Bernhard Spitzer, BSc

Automated Software Diversity with Unsound Randomization

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Dipl.-Ing., Andrea Höller, BSc

Institute for Technical Informatics

Dipl.-Ing. Dr. techn., Christian Kreiner

Graz, November 2015

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master‘s thesis dissertation.

Date

Signature

Kurzfassung

Hardware Komponenten in eingebetteten Systemen werden zunehmend fehleranfällig wegen der immer kleineren Bauweise und der Verwendung von handelsüblichen Standard-Hardware, die eingesetzt wird um mit den stetig steigenden Anforderungen, die an die Rechenleistung gestellt werden, Schritt halten zu können.

Redundante Systeme ermöglichen es Hardware-Fehler zu tolerieren. Es können jedoch nur solche Fehler gefunden werden, bei denen der Output von zwei oder mehr Instanzen unterschiedlich ist. Haben Hardware-Fehler eine gemeinsame Ursache, wie zum Beispiel Fehler in geteilten Resourcen, wird Diversität benötigt, damit das redundante System den Fehler erkennen kann.

In dieser Arbeit wird die Robustheit von Software gegenüber Mutationen verwendet um automatische Software Diversität zu erzeugen indem kleine und zufällige Transformationen in einem iterativen Prozess angewendet werden um voll funktionsfähige Programm-Varianten zu erhalten. Diese Transformationen werden auf Quellcode und auf Assembler Code angewendet. Es wird ein vierstufiges System zur Erzeugung von Software-Diversität implementiert, damit dieser Ansatz mit Diverse-Compiling Methoden, bezüglich der Fähigkeit Hardware Fehler zu erkennen, verglichen und erweitert werden kann.

Dieses vierstufige System wird außerdem verwendet um lauffähige Programme auf fehlerhafter Hardware zu erzeugen. Die häufigsten Hardware-Fehler eines ARM9 Prozessors werden mit einem QEMU basierten Fehler Injektions Framework simuliert.

Es werden 15 verschiedene Testprogramme von den Kategorien Industrielle Steuerung, Netzwerk und Telekommunikation verwendet um verschiedene Ansätze zu vergleichen. Die Ergebnisse zeigen, dass über 90 % aller eingefügten Instruktions Dekodierer und RAM Fehler, sowie über 47 % aller eingefügten Register Fehler mithilfe des vierstufigen Systems zur Erzeugung von Diversität maskiert werden können. Software-Mutationen hatten keinen Einfluss auf die Maskierung von Hardware-Fehlern.

Die Erkennungsrate von Hardware Fehlern konnte durch Software-Mutationen und der Verwendung eines Compiler Register-Flags gesteigert werden.

Abstract

Hardware components in embedded systems are becoming increasingly vulnerable because of technology scaling and the usage of commercial off-the-shelf hardware, which is caused by the necessity to put up with the computing performance requirements. Redundancy can be used to tolerate hardware faults. However, since the voter in redundant systems only detects faults if the outputs of two or more instances do not match, diversity is needed to detect common-cause failures, such as faults in shared resources. In this thesis, software-mutational-robustness is used to automatically introduce unsound software diversity by applying small and randomized program transformations to get neutral networks of fully functional program variants. Source code and assembler code are mutated as part of a diversity chain. This approach is combined with diverse compiling and register avoidance strategies giving a four level diversification chain. Each step of this diversification chain is optional to make possible the comparison of different diversification strategies concerning hardware fault detection.

Furthermore, the diversification chain is used to recover from permanent hardware faults, such as CPU register faults, instruction decoder faults and RAM faults.

In order to simulate the most common hardware errors for an ARM9 processor, a QEMU based fault injection framework is used.

The efficiency of the different strategies is quantified for 15 different test programs out of the categories Automotive and Industrial Control, Network and Telecommunications. The results concerning recovery showed that over 90 % of all introduced instruction decoder faults and RAM faults and over 47 % of all introduced register cell faults which lead to an error on the reference binary can be masked by using diverse compiling in combination with the fixed register flag. It was not possible to improve the recovery rates using unsound software diversity.

Furthermore, the results showed that source mutations and assembler mutations can improve the fault detection rate when the approach is combined with diverse compiling.

Danksagung

Diese Diplomarbeit wurde im Jahr 2015 am Institut für Technische Informatik an der Technischen Universität Graz durchgeführt.

Ich möchte mich bei meinen beiden Betreuern Andrea Höller und Chrisitan Kreiner sowohl für die hervorragende Unterstützung bei allen technischen und organisatorischen Fragen als auch für die Ermöglichung dieses spannenden Diplomarbeits-Themas bedanken.

Bei meinen Eltern sowie meiner gesamten Familie möchte ich mich für Ermöglichung des Studiums bedanken. Einen sehr großen Dank gebührt auch meiner Freundin Anna und ihrer Familie, die mich großartig unterstützt haben. Ebenso möchte ich mich bei meinen Korrektur-Leserinnen Anna und Lisa bedanken.

Schlussendlich möchte ich allen Freunden danken.

Graz, im Monat Jahr Name des Diplomanden

Contents

List of Figures

List of Tables

Chapter 1

Introduction

For embedded safety-critical systems, dependability is a key feature because failures could result in loss of life or financial damages [HRIK15]. Commercial off-the-shelf hardware is used with increased regularity to counter the growing demands for high computing performance and cost-efficiency. Since transistor sizes are continuously shrinking, these components are becoming increasingly vulnerable. Hardware redundancy and software redundancy can be used to detect hardware faults. Hardware redundancy is achieved by duplicating components of a system where the same software is executed in parallel. A hardware fault is detected when the comparison of the outputs of the parallel executions does not match. On the other side, when multiple independent programs are created from the same specification, these different versions can be executed sequentially on the same hardware. The diverse usage of registers and instructions makes it possible to detect hardware faults by comparing the output of the different versions. This thesis focuses on software redundancy in combination with a 1oo2 architecture (see Chapter 2.2.1).

1.1 Motivation

In [SFF^{+}14], the term mutational robustness is defined as "the fraction of random mutations to program code that leave a program's behavior unchanged". It is stated that mutational robustness is an inherent property of software since over 30 % of random mutations are neutral with respect to their specification.

According to the authors of [SFF^{+}14] the property holds across different classes of programs, for source code mutations as well as assembly instruction level mutations across various programming languages, where the test suite coverage influences the results only to a certain limit.

In this thesis the mutational robustness of software is used to create neutral mutations of programs. It is investigated how well suited these mutations are for hardware fault detection and fault tolerance. For this purpose, a diversification chain with five different diversification methods is implemented to allow the comparison of different approaches such as diverse compiling and fixed register transformations (see Chapter 3.2).

The authors of [Sch14] use a software evolution library written in common lisp to modify and evaluate existing software. The software evolution library is capable of:

- mutating the abstract syntax tree with Clang, CIL and LLVM,

- mutating the executable linkable format (ELF),
- mutating lisp source, and
- mutating assembly code.

This thesis focuses on source mutations generated with the Clang LibTooling library (see Chapter 3.4.1) and on assembly code mutations which are generated by a python module. The generated mutations have to be valid according to a provided test suite. In order to gain even more diversity, the implemented diversification chain allows the combination different approaches. It is examined which combinations are useful.

C-Programs out of the categories **Automotive and Industrial Control**, **Network** and **Telecommunications** taken from the MiBench benchmark suite [GRE$^+$01] are used for the simulation.

1.2 Goals

The goals of this thesis are:

1.2.1 Framework for Automatic Software Diversity

A framework is implemented for automatic software diversity with unsound randomization techniques (see Chapter 2.6.1) to

- mutate C-family source files with a tool based on the software evolution library [Sch14],[Sch15b],
- mutate assembler files,
- validate mutated programs against a test suite,
- provide evolution algorithms for source and assembler mutation,
- manage generated mutation files,
- diverse compile programs with GCC and Clang along with the mutation flags `-O0` to `-O3`,
- use the fixed register flag in combination with the GCC compiler,
- generate precompiled versions of the used libraries to enhance the mutation speed,
- compile programs to assembler for ASM mutation, and
- generate meta information used by the simulation algorithms.

1.2.2 Integration of FIES

To simulate hardware faults for an ARM9 processor, a QEMU based fault injection framework [Sch15a] should be used within the framework for automatic software diversity. The framework is adopted to allow the selection of a specific test input during runtime. Furthermore, an easy to use python module is implemented to allow the fast specification of

- permanent register faults with different fault modes,
- instruction decoder faults, and
- different RAM faults.

For this reason methods are implemented to examine the ELF format of binaries to get information about the used assembler instructions. In order to inject RAM faults efficiently, a method is implemented to log all accessed RAM addresses during the execution of a binary. Furthermore, methods for the simulation are provided to generate information used by fault detection mechanisms.

1.2.3 Hardware Fault Detection

Different diversification methods are compared regarding hardware fault detection and fault tolerance by using MooN architecture presented in Chapter 2.2.1.

1.2.4 Compiler Toolchain

A compiler toolchain for an Freescale i.MX28 EVK PCB REV D board [Sem11] board to run bare metal programs is implemented using Ubuntu 14.04.2 as host system.

1.2.5 Fault Recovery

An algorithm based on the diversification chain is implemented to recover from permanent hardware faults.

1.3 Outline

This thesis consists of six different chapters:

Chapter 2 gives background knowledge about the used components and methods. For the evaluation of automated software diversification methods presented in this thesis, an ARM processor is simulated using QEMU. To inject hardware faults into the simulated ARM processor a fault injection framework called FIES is used which is based on QEMU. The used automated software diversification method can be categorized as unsound randomization technique (see Chapter 2.6.1). There exist many other approaches for automated software diversity which are listed in Chapter 2.6. A program can be represented in different ways during the software life cycle. The representations used for automated software diversification in this thesis are presented in Chapter 2.9.

Chapter 3 presents the components used to create the framework for automated software diversity. As described in Section 3.1, QEMU is used by the fault injection tool to introduce faults into an Freescale i.MX28 EVK PCB REV D board (see Section 3.7). A diversification chain (see Section 3.2) implements the transformations presented in Section 3.5 and offers methods for diverse compiling (see Section 3.3). The presented AST transformations are implemented using LLVM (see Section 3.4). In Section 3.6 the used mutation algorithm is presented.

Chapter 4 provides insights into implementation details for

- the diversification chain,
- the mutation and evolution algorithms,
- the simulation with FIES,
- the modifications on FIES, and
- the creation of the compiler toolchain.

Chapter 5 presents the results of this thesis. The diverse compiling approach is compared to unsound randomization methods regarding fault detection. Furthermore, the diverse compiling approach is extended with unsound randomization methods to improve the results. Fault recovery experiments are shown with promising results. Additionally, the performance of creating software mutations is presented.

Chapter 6 provides a summary of this thesis and gives ideas for further work concerning hardware fault detection with unsound randomization methods.

Chapter 2

Technical Background and Related Work

This chapter provides background knowledge about the used components and methods. For the evaluation of automated software diversification methods presented in this thesis, an ARM processor is simulated using QEMU. To inject hardware faults into the simulated ARM processor a fault injection framework called FIES is used which is based on QEMU. The used automated software diversification method can be categorized as unsound randomization technique (see Chapter 2.6.1). There exist many other approaches for automated software diversity which are listed in Chapter 2.6. A program can be represented in different ways during the software life cycle. The representations used for automated software diversification in this thesis are presented in Chapter 2.9.

2.1 Fault Tolerance

„Fault tolerance is the ability of a system to continue performing its intended functions in presence of faults „ [Dub13]. *Software Fault Tolerance* aims at software faults and the goal of *Hardware Fault Tolerance* is to tolerate hardware faults. In this thesis hardware fault tolerance is addressed. The two phases of hardware fault tolerance are fault detection and system recovery [HRIK15]. System recovery can be performed with backward recovery or forward recovery. The goal of system recovery is to „remove errors and their effects from the computational state before a failure occurs“ [Pul01]. In this thesis an approach is presented to recover from permanent hardware faults using unsound randomization (see Chapter 2.6.1) along with different compiler flags. Furthermore, the presented diversification chain (see Chapter 3.2) is evaluated according its capabilities regarding fault detection. Faults can be detected by using redundancy techniques.

According to the authors of [Pul01], a distinction can be made between the terms *fault*, *error* and *failure*. The terms fault, error and failure are related to each other yet different in their exact meaning. An error is caused by an event. This event is called fault and describes an incorrect state. If a fault occurs, the specification is not fulfilled anymore. Only if the unfulfilled specification results in broken external services, a failure occurs [HRIK15].

A *fault* can be regarded as an incorrect state of a system. An *error* occurs if a fault is activated and the resulting state of the system deviates from the specification [HRIK15]. A *failure* occurs when the service produces incorrect results. The service deviates from the the specification.

According to the authors of [KML+06], there are three types of faults:

1. *Permanent Faults* are caused by irreversible physical hardware changes,
2. *Transient Faults* are caused by temporary environmental conditions, and
3. *Intermittent Faults* are caused by unstable hardware.

Furthermore, a distinction can be made between the effects of the mentioned faults. A fail-silent fault occurs when a program on a faulty hardware crashes. On the other side, when the program does not crash and produces a faulty output it is called a Byzantine fault.

2.2 Redundancy Techniques

According to the authors of [Pul01], redundancy is achieved by adding „additional resources that would not be required if fault tolerance were not being implemented“. In this thesis redundancy techniques are used for hardware fault detection.

2.2.1 M-out-of-N Architecture

A *M-out-of-N (MooN)* system consists of *N* replicas where at least *M* replicas have to work correctly [HRIK15],[koo15]. In this thesis a 1oo2 system is used. The results of the two programs are compared by a voter. It can be distinguished between time and spatial redundancy techniques. When spatial redundancy is used each replica is executed on different hardware. Time redundancy repeats the execution of programs using the same hardware. To detect transient hardware faults it is sufficient to use the same software. For detecting permanent hardware faults some sort of diversification has to be introduced to the programs. Otherwise it would be possible that both versions produce the same wrong output.

2.3 ARM926EJ-S Processor

The ARM926EJ-S processor is a general-purpose microprocessor which belongs to the ARM9 family [ARM08]. Two different instruction sets are supported: the 32-bit ARM and the 16-bit Thumb instructions. These two different instruction sets are used to choose between high performance and high code density. On this CPU, Java byte code execution performance is comparable to JIT with the advantage of avoiding code overhead. To ease debugging, the ARM debug architecture can be used. The Harvard cached architecture comes with a Memory Management Unit (MMU), different interfaces for instruction and data AMBA AHB buses and different interfaces for instruction and data TCM interfaces. The implemented v5TEj ARM architecture enables external coprocessors for floating-point or other hardware acceleration support.

2.3.1 MMU

Virtual memory features are enabled by the ARM architecture v5 MMU to support WindowsCE, Linux and Symbian OS. To control the memory region attributes, the address translation and permission checks, the page tables are stored in main memory. A single unified Translation Lookaside Buffer (TLB) is used to cache page table information and consists fo the main TLB, which is a two-way, set-associative cache for page table information with 32 to 64 entries and a lockdown TLB, which is fully-associative cache for locked TLB entries with eight entries. The lockdown TLB is used to avoid page table walks for a specific memory region. There are four different mapping sizes supported:

- sections with 1MB,
- large pages with 64KB,
- small pages with 4KB, and
- tiny pages with 1KB.

Hardware page table walks speed up the MMU performance. Subpage permissions are useful to specify permissions for quarters of large and small pages.

