Light-Weight Techniques for Improving the Controllability and Efficiency of ISA-Level Fault Injection Tools

Behrooz Sangchoolie, Roger Johansson, Johan Karlsson
Department of Computer Science and Engineering
Chalmers University of Technology
Gothenburg, Sweden
Email: {behrooz.sangchoolie, roger, johan}@chalmers.se

***Abstract*—ISA-level fault injection, i.e. the injection of bit-flip faults in Instruction Set Architecture (ISA) registers and main memory words, is widely used for studying the impact of transient and intermittent hardware faults. ISA-level fault injection tools can be characterized by different properties such as repeatability, observability, reachability, intrusiveness, efficiency and controllability. This paper presents two pre-injection analysis techniques that improve *controllability* and *efficiency* using object code analysis. To improve controllability, we propose a technique for identifying the type of data that is stored in a potential target location. This allows the user to selectively direct fault injections to addresses, data and/or control information. Experimental results show that the data type of 84-100% of the targets locations in 8 programs were successfully identified by this technique. The second technique improves efficiency by fault pruning, i.e., by avoiding injection of faults that is known *a priori* to be detected by the tested system. This technique leverage the fact that faults in certain bits in the program counter and the stack pointer are always detected by machine exceptions. We show that exclusion of these bits from the fault space could significantly prune the fault space and reduce the time it takes to conduct a fault injection campaign.**

***Index Terms*—ISA-level fault injection, data type identification, fault space optimization, controllability, efficiency**

1. Introduction

Fault injection is a widely used method for testing and evaluating error handling mechanisms in computer systems. It is also used to estimate different dependability measures such as the well known *error coverage* [1] [2], which is defined as the conditional probability of system recovery given the existence of a failure. The inclusion of fault injection as a highly recommended assessment method in the ISO 26262 standard [3] for functional safety of road vehicles is an example of the increasing use and importance of fault injection in the embedded systems industry.

Several fault injection techniques have been proposed in the past decades; they can be categorized into simulation-based techniques [4] [5] [6] [7] [8], software-implemented techniques [9] [10] [11] [12], hardware-based techniques [13] [14] [15], and test port-based techniques [8] [12] [16] [17].

Fault injection techniques could be characterized based on different properties such as *repeatability*, the ability to repeat the injection of a specific fault and obtain the same results; *observability*, the ability to observe and measure the effects of an injected fault; *reachability*, the ability to reach possible fault locations in a processor; *intrusiveness*, the level of unintended impact on the temporal and spatial behavior of the target system; *efficiency*, the time and effort needed to conduct a fault injection campaign (a fault injection campaign is a set of fault injection experiments using the same fault model on a given workload); and *controllability*, the ability to control when and where a fault is injected [12] [18] [19].

In this paper we present two pre-injection analysis techniques that improve the controllability and efficiency of fault injection techniques. Our techniques rely on object code analysis, i.e., no source code is required.

1.1. Improving Controllability

The first part of the paper deals with the controllability property of fault injection techniques. The ability to control when and where a fault is injected could help us design efficient fault injection campaigns. This is already addressed by works such as [12] where faults are identified by time-location pairs according to a fault-free execution of a program. In this paper, we improve controllability by identifying the type of *data-items* that are potential fault injection candidates, prior to conducting fault injection experiments. Here data-item refers to the content of a register or memory word and the type of a data-item could be a *data variable*, *memory address*, or *control information*.

The motivation for the data type identification comes from prior work [20] [21] [22] where we learned that the outcome of a fault injection experiment is highly dependent on the type of data-items targeted. For example, injecting faults in address data-items are more likely to raise a hardware exception mechanism, thus resulting in a program crash, compared to faults injected in data variables.

978-1-5090-5652-1/17/$31.00

Knowing the type of different data-items, provides us with a better control over the selection of locations where faults should be injected into. This helps us design cost-efficient fault injection campaigns that only target sensitive data-items, as well as providing us with useful information about locations that need to be hardened by fault tolerance mechanisms. Moreover, one could also design cost-efficient fault tolerant mechanisms that are data-type-specific, suitable for tolerating faults in data-items with specific data types.

1.2. Improving Efficiency

The focus of the second part of this paper is on efficiency property of fault injection techniques. The efficiency property of fault injection techniques is also sometimes referred to as the cost of conducting fault campaigns [18], which is relatively high. This is due to the fact that fault injection, in general, is a time-consuming technique. One way of dealing with the high temporal cost of fault injection campaigns is by injecting faults only in locations that would most likely disturb the system, instead of random selection of fault locations. In fact, authors in [14] [16] show that often 80-90% of randomly injected faults are not even activated.

A fault placed in a register just before the register is written into or faults that are injected into unused memory locations are examples of faults with no possibility of activation. In most tools the location and the time for fault injection are chosen randomly from the complete fault-space [23], which is typically extremely large. The statistical implication of this is that the cost of obtaining appropriate confidence levels of the dependability measures becomes unnecessarily high.

One way of injecting faults in a more efficient way is to use Barbosa et al.'s pre-injection analysis [23] to avoid injecting errors into "*dead*"[1] registers or memory locations in order to decrease the time it takes to run a fault injection campaign. This analysis is called pre-injection analysis as it uses knowledge of program flow and resource usage to choose the location and time where faults should be injected, before any experiment is performed.

Fault injection tools could use the pre-injection analysis to implement *inject-on-read* ISA-level fault injection technique, that selects the time of the injection so that errors are only injected in a register or a memory word immediately before it is read. According to Barbosa et al. [23], the pre-injection analysis reduced the fault space two orders of magnitude for the registers and four to five orders of magnitude for the memory locations. Tools such as GOOFI-2 [12] and FAIL* [8] use the inject-on-read fault injection technique.

Even though the pre-injection analysis would considerably reduce the fault space, the fault-space is still quite large when targeting the complete execution of programs under test; which is why most fault injection tools only sample a subset of the fault space or employ different heuristic pruning methods [7] [24] to cover the complete fault space. Therefore, it is still desirable to come up with other improvements that could help us reduce the number of fault injection experiments when evaluating error handling mechanisms.

