces. Segment boundaries are obtained using a run-time probe, which probes the "/proc" of LINUX when the instruction is executed. The distribution of segmentation causes is shown in the right pie chart of Figure 6. The major causes of segmentation errors are listed below.

5.1. Lack of Read or Write Permission.

This is primarily induced using unanticipated addressing modes. For the realigned instructions, fields of the instructions are reassembled, which can easily lead to unanticipated addressing modes. We show differences between the addressing modes in the original instructions and instructions with changed encoding by comparing the base register or index register used for memory addressing in Figure 7. The base register and index register for a specific instruction can be queried in its data trace. 81.6% of the original instructions involve accessing EBP or ESP for addressing. EBP points to the bottom of the current stack frame, and ESP points to the top of stack. A stack operation, such as loading local variables, uses ESP or EBP to find the stack segment. For instruction with changed encoding, the percentage of addressing modes using ESP and EBP drops to 19.4%. EAX has the largest

Table 3: Statistics on the distribution of alignment-based categories.

Category	Injected	Average rate (%)			STD for the outcome rates		
		SDC	Benign	Crash	SDC	Benign	Crash
Realigned	19295 (27.3%)	3.4	3.0	93.2	0.006	0.011	0.009
Preserved	50244 (71.0%)	13.7	26.2	59.1	0.031	0.047	0.039
Invalid	1195 (1.7%)	0	0	100	—	—	0
Data injection (baseline)	658411	14.4	43.5	41.4	0.054	0.091	0.098

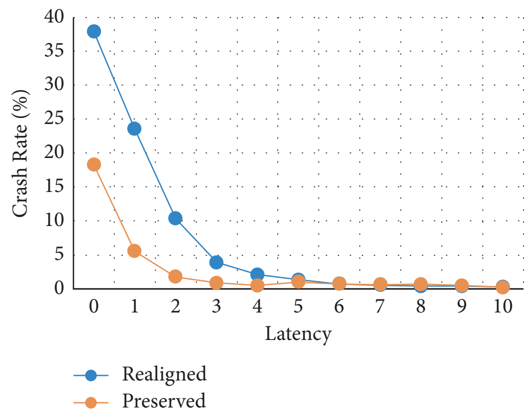


Figure 4: Crash rate for the realigned category and preserved category with 0 to 10 dynamic instructions.

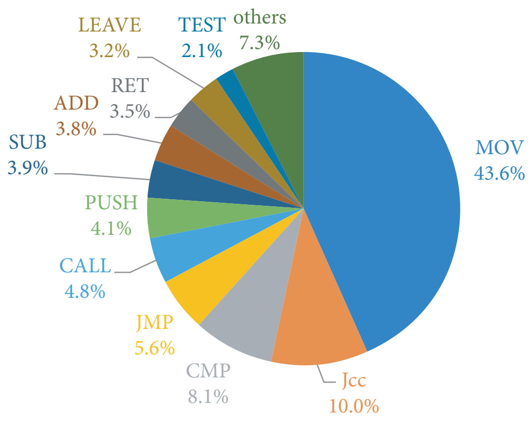


Figure 5: Instruction frequencies of the studied benchmarks.

percentage (40.2%) for addressing among all registers. The immediate byte 0x0 is interpreted as the MODR/M byte in many instructions with changed encoding, denoting an effective address [EAX], just as the example shown in Table 2. EBX (12.4%) and ECX (18.0%) also take up significantly larger portions for addressing than fault-free execution. These registers are commonly used for general storage of intermediate results during computation. When these registers are used for addressing, the address in these registers should be calculated and stored first. Therefore, the value of these registers may not be ready for addressing operation. Whether the value of the register is a legal address is tested to verify the effect of fault on addressing.

We investigated the value of EAX using TracePrinter to extract the value of registers during each dynamic instruction. TracePrinter probes the "/proc" system of LINUX to record the segment boundaries. We mapped each EAX value to a specific segment. The distribution of mapped segments of EAX values is shown in Figure 8. By analyzing the EAX values in the traces, we find that an average of 55.9% of EAX values does not belong to any segment. Most of these values are small positive numbers, pointing to the reserved section in memory. On LINUX, */proc/sys/vm/mmap_min_addr* controls the lowest virtual memory address, and the default setting is 0x10000. The results conform to the fact that EAX is often used to store temporary computation results and function return values. During execution of the realigned instructions, the value of EAX is seldom ready for addressing.

5.2. Executing Privileged or I/O Sensitive Instructions. Many generated instructions in the realigned instruction sequence belong to privileged and I/O sensitive instructions, including HLT, IN, INS, OUT, OUTS, CLI, and STI. The privileged instructions can be executed only when the CPL is 0 (most privileged). I/O sensitive instructions can be executed only if the current privilege level (CPL) of the program or task currently being executed is less than or equal to the IOPL. In a typical protection ring model, access to the I/O address space is restricted to privilege levels 0 and 1. Moreover, only the kernel and device drivers are allowed to execute I/O sensitive instructions. These conditions cannot be satisfied when the realigned instructions are executed.

The two primary reasons for segmentation error involve the addressing convention and permission check rule defined by the ISA. The convention should be followed by all programs developed under the ISA; thus, the standard deviation of crash rates is low. Both reasons cause a crash immediately; thus, the average crash latency is low.

6. Experimental Results for Preserved Category

The preserved category produces relatively high SDC and benign rates compared with the realigned category. The preserved category accounts for 91.3% of SDC cases and 95.8% of benign cases. In this section, we attempt to understand the reasons for various behaviors of preserved cases, especially for SDC propagation. Each field in the instruction encoding corresponds to a varied functionality, and a fault in each field causes different behaviors during injection, and thus, we discuss the results of each field separately. The results are described as follows: (1) opcode field, (2) MODR/M and SIB, and (3) displacement and immediate fields. The injection results for each instruction field are shown in Figure 9.

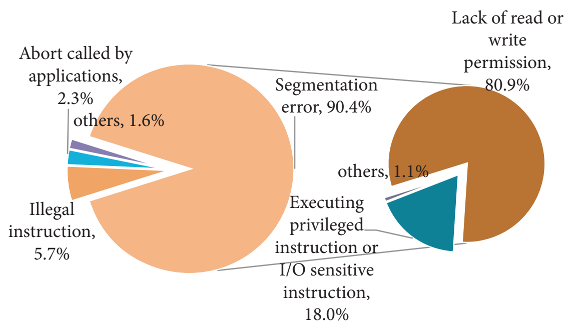


Figure 6: Distribution of crash causes for realigned cases.

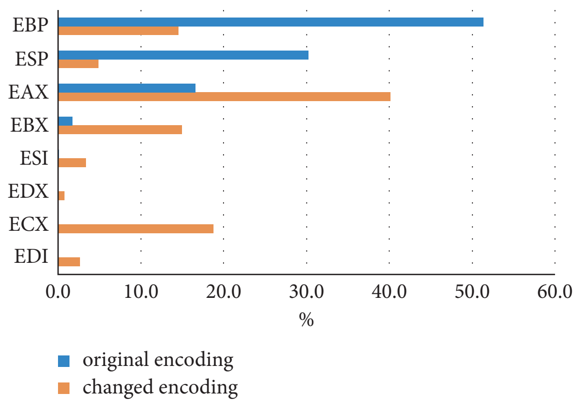


Figure 7: Distribution of register used for addressing in the original encoding and changed encoding.