2.3.2 Registers

A set of 37 32-bit registers are available on the ARM9EJ-S CPU with 31 general purpose and six status registers, where the accessibility is dependent on the processor state and operating mode. The registers r0 to r15 and the Program Status Register (CPSR) can be used when the CPU is in ARM state and user mode. The stack pointer (SP) is located in the register r13 and the link register (LR) can be found in r14. The program counter (PC), located in r15, is copied in this LR when a branch is executed. Using the thumb mode of the CPU reduces the number of usable registers. Only the registers r0 to r7 are available in this case.

2.4 Fault Injection

Fault injection techniques are useful to

- understand the effects of real faults,
- check the efficacy, failure coverage and latency of the provided fault tolerance mechanisms,
- test the target under different workloads,
- find weak spots in the design, and
- investigate the system's behavior in the presence of faults [ZAV$^+$04].

The following components can be used to construct a fault injection environment: The **fault injector** executes commands from the workload generator and injects faults. Fault types and fault locations, as well as timing information and hardware semantics are stored in the **fault library**. The input for the target is generated in the **workload generator**. To store examples for the workloads, a **workload library** can be used. A **controller** controls the experiment. Data collection is initiated by a **Monitor**, performed by a **data collector**, and analyzed by a **data analyzer** [ZAV+04].

There exist different types of fault injection methods:

- **Hardware-Based Fault Injection** uses special hardware to add faults into the target system. In can be differed between injection methods with contact and contactless injection methods. Methods with contact are possible, if the injector has physical contact with the target system, allowing the tester to use pinlevel active probes and socket insertion. External sources, to produce radiation or electromagnetic interferences, are used when contactless injection methods are applied. Hardware based fault injection can introduce faults to places on the chip, which are hard to test by other methods. Coverage and latency can be determined practically only with direct hardware insertion methods. The injected faults are very stable in their signal. Modifications of the target systems are not necessary by using hardware based fault injection. The near real-time execution speed of the experiments allows to run a large number of fault injection experiments. By using the real hardware, design faults can be found easily, since the actual hard and software is used for the tests. On the downside, hardware fault injection can damage the injected system. Furthermore, dense packaging of circuits and device integration can limit the possibility of injection. The set of injection points and injectable faults is limited by the actual hardware. Moreover special hardware is needed to perform the experiments.

  RIFLE is a system to inject deterministic and reproducible faults into processor pins, where errors can be found by tool without the usage of feedback circuits [MRMS94]. The microprocessor-based jet-engine controller of the Boing 747 and 757 aircrafts are tested with the FOCUS design automation environment [CMR+01].

  MESSALINE [Arl90] allows pin-level fault injection and supports stuck-at, open, bridging and complex logical faults.

  GOOFI (Generic Object-Oriented Fault Injection) allows to inject pre-runtime software implemented faults (SWIFI) and Scan-Chain implemented faults (SCIFI) [AVFK01]. With SCIFI faults can be injected into the pins and into the internal state elements of an circuit. Furthermore the internal state state can be observed. SWIFI enables to inject faults into the program and data areas of the target system. The authors state, that the GOOFI tool is highly portable between different hosts.

  Contact and contactless faults, to introduce transient faults, can be directly injected into a chip using FIST [KLD+94].

- **Software-Based Fault Injection** is used to inject hardware faults by simulating the target system. The set of introducable faults includes timing faults, missing messages, replays, corrupted memory, corrupted registers and faulty disk reads. Implementation details are important for software-based fault injection. The functional behavior of the target system is not affected by the injection software because of the independency of the injection software. No special-purpose hardware is needed, so this method is cheap and not complex, compared to hardware-based fault injection. Like hardware-based fault injection methods, the execution on real hardware enables to find design faults in the actual hardware. Drawbacks include the limited set of injection instants, the inability of injecting faults into all hardware locations, the requirement of modifying the source for the experiments, the limited controllability, the difficulty of modeling permanent faults and the difficulty to test hard real-time systems, because of the strict deadlines and the scheduler.

Faults can be injected during compile-time or run-time:

  - When the fault is introduced into the source code, before the program is loaded (*compile-time*), no additional software is necessary during runtime. This hard-coded fault effect allows the introduction of permanent faults.
  - *Run-time* methods require mechanisms to trigger fault injection. A simple time-out can be used to generate an interrupt for fault injection. Traps can be used to inject faults, whenever specific events or conditions occur. Code insertions are used to add instructions, which start events.

Software traps are used by FERRARI, to inject CPU, memory and bus faults[KKA92]. The program counter or a timer can be used to trigger events. Permanent and transient faults are supported.

FTAPE makes use of fault injection drivers to generate errors in CPU modules, memory locations and disk subsystems [TIJ96].

- **Simulation-Based Fault Injection** makes use of hardware description languages, like Very high speed integrated circuit Hardware Description Language (VHDL), to simulate the system under analysis and introduce faults, according to a distribution function. The full set of system abstraction levels are supported by this approach, including the electrical, logical, functional and architectural layer and it is still not intrusive but allows full control of both fault models and injection mechanisms. No special purpose hardware is necessary and the same software that will run in the field is used. Transient and permanent faults are supported as well as timing-related faults.

  The drawbacks of the simulation-based fault injection include the development efforts, the time consuming length of the experiment, the inability of real time fault injection and the inability of finding design faults.

  VERIFY modifies the VHDL language to enable multi-threaded fault injection by describing faults [STB97].

  MEFISTO-C comes with a variety of predefined fault models and makes use of VHDL simulation models to create fault injection experiments [FSK98].

  Permanent and transient faults can be simulated with HEARTLESS, which can be called a register-transfer-level-fault-simulator [RPB$^+$01].

- **Emulation-Based Fault Injection** is faster than simulation-based fault injection, but the costs for a FPGA based emulation board is quite high and only functional consequences of faults can be investigated.
- **Hybrid Fault Injection** methods use combinations of different fault injection techniques.

  LIVE is used to evaluate computer-based railway control systems with fault injection and software testing methods. Software-based and simulation-based fault injection methods are combined.

2.5 Fault Modes

This section desribes the different fault modes which are used for the simulation on QEMU. Faults are injected into CPU register cells, RAM register cells and into the instruction decoder.

2.5.1 RAM Fault Modes

To describe RAM faults a special notation can be used, consisting of three different parts: $< S/F/R >$. This triple is called a *fault primitive (FP)*. S is the *sensitizing operation sequence (SOS)* that leads to the fault [AAVdG01]. F is a binary number ($F \in \{0,1\}$) and contains the value of the faulty cell. R is used to represent the output value of a read operation. $R \in \{0,1,-\}$, where $-$ denotes that a write operations is used.

In Figure 2.1 a taxonomy of FPs is shown, classified by the number of accessed cells ($\#C$) and the number of operations ($\#O$).

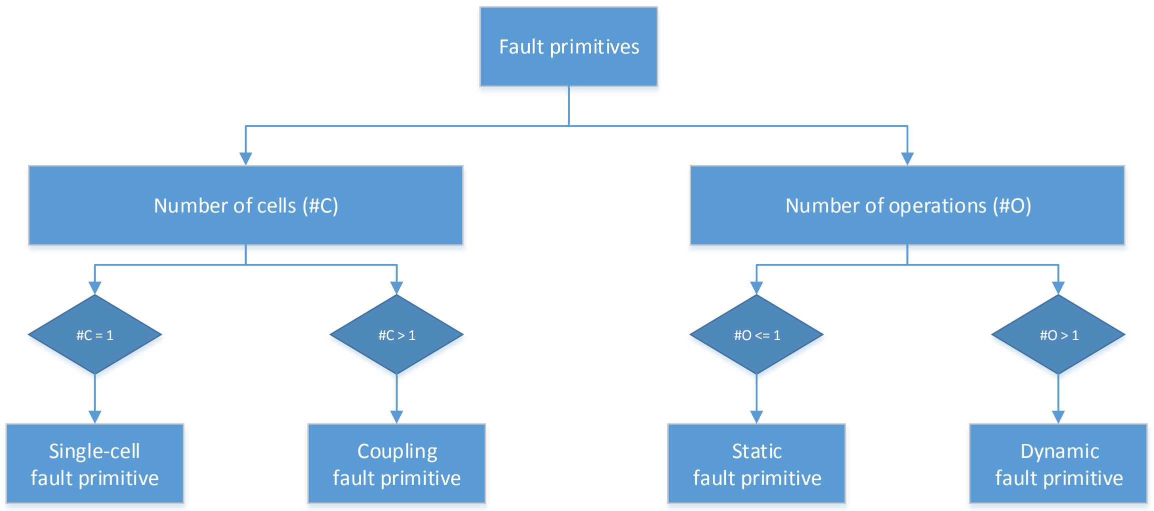


Figure 2.1: Taxonomy of fault primitives [AAVdG01].

In this thesis single cell FFMs are used. In Table 2.1 the supported dynamic and static FFMs are listed with the according FPs.

Functional fault model	*Fault primitives*
	Static FFMs
Stuck-at faults	$SAF_0 =< 0/1/- >$, $SAF_1 =< 1/0/- >$
Transition fault	$TF_\uparrow =< 0w1/0/- >$, $TF_\downarrow =< 1w0/1/- >$
Read disturb fault	$RDF_0 =< 0r0/1/1 >$, $RDF_1 =< 1r1/0/0 >$
Write disturb fault	$WDF_0 =< 0w0/1/- >$, $WDF_1 =< 1w1/0/- >$
Incorrect read fault	$IRF_0 =< 0r0/0/1 >$, $IRF_1 =< 1r1/1/0 >$
Deceptive RDF	$DRDF_0 =< 0r0/1/0 >$, $DRDF_1 =< 1r1/0/1 >$
	Dynamic FFMs
Read disturb fault	$RDF_{00} =< 0w0r0/1/1 >$, $RDF_{11} = 1w1r1/0/0 >$
	$RDF_{01} =< 0w1r1/0/0 >$, $RDF_{10} =< 1w0r0/1/1 >$
Incorrect read fault	$IRF_{00} =< 0w0r0/0/1 >$,$IRF_{11} =< 1w1r1/1/0 >$
	$IRF_{01} =< 0w1r1/1/0 >$,$IRF_{10} =< 1w0r0/0/1 >$
Deceptive RDF	$DRDF_{00} =< 0w0r0/1/0 >$,$DRDF_{11} =< 1w1r1/0/1 >$,
	$DRDF_{01} =< 0w1r1/0/1 >$,$DRDF_{10} =< 1w0r0/1/0 >$,

Table 2.1: Supported single cell FFMs [AHK15],[AAVdG01]

2.5.2 Instruction Decoder Faults

According to [HRIK15] instruction decoder faults can be classified as *inactive*, *wrong*, or *additional*. *Inactive* faults occur when an instruction is not executed. When too many instructions are executed it is classified as *additional* fault. The execution of wrong instructions is called a *wrong* fault. In this thesis inactive instruction decoder faults are used by specifying a 'No Operation' instruction. See Chapter 4.4.4 for the specification which is used by the fault injection framework presented in Chapter 4.4.1.

2.6 Automated Software Diversity

„*Automated software diversity* consists of techniques for artificially and automatically synthesizing diversity in software“ [BM14]. The term „Automated Software Diversity“ has been formed 20 years ago and this idea has been proven to be useful to counter attacks, by adding uncertainty to the target [LHBF14]. To create attacks, exact knowledge of the target software is essential. Therefore, diversity allows us to generate a broad defense line. Although homogenous software, along with standardizations allows us to scale systems, guarantee consistent behavior and simplifies the logistics of distribution [LHBF14]. Unfortunately, this homogeneity can be used by attackers, since the downloaded program can be probed for vulnerabilities. All systems running the vulnerable program can be attacked, when the vulnerability is turned into an exploit. The goal of automated software diversity is, that the attacker has to target each system individually, to raise the effort of the attacker.

2.6.1 Categorization

Randomization

Randomization can be used to automatically introduce diversity into applications. These randomization transformations can be categorized:

- *Static randomization* is used to create different source code versions of a program. In Section 2.6.2, methods and transformations to randomize source code on different levels, are presented. Obfuscation is used to prevent reverse engineering, but since obfuscating is automated, the generation of different versions of one single program is possible.

- *Dynamic randomization* adds randomization points to executables. While static randomization is applied during compile time, dynamic randomization is used during runtime and enables diverse executions, even under the same input. In [HNL$^+$13], just-in-time compilation randomization is presented. No randomization points are used, but the compiler itself is responsible for the randomization by inserting NOP instructions. *Data diversity* is introduced in [AK88] and allows to run a program in the presence of hardware failures. The input data is changed to bypass the failure. After the computation, an inverse transform function is used to calculate the actual output of the function. *Environment diversity* can be used to enable correct program executions on erroneous hardware by changing the environment.

- *Unsound program transformation:* While diverse compiling preserve the exact behavior of the original program, the random mutation approach may alter the semantics of a program. Transformations, which change the semantics of a program are called unsound program transformations. In [FS10] it is shown that the linking of two slightly different binary files is possible and useful to tolerate bugs. Transformation strategies on Java statements are investigated in [BAM14], where the term *sosie* of a program is defined as a set of source codes, which pass the same test suite. A large quantity of variants can be synthesized with different control and data flows. The impact of skipping a certain amount of loop iterations is investigated in [SDMHR11], where a trade-off between performance and accuracy is measured.

Domain-specific Diversity

Domain-specific knowledge can be used to create efficient diversification. The diversification of SQL statements is presented in [BK04]. All SQL keywords are prefixed with a specific token to prevent attacks from outside. Hardware faults can be found by applying program transformations to change numerical operations in a predefined manner [OMM02]. *Metamorphic engines* are used by computer viruses to constantly change itself [BFM10]. This makes it more difficult for the antivirus software to find all variants of a malign program. In the context of testing, the generation of random input test data is called *adaptive random testing* [CKMT10], when the generation approach depends on the previous generated test case.

Integrated Diversity

The automatic injection of different kind of diversity into a single program is called integrated diversity [BM14]. *Stacked diversity* integrates different forms of diversity into an application, where each diverse transformation improves the application in one perspective. In [WDHK01] a multi level program diversification is used to enable obfuscation. To protect a program against code injection attacks, the authors of [BDS03] used multiple randomization methods. They mutated the base addresses of memory regions, the order of stack-variables, and introduced random gaps in the memory layout.

The Genesis Diversity Toolkit [LDE$^+$07] can be used to create artificial diversity by

- Address Space Randomization (ASR),
- Stack Space Randomization (SSR),
- Simple Execution Randomization (SER),
- Strong Instruction Randomization (SIR), and
- Calling Sequence Diversity (CSD).

This toolkit is used by [WHD$^+$09], to establish a toolchain for diversity transformations at compile time, link time, load time and runtime. Runtime diversity is enabled by the Strata virtual machine technology. Compilation superoptimization is used to enable superdifersification by bytecode transformations[jac08]. The introduction of a new paradigm for software, aiming at massive-scale diversity is done by the authors of [Fra10] to counter reuse of software vulnerabilities. To protect distributed systems from men-at-the-end attacks, [CMMN12] use techniques to flatten the control flow, merge functions, dummy code addition, parameter reordering and variable encoding. A diversity scheduler is used to enable temporal diversity. When diversification methods are combined some sort of diversification controller is necassary. In [PSS12], the benefits of diversity management controllers are discussed. To detect security issues [CEF$^+$06] uses N-variant systems to create mutants of a program and run them in parallel [BM14]. Unlike N-version programming, N-variant systems are created automatically and not implemented by different teams.

2.6.2 Diversification Levels

Automated software diversity depends on the set of transformation functions and on the used software representation. The software representation changes during the software life-cycle. Simply spoken, it has to be determined, *what* and *when* diversification should take place.