The second pre-injection analysis technique proposed in this paper improves efficiency by fault pruning, i.e., by avoiding injection of faults that is known *a priori* to be detected by the tested system. This technique leverage the fact that faults in certain bits in the program counter and the stack pointer are always detected by machine exceptions [21]. We show that exclusion of these bits from the fault space could significantly reduce the time it takes to conduct a fault injection campaign.

The remainder of this paper is organized as follows. In Section 2, we present GOOFI-2, the fault injection tool that we use to evaluate our pre-injection analysis techniques. In Section 3, we present the background and some of the motivations behind this work. Data type identification pre-injection analysis technique is presented in Section 4 and the fault space optimization pre-injection analysis technique is presented in Section 5. We validate our proposed techniques in Section 6. Finally, we provide conclusions and some future work in Section 7.

2. GOOFI-2 Fault Injection Tool

The GOOFI fault injection tool was first presented in [25] and then enhanced with support for more target systems and new injection techniques in its next version GOOFI-2 [12]. GOOFI-2 can be configured to conduct fault injection experiments using two fault injection techniques, namely test port-based and software-implemented (SWIFI). Both techniques are capable of emulating transient hardware faults affecting the registers and memory of a microcontroller. Such errors are emulated using single or multiple bit-flip errors.

GOOFI-2's test port-based technique uses the Nexus [26] port to inject errors into ISA (instruction set architecture) registers and memory segments of MPC565 and MPC5554, PowerPC-based microcontrollers from Freescale. Nexus is a standard on-chip debug interface, which provides read/write access to processors resources. Using the test port-based technique, GOOFI-2 conducts fault injection without altering the software of the target system. GOOFI-2 also implements two software-implemented (SWIFI) techniques. The first one places the fault injection code in exception-handling routines intended for debugging, while the second one injects faults by instrumenting the executable file with fault injection code before it is downloaded to the target system.

GOOFI-2 uses a pre-injection analysis [23] to implement an ISA-level fault injection technique known as *inject-on-read* that selects the time of the injection so that errors are only injected in a register or a memory word immediately before it is read. In other words, all faults targeting a specific bit of a given register or memory word, from the time the

1. A register or memory location is considered to be dead if it holds data that will never be read again.

register or the memory word is written into until it is read, are considered equivalent. Note that in order to obtain an accurate estimation of different dependability measures, it would be necessary to apply a weight factor corresponding to the number of faults in each equivalence class, as also addressed in previous works [20] [23] [27]. Inject-on-read corrupts the content of the source register (or memory word) of a machine instruction, emulating soft errors that occur in a data-item during the time it resides in an ISA register or a memory word.

The current version of GOOFI-2 is also equipped with another ISA-level fault injection technique known as *inject-on-write* [20], where using this techniques, we could also inject faults into the destination register or memory word of an instruction. The inject-on-write is well suited for emulating errors that propagate into a register or a memory word as a result of particle strikes in hardware resources within microprocessor such as ALUs, caches and internal pipeline registers.

We define a fault injection experiment to be the injection of one fault and the monitoring of its impact on the program. During each experiment, GOOFI-2 controls the program under test using the winIDEA development environment [28] in conjunction with the iC3000 debugger [29]. GOOFI-2 stores the acquired data of each experiment into a database, which can later on be used to classify the outcome of the experiments. A fault injection campaign, on the other hand, is a set of fault injection experiments using the same fault model on a given workload. And a workload is a program running with a given input.

In this paper, we present two pre-injection analysis techniques, that strengthen controllability and efficiency properties; and evaluate them using GOOFI-2. The new features added were all lessons learned from developing and using GOOFI-2 in the past few years [20] [21] [22] [30].

3. Background and Motivations

3.1. Data Type Identification

The motivation for the data type identification comes from prior work [20] [21] [22] where we learned that the outcome of a fault injection experiment is highly dependent on the type of data-item targeted. Here data-item refers to the content of a register or memory word, which could be a *data variable*, *memory address*, or *control information*. For example, injecting faults in address data-items, are more likely to raise a hardware exception mechanism compared to faults injected in data variables.

The first paper [20] compares two techniques for ISA-level fault injection, namely inject-on-read and inject-on-write. In addition, the paper compares two variants of inject-on-read, one where all faults are given the same weight and one where weight factors are used to reflect the time a data-item spends in a register or memory word. The paper shows significant differences in the results obtained with the three techniques and concludes that one of the main reasons for the differences observed is the distribution of data-items targeted by each technique.

The second paper [22] investigates the impact of compiler optimizations on the error sensitivity of programs. Error sensitivity is the probability that a silent data corruption (SDC) occurs in the programs output, given that a transient hardware error has occurred in a register or memory location. SDC is caused when the program under test terminates normally, but the output is erroneous and there is no indication of failure. In other words, SDC is an uncovered error. The results show that the level of optimization in which a program is compiled with could affect the program's error sensitivity. Type of data-items targeted plays a significant role in explaining these differences, especially since different GCC optimization flags significantly reduce the number of access to the memory, minimizing the percentage of injections in address data-items.

In [22] we investigate the impact of the source code implementation of a program on its error sensitivity. The paper shows significant variations in the error sensitivity among different implementations of a program. These variations are caused as a result of the varying characteristics of the implementations, e.g., in terms of number of dynamic instructions, number of memory accesses, type of data structures used in the program, etc. The latter (type of data structures) could have a major impact on the error sensitivity of programs. The paper further investigated this through a detailed analysis of the types of data-items used. The investigation is performed manually, by tracing the content of registers and memory segments targeted, which is very time-consuming.

When measuring the error sensitivity of a program, single bit-flip fault model has been a valid engineering approximation to mimic errors that originate from transistor-level faults or direct hits into the instruction set architecture registers and memory segments. However, ideally, the fault model should account for both single and multiple bit errors. There has been a number of studies [21] [31] comparing the commonly used single bit-flip model with the double bit-flip model, as one variation of multiple bit-flip model. Results of these studies show that the percentage of SDCs in the output of the programs under study, is marginally different for the two fault models. However, we may still need to target programs with other numbers of injections to improve the accuracy of the error sensitivity measure.