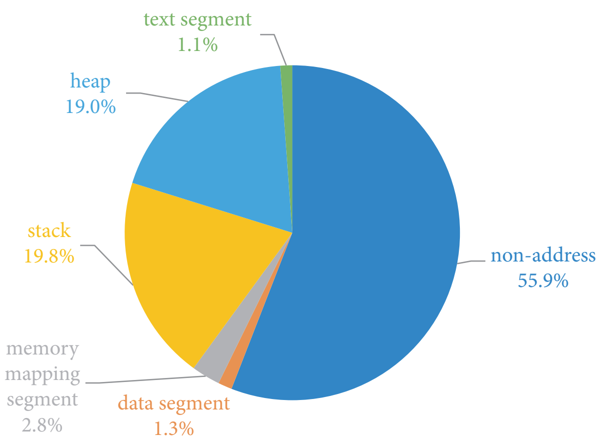


Figure 8: Distribution of mapped segments of EAX values.

6.1. Opcode Field. The opcode field produces an SDC rate of 13.9%, which is close to the baseline value (14.4%). A bit flip in the opcode can change the class of instructions and thus change the functionality of the instruction. In total, 466 types of instruction class mutations were found in the experiment. Figure 10 shows the instruction class mutations that appeared more than 50 times. The thickness of the line indicates the number of cases. The largest number of mutations observed is MOV ⟶ OR, accounting for 6.4% of all cases. We categorize the injection results of instruction class mutations by the original instruction's group. The groups include data transfer instruction group, call-related instruction group, control transfer instruction group, and arithmetic instruction group. Different colors are used in Figure 10 to differentiate instruction groups. Statistics of instruction class mutations are shown in Table 4. We describe the features of program behavior for each group and introduce typical cases in the following subsections.

6.1.1. Call-Related Instruction. To understand the functionality of call-related instruction, we introduce the entire process of a call to procedure. Figure 11 shows a typical assembly code of a call and data dependence graph. Call-related instructions refer to CALL/PUSH/MOV ESP, EBP/SUB ESP, offset/ADD ESP, and offset/LEAVE/POP/RET. These instructions represent the essential calling convention steps. The line starting from a box with the site name denotes the value stored in the site. The black node represents the value written to a site, and the white node represents the value read from a site. The edge denotes the dependence between nodes. The white node depends on the value stored in the corresponding site and the address of the site if one exists. The value of a site after a write operation depends on the black node. Moreover, the value produced by an instruction depends on some source operands of the instruction. We list the essential instructions in a calling operation and explain the effects of the instructions on the stack.

(i) Assume procedure A calls procedure B. The calling operation starts by using a CALL instruction. CALL B pushes the return address of procedure A to the stack and loads the starting address of procedure B in the EIP register. The return address points to the instruction where execution of procedure A should resume following a return from procedure B.

(ii) Execution of procedure B starts by using PUSH EBP to save old frame pointer. EBP and ESP are subsequently initialized for the stack frame of procedure B. MOV EBP, ESP copies the contents of ESP into EBP. SUB ESP, offset reserves place for local uses.

(iii) When the invocation is over, EBP and ESP must be restored to match the state of procedure A. EBP and ESP can be restored in two ways. The first way is to use a LEAVE instruction, which we call LEAVE-way restoring. The effect of LEAVE equals executing MOV ESP, EBP and POP EBP. MOV ESP, EBP restores ESP, and POP EBP restores EBP. The other way is to use ADD ESP, offset and POP EBP, which we call ADD-way restoring. ADD ESP, offset frees space allocated for procedure B, and thus, ESP is restored. In the end, RET loads the return address of procedure A and procedure A resumes execution.

The red line in Figure 11 is used to mark the propagation path that affects return address. In ADD-way restoring, the return address when executing the RET instruction only depends on ESP, and in LEAVE restoring, return address when executing the RET instruction is affected by both ESP

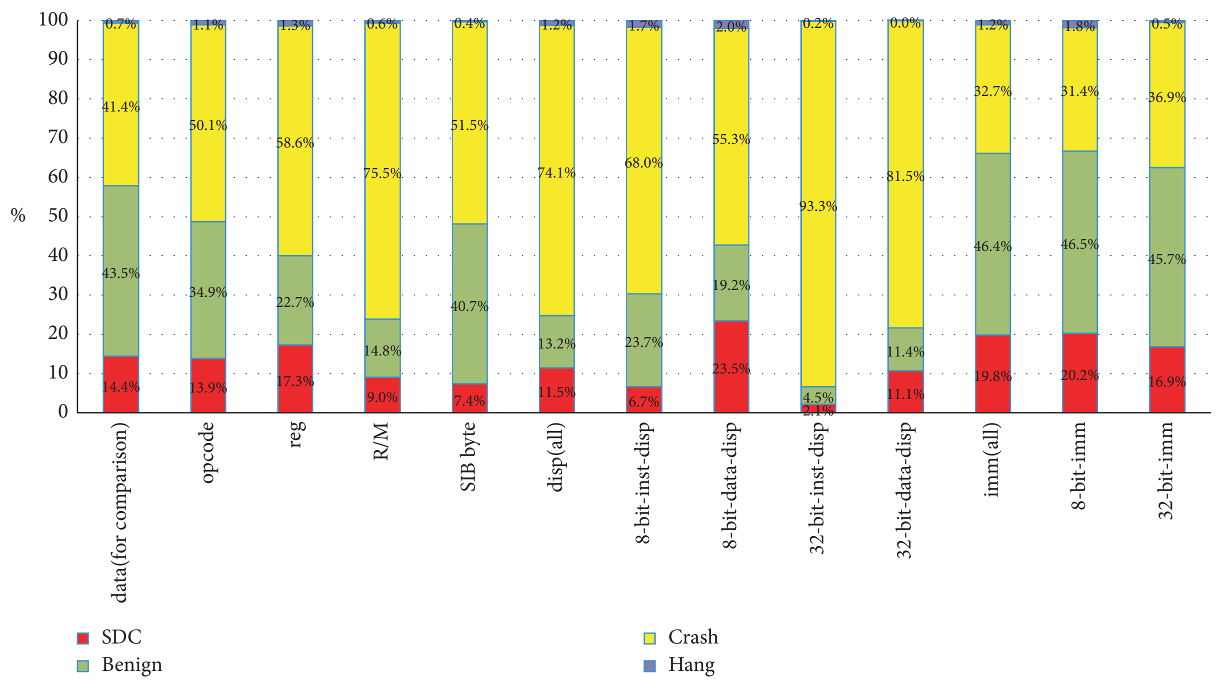


Figure 9: Outcome distribution of preserved cases grouped by instruction field.

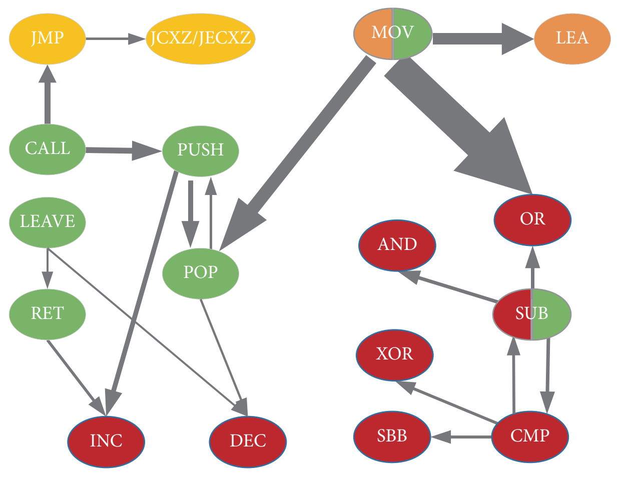


Figure 10: Instruction class mutations due to a bit flip in the opcode field (number of cases ≥50). Green node denotes call-related instructions. Yellow node denotes control transfer instructions. Orange node denotes data transfer instructions. Red node denotes arithmetic instructions.

and EBP. ESP affects return address during the execution from CALL B to MOV EBP, ESP and from LEAVE to RET. EBP affects return address during the execution from MOV EBP, ESP to LEAVE.

We take CALL instruction as an example to describe the effect of a fault on a call-related instruction. There are two mutations from the CALL instruction, which leads to the preserved category, CALL ⟶ JMP, and CALL ⟶ PUSH.