There are many possible diversification-levels available:

- **Instruction Level:** Instructions of basic blocks are permuted. A sequence of instructions is called basic block, if the execution of the first instruction, guarantees the executions of the other instructions of the block. *Equivalent Instruction Substitution* can be used to exchange two equivalent instructions. *Equivalent Instruction Sequences* are used when the functionality of instruction sequences overlaps [LHBF14].

Instruction Reordering works as long as the dependencies within the instructions are preserved. *Garbage Code Insertion* allows to generate infinitely many program variants, by adding no-operation instructions (NOPs) or more complex statements.

- **Basic Block Level:** *Basic Block Reordering* is based on the last instruction of a basic block, since this instruction may branch to another block. *Branch Function Insertion* works by inserting additional code to calculate the return address of a function based on the caller.
- **Function Level:** To counter buffer overflow attacks,
  - stack frame padding,
  - stack variable reordering,
  - stack growth reversal, and
  - non-contiguous stack allocation

  can be used for the transformation. *Function Parameter Randomization* can be used to permute existing parameters. The insertion of new parameters is also possible, when all function-calls are adopted to the changes. *Inlining* a function means, that the actual function call is avoided, by replacing the function call with the contents of the function. *Outlining* is the process of extracting a basic block into a new function. If code of a function is outlined, the transformation is called *Splitting*. *Control Flow Flattening* allows to connect basic blocks with indirect jumps through jump tables.
- **Program Level:** The order of functions within executables and libraries can be chosen arbitrarily, so randomization can be done with less effort. *Address Space Layout Randomization (ASLR)* can be implemented, because the virtual address space of each process is private, therefore the starting address can be chosen at random. Currently, ASLR is the „only deployed probabilistic defense“ [LHBF14]. *Program Encoding Randomization* is the method of change the encoding of a program. The changed encoding of a program can be reversed by a virtual machine which either interprets the changed encoding or emulates a machine for the changed encoding. *Data Randomization* can be used to defend against memory corruption attacks, by
  - permutate static variables and add padding, by
  - blinding constants, where the actual value of the constant is retrieved during runtime, by
  - randomizing composite data structures such as classes and structs, and by
  - adding random padding to objects on the heap.
- **System Level:** Transformations in this category use knowledge about the system software. An example would be *System Call Mapping Randomization*, which is used to diversify the system call interface between processes and the operating system.

2.6.3 Diversification During The Software Life-Cycle

The life-cycle of software can be split into the following categories: implementation, compilation, linking, installation, loading, executing and updating.

- **Implementation:** N-Version programming is a well known method to diversify software and is based on *design diversity*[AC77]. Software components are designed and implemented by using separate teams and programming languages. The goal is to minimize the probability, that the resulting software contains similar errors. A voter compares the output of the diverse instances. *Design diversity* is expensive to implement and therefore only used by some domains like aerospace or automotive software.

- **Compilation and Linking:** Diverse compiling avoids changing the source code by automatizing the process of diversification. This process is limited, since compilers do not know the specifications of the program, only the source code is visible. Additionally, the semantic of the source has to be kept, so high-level transformations to change algorithms, are not supported.

  Using a compiler for diversification avoids the need for dissassembly. Furthermore, multiple hardware platforms can be targeted easily and transformations can be added relatively easy, since compilers contain the needed source analysis tools [HRIK15].

  In contrast to compile-time diversification, link-time diversification works with proprietary compilers and linkers.

- **Installation:** When diversification is done during, or after installation, the disassemble of stripped binaries is a challenging task, since errors are introduced when no debugging symbols are available. *In-place diversification* [PPK12] rewrites code, reachable from the program entry point. Unreachable code is not diversified. Many approaches for diversification are implemented on multiple steps in the software life cycle. Instruction location randomization is used to create a new program encoding by rewriting binaries [HNTC^{+}12].

  The advantages of post-distribution diversification approaches are:

  - Since no source code is needed, proprietary software and legacy software can be transformed.
  - The diversification approach is compatible to current software distribution systems, because transformation is done on the end user system.
  - In contrast to pre-distribution diversification methods, the computing power needed for the transformations is spread among the entire user base.

  There are some drawbacks as well:

  - Since diversification is done on client side, there is no protection against client side attacks.
  - The diversification engine must be installed on all clients and therefore becomes an important target for attacks.

  – This method is not applicable for operating system diversification.

  [LHBF14]

- **Loading:** Diversification is added while the program is loaded into memory by the operating system. Signed binaries are supported by this approach, since the on-disk representation is not altered. *Dynamic binary rewriting* allows to rewrite code on the fly. A code cache is used to store translated code pages. The downside of this approach is the runtime overhead, since the translation adds to execution time. Another drawback of this method is the inability to share code pages for randomized libraries. In [BAFS05], the instruction set encoding is randomized. Before execution the instructions are decoded with the valgrind dynamic rewriter [NS07].
- **Execution:** Dynamically allocated data can be located randomly in memory. The memory allocator can be changed to randomize one object per time [NB10]
- **Updating:** Before a patch is generated, the software can be diversified to avoid the exploiting of bugs which can be found by comparing the updated software with the original software [CDSDB13].

2.7 Managed Software Diversity

Unlike automated software diversity, managed software diversity approaches control the introduced diversity in different ways:

N-Version diversity can be applied on the product itself, on the process and the environment [Avi95]. *N-Version programming*, as discussed in Section 2.6.3, can be categorized as a „design diversity“ technique. The term „N-version design“ is defined as „the independent generation of $N \geq 2$ functionally equivalent programs from the same initial specification“ [AK84] [BM14].

Managed Natural Software Diversity is defined as „the existence of different software solutions that provide similar functionalities and which spontaneously emerge from software development processes“ [BM14]. Web browsers, operating systems, firewalls, database management systems, virtual machines and servers are examples for natural software diversity. In [WWB03], different versions of servers which run on different operating systems improve the security level of a system. The term „N-version protection“ is defined in [OCJ08] and makes use of different antivirus and malware programs, where a cloud is used to put up with the performance overhead.

Managed Functional Diversity Software abstractions are used to hide the diversity of implementation details and offer a unified abstraction. The Unix file capture concept is an example for this kind of diversity. A very popular software abstraction is the *objekt-oriented software paradigm*, where polymorphism is used to enable code-calls in a non predefined manner [BM14]. The late binding of code-calls is essential for enabling diversity. *Software product lines* are used to handle a diversity of requirements with a diversity of software implementations. A feature model can be used to model the requirements and software solutions. The main challenge is to identify existing parts of software systems for reuse.

Plugin based software architecture is used to encapsulate functionality and dependency information to enable open software systems. Eclipse, GIMP and Audacity are just some examples of successful projects using this approach. Since plugins can be combined in many variations, the software diversity of such systems is very high.

2.8 Evolutionary Algorithms

The idea behind evolutionary algorithms is to evaluate individuals in a population according to a quality function and to apply selection methods where the fittest candidates have a higher probability to be chosen as parents for the next generation [ES03]. The next generation can be created by applying mutation transformations on the existing individuals or by recombination of existing individuals. Algorithm 1 shows the basic structure of evolutionary algorithms in pseudo code. Furthermore, the general scheme can be seen in Figure 2.2.

Algorithm 1 General Evolutionary Algorithm

```
1: procedure EVOLUTION
2:     INITIALISE population with random candidates
3:     EVALUATE each candidate
4:     while TERMINATION CONDITION do
5:         SELECT parents
6:         RECOMBINE pairs of parents
7:         MUTATE the resulting offspring
8:         EVALUATE new candidates
9:         SELECT individuals for the next generation
```

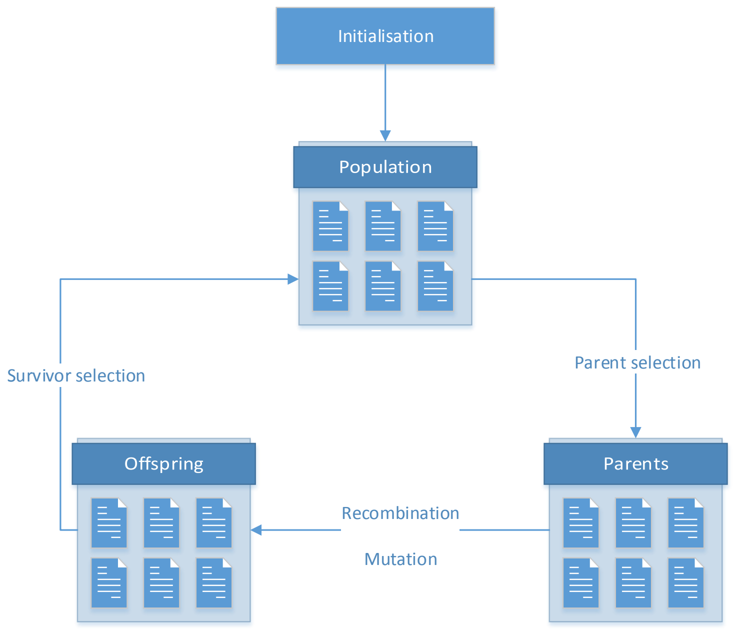


Figure 2.2: The general scheme of an evolutionary algorithm [ES03]

2.9 Program Representation Tools

Programs can be represented by hierarchical trees (AST) or by linear vectors (ASM) [Sch14]. Common tools for the high-level program representation are based on

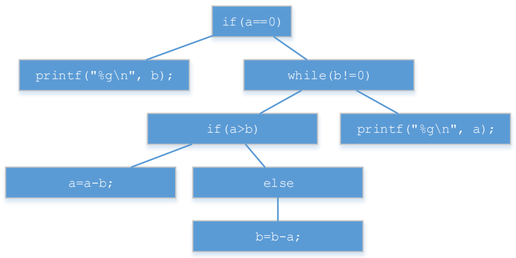
- CIL [NMRW02a], and on
- C Language family frontend for LLVM (CLang) [Lat08a].

Low-level program representations can be based on

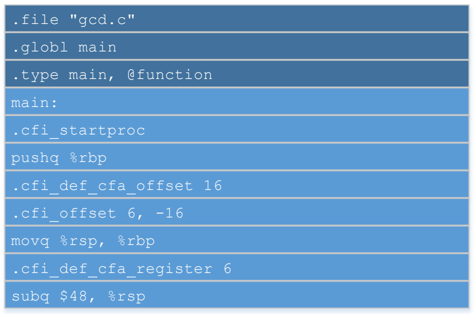
- argumented ASM code [SFW10], and on
- Low Level Virtual Machine (LLVM) IR [LA04].

```
if (a==0){
  printf("%g\n", b); }
else {
  while (b!=0){
    if (a>b){ a=a-b; }
    else    { b=b-a; } } }
printf("%g\n", a);
```

(a) Source

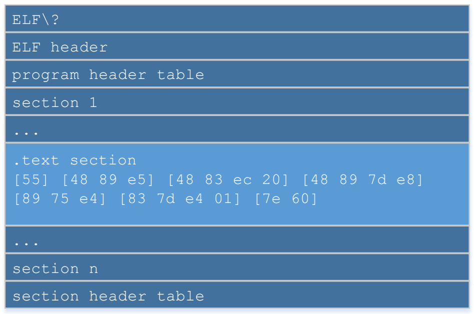


(b) AST



```
.file "gcd.c"
.globl main
.type main, @function
main:
.cfi_startproc
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
subq $48, %rsp
```

(a) Assembler



```
ELF\?
ELF header
program header table
section 1
...
.text section
[55] [48 89 e5] [48 83 ec 20] [48 89 7d e8]
[89 75 e4] [83 7d e4 01] [7e 60]
...
section n
section header table
```

(a) ELF

Figure 2.3: Program representations [Sch14].

In this thesis, the CLang-AST representation and the assembler representation are used for generating mutations (see Figure 2.3). According to [Sch14], the CLang-AST representation is closest to the source code, written by humans. As an result, fine-grained mutations are possible by using the CLang-AST. The effects of this fine-grained mutation are discussed in Chapter 5. The GCC compiler is able to generate ASM code, by using the `-S` flag. This program representation is mutated per line, so no further transformation is needed. In Figure 2.4 it is shown how the program representation influences the mutation transformations. It is distinguished between AST based and vector based approaches.

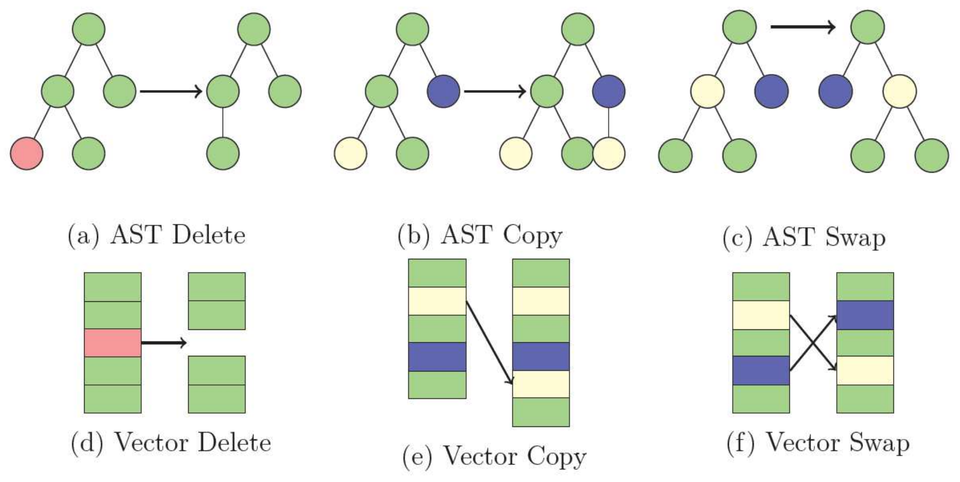


Figure 2.4: The used mutation transformations [Sch14].

2.10 Software Mutational Robustness

It already has been stated that mutational robustness is an inherent property of software, since over 30 % of random mutations are neutral with respect to their specification. A formal definition of software mutational robustness, can be given as follows:

$$MutRB(P,T,M) = \frac{|\{P'|m \in M \wedge P' = m(P) \wedge T(P') = true\}|}{|\{P'|m \in M \wedge P' = m(P)\}|}$$

P is the program, M is the set of all mutations and T represents the testsuite [SFF+14]. A specific mutation $m \in M$ can be understood as a function, which takes a Program P as input and gives a mutated program P'. In [SFF+14] it is shown, that the mutational robustness of software does not rely strongly on T and M. Abstract syntax trees (AST), generated with the CIL toolkit [NMRW02a], and assembly code (ASM) are used for the program representation. CIL is used to create a simpler version of a given program and allows source to source translations, which is used for mutation operations. The assembler code is generated with the compiler command `gcc -O2 -S`, where mutations are realized per line. The tree based high level AST representation and the linear based low level assembler representation give similar results, so the authors of the paper conclude, that the mutational robustness does not depend strongly on the chosen representation. Three language-independent mutation operators are used, as shown in Figure 2.4:

- **Copy** duplicates a statement and inserts it at a random index.
- **Delete** remove a statement at a random index.
- **Swap** exchanges two randomly chosen statements (see Figure 2.4).

To measure the software mutational robustness, it is important to use programs with high quality test suites. In [SFF+14], fourteen off-the-shelf programs are selected, along with four programs from the Siemens Software-artifact Infrastructure Repository [DER05].

For the experiment, a given source code is duplicated over 200 times. Each of these 200 clones is mutated with one of the mutation operators, mentioned in Figure 2.4. Only lines, which are tested by at least one test of the test-suite, are taken into account. This procedure gives first-order mutations. The results shows, that the average software mutational robustness is at least 36.8 % and that the specific program influences the result only to a certain limit.