Conducting fault injection experiments for all cases of multiple bit-flips may be unrealistic as one could easily end up at error space explosion. However, one could predict or reason about the error sensitivity of programs targeted by multiple bit-flip errors by analyzing the data-item distribution as well as error propagation of programs. For example, injecting multiple faults in a program that has a significantly high portion of address data-items would most likely result in a lower percentage of silent data corruptions. This is because, multiple injections in this program increase the probability of a hardware exception to be raised.

The significant impact of different data types on the error sensitivity of a program as well as the cumbersome

process of manual identification of data-item types have motivated us to improve GOOFI-2's pre-injection analysis by implementing an automatic data-item type identification module.

3.2. Fault Space Optimization

As mentioned in Section 2, GOOFI-2 uses a pre-injection analysis [23] that selects the time of the injection so that errors are only injected in a register or a memory word immediately before it is read. Even though the pre-injection analysis would considerably reduce the fault space, the fault-space is still quite large when targeting the complete execution of programs under test. Therefore, it is still desirable to come up with other fault space optimizations that could help us reduce the number of fault injection experiments when evaluating error handling mechanisms.

In [21], we present detailed statistics about the error sensitivity of different target registers and memory locations, including bit positions within registers and memory words. We observed that the error sensitivity varies significantly between different bit positions. An important observation in this study is that the impact of injecting errors in certain bit positions could be identified *a priori*. Having this information, the fault injection space could be reduced (optimized) by removing these locations from the fault injection space, while including their results in error sensitivity calculation. According to the results presented in [21], potential target locations that could contribute to the optimization of fault space are:

- *Program counter register.* Every error injected into bits 17 to 32 is detected by hardware exceptions. This is due to the fact that faults injected into the program counter register, that cause the register to point to somewhere outside the scope of a program (text segment), will be eventually detected by hardware exceptions. The first (least significant) bit that is fully covered by hardware exceptions is highly dependent on the program size, which could be automatically measured in the pre-injection analysis phase.
- *Stack pointer register.* Every error injected in bits 17 to 22 is detected by hardware exceptions. This is because faults injected in the stack pointer register, that causes the stack pointer to point to somewhere with a lower address than the lowest addressable area of the SRAM (static random access memory) section, triggers hardware exceptions, such as *Machine Check Exception (MCE)*. Note that the stack pointer always contains an address to the SRAM area.

Even though the above two fault space optimizations are program/platform dependent, they can be automatically integrated into any program or platform. It is also important to note that, general purpose registers and memory words could also hold address data-items that point to somewhere in the text or the SRAM segments. Therefore, one needs to first perform data-item type identification to connect each target location to a specific data-item and then use that information to reduce the size of the fault space.

TABLE 1. FAULT INJECTION LOCATIONS

Instruction Set Architecture Registers (ISA registers)	**Memory segments**
General purpose registers (GPR)	Stack
Floating point registers (FPR)	Data
Program counter register (PCR)	Sdata
Condition register (CR)	Bss
Link register (LR)	Sbss
Integer Exception Register (XER)	

4. Data Type Identification

In this section we present data type identification, which is a pre-injection analysis technique that offers more controllability over the selection of target data-items in a fault injection campaign. Note that a data-item could be a *data variable*, *memory address*, or *control information*.

We implement the data type identification on GOOFI-2, which defines faults as time-location pairs according to a fault-free execution of a workload. The location could be any register or memory segment, also known as *target locations*, that is used in the assembly code representation of a target program compiled with the GCC compiler. GOOFI-2 normally selects its targets from locations specified in Table 1.

The type of data-items stored in some of the target locations could be identified without the need to do any type of analysis. For example, floating point registers (FPR) always hold data variables, whereas the program counter register (PC) always holds a memory address. However, this is not the same case for some of the other target locations such as general purpose registers (GPR) and the stack segment. In fact these target locations could hold a data variable as well as a memory address.

The content of general purpose registers and stack segment words need to be analyzed to be able to identify the type of data-items they hold. Transient hardware faults in these target locations are very likely to have an impact on the program execution as these locations are frequently accessed throughout the program's execution.

Table 2 shows a detailed list of data-item types that can be identified by GOOFI-2 using the data type identification presented in this section. A fault injection campaign can be created selecting any combination of these data-items. This way, faults will only be injected in the data-items selected. The data-item could be stored in any of the target locations specified in Table 1. For example, a text segment address could be stored in the program counter register, a general purpose register or a memory segment. Throughout the rest of this section, we present our data type identification technique that can be used to automatically map a location to a data-item.

TABLE 2. DATA-ITEM TYPES IDENTIFIED BY GOOFI-2 USING THE DATA TYPE OPTIMIZATION TECHNIQUE PRESENTED IN SECTION 4

Data variable	Control information
Text segment address	Stack segment address
Data segment address	Sdata segment address
Bss segment address	Sbss segment address
Unclassified address	Unclassified data-item

4.1. Target-Specific Data Type Identification

In this section, we present the techniques that one could use to classify the type of different target locations using information of the targeted processor, such as its addressing modes. Even though we implement these techniques on object codes generated for PowerPC architecture (PPC), our data type identification scheme can be easily generalized for arbitrary architectures.

4.1.1. Instruction-Based Data Type Identification. The type of data-items stored in some of the target locations could be identified knowing the type of machine instruction used to access those locations. Fig. 1 illustrates three examples of this type of data type identification.

Fig. 1(a) shows a sample assembly code where *r9* can be classified as an Address data-item. This is due to the fact that the `lbz` instruction loads a byte from the memory that is addressed by *r9+4*. This means that *r9* register is definitely an Address data-item. The same justification could be used for source registers of many more instructions, such as `lbzu`, `stb`, `stw`, etc. We could also check the content of this register to have a more precise data-item classification, i.e., finding the memory segment in which this address points to.

The two source general purpose registers illustrated at address `2180` in Fig. 1(b) (*r0* and *r9*) could also be identified as Address data-items. This is because the sum of the values stored in these registers would generate the address in which the `lbzx` instruction uses when reading the memory. In other words, both of these registers have an impact on the effective address generated by the `lbzx` instruction, which is why they are both classified as Address data-items. The same justification could be used for source registers of many more instructions, such as `lbzux`, `lhzx`, `stbx`, `stwx`, etc.