(i) The mutation CALL ⟶ JMP causes crashes at a 96.0% rate. The CALL and JMP instructions use the same displacement field, which means that, after the execution of JMP, the processor that jumps to the starting location of the procedure CALL instruction was invoked. However, the JMP instruction does not record return information. When the called procedure finishes execution, it loads the value from the top of the stack, which is likely to be a procedure parameter. Since the value is not a legal address that points to text segment, the return operation should cause a segmentation error.

(ii) The mutation CALL ⟶ PUSH causes crashes with a substantially lower rate (50.8%) than CALL ⟶ JMP. Although the push operation affects ESP, it may not affect the return operation if the procedure applies LEAVE-way restoring. EBP is unaffected, and thus, the correct return address is loaded when the RET instruction is executed. Otherwise, if the procedure applies ADD-way restoring, it is likely to cause a segmentation error because ESP is corrupted and a bad return address is used, which is similar to the situation of mutation to JMP. We find that, in the CALL ⟶ PUSH mutation, the caller procedures apply LEAVE-way restoring in all SDC or benign cases.

MOV EBP, ESP is a heavily used instance of MOV reg1, reg2. The mutation MOV reg1, reg2 ⟶ OR reg1, reg2 incurs a high crash rate (63.4%). The destination operand EBP is used for addressing, and even slight changes to EBP cause illegal access to memory. Moreover, SUB ESP, offset is a heavily used instance of SUB reg, imm. SUB ESP, offset is used to reserve a place for local use. Therefore, the mutation causes an erroneous ESP and incurs a high crash rate. Other call-related instructions PUSH/LEAVE/RET also produce crashes with high rate.

Table 4: Statistics on bit flips in opcode field.

Inst group	Original inst ⟶ changed inst	Injected	Average rate (%)			Sites may be affected				
			SDC	Benign	Crash (%)	EIP	Return address	EFLAGS	Dest operand	Source operand
Call-related instruction	CALL ⟶ JMP	177	2.3%	0%	96.0		√			
	CALL ⟶ PUSH	177	31.6%	16.9%	50.8	√	√			
	MOV reg1, reg2 (MOV EBP, ESP) ⟶ OR reg1, reg2	112	8.9%	27.7%	63.4		√	√	√	
	SUB reg, imm (SUB ESP, offset) ⟶ CMP reg, imm	108	4.6%	26.9%	67.6		√	√	√	
	SUB reg, imm (SUB ESP, offset) ⟶ AND reg, imm	108	6.5%	14.8%	77.8		√	√	√	
	SUB reg, imm (SUB ESP, offset) ⟶ OR reg, imm	108	6.4%	26.9%	64.9		√	√	√	
	PUSH ⟶ POP	151	3.3%	4.0%	92.7		√			
	PUSH ⟶ INC	151	3.3%	6.6%	90.1		√	√		
	LEAVE ⟶ DEC	62	0	0	100		√	√		
	LEAVE ⟶ RET	63	0	0	100	√	√			
	RET no argument ⟶ RET intersegment	90	0	0	100	√	√			
Control transfer instruction	JMP ⟶ JCXZ	66	27.3%	56.1%	16.7	√				
	Jcc ⟶ Jcc (ttt-bit)	636	21.1%	58.0%	18.9	√				
	Jcc ⟶ Jcc (n-bit)	212	46.7%	13.7%	37.7	√				
Arithmetic instruction	CMP mem, imm ⟶ SUB mem, imm	91	12.1%	76.9%	11.0			√	√	
	CMP mem, imm ⟶ XOR mem, imm	91	13.2%	75.8%	9.9			√	√	
	CMP mem, imm ⟶ SBB mem, imm	91	12.1%	75.8%	12.1			√	√	
Data transfer instruction	MOV mem, reg ⟶ OR mem, reg	382	24.1%	33.8%	40.3			√	√	
	MOV mem, reg ⟶ LEA reg, mem	350	22.3%	21.7%	52.9				√	√
	MOV mem, reg ⟶ MOV reg, mem	382	23.6%	23.0%	50.8				√	√
	MOV reg, mem ⟶ OR reg, mem	478	15.7%	43.3%	40.8			√	√	
	MOV reg, mem ⟶ POP	401	12.7%	13.7%	73.3		√		√	
	MOV reg, mem ⟶ MOV mem, reg	478	15.9%	25.7%	56.0				√	√

To conclude, bit flips in the opcode field of call-related instructions probably cause crashes because it affects loading of return address. Whether loading of return address is affected depends on the restoring style of the caller procedure (ADD-way or LEAVE-way). If the return address is affected, all cases cause crashes without any exceptions in the experiments. The result is aligned with our prior approach, which tried to inject faults to ESP or EBP [27]. The prior approach showed that if a proper return address was not loaded it tended to incur crash.

6.1.2. Control Transfer Instruction. The control transfer instruction discussed here includes JMP and Jcc instructions. JMP instruction transfers program control to a different point in the instruction stream without recording return information. One finds that the crash rate with JMP instructions is much lower than CALL instruction. The mutation JMP ⟶ JCXZ produces a high benign rate. JCXZ checks ECX or CX for 0. If ECX equals 0, it performs a jump to the target address, which is the same as in the JMP instruction, so the bit flip does not cause any change to the execution and leads to benign. If ECX does not equal 0, it continues with the instruction following the JCXZ instruction that may lead to an SDC.

The Jcc instruction checks the state of the EFLAGS register, and if the flags are in the specified state, a jump to the target instruction is performed as specified by the destination operand. ttt-bit and n-bit specify a condition asserted or negated for the Jcc instructions. Flips in ttt-bit or n-bit of the opcode field change the branch condition instead of the instruction class. n-bit indicates whether the condition for Jcc instruction or its negation should be used. Injections on n-bit produce the highest SDC rate (46.7%) in the experiment. A bit flip in n-bit causes execution of the opposite branch. We show an example in Figure 12. In the original source code, the condition of an if statement is $proc! = 0$. When the n-bit is flipped, the JZ instruction (encoding 74) is