2.10.1 Test Suite Quality

In [SFF+14] it is stated, that the software mutational robustness is not fully related to the test-suite quality. In their approach, they only mutated statements, which are covered by the test-suite. Though the statement-coverage is not very precise, since it only tells us, whether the line of code was executed by a test-case or not. A fully tested program reaches a software mutational robustness over 20 %, whereas untested programs reach about 84.8 %.

2.10.2 Cumulative robustness

A given ASM source code is mutated until 100 neutral first-order mutations are found. Then, a random generator selects one of the 100 neutral variants and applies a mutation. This procedure is repeated until 100 neutral second-order mutations are found. The authors of [SFF+14] apply this approach until the mutations are 250 steps away from the original program. In Figure 2.5 it can be seen, that the percentage of neutral variants rises, as the number of applied mutation steps gets bigger. During the evolution, the average number of lines of code rises as well.

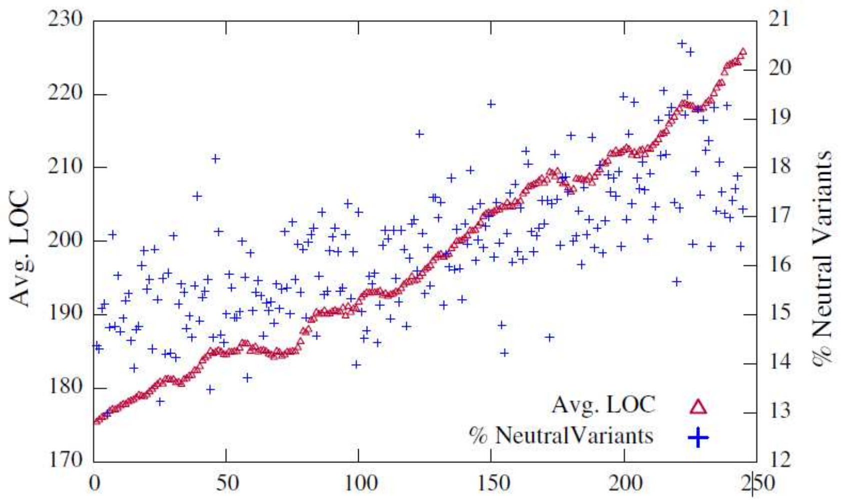
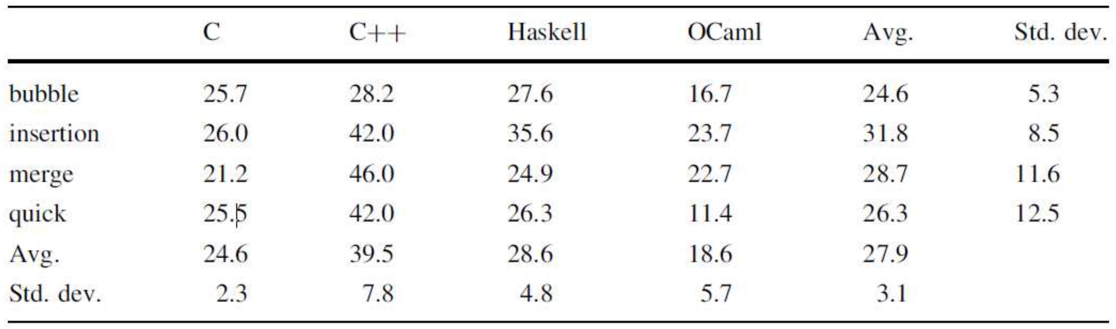


Figure 2.5: Number of applied mutations [SFF+14].

Figure 2.6 shows that the mutational robustness holds on multiple programming languages. Well known implementations of sorting algorithms were used for this purpose.



	C	C++	Haskell	OCaml	Avg.	Std. dev.
bubble	25.7	28.2	27.6	16.7	24.6	5.3
insertion	26.0	42.0	35.6	23.7	31.8	8.5
merge	21.2	46.0	24.9	22.7	28.7	11.6
quick	25.5	42.0	26.3	11.4	26.3	12.5
Avg.	24.6	39.5	28.6	18.6	27.9	
Std. dev.	2.3	7.8	4.8	5.7	3.1	

Figure 2.6: Mutational robustness of sorting algorithms [SFF+14].

2.10.3 Repairing Bugs

In [SFF+14], the following five bugs-categories are used, to generate faulty programs:

- missing conditional clause,
- extra statement,
- constant should have been variable, and
- wrong parameter.

The defect lines in the programs are covered with test-cases, so it is possible to check, whether the mutations can fix the bug. For this scenario, 5000 first-order mutations are generated, by applying the operators which are explained in Chapter 2.10. They show, that in practice, 5000 mutations are enough to fix at least one bug, in the average case. When more than 5000 mutations are used, the influence in the performance of repairing bugs, can be neglected. In 88 % of all cases, the bug repairs are compensatory. That means, the mutated line of code is unchanged, but other regions of the program are mutated with the effect of repairing the introduced malfunction.

Chapter 3

Concept and Design

In this chapter the components used to create the framework for automated software diversity are presented. As described in Section 3.1, QEMU is used by the fault injection tool to introduce faults into an Freescale i.MX28 EVK PCB REV D board (see Section 3.7). A diversification chain (see Section 3.2) implements the transformations presented in Section 3.5 and offers methods for diverse compiling (see Section 3.3). The presented AST transformations are implemented using LLVM (see Section 3.4). In Section 3.6 the used mutation algorithm is presented.

3.1 QEMU

QEMU can be used to run operating systems in a virtual machine, supporting many different host operating systems, like Linux, Windows and Mac OS X. Since the state of a virtual machine can be inspected easily, QEMU is often used for debugging purposes. The following subsystems are used:

- CPU emulator (x86, PowerPC, ARM, Sparc)
- Emulated devices (VGA display, serial port, mouse and keyboard, hard disk, network card)
- Generic devices are used to connect host devices to emulated devices.
- Machine descriptions to make instants of the emulated devices
- Debugger
- User interface

[Bel05]

The **full system emulation** operating mode of QEMU is used to emulate a full system with a processor. Multiple target operating systems can be launched in parallel [dev15]. To launch processes for a different CPU yet the same operating system as on the host system, **user mode emulation** can be used. This mode makes cross-compilation much easier.

Dynamic translation is used to get native code for gaining speed. Furthermore, self-modifying code, precise exceptions and floating points are supported. A full software emulation is possible as well as native host FPU instruction usage. For user mode emulation, generic Linux system calls are converted, the Linux scheduler for threads is used and the remapping of host signals allows accurate signal handling.

For system emulation, a full software MMU is used. Optionally an in-kernel accelerator can fasten guest code natively. Symmetric multiprocessing (SMP) can be used, even when the host system is single-cored.

3.1.1 QEMU compared to other emulators

Dynamic compilation makes QEMU much faster than bochs [Law96]. QEMU is capable of simulating several processors, while bochs is tied to x86.

Valgrind [NS07] is used for memory debugging and can track uninitialized data, which is impossible with QEMU. User space emulation and dynamic translation is supported by valgrind, but closely tied to x86 hosts.

Compared to commercial products (VMWare, VirtualPC, TwoOStwo), QEMU is slower, but many commercial products need unsafe host drivers and they can not provide cycle exact simulation. Non commercial QEMU based systems are VirtualBox, Xen and KVM.

3.2 Diversification Chain

A stacked diversity approach (see Chapter 2.6.1), based on the software evolution library, is implemented, with four different diversification methods:

- Source Mutation
- Register Transformation
- Diverse Compiling
- ASM Mutation

Each step in this chain is optional and can be combined freely. In Chapter 5, the four methods and combinations of them, are compared to each other.

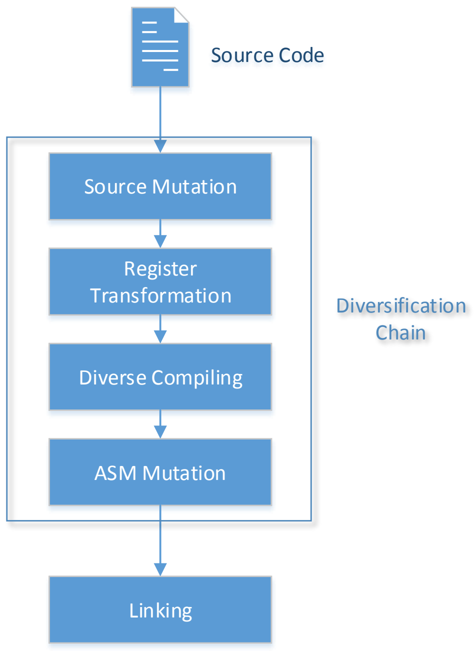


Figure 3.1: The four different diversification methods, used in this chain.

1) AST Mutation: A given source code is mutated to generate a neutral landscape with respect to the test suite. For this purpose, a software evolution framework is developed, where some elements are based on the Software Evolution Library. Parameters for the

- optimal number of applied mutations, and
- the optimal amount of neutral variants

are investigated during the experiment. The mutational transformations (Figure 2.4) are implemented with a C++ application based on LLVM (Section 3.4).

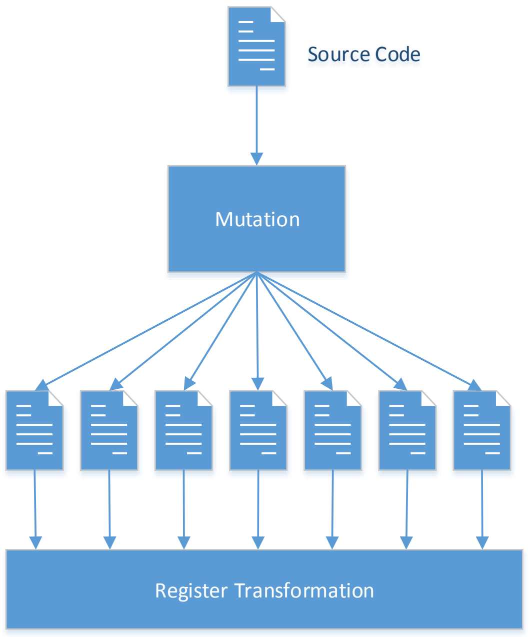


Figure 3.2: A given source code is mutated to generate a neutral landscape with respect to the test suite.

2) Register Transformation: To recover from permanent register faults, a method is implemented to avoid the usage of the specified registers. The compiler reserves that registers entirely for this use within the current compilation. It is necessary to recompile other source code and the used libraries, where the variable is not declared explicitly with the `-ffixed-reg` compiler option. For that reason, it is not possible to apply this transformation, when the source of an used library is not available or when Inline-Assembler-Statements are used, where the specified register is accessed. It is planned to create a version for every general purpose register. To gain compilation speed, the core libraries for the i.MX28 EVK board are compiled in advance, getting 15 different versions for each general purpose register. This transformation is realized during compilation, no source modification is necessary. CLANG does not support the `-ffixed-reg` flag, restricting this approach to GCC.

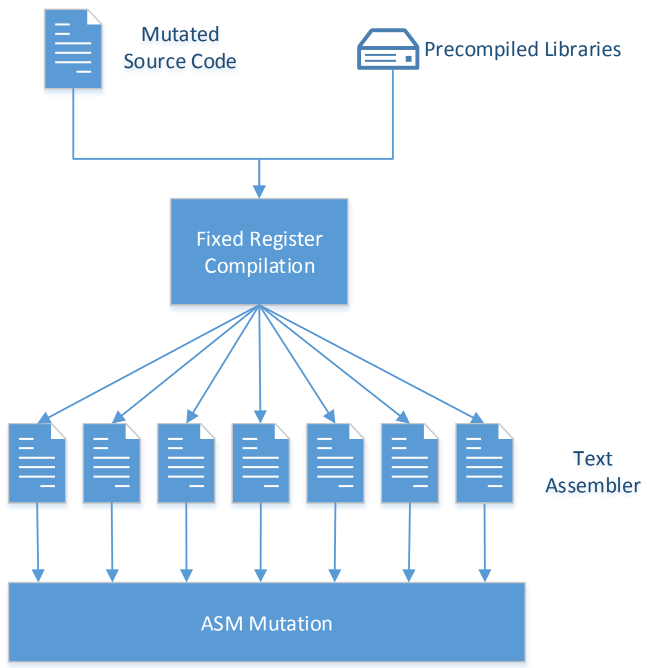


Figure 3.3: The -ffixed-reg flag is used to avoid the usage to the specified register along with precompiled libraries.

3) Diverse Compiling: A simple but efficient way to create automated software diversification is called diverse compiling, where different compilers and optimization flags are combined. I will follow the approach as presented in [HRIK15], where the Gnu Compiler Collection and the LLVM Clang compiler are used in combination with the ARM architecture (see Section 3.3). During the experiment, it is investigated,

- which optimization flags,
- what number of replicas, and
- which combination of replicas

should be used to get the best result. The used libraries are precompiled with the specified optimization flags, to avoid long compilation times. Each combination of compiler and optimization flag is used during this precompilation process. Diverse compiling is strongly connected to register transformation, since register transformation is done by using a special flag. So the number of precompiled library versions is: number of compilers $\times$ number of optimization flags $\times$ number of fixed registers. In fact, the number of precompiled library versions is not exactly this big, since CLANG does not support certain flags.

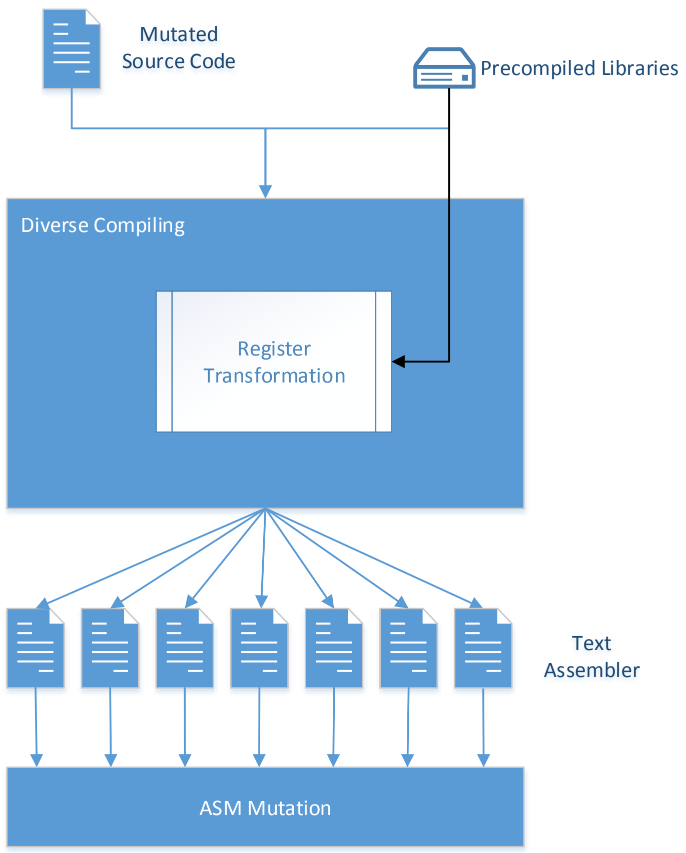


Figure 3.4: The mutated source code is compiled to text assembler with GCC and CLANG compilers, using different flags for optimization.

4) ASM Mutation: This mutation step works on text assembler, generated by the previous steps, to create a larger neutral landscape with respect to the test suite. For this purpose, an approach, based on the Software Evolution Library is implemented, where the parameters for

- the optimal number of applied mutations, and
- the optimal amount of neutral variants

are investigated during the experiment. The final step in the diversification chain would be to compile the text assembler files to executable programs using the Gnu Compiler Collection and the LLVM Clang compiler.