There are also many instructions that only deal with floating-point registers, see Fig. 1(c). The type of data-items stored in registers used by these instructions could easily be classified as Data variable. Examples of such instructions are `fadd` (floating-point addition), `fdiv` (floating-point division), etc.

4.1.2. Location-Based Data Type Identification. Registers and memory segments used by an instruction could also reveal information about the type of data-item they hold. In other words, there are certain locations that always hold the same type of data-item, irrespective of the type of machine instruction. Table 3 shows examples of these locations along with the data-items they hold.

(a)
```
217c:  lwz     r9, 24(r31)
2180:  lbz     r0, 4(r9)
2184:  clrlwi  r0, r0, 24
```

(b)
```
217c:  addi    r9, r9, 286
2180:  lbzx    r0, r9, r0
2184:  clrlwi  r0, r0, 24
```

(c)
```
223c:  lfd     f0, 100(r9)
2240:  fmul    f0, f1, f0
2244:  fadd    f1, f1, f0
```

Figure 1. Three sample assembly code snippets showing examples of instructions that can be used to identify the type of data-items stored in (a) *r9*, (b) *r0* and *r9*, and (c) *f0* and *f1*

TABLE 3. FAULT INJECTION LOCATIONS THAT ALWAYS HOLD THE SAME TYPE OF DATA-ITEM

Target Fault Location	**Data-Item**
Stack pointer register (r1)	Stack segment address
Floating point registers (FPR)	Data variable
Program counter register (PCR)	Text segment address
Condition register (CR)	Control information
Link register (LR)	Text segment address
Data & Sdata memory segments	Data variable

(a)
```
2214:  lwz     r0, 4(r11)
2218:  mtlr    r0
221c:  lwz     r31, -4(r11)
```

(b)
```
21f0:  addi    r3, r9, 8916
21f4:  mr      r1, r2
21f8:  bl      2278
```

Figure 2. Two assembly code snippets showing examples of instructions that can be used to identify the type of data-items stored in (a) *r0* and (b) *r2* using our prior knowledge about the data type of link register (LR), holding the old instruction pointer, and stack pointer register (r1).

One could also combine the knowledge we have from the type of machine instruction with the locations used by that instruction to classify more target locations. We illustrate two examples of this type of data-item identification in Fig. 2.

Fig. 2(a) shows a sample assembly code where *r0* can be classified as a Text segment address data-item. This is because we already know that the `mtlr` instruction places a general purpose register into the link register (LR), which always points to somewhere in the Text segment, as this register can be thought of as the old instruction pointer. Thus, the data type of the source register (*r0*) should be Text segment address.

Fig 2(b) shows a sample assembly code where *r2* could be classified as a Stack segment address data-item. This is due to the fact that an `mr` instruction places the content of its source register (in this case *r2*) into its destination register (in this case *r1*). As the *r1* register is the stack pointer register, always holding a Stack address data-item, *r2* register should also hold a Stack address data-item.

```
21f4:  mr    r1, r2
21f8:  addi  r4, r2, 8
21fc:  mr    r5, r4
2200:  lwz   r5, 8(r5)
2204:  addi  r6, r5, 128
2208:  stw   r2, 8(r6)
```

Figure 3. Sample assembly code snippet showing how a location's backward/forward chains could be used to identify the type of a data-item.

Using target-specific data type identification techniques such as the ones presented in Section 4.1.1 and 4.1.2, one could identify the data-item type of many locations. Note that the sample data type identifications presented in these sections are just a subset of target-specific techniques that could be used to identify data-item types. However, there are two limitations concerning these techniques. The first one corresponds to the fact that these techniques do not take into account the knowledge they have gained about the data type of locations they have classified up to the current instruction at run-time. The second limitation deals with the fact that both of these techniques are tightly bounded by a certain types of machine instructions and target locations.

In the remaining of this section, we present two enhancements that could help us overcome the addressed limitations and strengthen the data type identification. The first enhancement (see Section 4.2) keeps track of the data type of all classified data-items hoping that they can be used to classify data-items used by future instructions. In other words, using this enhancement, we can make use of the type of data-items in backward instruction chain of all instruction. The second enhancement (see Section 4.3), on the other hand, looks into future instructions to see if a certain data-item can be classified using instructions that are in the forward instruction chain of the current instruction.

4.2. Enhancement I: Backward-Chain-Based Data Type Identification

Fig. 3 presents the idea behind backward-chain-based data type identification. Here we can see that the *r2* register at address `21f4` could be classified as Stack address data-item according to the technique presented in Section 4.1.2. However, this technique does not use the knowledge we gained from this classification when classifying register *r4* at address `21f8`. In other words, one could classify *r4* as Stack address data-item as *r2* register has not been updated since the execution of the `mr` instruction at address `21f4`.

Fig. 4 shows the data type identification technique after applying the first enhancement, where the ITERATE_OVER_INSTRUCTION_TRACE(*objectCode*) procedure at line 1 is called to identify the type of all data-items stored in the source target registers and memory words of an *objectCode*.

Backward-chain-based data type identification uses the knowledge we gained from classifying the data-items up to the current instruction to see whether they could be used to classify the operands of the current instruction. To do this,

```
1:  procedure ITERATE_OVER_INSTRUCTION_TRACE(objectCode)
2:      global lookup_Table[] ← NULL
3:      global srcList, dstOperand
4:      for all inst in objectCode.instructionTrace do
5:          srcList = inst.sourceOperands
6:          dstOperand = inst.destinationOperand
7:          ITERATE_OVER_SRCLIST(inst)
8:      end for
9:  end procedure
10:
11: procedure ITERATE_OVER_SRCLIST(inst)
12:     invalidate_dstOperand = TRUE
13:     for all srcOperand in srcList do
14:         if lookup_Table[srcOperand] != NULL then
15:             dataType = lookup_Table[srcOperand]
16:         else
17:             dataType = TARGET_SPECIFIC(inst, srcOperand)
18:         end if
19:         lookup_Table[srcOperand] = dataType
20:         if dataType != NULL then
21:             if POSSIBLE_TO_CLASSIFY_DSTOPERAND (inst) then
22:                 lookup_Table[dstOperand] = dataType
23:                 invalidate_dstOperand == FALSE
24:             end if
25:         end if
26:         UPDATE_DATABASE(inst, srcOperand, dataType)
27:     end for
28:     if invalidate_dstOperand == TRUE then
29:         lookup_Table[dstOperand] = NULL
30:     end if
31: end procedure
```

Figure 4. Data type identification after applying the 1st enhancement.

we maintain the data type of each target location throughout the execution of a program, using a *lookup_Table* defined at line 2, so that if needed in future instructions, we could use it to identify the type of a data-item (see line 15). In case we do not already know the data type of the data-item (see line 14), we use the target-specific technique presented in Section 4.1 to identify the data type of the data-item (see line 17).