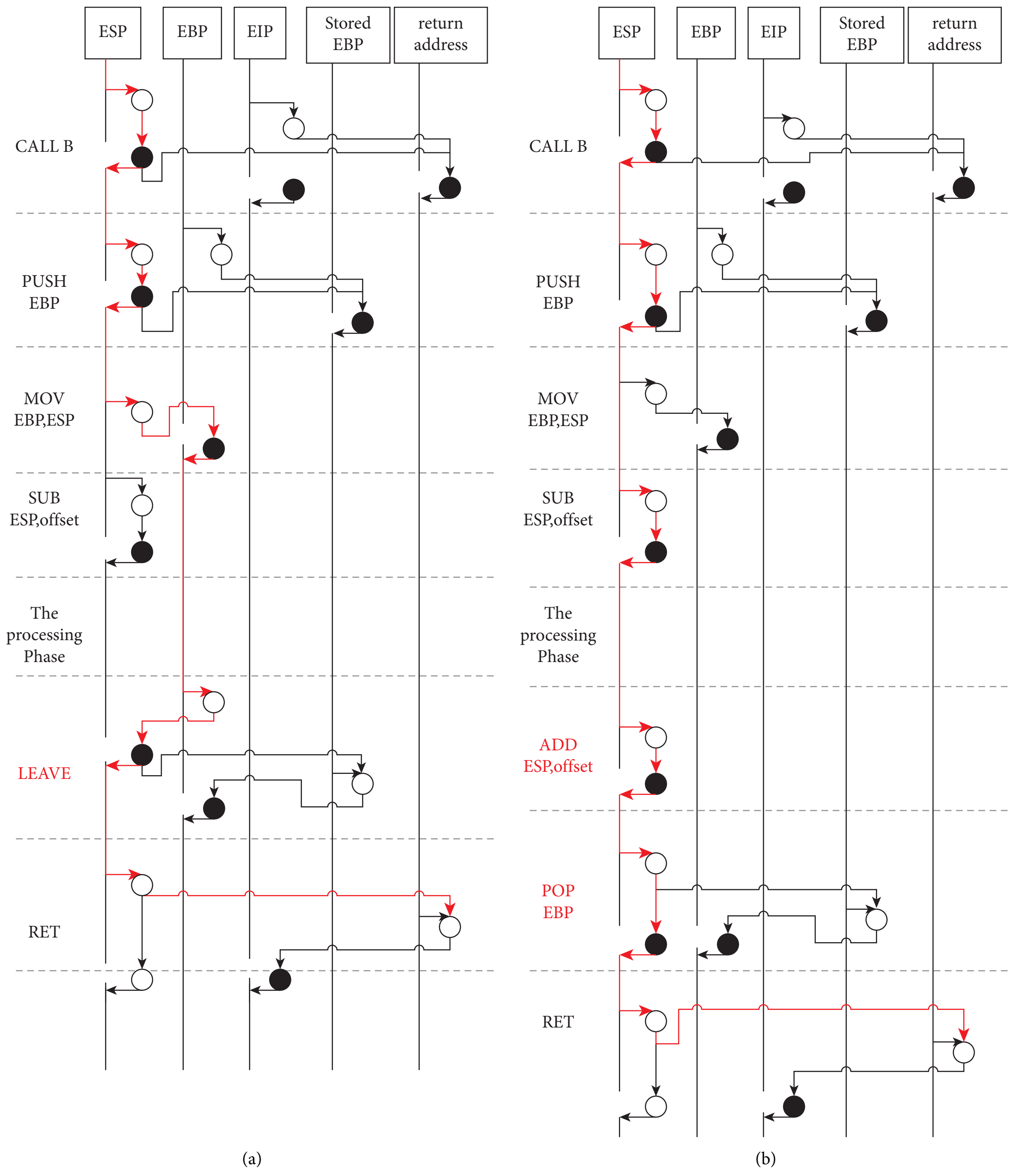


FIGURE 11: A snippet of typical assembly code of a call to procedure. (a) LEAVE-way restoring. (b) ADD-way restoring.

changed into JNZ instruction (encoding 75). The condition is changed in every execution after the flip; i.e., this is a permanent effect. The effect is equivalent to introducing a logic bug in the if statement, thus changing the condition into $proc = 0$. Benign cases make up 13.7% of the results. Using the definition proposed by Wang et al. [28], a branch instruction is considered a Y-branch if the negated branch outcome does not affect the final output from the program. Most benign cases involve the structure of the if statement, which tests several conditions. If one of these conditions is true, it returns true. If at least two of the conditions evaluate to be true, the flip in n-bit of one Jcc instruction does not change the execution of the body of the if statement.

The ttt-bit gives the condition to test and produces benign with a high rate (58.0%). The primary cause of the high benign rate is one-to-many mappings from the value of EFLAGS to the conditions denoted by the encoding of ttt-bit. In the example shown in Figure 12, the ttt-bit is flipped and the JZ instruction is changed into a JBE instruction. The condition of the if statement is changed into $proc > 0$. If the variable *proc* is greater than 0 when executing the if statement, both *proc*! = 0 and *proc* >0 are satisfied, and thus, the bit flip does not modify the direction of the branch.

6.1.3. Arithmetic Instruction. Bit flips in the opcode field of an arithmetic instruction produce prominently higher rates of benign (around 75%) compared with the baseline (the benign rate of data injection, 43.5%). The CMP instruction sets the status flags in the EFLAGS register according to the comparison results. Following CMP instruction, Jcc instruction performs a jump if EFLAGS is in the specified state. CMP instruction can be mutated into other arithmetic instructions, including SUB/XOR/SBB instructions. These mutations can alter EFLAGS and the destination operand, and they share a similar benign rate. CMP mem, imm ⟶ SBB mem, imm or XOR mem, imm changes EFLAGS and the destination operand. CMP mem, imm ⟶ SUB mem, imm only modifies the destination operand. The SUB instruction modifies EFLAGS in the same manner as the CMP instruction, and thus, the subsequent Jcc instruction is unaffected by the flip. The difference between the two instructions is whether to store the subtraction result. After subtraction, the SUB instruction stores the result in the destination operand, while the CMP instruction keeps the original operand. If the faulty value is not loaded again, the effect of executing CMP instruction is equivalent to executing SUB instruction, and the fault is masked.

6.1.4. Data Transfer Instruction. The data transfer instructions move data between memory and the general purpose and segment registers. The MOV instruction is used most frequently and makes up 43.6% of the total executed instructions. MOV reg1, reg2 was discussed in the call-related instruction section. Here, we discuss mutations from MOV mem, reg or MOV reg, mem, as shown in Table 4.

(i) The mutation MOV ⟶ OR encompasses the largest proportion of bit flips in the opcode field. The MOV and OR instruction encodings only differ in the most significant bit, and they share the same destination operand and source operand encoding. The mutation affects the value of destination operand and EFLAGS. The fault in EFLAGS is often masked by the next CMP or other arithmetic instruction. The change in the value of destination operand is the main factor that causes SDC-prone propagation. Whether or not the fault incurs an erroneous destination operand depends on the value of destination operand and source operand.

A bitwise comparison between the destination operands of MOV and OR instruction is shown in Table 5. Only if the bit of destination operand before execution is 1 and bit of source operand is 0, MOV and OR will have different corresponding bit values of destination operand. Otherwise, they arrive at the same bit value of the destination operand. For example, execution of MOV AL, BL produces AL = 13 when AL = 5(0101b) and BL = 13(1101b). If it executes OR AL, BL, we still have the result AL = 13. In particular, when the destination operand equals 0, MOV and OR reach the same value of destination operand. Therefore, the mutation may not cause any change to the value of destination operand.

We assume that the value of the destination and source operand bits obey uniform distributions; i.e., the probability of the bit value 1 is 0.5, and n is used to represent the length of the destination operand. The mask probability is $(0.75)^n$ when the MOV and OR instructions produce the same value of destination operand. When the length of the destination operand is 8 bits, the mask probability is equal to $0.75^8 \approx 10.0\%$. The distribution of EAX values is shown in Figure 8. Most non-address values are small positive numbers. These values may possess many zero bits, especially in higher bits, which increase the actual mask probability.

(ii) MOV mem, reg can be mutated into LEA reg, mem. The LEA instruction computes the effective address of the second operand and stores it in the first operand. For example, by flipping a bit, MOV [ESP], EAX is mutated into LEA EAX, [ESP]. The effective address of [ESP] is ESP, thus executing LEA EAX, [ESP] has the same effect as executing MOV EAX, ESP. The mutation affects the value of the register and memory location. The register is often for temporary use. We investigated the traces of such a mutation. After execution of LEA reg, mem, the specific register written by the LEA instruction is always rewritten by another instruction

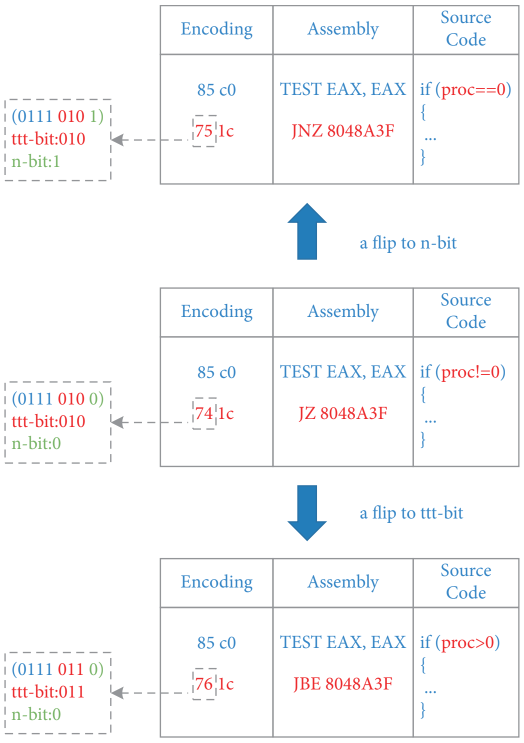


Figure 12: Example of flips in ttt-bit and n-bit in the Jcc instruction.

before the next read operation, which means the result of LEA is not used by any other instructions. Although the LEA instruction changes the value of the register, it has no effects on subsequent computation. According to the definition in AVF analysis, LEA reg, mem is a dynamically dead instruction [13]. Because the memory location is not written using LEA instructions, the value of the memory location should be different from that after the original MOV instruction is executed. The major factor that causes SDC-prone propagation is the faulty memory location. Therefore, the effect of executing LEA instructions is equivalent to that of deleting the original MOV instruction.