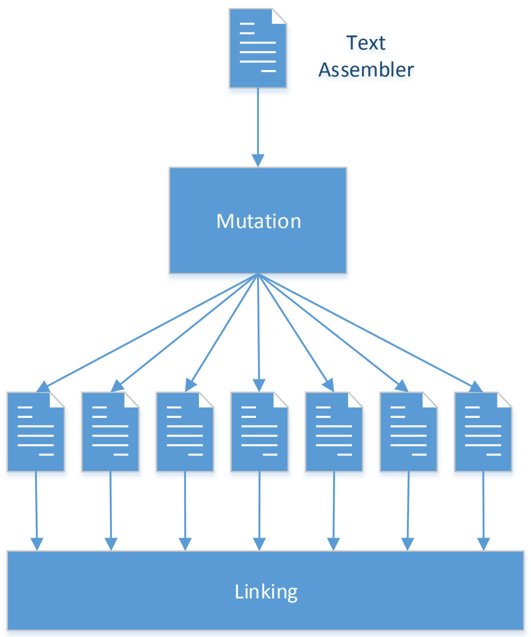


Figure 3.5: ASM mutation of text assembler.

3.3 Diverse Compiling

In [HRIK15], the diverse compiling approach is evaluated, in the context of finding faults in processors. The outcomes are, that 90 % of all register faults and 70 % of all instruction decoder faults can be detected. Time redundancy is used for fault detection, so the results of different subsequently runs are compared together. Unmodified compilers are used, with well tested optimization flags. The development costs can be kept low, because of this reason. Features of a processor are differently used by diverse compiled versions of the same source. Therefore, the possibility is very high, that a hardware fault results in different outputs of two diverse replicas. A simple output comparison check can detect the fault.

3.4 LLVM

The Low Level Virtual Machine (LLVM), is a

> „compiler framework designed to support transparent, life-long program analysis and transformation for arbitrary programs, by providing high-level information to compiler transformations at compile-time, link-time, run-time, and in idle time between runs“ [LA04].

A low-level code representation is defined with the following features:

- a language-independent type-system

- an instruction for typed address arithmetic
- a simple mechanism to implement exception handling in high-level languages

The key operations of ordinary processors are captured by the LLVM instruction set. Low-level calling conventions, pipelines, physical registers and other machine-specific constraints are avoided. As a load/store architecture, LLVM transfers values between registers and memory only via load and store operations. The avoidance of multiple opcodes for the same operation, the usage of overloaded opcodes and the three-address form of nearly all opcodes, make it possible, that the LLVM instruction set can be implemented with 31 opcodes.

The **LLVM-GCC 4.2 Front-End** is compatible with GCC and existing makefiles and provides support for C, C++, Objective-C, Ada and FORTRAN [Lat08b]. Using LLVM as back-end, enables many optimizations of source files, including inlining and constant propagation. Furthermore, a faster optimizer and a slightly better codegen can be used.

In [Lat08b], it is stated, that „GCC's front-end is slow and memory hungry“, and that important functionality for the usage in an IDE is missing. The limited source-level-information and the steep learning curve for developers are stated as well. The **CLANG Front-End** for LLVM is introduced to fix this problems, by fulfilling the following goals:

- A unified parser for C, Objective C and C++ with good error and warning messages.
- The library based architecture with C++ API's is replaceable, reentrant, composable and extensible.
- Many different usages are possible, like source to source tools, refactoring for IDE's, indexing, static analysis and code generation.
- The fast compilation and low memory footprint make CLANG a high performance front-end for LLVM.

3.4.1 CLANG LibTooling

The library LibTooling [Tea15a] is used, to create a standalone, clang-based tool. According to [Tea15f], the clang AST is closely related to the written C++ code and the C++ standard [Tea15f]. This is the main difference between the Clang AST and the ASTs from other compilers. A tool to „manipulate C-family ASTs with Clang“ [Sch15b] is used to give an example of the Clang AST. The following source gives the AST which is shown in Figure 3.6.

```
int f(int x) {
  int result = (x / 42);
  return result;
}
```

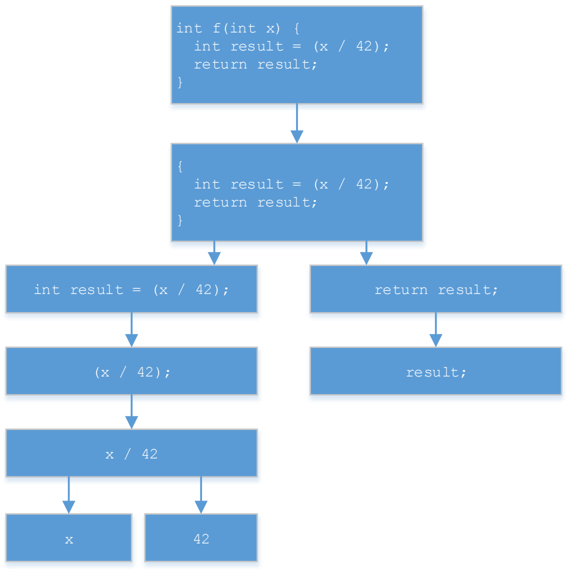


Figure 3.6: AST of C Source.

The class **ASTContext** holds all information about the AST of a translation unit. „A translation unit is the ultimate input to a C compiler from which an object file is generated“ [Wik15]. The full AST can be traversed by starting with TranslationUnitDecl, which is the top declaration context. There are two basic nodes in the Clang AST, statements and declarations. Expressions are treated as statements. To visit each node of the Clang AST in preorder depth-first traversal, the **RecursiveASTVisitor** is implemented. Figure 3.7 shows a minimal inheritance diagram. Three different tasks are performed:

1. Traversal of the Clang AST.
2. For every node, walk up in the class hierarchy.
3. For a given node and class combination, a user-overridable function is called.

Task one is done by the method TraverseDecl(Decl *x), which is the entry point for the traversal. Task two is implemented by WalkUpFromFoo(Foo *x) and task three is managed by VisitFoo(Foo *x). The calling sequence starts with Traverse* which calls WalkUpFrom* and finally Visit* is called.

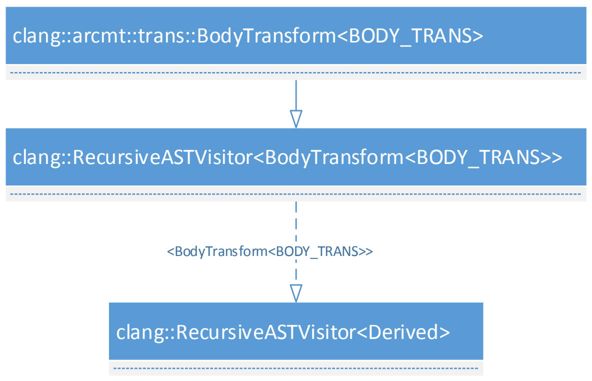

clang::arcmt::trans::BodyTransform<BODY_TRANS>
clang::RecursiveASTVisitor<BodyTransform<BODY_TRANS>>
<BodyTransform<BODY_TRANS>>
clang::RecursiveASTVisitor<Derived>

Figure 3.7: Inheritance diagram for the RecursiveASTVisitor class.

To be independent of the AST producer, the abstract interface **ASTConsumer** is implemented. Methods to add listeners and enable customized initialization are defined within the ASTConsumer class. An inheritance diagram is shown in Figure 3.8 [Tea15c]

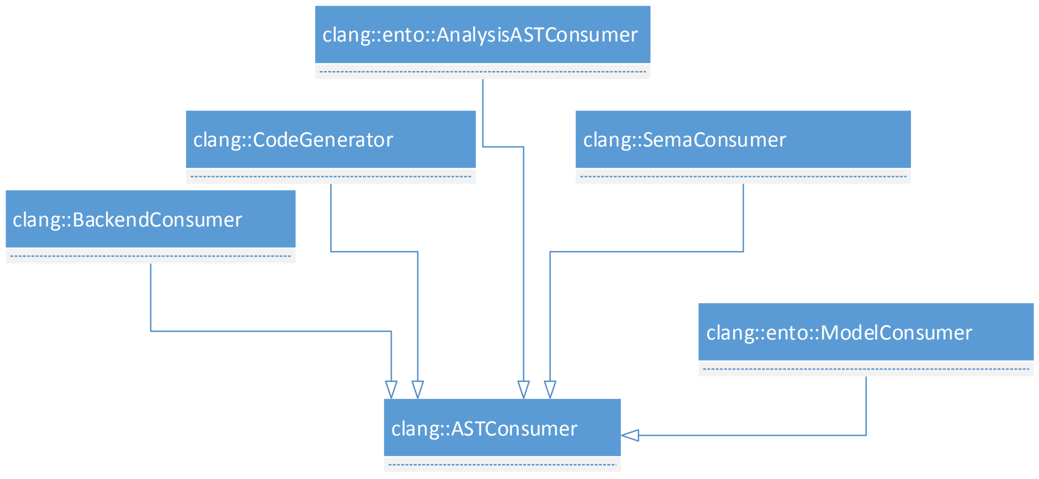


Figure 3.8: Inheritance diagram for the ASTConsumer class.

3.5 Transformations

According to [PLMK08], the three simple mutation transformations copy, delete and swap, in combination with a crossover transformation, are sufficient to evolve novel behavior [Sch14]. These four transformations do not use domain knowledge of the program which is manipulated. The transformations can be used on different program representations and different languages. In this work, AST and ASM are used as program representations. In [KBLN04] it is stated, that even human software developers use these kind of transformations commonly during programming. The fact, that these transformations do not produce new code is based on the intuition that „most extant programs already contain the code required to implement any desirable behavior related to their specification“ [Sch14].

3.6 Software Neutral Networks

In Chapter 2.10.2, the process of generating high level neutral variants is discussed as it is implemented by [Sch14]. In this work, Algorithm 2 is implemented, to allow the controlled generation of mutations. The method `mutateFile()` is explained in detail in Section 4.2.2 and in Section 4.2.4.

Algorithm 2 Evolution

```
 1: procedure EVOLUTION(maxMutationDepth, minimalNumberOfMutationsPerLevel,
    numberOfRandomMutations, workingDirectory)
 2:     while true do
 3:         mutationDepth ← getHighestMutationDepth(workingDirectory)
 4:         listOfMutations ← getListOfMutations(workingDirectory, mutationDepth)
 5:         if mutationDepth == maxMutationDepth ∧ length(listOfMutations) ≥ mini-
    malNumberOfMutationsPerLevel then
 6:             return
 7:         if mutationDepth ! = None ∧ length(listOfMutations) < minimalNumberOf-
    MutationsPerLevel then
 8:             mutationDepth ← mutationDepth −1
 9:             listOfMutations ← getListOfMutations(workingDirectory, mutationDepth)
10:         fileToMutate ← random(numberOfMutations)
11:         mutateFile(fileToMutate, numberOfRandomMutations)
```

The parameters can be used to control the evolution:

- **maxMutationDepth**: The procedure will apply the mutation operators, until the mutations are maxMutationDepth - steps away from the original program. In Figure 3.9, the correlation between maxMutationDepth and minimalNumberOfMutationsPerLevel, is shown.
- **minimalNumberOfMutationsPerLevel**: On each level, the same amount of mutations are generated. For the generation, a random mutation from the level below is used (algorithm 2, line 10)
- **numberOfRandomMutations**: This parameter is used for performance issues. The generation of mutations is very fast, since AST operations, or linear vector operations on ASM, are cheap. However, the verification of mutations is very slow. The extern programs for AST and ASM mutations are written in C++ and Python, so every time a mutation is planned, the program for mutation has to be loaded. To reduce the number of program loads, numberOfRandomMutations many mutations are generated at once. Figure 3.10 shows the selection and generation of mutations.

- **workingDirectory**: The whole diversity chain is based on a special data structure, as discussed in Chapter 4.

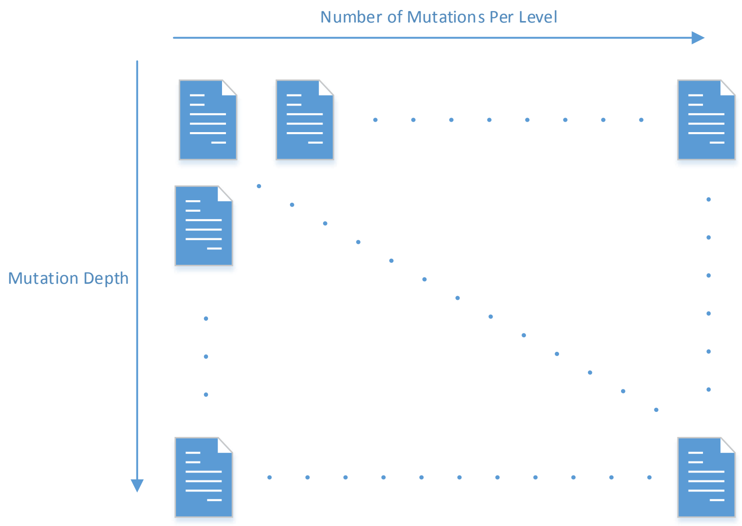


Figure 3.9: The mutation matrix to show the correlation between number of mutation per level and the mutation depth.

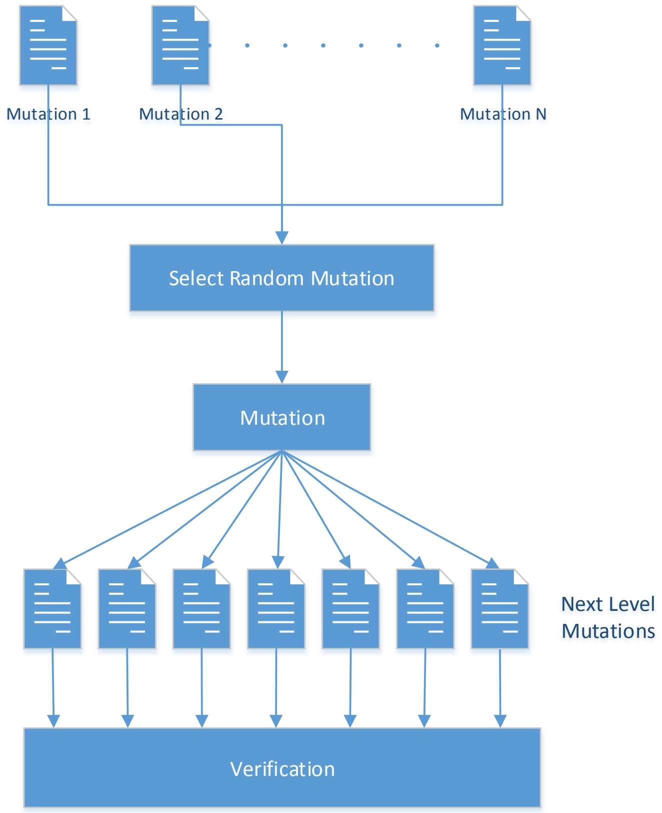


Figure 3.10: The visualization of the mutation step.

3.6.1 Fitness Evaluation

To test a given mutation of a program for its fitness or correctness, an executable has to be generated. The test-suite is then applied on this executable to evaluate the functional correctness of the mutation. Functional evaluation checks if the behavior of a program is correct or acceptable [Sch14]. No semantic tests are applied during functional evaluation, so in many cases the program variant computes a different function than the original. Furthermore, the formal program specification can not be checked during functional evaluation.

Nonfunctional evaluation, i.e. tests for nonfunctional runtime properties, are not considered in this work. In [Sch14] a post-compilation, workload-driven optimization technique for nonfunctional evaluation is presented.