The *lookup_Table* maintains the data types of both source and destination operands (refer to lines 19 and 22, respectively). Fig. 3 can be used to explain this. Here, the *r2* register at address `21f4`, can be classified as Stack address data-item according to the target-specific data type identification technique presented in Section 4.1 that is called at line 17 in Fig. 4. This register's classification can be used in the next instruction (instruction at address `21f8`) to classify *r4* register, as *r2* is the source register in this instruction and the destination register of this instruction (*r4*) should also have the same data type. This way, we can maintain the data type of both of these registers hoping that they could be used when classifying future data-items. In fact, Fig. 3 shows that *r4* is read at address `21fc` allowing us to classify the destination register in that instruction (*r5*).

The data type of the destination operand in the *lookup_Table* is invalidated at line 29 in case it cannot be classified by the source operand. This is because the data-item stored in the destination operand is updated after executing an instruction. Fig. 3 can also be used to explain this. As mentioned before, the *r5* register can be classified

as Stack address data-item after executing the instruction at address `21fc`. This register is then used in the next instruction to load a data-item from the memory. Even though, we know the data type of this register before executing this instruction, its data type after the item is loaded is not known, given that we do not know the data type of the memory data-item. This means that *r5* register needs to be invalidated after executing this instruction. This also means that we are not able to use *r5* register at the next instruction to classify the destination register (*r6*). The check as to whether it is possible to use the data type of the source operand to classify the data type of the destination operand is done using the POSSIBLE_TO_CLASSIFY_DSTOPERAND ($inst$) procedure at line 21 in Fig 4.

Using the enhancement presented in this section, we can identify the type of many more data-items. This enhancement is very light-weight and cheap as we would only need to maintain an accurate list of target locations along with their data-item types throughout the pre-injection run of the program under test. However, there are still locations that cannot be classified. In the next section, we present the second enhancement that could further strengthen our proposed data-item identification scheme.

4.3. Enhancement II: Forward-Chain-Based Data Type Identification

The idea behind the forward-chain-based enhancement is to look into the forward-chain of a target location and see whether that could help us classify the location. We present the motivation behind using this enhancement in Fig. 3, where the target-specific data type identification technique was unable to classify the *r5* register at address `2204`. However, looking at the forward-chain of this register, we can classify this register as an Address data-item. This is because after executing the instruction at address `2204`, the *r6* register should have the same data type as *r5* and looking at *r6* register at instruction `2208`, we can see that this register should be classified as an Address data-item. Therefore, using the forward-chain of *r5* register, we can classify this register to hold an Address data-item.

As the focus of this technique is on the forward-chain of a target location, we could use the locations inside the forward chain of an instruction to perform the classification. This significantly helps us classify more locations as classifying any of the locations in the forward chain would lead to the classification of the target location under investigation.

Fig. 5 shows the data type identification after applying the second enhancement. We maintain a list of operands that if classified by the target-specific technique presented in Section 4.1, we could identify the data type of the data-item under investigation. This list is referred to as *vitalList*. Lines 42-53 check to see if any item in the *vitalList* can be classified using the target-specific technique.

The first operand added to the *vitalList* is the operand in which we would like to identify the type of its data-item (see line 31). We then update this list with destination operands of the instructions in the forward-chain at lines 56 and 61, depending on the type of instruction and whether it is possible to conclude that the data type of the source and destination operands should be the same.

The search for a data-item classification for a location using the forward-chain identification continues until *(i)* the last dynamic instruction of the program under test, to make sure that there is no data-item type inconsistencies; *(ii)* or until all operands in the *vitalList* are invalidated (see line 37), which means that we are unable to classify the location under investigation. The former (data-item type inconsistencies) occurs where a location could potentially be classified as multiple data-item types. For example, a location may be classified as a Data variable data-item using a certain instruction in the forward-chain, while it could also be classified as an Address data-item at another instruction. Even though we observed very few cases as such, we should make sure that we identify them and invalidate them (see lines 44-52).

5. Fault Space Optimization

In this section we present fault space optimization improving the efficiency of fault injection campaigns. The optimizations proposed in this section deal with two of the data-items identified in Section 4, namely Text segment address and Stack segment address. Here we use lessons learned from prior work [21] to reduce the number of fault injections needed when evaluating error handling mechanisms in computer systems. The idea behind this optimization comes from the fact that injecting faults in certain bits of the specified data-items would always raise an exception mechanism. These bits could be removed from the fault injection campaign, while still being included in the error sensitivity estimation of the program under test.

5.1. Fault Space Optimization for Text Segment Address Data-Items

Errors occurring in certain bits of locations holding a Text segment address data-item would always raise a hardware exception. These exceptions could be raised *(i)* by memory management units, in case the program under test attempts to access an illegal address *(ii)* or by attempting to execute instructions that are not implemented, in case the memory management units do not raise an illegal address exception. In both cases, these bits could be identified and removed from the fault injection space as we already know the failure mode classification of these locations. To perform the fault space optimization, we first identify the Text segment address data-items using the techniques presented in Section 4 and then we identify the bits that would always raise hardware exceptions in case targeted by errors.