(iii) MOV mem, reg⟶ MOV reg, mem changes the direction of data operation. The mutation may affect the register and memory location. MOV reg, mem⟶ MOV mem, reg also changes the direction of data operation. Both mutations produce higher SDC rates than the baseline. Whether it causes SDC depends on how the memory location and register are used in the subsequent computation, and thus, error propagation is context-dependent.

To conclude, all mutations of an MOV instruction change the value of the destination operand. In some mutations (MOV mem, reg⟶LEAreg, mem, MOV mem, reg⟶MOVreg, mem, MOV reg, mem⟶MOVmem, reg), the value of the source operand is also affected. The SDC rates of these mutations vary due to differences in the effects of the operand on computation.

6.2. MODR/M and SIB. The MOD field (2 bits) combines with the R/M field (3 bits) to form 32 possible values (8 registers and 24 addressing modes) [9]. The MOD field determines whether a displacement byte follows the MODR/M byte. The instruction lengths according to MOD, R/M, and BASE fields are shown in Table 6. We can find 4 sets of formats with identical instruction length, {#1,#7},{#2,#6},{#3,#8}, and {#4,#9}.

Either {#2,#6} or {#4,#9} has at least 2 different bits, and thus, a bit flip cannot transform one format to another, and a flip in the MOD field that leads to the preserved category has two possible modes, as shown in Figure 13. The experimental results show that most cases fall in the realigned category. 96.8% and 1.5% of all injections fall in the realigned and preserved categories, respectively.

Regarding bit flips in the R/M field, the modes in which the instruction length is changed are listed in Figure 14. This can also change the memory location or the register represented by the R/M field. 64.4% of all injection results fall into the preserved category. Injection on the R/M field produces a crash rate of 75.5% because it specifies the addressing mode. The reg field specifies a general purpose register operand and produces a higher SDC rate (17.3%) than the baseline (14.4%). A change in the reg field may affect the value of the original register and the register with the changed encoding.

The SIB is required by certain encodings of the MODR/M byte as a second addressing byte. The base-plus-index and scale-plus-index forms of 32 bit addressing require the SIB. The SIB includes scale, index, and base fields. For the SIB, the flip mode that leads to an instruction length change is #1⟶#2 or #2⟶#1. The preserved category encompasses up to 93.2% of injection results. Injection on the SIB produces a benign rate of 42.1%. By investigating the traces, we find that 59.2% of benign cases are caused by none encoding of index register. The index field specifies the register number of the index register. It is allowed that none of the index registers is specified in the index field (encoded "100b"). When none is specified in the index register, only the base field determines the effective address, and thus, a bit flip in the scale field has no effect on the address. For example, the SIB 0x20 (00 100 000b, scale field = 00b, index field = 100b) and SIB 0x60 (01 100 000b, scale field = 01b, index field = 100b) refer to the same displacement [EAX]. Thus, the bit flip in the SIB from 0x20 to 0x60 does not change the effective address and produces a benign case.

6.3. Displacement and Immediate Fields. Injection on the displacement field produces a higher crash rate (74.1%) than the baseline (41.4%). The results are shown in terms of the number of bits (8 bit, 32 bit) and usage (instruction address, data address) in Figure 9. Displacement as instruction address is used by control transfer instruction, such as CALL and JMP. The crash rate produced by injection on the displacement field as an instruction address is approximately 12% higher than the case where injection on the displacement field is a data address. The number of valid addresses with execution permission is often much lower than the number of valid addresses with read and write permissions. Usually, only text segment and certain memory mapping segments have execution permission. The crash rate produced with a 32 bit displacement is approximately 25% higher than that produced with an 8 bit displacement. Faults in higher bit are more likely to reference a different segment, and thus, it is easier to produce a segmentation error.

Injection on the immediate field produces a higher rate of SDC (19.8%) than the baseline (14.4%). Immediate is often used to represent a data value instead of an address, and a fault in the immediate field leads to a high rate of benign and SDC. The statistics of the immediate field do not show a significant gap between varied numbers of bits, which is different from displacement field. The difference between the SDC and benign rates for 32 bit and 8 bit immediate fields is less than 5%.

7. Experimental Results for Invalid Category

When an invalid instruction occurs, we use XED to decode the flipped encoding to find which part of the encoding causes the error. XED is a software library for encoding and decoding IA-32 instructions [9]. Two types of errors obtained from XED are "GENERAL_ERROR" and "BAD_REGISTER." 99.6% of the invalid instruction cases

Table 5: Comparison of execution results for MOV and OR instructions.

Bit of dest operand before execution	Bit of source operand	Bit of dest operand after OR executes	Bit of dest operand after MOV executes	Having same dest bit value
0	0	0	0	√
1	0	1	0	×
0	1	1	1	√
1	1	1	1	√

Table 6: Instruction length according to binary encodings of MOD, R/M, and BASE fields.

#Format	MOD	R/M	BASE(SIB)	Instruction length (bits)
1	00	100	101	48
2			000~100,110,111	16
3		101	—	40
4		000~011,110,111	—	8
5	01	100	—	24
6		000~011,101~111	—	16
7	10	100	—	48
8		000~011,101~111	—	40
9	11	—	—	8

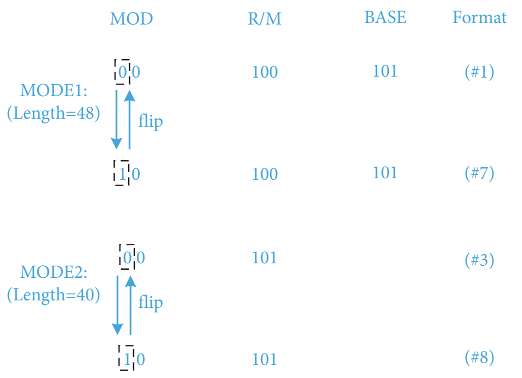


Figure 13: Bit flip in the MOD field, leading to the preserved category.

refer to "GENERAL_ERROR," while the remaining cases refer to "BAD_REGISTER."

(i) "GENERAL ERROR" indicates an invalid MODR/M byte encoding. LEA instruction computes the effective address of the source operand and stores it in the destination operand. The encoding 11B in the MOD field of LEA instruction is not allowed by ISA. 11B in the MOD field denotes that both source operand and destination operand are registers, which violates the definition of LEA instruction. The source operand of LEA instruction should be a memory address. The instruction loads a far pointer from the second operand into a segment register and also reserves 11B in the MOD field, including LDS (load pointer to DS), LES (load pointer to ES), and LFS (load pointer to FS).

(ii) "BAD_REGISTER" indicates an invalid register encoding. When an instruction operates on a segment register, the reg field in the MODR/M byte is called the sreg field and is used to specify the segment register. The IA-32 architecture has 6 segment registers. The sreg field encoding can be 3 bits long, and two encodings (110B and 111B) are reserved.