3.6.2 Evolutionary Algorithm

As discussed in Chapter 2.8 an evolutionary algorithm consists of the following parts: Initialisation, Mutation and Selection. In Figure 3.11 the main parts of the evolutionary algorithm are shown. This algorithm is used to improve the hardware fault detection concerning register faults throughout this thesis. The data structure contains the inherent property that given a mutation m_1 with mutation depth x: $\texttt{depth}(m_1) = x$ so that $x > 1$ fulfills the property: $\forall y : \texttt{depth}(y) < \texttt{depth}(x) \rightarrow \texttt{diff}(x, y) = 0$. This property is used to avoid the creation of equivalent mutations and can be referred as a silent evaluation function. The actual evaluation function is made of register fault injection experiments. The more faults are detected, the better is the fitness of the given individual. Since a 1oo2 architecture is used the binary GO3F03 is chosen as the second binary, because of the results presented in Chapter 5.

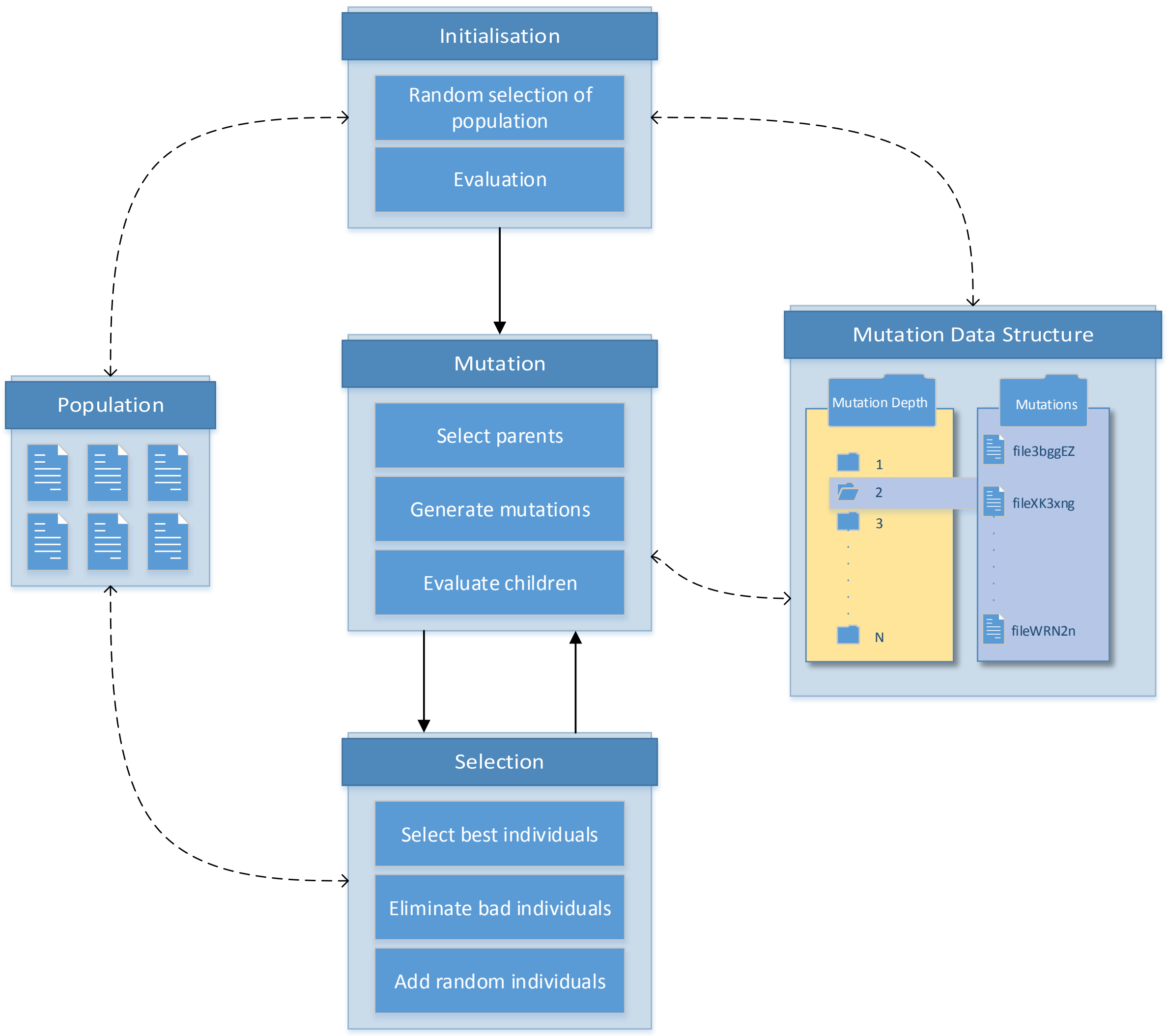


Figure 3.11: The visualization of an evolutionary algorithm used for experiments throughout this thesis.

3.7 Target System

The Freescale i.MX28 EVK PCB REV D board [Sem11] is used as target system, to run bare metal programs. An ARM926EJ-S core with 454MHz and 32KB cache is contained, as well as 128KB on-chip RAM. The NAND Flash socket can be used as data storage media or as boot device.

3.8 Software Evolution Library

The framework for automated software diversity presented in this thesis is influenced by software evolution library created by Eric Schulte [Sch14, SFF$^+$14]. This chapter gives a short overview of the main parts of the software evolution library.

The library is used in [Sch14, SFF$^+$14] to „enable the programmatic modification and evaluation of extant software“ [Sch14]. Developed during the genprog project, the software evolution library can be used for automated program repair. Many software objects, like linked ELF binaries, compiled assembler, LLVM IR, abstract syntax trees (AST), can be used due to a common interface. Methods for mutation and evaluation use this interface to support Search Based Software Engineering (SBSE).

3.8.1 Implementation

Common Lisp is used to implement the software evolution library. Many different types of software objects are supported:

- **ASM:** Assembler Code
- **CIL:** C Intermediate Language
- **CLANG:** C AST
- **LLVM:** Low Level Virtual Machine AST
- **ELF Mips:** Executable Linkable Format (ELF) binaries in MIPS architectures
- **ELF-x86:** Executable Linkable Format (ELF) binaries in x86 architectures
- **LISP:** Lisp Source

The following software methods are implemented on top of the abstraction: The function **genome** gives the data structure of the underlying object. A tree structure is used for high level source code, whereas linear structures represent low level objects like assembler code. A method to return an executable version of a given source code is called **phenome**. Deep copies are possible with **copy**. To select random elements of a genome, **pick** can be used. The function **mutate** mutates the software object by using one of the three specified mutation operators. To get the crossover of two software objects the function **crossover** can be used. Methods, used for file operations are:**from-file** and **to-file**.

To configure and interact with the search process, global variables are used. The list of currently used software objects is stored in the **population** variable. To limit the allowable population size **max-population-size** is used. During the selection, the variable **tournament-size** specifies the number of mutants. The fitness of individuals is compared with an operator, specified in **fitnes-predicate**. The chance of using cross over instead of mutation is defined in **cross-chance**. Each individual has a mutation rate of **mut-rate**. To track the number of fitness evaluations, **fitness-evals** is used.

There are two implemented search functins: **evolve** and **mcmc**. Evolve tries to imitate natural selection, whereas mcmc performs a markov chain monte carlo search [Gil05]. The runtime of these two methods can be limited by the maximum number of evaluations and the maximum number of passed seconds. An overview of the mentioned methods and variables of the software evolution library API can be seen in Figure 3.12.

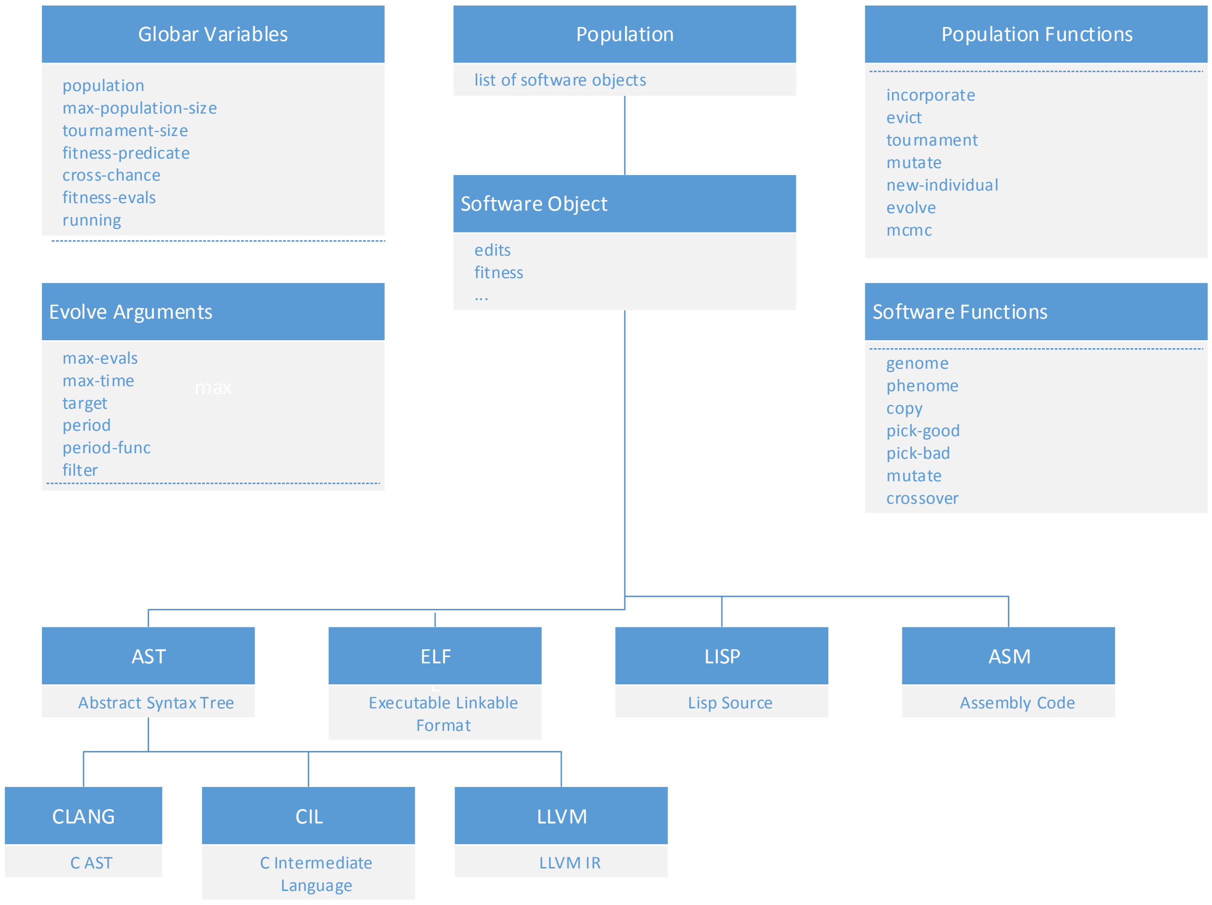


Figure 3.12: Software Evolution API.

Chapter 4

Implementation

In this chapter insights into implementation details for

- the diversification chain,
- the mutation and evolution algorithms,
- the simulation with FIES,
- the modifications on FIES, and
- the creation of the compiler toolchain

are provided.

4.1 Diversification Chain

The high-level aspects of the diversification chain are written in Python, whereas shell scripts are used for implementing low-level features. In Figure 4.1 the main parts of the implementation can be seen:

- **Python**:
  - The **evolution** module implements methods to automatically generate diverse versions of the specified program source. Methods of the simulation and wrapper modules are used.
  - The **simulation** module is capable of simulating different hardware errors and implements algorithms and methods for error dedection.
  - The **mutation** module is responsible for the mutation process and offers methods to compile and link a program with the specified mutations.
  - The **wrapper** is the interface between high-level aspects and low-level features.
- **Shell Scripts**:
  - The **file system** is directly used as information storage, too ease and fasten the **compilation** and mutation process.

  - To **verify** mutations, the output is compared to the original program, which is compiled with no optimization flags using the GCC compiler.
  - The **execution** of programs is done by calling Qemu with the specified parameters.

- **External Binaries**:
  - **Clang mutate**, based on [Sch15b], is used to mutate source code, by modifying the AST of input files.

- **LISP**:
  - The **software evolution library** [Sch14] can be used for mutation.

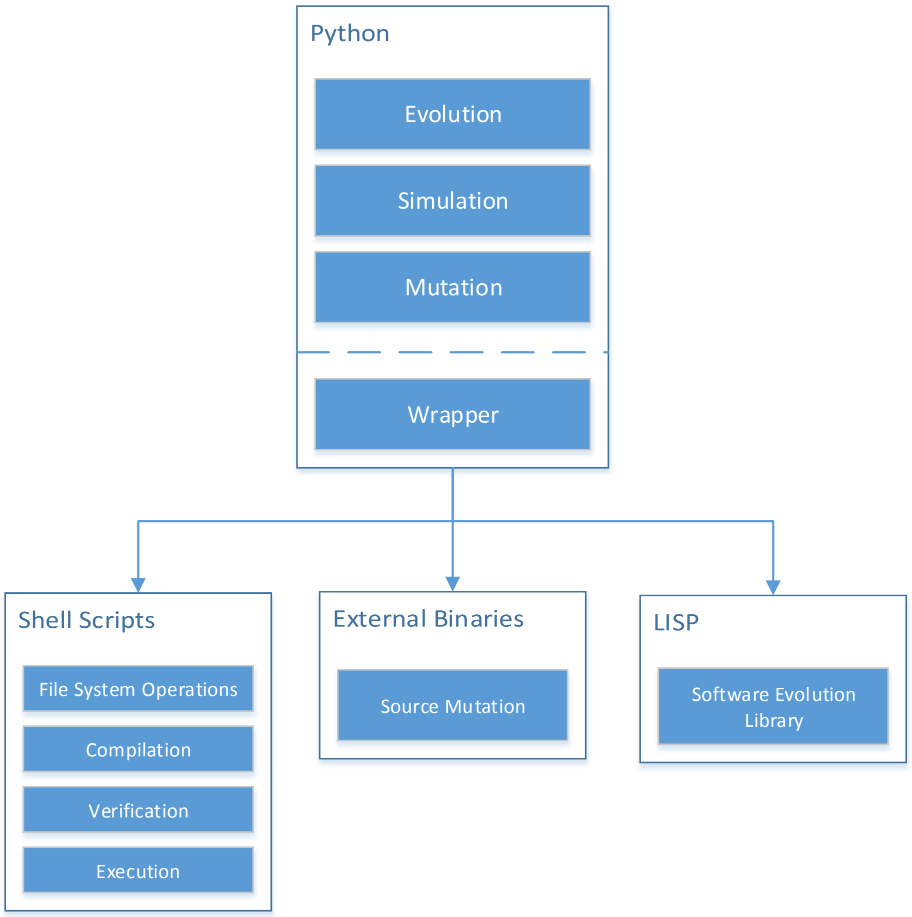


Figure 4.1: Overview of the implented modules, used languages and interactions.

4.2 Mutation

4.2.1 LLVM Source Mutation

To mutate C source files, the library LibTooling [Tea15a] is used to create a standalone clang-based tool. The mutation program is written in C++ and based on [Sch15b]. Revision 182049 of LLVM is used, which must be built with the `.configure` and `make` commands. Note that the framework takes a couple of hours for the compilation and installation process. The tool can be found in `~/diversity_chain/ClangMutate/clang-mutate/`. **To compile the tool, GCC has to be used whereas CLANG is needed for the linking process.** The shell scripts `build_clang.sh` and `build_gcc.sh`, provided in the `clang-mutate` directory can be used for this purpose.