Finding the bits that would always raise hardware exceptions, in case targeted by errors, is dependent on *(i)* the static size of program's under test *(ii)* and its base address in the Text segment. These values could be calculated from the program's object code. When a data-item is identified to be a

```
1: procedure ITERATE_OVER_INSTRUCTION_TRACE(objectCode)
2:     global lookup_Table[] ← NULL
3:     global srcList, dstOperand
4:     for all inst in objectCode.instructionTrace do
5:         srcList = inst.sourceOperands
6:         dstOperand = inst.destinationOperand
7:         ITERATE_OVER_SRCLIST(inst)
8:     end for
9: end procedure
10:
11: procedure ITERATE_OVER_SRCLIST(inst)
12:     invalidate_dstOperand = TRUE
13:     for all srcOperand in srcList do
14:         dataType = SEARCH_FORWARD_CHAIN(inst, srcOperand)
15:         lookup_Table[srcOperand] = dataType
16:         if dataType != NULL then
17:             if POSSIBLE_TO_CLASSIFY_DSTOPERAND (inst) then
18:                 lookup_Table[dstOperand] = dataType
19:                 invalidate_dstOperand == FALSE
20:             end if
21:         end if
22:         UPDATE_DATABASE(inst, srcOperand, dataType)
23:     end for
24:     if invalidate_dstOperand == TRUE then
25:         lookup_Table[dstOperand] = NULL
26:     end if
27: end procedure
28:
29: procedure SEARCH_FORWARD_CHAIN(inst, srcOperand)
30:     dataType = NULL
31:     vitalList.add(srcOperand)
32:     if lookup_Table[srcOperand] != NULL then
33:         dataType = lookup_Table[srcOperand]
34:     end if
35:     for all fwd_inst in FORWARD_CHAIN(inst) do
36:         invalidate_dstOperand = TRUE
37:         if vitalList.isEmpty then
38:             return dataType
39:         end if
40:         srcListTmp = fwd_inst.sourceOperands
41:         dstOperandTmp = fwd_inst.destinationOperand
42:         for all item in vitalList do
43:             dataTypeTmp = TARGET_SPECIFIC(fwd_inst, item)
44:             if dataTypeTmp != NULL then
45:                 if dataType != NULL then
46:                     if dataType != dataTypeTmp then
47:                         return NULL
48:                     end if
49:                 else
50:                     dataType = dataTypeTmp
51:                 end if
52:             end if
53:         end for
54:         if POSSIBLE_TO_CLASSIFY_DSTOPERAND (fwd_inst) then
55:             if vitalList has an operand in srcListTmp then
56:                 vitalList.add(dstOperandTmp)
57:                 invalidate_dstOperand == FALSE
58:             end if
59:         end if
60:         if invalidate_dstOperand == TRUE then
61:             vitalList.Remove(dstOperandTmp)
62:         end if
63:     end for
64:     return dataType
65: end procedure
```

Figure 5. Data type identification after applying the 2nd enhancement.

Text segment address, the fault space optimization technique evaluates all of its bits during the pre-injection analysis, to see if injecting faults in these bits would result in an address beyond the program's addressable area, in which case, the fault space could be pruned by excluding these bits from the fault space and classifying their failure modes to detected by hardware exception (aka crash).

5.2. Fault Space Optimization for Stack Segment Address Data-Items

Faults injected in certain bits (17 to 22) of the Stack segment address data-items would also always raise hardware exceptions. This can be explained by studying the internal memory block of our target platform.

The internal memory is 4Mbytes that resides in `0x0000 0000` to `0x003F FFFF` address block and the SRAM is located from the address `0x003F 7000` to `0x003F FFFF`. The stack pointer always contains an address to the SRAM area. Therefore, all the addresses referring to the SRAM contain 1 in bits 17 to 22. Flipping any of these bits from 1 to 0 will result in an address smaller than the SRAM base address which triggers hardware exceptions.

We also expected accesses above the SRAM upper bound to be detected by hardware exceptions. However, the errors in bits 23 to 32 are not always detected by hardware exceptions. The reason for this behavior is likely to be related to implementation of the address decoding logic on the processor board that we use for our experiments. However, the bits that always raise a hardware exception could also be easily calculated for different hardware platforms.

Finding the bits that would always raise hardware exceptions, in case targeted by errors, is dependent on SRAM addressable area. When a data-item is identified to be a Stack segment address (using the techniques presented in Section 4), the fault space optimization technique prunes the fault space by excluding bits 17 to 22 from the fault space and classifying their failure modes to detected by hardware exception (aka crash).

6. Validation

To validate the techniques presented in Section 4 and Section 5, we conduct a set of fault injection experiments on eight programs selected from the MiBench benchmark suite [32]. We select a diverse set of programs with respect to implementation, code size, input type/size. These programs include five different implementations of the bit count algorithm, in addition to binary-to-integer convertor, square root calculator and the cubic equation calculator.

Table 4 shows the number and percentage of data-items identified for different programs under test according to the technique presented in Section 4. Here we can see that using our technique and the proposed enhancements we can successfully identify the type of all data-items in the five bit count implementations, 98% of the data-items for binary-to-integer convertor, 86% of the data-items for the square root calculator and 84% of the data-items for the cubic equation calculator. Our technique is very powerful in identifying

TABLE 4. NUMBER AND PERCENTAGE OF DIFFERENT TYPES OF DATA-ITEMS IDENTIFIED.

	BitCnt1	BitCnt2	BitCnt3	BitCnt4	BitCnt5	BinToInt	Isqrt	Cubic
Text segment address	93 (38%)	61 (42%)	40 (42%)	63 (40%)	153 (46%)	1311 (44%)	904 (37%)	5676 (41%)
Stack segment address	55 (22%)	26 (18%)	16 (17%)	48 (30%)	85 (26%)	415 (14%)	499 (21%)	378 (3%)
Data segment address	0 (0%)	0 (0%)	23 (24%)	21 (13%)	50 (15%)	236 (8%)	0 (0%)	0 (0%)
Sdata segment address	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)
Bss segment address	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	6 (<1%)
Sbss segment address	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	1 (<1%)
Unclassified address	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	140 (5%)	0 (0%)	265 (2%)
Control information	16 (6%)	0 (0%)	0 (0%)	0 (0%)	8 (2%)	242 (8%)	130 (5%)	1348 (9%)
Data variable	83 (34%)	58 (40%)	16 (17%)	26 (17%)	35 (11%)	559 (19%)	551 (23%)	3999 (29%)
Unclassified data-item	0 (0%)	0 (0%)	0 (0%)	0 (0%)	0 (0%)	63 (2%)	346 (14%)	2211 (16%)

Address segment data-items and almost all the data-items that could not be classified (Unclassified data-items) belong to the Data variable data-item type.