8. Implications

The main challenge in designing resilient instruction encoding is to reduce the SDC rate. The injection results show that faults in certain fields can easily cause crashes, and only a few encodings of instruction fields are SDC-prone. The observations can potentially help reduce the SDC rate in ISA or fault-tolerant systems. In this section, we compare the results of the typical RISC architecture and IA-32 architecture, and we provide some advice on the design of resilient instruction encoding and validate our proposed resilient instruction encoding through fault injections.

8.1. Comparison with the RISC Architecture. We compare the SDC rates and crash rates of fault injections on typical RISC and CISC ISA. The impact of an ARM Cortex-M0 Thumb ISA on reliability to soft errors was examined by Martinez [10]. The ARM Cortex-M0 Thumb ISA is a RISC architecture with fixed length of 16 bit instructions. Their evaluation shared the same fault model and injection setup as that presented here, so their injection results can be compared with ours. Their injection campaign was also exhaustive. All bits in all instructions for each test program were tested. Once each test ended, the program was reset to guarantee that it was error-free before running a new test. The tests were run with Keil μVision Debugger. The SDC rate of the injections on the Thumb ISA varies significantly from that on the IA-32 architecture. The programs in the Thumb ISA produce an SDC rate ranging from 50.1% to 58.0%. In our experiment, the SDC rates range from 7.2% to 14.2%. Their subject programs are matrix multiplication, quicksort,

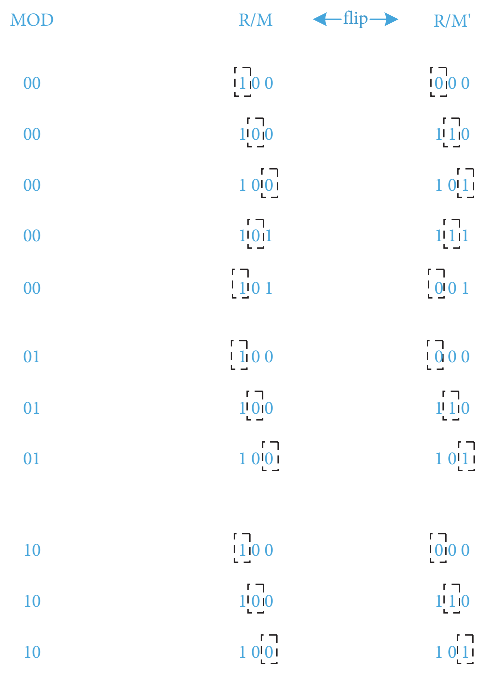


Figure 14: Bit flips in R/M field leading to the realigned category.

bubble sort, and binary search, which include algorithms using iterations, recursion, function calls, and arrays. The same program *quicksort* was tested in their and our injection campaigns. Their results with *quicksort* show an SDC rate of 50.1%, which is 39.6% higher than the SDC rate observed in our experiment. IA-32 ISA can produce lower SDC rates than Thumb ISA, and thus, it is more resilient to soft errors. Without any error detector deployment, the SDC rate can be decreased by switching from Thumb ISA to IA-32 ISA.

We analyze the factors that contribute to the SDC rate of ISA. The instructions in the RISC architecture have a fixed length, which is a vital factor determining fault propagation. A change in the instruction encoding in the RISC architecture does not incur an alignment change, and thus, only one instruction's encoding is altered. We have shown that this leads to a high crash rate if the instruction alignment is not preserved. The reason why the realigned instruction sequence can easily cause a crash is explained in Section 5. Although the experiment was performed in IA-32 ISA, one can infer that a variable instruction length in another CISC ISA may also produce a high crash rate. When the fault causes realignment of the instruction sequence, multiple instructions' encoding may be altered. The instruction in the realigned instruction sequence is unpredictable since any field can be interpreted as opcode field, producing abnormal behavior.

Other factors, such as addressing mode, memory organization, and calling conventions of specific ISA architectures, may affect fault propagation and further affect SDC rate or crash rate. In Thumb ISA, the return address is stored in *link register* (LR) instead of on the stack. Therefore, bit flips in the opcode field of call-related instructions do not directly affect return address. A fault in a call-related instruction can alter return address in the IA-32 architecture, as shown in Section 6.1. Furthermore, the addressing mode of Thumb ISA is much simpler than that in the IA-32 architecture. Thumb ISA supports two basic addressing modes ([reg + reg], [reg + disp]), and the length of the displacement field is 5 bits. Aside from these two modes, the IA-32 architecture also supports [reg + reg + disp], [reg ∗ scale + reg + disp] and a longer displacement field (8 bit, 16 bit, or 32 bit). A fault in a longer displacement field tends to cause crashes at a higher rate, which can be validated by our fault injections on displacement field. The crash rate produced by a fault in a 32 bit displacement field is approximately 25% higher than the crash rate produced by a fault in an 8 bit displacement field as a different segment is more likely to be addressed.

The comparison shows that the SDC rate observed in the IA-32 ISA architecture is much lower than that seen with Thumb ISA. The variable instruction length in CISC ISA can produce a high crash rate in the presence of a fault and reduce the chance that an SDC occurs. The conclusion may be helpful to decide between two alternatives having similar performance, overhead, or power, and then, considering reliability could serve to do a tiebreak.

8.2. Implication for Design of Resilient Instruction Encoding. We propose several modifications to the instruction set encoding scheme that increases the Hamming distance between certain vulnerable encodings and increase the probability of a crash. We use the Hamming distance to denote the number of different bits between two encodings. Encoding B is a neighbor of encoding A if their Hamming distance is 1. As we assume faults are caused by a single-bit flip, the encoding becomes its neighbor in the Hamming space after the flip. Mutations of different instruction classes produce different SDC or crash rates. Changing the neighbor of a specific encoding can affect the SDC rate or crash rate. By reshuffling the encoding of the current instruction set, one finds a trade-off where the crash rate can increase, while the SDC rate can decrease. We show some possible changes in the following fields by applying the findings obtained in the experiment.

(i) opcode field. Certain mutations from frequently used instructions produce high SDC rates. For example, MOV $\longrightarrow$ OR takes up the largest portion in all mutations and produces a high SDC rate. We can modify the encoding to set MOV and RET instructions as neighbors in the Hamming space instead of MOV and OR instructions. The RET and MOV instructions differ in their instruction lengths. The mutation causes realignment of the instruction sequence, which tends to cause a crash. Moreover, as a call-related instruction, RET probably causes a crash when it is generated by a bit flip. When a RET instruction is flipped into an MOV instruction, it is also likely to cause a crash. The overall SDC rate due to faults in the opcode field of ISA is dominated by mutations from frequently used instructions. After reshuffling the encodings, the overall crash rate should increase and the SDC rate should decrease.

(ii) reg field. The original and redesigned encodings of the reg field are shown in Figure 15. For example, the original encoding of EAX in the reg field is 000b, and its neighbors in the Hamming space are ECX (001b), EDX (010b), and ESP (100b). We can swap the encoding of EBP and ECX to set EBP as a neighbor of EAX. The values of EAX seldom belong to any segment, as was described in Section 5. Moreover, it was concluded in Section 6.1 that changes in the value of EBP can easily cause a crash, so the mutations EAX $\longrightarrow$ EBP or EBP $\longrightarrow$ EAX probably cause a crash. After the swap, one would expect the new encoding to produce a higher crash rate than the original encoding. Moreover, we swap encoding (EDX, ESI) to set ESI as the neighbor of EAX. Usually, ESP, EBP, ESI, and EDI store addresses, which are denoted by red nodes in Figure 15. EAX, EBX, ECX, and EDX often store non-address values, which are mostly small positive numbers (the distribution of the EAX value is shown in Figure 7). The swaps put EAX/EBX/ECX/EDX in the neighbor position of ESP/EBP/ESI/EDI. The mutation of any reg encoding probably creates an erroneous address and incurs a crash case. Either two encodings of EAX/EBX/ECX/EDX have the Hamming distance of 2, which prevents the occurrence of SDC-prone mutations.