A path to the source file and a number of mutations must be specified for the tool to work correctly. An example would be: `./clang-mutate -numberOfMutations 10 file.c --`. The `--` statement means, that there are no **translation database** available [Tea15g]. First the AST is traversed to count the available number of statements. Then, one of the mutation operators(delete, insert and swap) is applied in a loop until the expected number of mutation parameter is reached. **Note that there is a known bug, which randomly kills the AST structure, so there is no guarantee that the expected number of mutations is reached during on run of clang-mutate.**

All mutations are placed on the `/tmp` folder and renamed by the `std::tmpnam` function to get a list of files which can be used by the diversification chain.

4.2.2 High Level Source Mutation

To unterstand the sequence diagram, shown in Figure 4.4, the file structure is explained in Figure 4.2. A example project written in C is used with the three files: `Main.c`, `Calc.c`, and `Calc.h`. The file to mutate is `Calc.c` which contains a single function for doing some calculation. As it can be seen in Figure 4.2 a folder with the name `Calc.c_mutations` is generated by the diversification chain, containing a list of subfolders, where the name of the folder shows how many mutation steps the contained mutated files are away from the original version. In the example, **folder 2** is selected, giving mutation files which are two steps away from the base version. Each mutation is named randomly and comes with a mutation information file, where the predecessors of the file are listed to allow the generation of mutation trees.

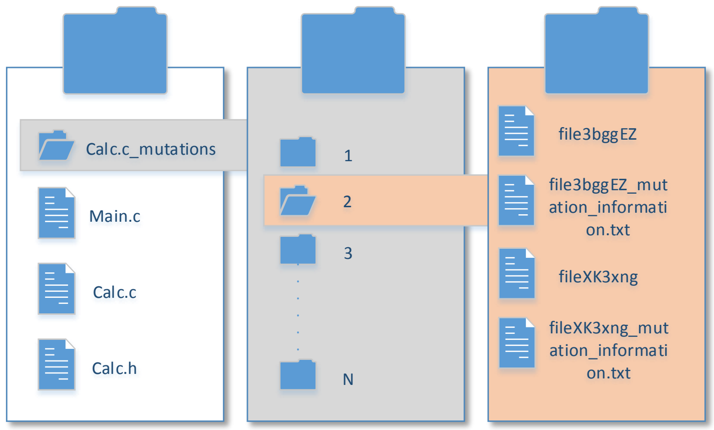

Calc.c_mutations
Main.c
Calc.c
Calc.h
1
2
3
N
file3bggEZ
file3bggEZ_mutation_information.txt
fileXK3xng
fileXK3xng_mutation_information.txt

Figure 4.2: The mutation folder structure of an example project generated by the diversification chain.

During the mutation process, a `temp` folder is generated, where the mutations are compiled and tested (Figure 4.3):

- `build`: Contains the precompiled object files which are not mutated during the current run. This is just a temporary folder since the object files are copied to the parent folder for further processing as it can be seen in Figure 4.3.
- `Calc.c_compiled_and_tested`: A folder where the tested binaries are located.
- `Calc.c_compiled_and_tested_source`: The source of the valid mutations of the current run are located in this folder.
- `Calc.c_compiled_mutations`: Contains successfully compiled mutations.
- `Calc.c_mutations_untested`: Untested mutations are first copied to this folder. This is the content of the `/tmp` folder, when clang-mutate is used for C-AST mutation.
- `Main.o`: The precompiled object file of a file, which is not mutated during this run.
- `Calc.c`: The current mutation which is tested.

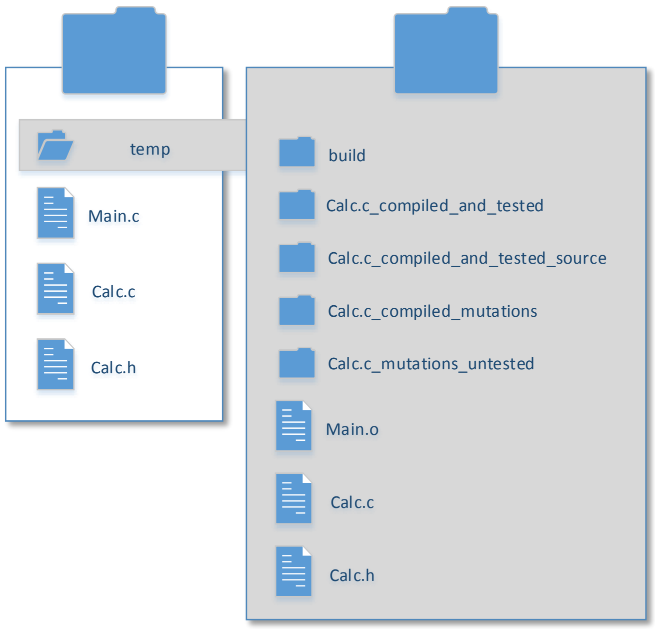

temp
Main.c
Calc.c
Calc.h
build
Calc.c_compiled_and_tested
Calc.c_compiled_and_tested_source
Calc.c_compiled_mutations
Calc.c_mutations_untested
Main.o
Calc.c
Calc.h

Figure 4.3: The folder structure of the temporary folder used for mutation, compilation and testing.

In Figure 4.4 a sequence diagram shows the generation of source code mutations. The Python module `mutator` contains a method `mutateSourceFile` with the parameters `pathToFile` and `numberOfMutations` to specify the file and the number of applied mutations. To communicate with the file system, a wrapper is used where shell scripts are called. First, the temporary project structure is created as discussed in Figure 4.3. For this purpose, the old folder is deleted and replaced with the current content of the project. Then, clang-mutate is used to generate the expected number of mutations. The wrapper opens a subprocess with the clang-mutate tool which creates the mutated files in the `/tmp` folder of the system in use. The mutations are then copied into the `/temp/Calc.c_mutations_untested` directory when the example is used as in the description above Figure 4.3. Files which are not mutated in this run can be precompiled to object files. The next step is time consuming because a loop iterates over each mutation m and tries to compile and verify the mutation. All valid mutations are then copied to the folder `Calc.c_compiled_and_tested_source` and then transferred to the corresponding file in the main project structure as it can be seen in Figure 4.2.

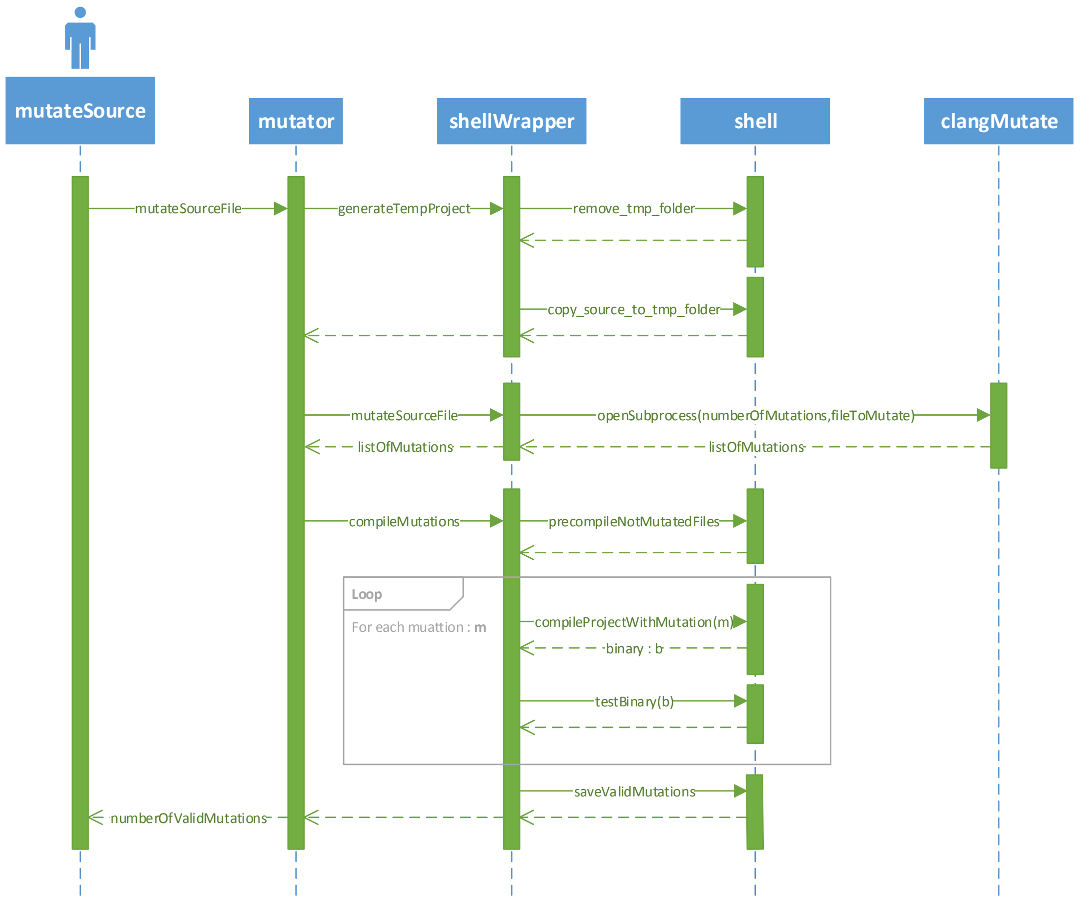


Figure 4.4: The sequence diagram of a source mutation call. Interactions between the Python module mutator and the wrapper are shown as well as interactions between the wrapper and the shell and the external binary clangMutate.

Fast Source Mutation

The process of compilation is very slow when the compiler `gcc-arm-none-eabi` (see Section 4.5) is used. Furthermore, the simulation on QEMU is very time consuming for generating test results. A method is implemented to avoid these time consuming tasks by simulating the program on the actual operating system. For this reason commonly used methods are rewritten and placed in the folder `~/imx28_simulation/`. The source files are compiled and tested with the GCC 4.9.2 compiler. Note that this approach only works if all hardware specific methods are simulated correctly and no precompiled libraries are used within the project. Obviously this approach can not be applied to assembler mutations since already cross compiled assembler code can not be simulated on the host system.

4.2.3 High Level ASM Mutation

Many methods which are used by the source mutation can be used directly for the ASM mutation, since the main approach is the same. Yet there are some important differences. To mutate assembler files, the source files have to be compiled before. The compilation process is described in the diverse compilation chapter. In Figure 4.5 a new folder named `asm` can be seen, where subfolders a listed with rising numbers. The numbers are chosen arbitrarily and have no deeper meaning in opposite to the `Calc.c_mutations` subfolders, where the name of the folder is equal to the number of mutation steps applied to the mutation. A file named `mutationInformation.txt` contains information about the compilation process (see Figure 4.7). In Figure 4.5 the files `Main.s` and `Calc.s` can be seen, which are generated by compiling `Main.c` and `Calc.c` with the `-S` flag by GCC or CLANG. Mutations of the `Calc.s` file are located in the folder `Calc.s_mutations` where the name of the subfolders contain information about the mutational distance to the unmodified file. In this example the contents of folder 2 are visible, where two mutations with the corresponding `mutation_information.txt` files can be seen.

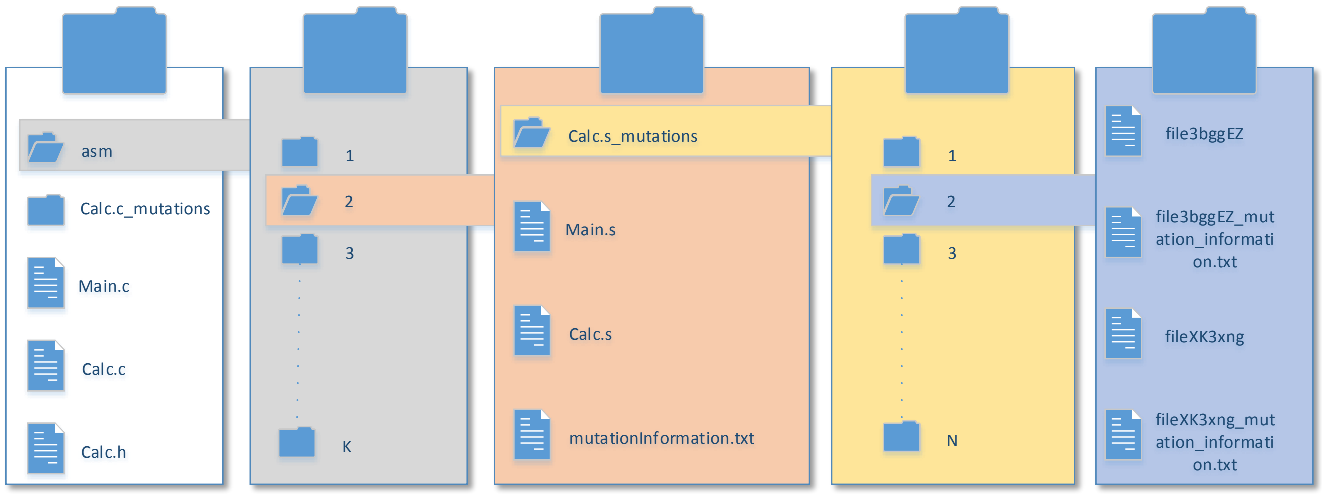


Figure 4.5: The folder structure of assembler mutations, where the second-order mutations of the second assembler project are listed. A mutation information file is used to save compiler options.

4.2.4 Low Level Mutation

Source Mutation

In Listing A.3 the method `mutateSourceFile(pathToFile, numberOfMutations, strictMode=True)` is responsible for the AST mutation of C-family files. The parameter `strictMode` is set to `True` by default. This means that all mutations have to be different when compared to all predecessors of the specified file. The list of predecessors is saved in the file `mutation_information.txt`(Figure 4.2). It is possible to mutate already mutated files so `pathToFile` can either point to the original file or to a mutation. According to

Figure 4.2 the mutations of a file are located in a special sub directory in the project folder. If the file is a mutation, the path to the original file has to be determined. The `/tmp/` directory of the system is used to temporarily save the mutations of file, so it has to be cleared at the beginning of this method. To avoid changes to the original project, a subfolder called `<path-to-project->/temp/` is created within the project file structure (see Figure 4.3). The mutated file is then copied to this folder and a sub-process is started to call the external mutation program with the specified number of mutations. All mutations are copied from the `/tmp/` directory to the `_mutations_untested` subfolder in the temporary project, compiled and tested. All successful mutations are saved to the according subfolder in the main project. The number of successful mutations is saved for statistical purposes.

Assembler Mutation

The mutation of of assembler files is done by the method `mutateAssemblerFile` (Listing A.3) and very similar to source mutation. In contrast to source mutation, a mutation information file is loaded with information about the compilation process necessary for the linking process. The mutation is done by a Python script so no external binaries are included.

4.3 Evolution

Algorithm 2 is implemented using Python(see Listing A.1, function `evolutionAlgorithm`). The sequence diagram can be seen in Figure 4.6, where the user starts the evolution Algorithm with the parameters:

- `pathToFile`,
- `numberOfRandomMutations`,
- `maximalMutationDepth`,
- `minimalNumberOfMutationsPerLevel`, and
- `mutationType`.

The evolution algorithm is capable of mutate a single source file which is specified by a string in the parameter `pathToFile`. The `numberOfRandomMutations` is an integer value specifying the number of mutations generated by the external mutation tool to boost the performance. The parameter `maximalMutationDepth` defines the stopping criteria of the evolution algorithm. When enough mutations are generated (as defined in `minimalNumberOfMutationsPerLevel`) on the final level the algorithm is stopped. The `mutationType` is a string which is either

- `Source`,
- `ASM`, or
- `SourceSimulationOnCurrentOS`

to tell the algorithm about the used mutation. The option `SourceSimulationOnCurrentOS` is used for fast source mutation and described in Chapter 4.2.2. In Figure 4.6 the evolution algorithm collects information about the current mutation depth and the current mutations by communicating with the wrapper. Then, a file to mutate is selected randomly. This file is mutated as described in the Chapters 4.2.3 and 4.2.2.