Table 4 also shows that the majority of data-items accessed are address data-items (on average 68% over all programs). As mentioned before, Address data-items are significantly less sensitive than Data variable data-items, as injecting faults in Address data-items are mostly detected by hardware exception mechanisms. Therefore, in case one wants to remove these data-items from the fault space, only three of the data-items presented in Table 4 need to be selected in a fault injection campaign, namely Data variable, Control information, and Unclassified data-items. This can significantly reduce the number of experiments needed to find sensitive parts of the program under test. Note that, ideally, every single data-item needs to be targeted with faults. However, one could target all data-items after hardening the parts of the program that are proved to be significantly more sensitive to faults.

Table 5 shows the number of bits optimized in the fault space of Text address and Stack address data-items according the techniques presented in Section 5. These are bits that we can exclude from our fault injection campaign, as we already know that faults injected in these bits would raise a hardware exception. According to the result of our fault injection campaigns, on average, these bits constitute of around 25% of the unoptimized fault space, meaning that using the fault space optimization technique, we could, on average, prune 25% of the fault space.

Table 5 also shows the time that we could save by excluding the bits identified by our fault space optimization technique from a fault injection campaign. Here, we assume that conducting 1000 experiments would take around one hour; thus using our fault space optimization technique and the set of programs presented in this table, we could save 1 to 105 hours of fault injection time, corresponding to the smallest (BitCnt3) and biggest (Cubic) programs under test, respectively. It is important to mention that the time that it takes to perform the analyses presented in Section 4 and Section 5 is only in the orders of a few seconds to a few minutes, which is significantly smaller than the time that could be saved by only targeting sensitive data-items and optimized bits.

TABLE 5. NUMBER OF BITS OPTIMIZED IN THE FAULT INJECTION SPACE OF TEXT ADDRESS AND STACK ADDRESS DATA-ITEMS ALONG WITH THE TIME THAT WE SAVE BY EXCLUDING THESE BITS.

	BitCnt1	BitCnt2	BitCnt3	BitCnt4	BitCnt5	BinToInt	Isqrt	Cubic
Text segment address	2,084	1,380	876	1,391	3,488	29,195	20,556	102,756
Stack segment address	336	162	102	294	534	2,622	3,000	2,172
Sum	2,420	1,542	978	1,685	4,022	31,817	23,556	104,928
Time saved (hour)	2.5	1.5	1	1.6	4	31	23.5	105

7. Conclusions and Future Work

The outcome of a fault injection experiment is highly dependent on the type of data-item targeted. For example, data variable data-items are significantly more sensitive than Address data-items, as injecting faults in these data-items are more likely to result in a silent data corruption compared to when an Address data-item is targeted. Thus, it would be beneficial if one could identify the type of different data-items used in a program prior to performing fault injection experiments.

In the first part of this paper, we present a pre-injection analysis technique that improves the controllability property of ISA-level fault injection techniques by identifying the type of different data-items used in a program prior to performing fault injection experiments. We show that using this technique, we can successfully identify the type of data-items in 84-100% of target locations. This allows us to have a better control over the selection of sensitive fault injection locations. As part of our future work, we would like to improve our data-item identification technique by implementing additional techniques that could help us identify the type of data-items in all target locations.

Injecting faults in certain bits of specific registers and memory segments would always raise a hardware exception. Therefore, in the second part of this paper, we suggest another pre-injection analysis technique that could automatically identify these bits and exclude them from the fault space. We show that exclusion of these bits from the fault space could significantly prune the fault space and reduce the time it takes to conduct a fault injection campaign.

Note that the techniques presented in this paper rely on object code analysis, i.e., no source code is required. Even though in this paper we implement these techniques on object codes generated for PowerPC architecture (PPC), it should be straightforward to generalize our techniques for arbitrary architectures, which is also one of the directions of our future work.

References

[1] W. G. Bouricius, W. C. Carter, P. R. Schneider, "Reliability modeling techniques for self-repairing computer systems," in Proc. of the 1969 24th National Conf., pp. 295-309, 1969.

[2] T. F. Arnold, "The Concept of Coverage and its Effect on the Reliability Model of Repairable Systems," in IEEE Transactions Computers, vol. 22, no. 3, pp. 251-254, March 1973.

[3] ISO 26262:2011, Road vehicles Functional safety.

[4] E. Czeck, D. Siewiorek, "Observations on the Effects of Fault Manifestation as a Function of Workload," IEEE Transactions on Computers, vol. 41, no. 5, pp. 559-566, May 1992.

[5] K. Goswami, R. Iyer, "Simulation of Software Behavior Under Hardware Faults," in Proc. Of 23rd Int. Symp. on Fault- Tolerant Computing (FTCS-23), Toulouse, France, pp. 218-227, June 1993.

[6] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, J. Karlsson, "Fault Injection into VHDL Models: the MEFISTO Tool," in 24th Int. Symp. on Fault-Tolerant Computing (FTCS-24), Austin, TX, USA, pp. 66-75, June 1994.

[7] S. K. Sastry Hari, S. V. Adve, H. Naeimi, and P. Ramachandran, "Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults," in Proc. of the 17th Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS 12). New York, NY, USA: ACM Press, 2012, pp. 123-134.

[8] H. Schirmeier, M. Hoffmann, C. Dietrich, M. Lenz, D. Lohmann and O. Spinczyk, "FAIL*: An Open and Versatile Fault-Injection Framework for the Assessment of Software-Implemented Hardware Fault Tolerance," Dependable Computing Conference (EDCC), 2015 Eleventh European, Paris, 2015, pp. 245-255.