(iii) MOD, R/M, and SIB fields. Modifications to certain encodings in the MOD, R/M, and SIB fields can be made to increase the percentage of faults in the realigned category when bit flips occur in these fields. We take the MOD field as an example. In Section 6.2, it was shown that a bit flip in the MOD field that causes a fault in the preserved category only has two modes (shown in Figure 13): {#1, #7} and {#3, #8}. We can alter the encoding of the MOD field to increase the Hamming distance between {#1, #7} and {#3, #8}. Therefore, all flips in the MOD field lead to faults in the realigned category. As stated before, faults in the realigned category produce crashes with a much higher rate than faults in the preserved category, and thus, the modification increases the overall crash rate.

8.3. Evaluation of New Encoding Scheme. We propose a change to the encoding of opcode field to set RET instruction and MOV instruction as neighbors in the Hamming space instead of MOV and OR instruction. The neighboring encodings of MOV mem, reg are shown in Figure 16. The MOV and OR instructions only differ in the most significant bit, and they share the same destination operand and source operand encoding. We describe the injection results of MOV$\longrightarrow$ OR in the discussion of data transfer instruction of Section 6.1. The RET (1 byte) and MOV ($\geq$2 bytes) instructions differ in their instruction lengths. In the old encoding scheme, the mutation MOV$\longrightarrow$ OR results in a preserved case. According to our new encoding scheme, the mutation MOV$\longrightarrow$ RET results in a realigned case.

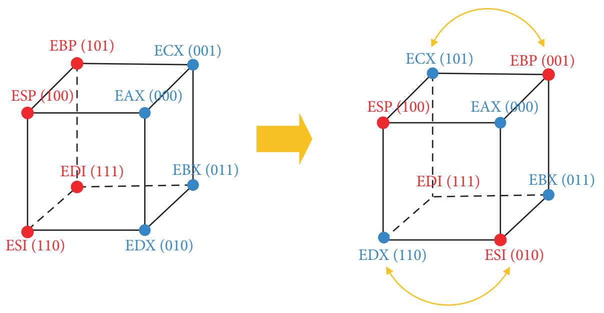


FIGURE 15: Hamming space of the original and redesigned reg field encodings.

The idea is achieved by reshuffling the encoding of the current instruction set. Table 7 shows the mapping from the encoding in the old instruction set to the encoding in the new instruction set. The encoding of opcode field of MOV mem, reg is 0x89, which is neighboring to the encoding of opcode field of OR mem, reg(0x09). Note that MOV mem, reg has MOD byte, so it is at least two bytes. The encoding of RET instruction is 0xC3, and it is only one byte.

After the reshuffling, the encoding of opcode field of OR mem, reg is set to 0xC3. The encoding of RET is set to 0x09. So, RET instruction becomes a neighbor of MOV instruction in the Hamming space.

We construct a hypothetical processor structure for evaluating the new encoding scheme, which is proposed by Xu et al. [12]. Assuming the existence of a hypothetical processor that incorporates the new instruction encoding, whenever an instruction from the text segment is picked for fault injection, it is mapped from the old encoding to the new one. Then, a bit in the mapped new instruction is selected to obtain an erroneous instruction in the new encoding. The erroneous instruction is then mapped back to the old instruction encoding and is executed on the processor. We believe that this process can accurately emulate fault injection for the new encoding on the current processor.

For example, consider instruction MOV DWORD PTR [ESP], EAX (encoding = 0x89 04 24) from the text segment of a current IA-32 executable. Assume that the most significant bit is flipped (from 1 to 0), and it results in 0x09 04 24. This value is mapped back to the old encoding scheme using the table, resulting in 0xC3 04 24. So, the instruction to be executed should be RET (0xC3) and ADD AL, 0x24 (0x04 24). RET is then executed on the current processor.

The results for the injections on the most significant bit of opcode field of MOV instruction under old encoding scheme and new encoding scheme are shown in Figure 17. In the original fault injection experiment, the mutation MOV mem, reg - > OR mem, reg incurs a high rate of SDC (15.7%). The fault injections on the instruction encoding with the new encoding scheme show that the mutation MOV mem, reg - > RET incurs a crash rate of 100% and no SDCs. So, the opcode field of MOV mem, reg is much less SDC-vulnerable than that of original encoding scheme.

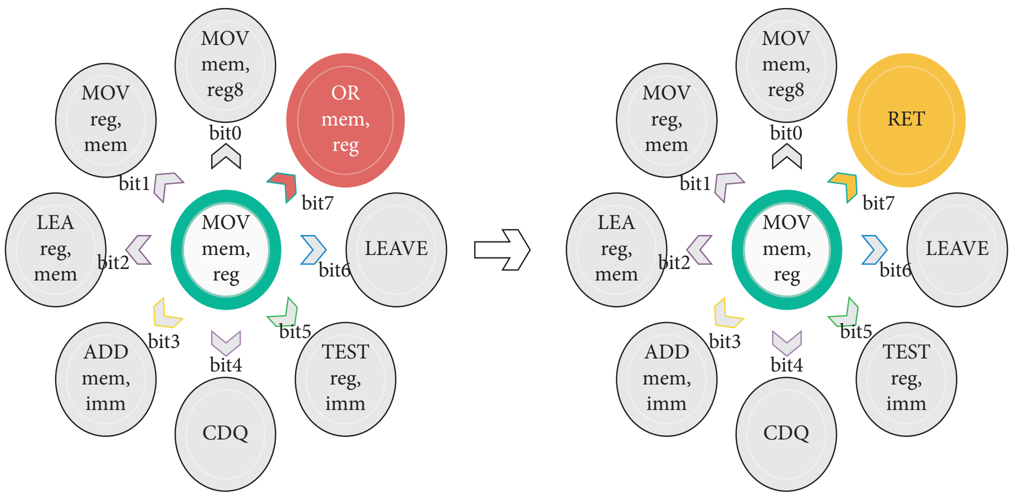


Figure 16: Neighboring encodings of MOV mem, reg in the old encoding scheme and new encoding scheme.

Table 7: OR and RET instruction encoding mapping.

Mnemonics	Old encoding of opcode field	New encoding of opcode field
OR mem, reg	09	C3
RET	C3	09

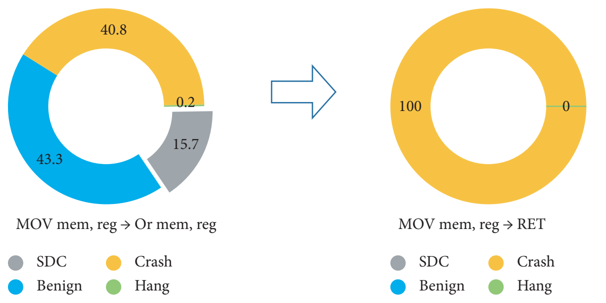


Figure 17: Results for the injections on the most significant bit of opcode field of MOV instruction under old encoding scheme and new encoding scheme.

9. Conclusion

In this study, we characterized the behavior of a program under faulty instruction encodings. We injected over 70,000 faults into binary executables and analyzed trace files to seek causes of SDC-prone or crash-prone errors. The key findings from the experiments are summarized as follows:

(i) Whether the alignment of instruction sequence is preserved strongly affects the observed crash rates. Faults in the realigned category produce crashes with high rate (93.2%), short latency, and low standard deviation. If the alignment is preserved, the SDC rate (13.7%) is close to the SDC rate of injections on data (14.4%), meaning that instruction encoding can also be major source of SDC.

(ii) SDC-prone fields include the reg field, opcode field, and immediate field. Bit flips in n-bit produce the highest SDC rate (46.7%) because a bit flip causes the opposite branch to execute. These fields are vulnerable parts of code segments and need to be protected against soft errors or bit flip attacks.