[9] W. Kao, R. K. Iyer, T. D. Tang, "FINE: A Fault and Monitoring Environment for Tracing the UNIX System Behavior under Faults," IEEE Transactions on Software Engineering, vol. 19, no. 11, pp. 1105-1118, November 1993.

[10] G. Kanawati, N. Kanawati and J. Abraham, "FERRARI: a Tool for the Validation of System Dependability Properties," in 22nd Int. Symp. on Fault-Tolerant Computing (FTCS-22), Boston, MA, USA, pp. 336-344, July 1992.

[11] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Barton, D. Rancey, A. Robinson, and T. Lin, "FIAT-fault injection based automated testing environment," in 18th Int. Symp. on Fault-Tolerant Computing (FTCS-18), Tokyo, Japan, pp. 102-107, June 1988.

[12] D. Skarin, R. Barbosa, J. Karlsson, "GOOFI-2: A Tool for Experimental Dependability Assessment," in IEEE/IFIP Int. Conf. on Dependable Systems and Networks (DSN), Chicago, IL, USA, pp. 557-562, June 2010.

[13] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, D. Powell, "Fault injection for Dependability Validation: A Methodology and Some Applications," IEEE Transactions on Software Engineering, vol. 16, no. 2, pp. 166-182, February 1990.

[14] H. Madeira, J.G. Silva, "Experimental Evaluation of the Fail Silent Behavior in Computers Without Error Masking," in Proc. of 24th Int. Symp. on Fault-Tolerant Computing (FTCS- 24), Austin, TX, USA, pp. 350-359, June 1994.

[15] G. Miremadi, J. Karlsson, U. Gunneflo, J. Torin, "Two Software Techniques for On-line Error Detection," in 22nd Int. Symp. on Fault-Tolerant Computing (FTCS-22), Boston, MA, USA, pp. 328-335, July 1992.

[16] P . Yuste, J. Ruiz, L. Lemus, P . Gil, "Non-intrusive software-implemented fault injection in embedded systems," in Proc. of 1st Latin-American Symp. on Dependable Computing (lADC 2003), Sao Paulo, Brazil, pp. 23-38, October 2003.

[17] A. V. Fidalgo, G. R. Alves, J. M. Ferreira, "Real time fault injection using a modified debugging infrastructure," in Proc. of 12th IEEE Int. On-Line Testing Symp. (IOLTS'06), Lake Como, Italy, pp. 242-250, July 2006.

[18] M. C. Hsueh, T. K. Tsai and R. K. Iyer, "Fault injection techniques and tools," in Computer, vol. 30, no. 4, pp. 75-82, April 1997.

[19] R. Svenningsson, J. Vinter, H. Eriksson, M. Trngren, "MODIFI: A MODel-Implemented Fault Injection Tool," in Proc. of 29th Int. Conf. on Computer Safety, Reliability, and Security (SAFECOMP), Vienna, Austria, pp. 210-222, 2010.

[20] B. Sangchoolie, F. Ayatolahi, R. Johansson, J. Karlsson, "A Comparison of Inject-on-Read and Inject-on-Write in ISA-Level Fault Injection," Dependable Computing Conference (EDCC), 2015 Eleventh European, Paris, 2015, pp. 178-189.

[21] F. Ayatolahi, B. Sangchoolie, R. Johansson, J. Karlsson, "A Study of the Impact of Single Bit-Flip and Double Bit-Flip Errors on Program Execution," in Proc. of 32nd Int. Conf. on Computer Safety, Reliability, and Security (SAFECOMP), Toulouse, France, pp. 265-276, September 2013.

[22] B. Sangchoolie, F. Ayatolahi, R. Johansson, J. Karlsson, "A Study of the Impact of Bit-flip Errors on Programs Compiled with Different Optimization Levels," in Proc. of 10th European Dependable Computing Conference (EDCC), Newcastle upon Tyne, UK, pp. 146-157, May 2014.

[23] R. Barbosa, J. Vinter, P. Folkesson, J. Karlsson, "Assembly- Level Pre-Injection Analysis for Improving Fault Injection Efficiency," in Proc. of 5th European Conf. on Dependable Computing (EDCC), Budapest, Hungary, pp. 246-262, April 2005.

[24] J. Li and Q. Tan, "SmartInjector: Exploiting intelligent fault injection for SDC rate analysis," 2013 IEEE Int. Symp. on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS), New York City, NY, 2013, pp. 236-242.

[25] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson. GOOFI: Generic object-oriented fault injection tool. In Proc. Int. Conf. on Dependable Systems and Networks (DSN 2001), pages 83-88, July 2001.

[26] "Nexus 5001TM Forum," IEEE-ISTO, 1999. [Online]. Available: http://www.nexus5001.org. [Accessed November 2016].

[27] H. Schirmeier, C. Borchert and O. Spinczyk, "Avoiding Pitfalls in Fault-Injection Based Comparison of Program Susceptibility to Soft Errors," 2015 45th Annual IEEE/IFIP Int. Conf. on Dependable Systems and Networks, Rio de Janeiro, 2015, pp. 319-330.

[28] "winIDEA iSystems Integrated Development Environment,". [Online]. Available: http://www.isystem.com/products/software/winidea. [Accessed November 2016].

[29] "iC3000 debugger,". [Online]. Available: http://www.isystem.com/products/11-products/89-ic3000-activeemulator. [Accessed November 2016].

[30] D. Di Leo, F. Ayatolahi, B. Sangchoolie, J. Karlsson and R. Johansson, "On the Impact of Hardware Faults An Investigation of the Relationship between Workload Inputs and Failure Mode Distributions," in Proc. of the 31st Int. Conf. on Computer Safety, Reliability, and Security (SAFECOMP), Magdeburg, Germany, 2012.

[31] Q. Lu, M. Farahani, J. Wei, A. Thomas and K. Pattabiraman, "LLFI: An Intermediate Code-Level Fault Injection Tool for Hardware Faults," Software Quality, Reliability and Security (QRS), 2015 IEEE Int. Conf. on, Vancouver, BC, 2015, pp. 11-16.

[32] "MiBench Version 1.0," University of Michigan, [Online]. Available: http://www.eecs.umich.edu/mibench/. [Accessed November 2016].