(iii) By investigating the injection traces, we reveal the following crash patterns. By leveraging realignment of instruction sequence and calling convention violation, we also show some implications for designing resilient instruction encoding.

(i) Realignment of instruction sequence, activated by faults in opcode, MOD, and R/M fields.

(ii) Addressing failure, activated by faults in displacement. The crash rates of the displacement field vary wildly due to the number of displacement bits and their usage.

(iii) Calling convention violation, activated by faults in the opcode field of call-related instructions. Crashes are caused since the loading of the return address is affected.

In future studies, we plan to estimate the reliability of specific ISA by calculating the mutation probability and SDC rate. For these instruction types that are not targeted in this study, if the mutation from the uncovered instruction to its neighbor matches any crash patterns or SDC patterns, we can predict the program behavior when the encoding of the instruction is flipped. We intend to evaluate the detection rate of software-implemented fault tolerance in the presence of faulty instruction encoding. Furthermore, we plan to perform fault injections on RISC ISA to get insight into the fault propagation under RISC ISA and make a comparison between the propagation patterns of different ISAs.

Data Availability

The dataset could be found at https://sir.csc.ncsu.edu/portal/bios/tcas.php#siemens.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was funded by the Natural Science Foundation of China (No. 62002030) and Key Research and Development Plan Project of the Shaanxi Province, China (No. 2019ZDLGY17-08, 2019ZDLGY03-09-01, 2019GY-006, and 2020GY-013).

References

[1] A. P. Shah, S. K. Vishvakarma, and M. Hübner, "Soft error hardened asymmetric 10T SRAM cell for aerospace applications," *Journal of Electronic Testing*, vol. 36, no. 2, pp. 255–269, 2020.

[2] A. Mahmoud, R. Venkatagiri, K. Ahmed et al., "Minotaur: adapting software testing techniques for hardware errors," in *Proceedings of the Architectural Support for Programming Languages and Operating Systems(ASPLOS)*, pp. 1087–1103, ACM, New York, United States, April 2019.

[3] G. Li, K. Pattabiraman, S. K. S. Hari, M. B. Sullivan, and T. Tsai, "Modeling soft-error propagation in programs," in *Proceedings of the Dependable Systems and Networks(DSN)*, pp. 27–38, IEEE, Luxembourg, Luxembourg, June 2018.

[4] N. Yang and Y. Wang, "Predicting the silent data corruption vulnerability of instructions in programs," in *Proceedings of the IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, pp. 862–869, IEEE, Tianjin, China, December 2019.

[5] X. Li, S. V. Adve, P. Bose, and J. A. Rivers, "SoftArch: an architecture-level tool for modeling and analyzing soft errors," in *Proceedings of the Dependable Systems and Networks (DSN)*, pp. 496–505, IEEE, Yokohama, Japan, July 2005.

[6] J. A. Martinez, J. A. Maestro, and P. Reviriego, "A scheme to improve the intrinsic error detection of the instruction set architecture," *IEEE Computer Architecture Letters*, vol. 16, no. 2, pp. 103–106, 2016.

[7] M. Atamaner, O. Ergin, M. Ottavi, and P. Reviriego, "Detecting errors in instructions with bloom filters," in *Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pp. 1–4, IEEE, Cambridge, UK, October 2017.

[8] S. Wang and G. Duan, "On the characterization and optimization of system-level vulnerability for instruction caches in embedded processors," *Microprocessors and Microsystems*, vol. 39, no. 8, pp. 686–692, 2015.

[9] Intel, *The Intel 64 and IA-32 Architectures Software Developers Manual, Volume 1: Basic Architecture*, Intel, 2006.

[10] J. A. Martinez, J. A. Maestro, and P. Reviriego, "Evaluating the impact of the instruction set on microprocessor reliability to soft errors," *IEEE Transactions on Device and Materials Reliability*, vol. 18, no. 1, pp. 70–79, 2018.

[11] J. Martinez, M. Atamaner, P. Reviriego, O. Ergin, and M. Ottavi, "Opcode vector: an efficient scheme to detect soft errors in instructions," *Microelectronics Reliability*, vol. 86, pp. 92–97, 2018.

[12] J. Xu, S. Chen, Z. Kalbarczyk, and K. lyer, "An experimental study of security vulnerabilities caused by errors," in *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pp. 421–430, IEEE, Gothenburg, Sweden, July 2001.

[13] S. S. Mukherjee, C. Weaver, and J. Emer, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in *Proceedings of the Microarchitecture*, pp. 29–40, IEEE, San Diego, CA, USA, Dec. 2003.

[14] 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.

[15] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: a remote software-induced fault attack in Javascript," in *Proceedings of the Detection of Intrusions and Malware, and Vulnerability Assessment(DIMVA)*, pp. 300–321, Springer, San Sebastián, Spain, July 2016.

[16] V. Van Der Veen, F. Yanick, L. Martina et al., "Drammer: deterministic rowhammer attacks on mobile platforms," in *Proceedings of the Computer and Communications Security(CCS)*, pp. 1675–1689, ACM, New York, USA, October 2016.

[17] A. T. Olesen, *Off by a Bit: Exploring Bit-Flip Vulnerabilities through Program Emulation and Symbolic Execution*, Master's thesis, Aalborg University, Aalborg, Denmark, 2017.

[18] F. K. Lodhi, S. R. Hasan, O. Hasan, and F. Awwad, "Analyzing vulnerability of asynchronous pipeline to soft errors: leveraging formal verification," *Journal of Electronic Testing*, vol. 32, no. 5, pp. 569–586, 2016.

[19] C. K. Luk, R. Cohn, and R. Muth, "Pin: building customized program analysis tools with dynamic instrumentation," *ACM SIGPLAN Notices*, vol. 40, no. 6, pp. 190–200, 2005.

[20] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria," in *Proceedings of the International Conference on Software Engineering (ICSE)*, pp. 191–200, IEEE Computer Society Press, Sorrento, Italy, May 1994.

[21] 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," in *Proceedings of the IEEE international workshop on workload characterization. WWC-4*, pp. 3–14, IEEE, Austin, TX, USA, December 2001.

[22] S. Rehman, M. Shafique, P. V. Aceituno, F. Kriebel, J. J. Chen, and J. Henkel, "Leveraging variable function resilience for selective software reliability on unreliable hardware," in *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 1–6, IEEE, Grenoble, France, March 2013.

[23] C. Liu, J. Gu, Z. Yan, F. Zhuang, and Y. Wang, "SDC-causing error detection based on lightweight vulnerability prediction," in *Proceedings of the Asian Conference on Machine Learning(PMLR)*, pp. 1049–1064, WINC AICHI, Nagoya, Japan, November, 2019.

[24] Q. Lu, K. Pattabiraman, M. S. Gupta, and J. A. Rivers, "SDCTune: a model for predicting the SDC proneness of an application for configurable protection," in *Proceedings of the Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pp. 1–10, IEEE, Uttar Pradesh, India, October 2014.

[25] G. Li, Q. Lu, and K. Pattabiraman, “Fine-grained characterization of faults causing long latency crashes in programs,” in *Proceedings of the Dependable Systems and Networks(DSN)*, pp. 450–461, IEEE, Rio de Janeiro, Brazil, June 2015.

[26] A. H. Ibrahim, M. Abdelhalim, H. Hussein, and A. Fahmy, “Analysis of x86 instruction set usage for windows 7 applications,” in *Proceedings of the International Conference on Computer Technology and Development*, pp. 511–516, IEEE, Cairo, November 2010.

[27] J. Ma and Y. Wan, “Characterization of stack behavior under soft errors,” in *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition(DATE)*, pp. 1534–1539, IEEE, Lausanne, Switzerland, March 2017.

[28] N. Wang, M. Fertig, and S. Patel, “Y-branches: when you come to a fork in the road, take it,” in *Proceedings of the 12 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 56–66, IEEE, New Orleans, LA, USA, September 2003.