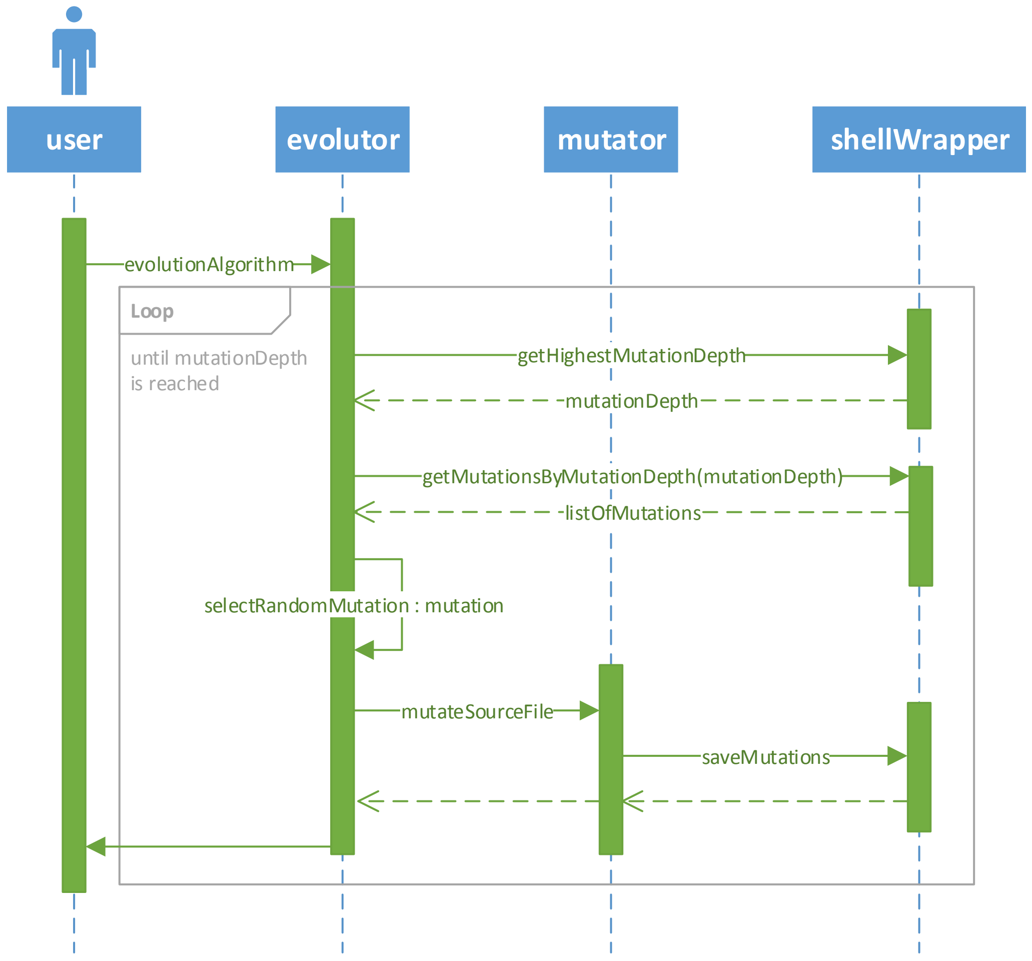


Figure 4.6: The sequence diagram of the evolution algorithm. Interactions with the python modules evolutor and mutator are shown as well as wrapper calls.

4.4 Simulation

4.4.1 Fault Injection Framework

The simulation of hardware errors is done with a fault injection framework based on QEMU [Sch15a]. The starterscript `automattest.sh` located in the QEMU folder, is used to start the simulation by specifying a binary, a fault XML file and a parameter for the used input (Section 4.4.4).

In Listing 4.1 the main file of a simple project can be seen. Arguments are used to transfer the input values to the program. The actual input is generated by calling `generateInput` with the parameter `argv`. The calculation is done in the method `doCalc` where the generated input is passed as parameter. Since no operating system is used on the target system the usage of argument values is not possible. For this reason, the fault injection framework is modified to allow the selection of the used input during runtime (see Section 4.4.2).

To simulate hardware faults during the execution of `doCalc()` the three variables `sbst_cycle_count`, `fault_counter` and `input_data_id` have to be defined and initialized according to Listing 4.2.

The variable `sbst_cycle_count` has to be initialized to -1, `fault_counter` needs to be set to 1 and `input_data_id` has to be set to -2. To retrieve the value for `input_data_id` as specified in the starter script, the method `initializeInputDataID()` is called (see Listing 4.3). This method increments `input_data_id` to signal FIES that the actual value can be stored in the variable. Since compiler optimizations are used throughout the experiments in this thesis, the dummy call of the method `writeIntegerToOutput(output, input_data_id)` is used to ensure that no math optimization is applied to the incrementation of `input_data_id`. The method `selectInput(input_data_id)` replaces the method `generateInput(argv)`. A mechanism to generate input data has to be provided within the program. The variable `input_data_id` can be used as seed for this mechanism or to chose between already defined test inputs.

To tell the fault injection framework about the start of the experiment `sbst_cycle_count` is used. This is realized as software to hardware communication [Sch15a]. For this reason the variable `sbst_cycle_count` is incremented before `doCalc()` is called (Listing 4.2 line 10). The variable `fault_counter` on the other side is used to collect the number of detected faults within software based self tests. Since no software based self tests are used within this thesis, the variable `fault_counter` is not used.

To terminate a fault injection experiment the `sbst_cycle_count` is incremented in a loop until no further fault injection experiments are defined within the used fault injection XML file (Listing 4.2 line 12-13). This while loop replaces the `return 0` statement. Increment the variable `sbst_cycle_count` within a infinite loop `while(1)` would cause the compiler to delete the code when the optimization flag `-O3` is activated. Furthermore, the infinite loop would not work in combination with the fast source mutation approach as described in Chapter 4.2.2 since no fault injection framework is used there. Therefore, the provided form with the incrementation in the loop header should be used. The number `1` is chosen arbitrarily and can be replaced with any number which is greater than 0 when only one fault injection experiment is provided within the XML file (Section 4.4.4).

Listing 4.1: The original main file of a simple project where only doCalc() is called.

```
#include "calc.h"

int main( int argc, char* argv[] ) {
  int *input = generateInput(argv);
  doCalc(input);
  return 0;
}
```

Listing 4.2: The modified main file of with the introduced variables used by the fault injection framework.

```c
#include "calc.h"

int sbst_cycle_count = -1;
int fault_counter = 1;
int input_data_id=-2;

int main(void) {
  initializeInputDataID();
  int *input = selectInput(input_data_id);
  sbst_cycle_count++;
  doCalc(input);
  while (sbst_cycle_count++ < 1) {
  }
}
```

Listing 4.3: A function for retrieving the selected input id.

```c
void initializeInputDataID() {
  // Signal QEMU
  input_data_id++;

  // This call avoids -O3 flag optimization
  writeIntegerToOutput(output, input_data_id);
}
```

4.4.2 Modifications on FIES

Input Selection

To enable the specification of the used input by the starter script (see Listing A.7) the following modifications are applied on the source code of FIES:

- `vlv.v`: In line 3679, the switch is extended to read the address of the variable `input_data_id` and the actual input id which is written to the address. The address and the value are specified by the starter script (see Listing A.7).
- `fault-injection-data-analyzer.c`: A new method is introduced (`set_input_file_to_use(int num)`) to actually set the variable `input_data_id` to the specified value.
- `fault-injection-controller.c`: The method `start_automatic_test_process` is adopted to call the method for setting the variable `input_data_id`. This is done by polling on the value of `input_data_id`. When the simulated binary sets the variable to -1 the value for the input variable from the starter script is assigned.
- `fault-injection-config.h`: The two variables `extern unsigned int file_input_to_use` and `extern unsigned int file_input_to_use_address;` are added.

Ubuntu 32-Bit

FIES uses a parser to read XML files which contains information about the introduced fault (see Section 4.4.4). To parse character arrays the method `strtol` is used with `long int` as return type. On the used host system, a 32-bit linux distribution, `long int` has 4 bytes. So numbers higher than `0x7FFFFFFF` can not be read by this function. Therefore the method `strtol` is replaced by the method `strtoul` which returns an `unsigned long int`.

Print Accessed RAM Addresses

Since the RAM address space is rather big it is not effective to use each register for the fault detection tests. Therefore a mechanism is implemented to enable the logging of accessed RAM addresses into a file. The following changes are introduced:

- `include/exec/softmmu_header.h`: The static inline function `glue` is extended with a conditional. When the variable `SAVE_ACCESSED_RAM_ADDRESSES_TO_FILE` is set to 1, the current RAM address is written to a file buffer using the hexadecimal representation.
- `fault-injection-config.h`: The two variables `extern unsigned int SAVE_ACCESSED_RAM_ADDRESSES_TO_FILE` and `extern FILE *outfile` are added. Furthermore the line `#define OUTPUT_FILE_NAME_FOR_ACCESSED_MEMORY_ADDRESSES "memory_addresses.txt"` is introduced to specify the name for the output variable.
- `fault-injection-controller.c`: The method `start_automatic_test_process` is adopted to open and close the the output file where the accessed RAM addresses are written to.
- `fault-injection-library.c`: The function `validateXMLInput` now supports `"PRINT ADDRESSES TO FILE"` as target. `SAVE_ACCESSED_RAM_ADDRESSES_TO_FILE` is set to 1, when the above defined target is specified in an XML file.

4.4.3 Python Simulator

For each simulated run a timeout can be specified. In Figure 4.7 a working directory can be seen on the left side with binaries and `.info` files which give information about the binary. In the current example GCC is used as compiler with the optimization flag `-O3`. The register `r3` is avoided by the compiler. The AST mutation depth is 50 and 20 mutations are applied on the ASM representation of the `calc.c` file. The binaries in a working directory can be filtered with the module `BinaryInformation.py`. To filter all binaries with source mutations `BinaryInformation.filterSourceMutation(listOfBinaries)` can be used on a list of binaries which is created by the method `Simulator.loadDirectory(pathToDirectory)`. To filter out binaries which contain source mutations and are compiled with the GCC compiler multiple filters can be used:

```
BinaryInformation.applyFilters([filterSourceMutation, filterCompilerGCC],
listOfBinaries)
```

Hardware errors are specified by a XML file which is called fault library 4.4.4. The generation of fault libraries is done dynamically using the `Simulator` module. To generate a fault library with a permanent register error on register 4, the command

```
Simulator.generateFaultLibRegisterPermanent("0x4", "0x00FF")
```

is used. The second parameter `0x00FF` specifies the erroneous content of the register. In Listing A.2, the method `generateFaultLibRegisterPermanent` is responsible for creating the necessary parameters to fill the template fault library with content. The wrapper is called to actually write content to the file.

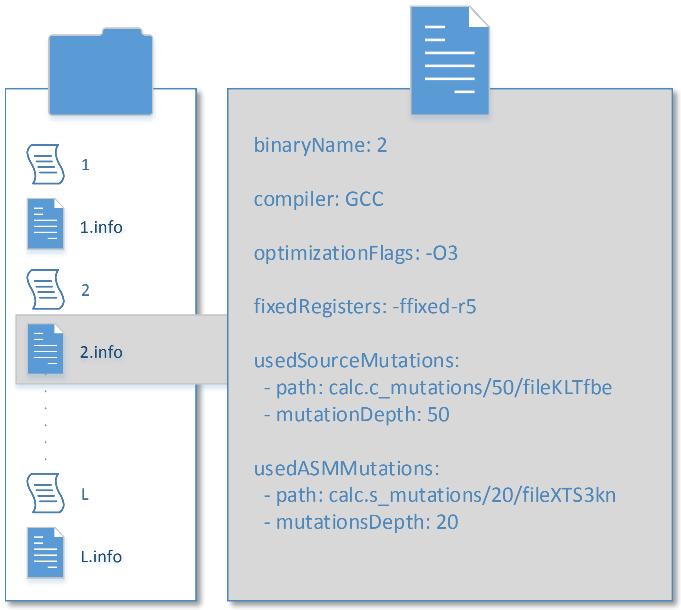


Figure 4.7: The simulation file structure can be seen on the left side. Each binary comes with a .info file holding information about the compilation process.

4.4.4 Fault Library

XML Templates

To define faults for the Qemu based fault injection framework [Sch15a], a XML format is used. Listing A.8 shows a template XML file for permanent register faults:

- `<component>`: The used entry `REGISTER` specifies the faulty component. `CPU` and `RAM` are supported too.
- `<target>`: The target within the component. `REGISTER CELL` is used in Listing A.8. For register components `ADDRESS DECODER` would be another supported target. For CPU faults the target can be `INSTRUCTION DECODER`, `INSTRUCTION EXECUTION` or `CONDITION FLAGS`. In Listing A.9 an instruction decoder fault is shown with `{address}` as placeholder for the faulty instruction. RAM faults can target the `ADDRESS DECODER` or a single `MEMORY CELL`.
- `<mode>` is used to define the fault mode. In Listing A.9 the `NEW VALUE` mode is used.
- `<trigger>`: To trigger faults `ACCESS`, `TIME`, or `PC` can be used. In this work the access trigger is used exclusively.
- `<type>`: `PERMANENT`, `TRANSIENT` and `INTERMITTEND` fault types are supported by the library. In Listing A.9 a permanent fault is introduced. For transient faults, a `<duration>` has to be defined and intermittend faults require the `<interval>` field.
- `<params>` is used to define the parameter description:

  - `<address>`: Contains a hexadecimal value to describe a memory address. For permanent register faults the registers from `0x1` to `0x15` are supported.
  - `<mask>`: For permanent register faults (Listing A.8) the inserted value is defined by this tag.
  - `<instruction>`: Specifies the instruction to be inserted. In Listing A.9 a NOP operation is created by using `0xDEADBEEF` within this tag.

Simulation with Python

The folder `diversity_chain/qemu_fault_libs` contains template XML files for fault intruduction which are used by the diversity chain.

- `instruction_decoder_nop.xml` represents a permanent instruction decoder fault by introducing NOP instructions. The faulty instruction can be specified by using the placeholder `address`. In Listing A.2 the python based simulator is shown with the method `generateFaultLibInstructionDecoderNOP(instructionToReplace)` which takes an instruction address and returns the path to the generated fault library.

- `no_error.xml` is used to generate reference outputs of binaries. No error is introduced.

- `register_permanent_template.xml` defines a permanent register fault. In Listing A.2 the method `generateFaultLibRegisterPermanent(address, mask, faultMode)` generates the path to the fault library by taking the parameters `address` and `mask` which are hexadecimal numbers. A fault mode as listed in `simulation/FaultModes.py` has to be used. The XML template can be seen in Listing A.8

- `print_addresses_to_file.xml` is used to tell FIES to print all accessed RAM addresses. See Section 4.4.2 for all implementation details.

- `ram_address_decoder.xml` introduces a fault into the RAM address decoder. A mask and the bits to set have to be specified when this fault is generated.

- `ram_memory_cell.xml` adds a fault to the specified RAM register cell. A mask and the bits to set have to be specified when this fault is generated.

4.5 Toolchain

4.5.1 Packages and Files

To compile programs for the Freescale i.MX28 EVK PCB REV D board, a GCC bare metal cross compiler for embedded ARM-Cortex chips is used. The exact version number of the **gcc-arm-none-eabi** package is 4.8.2-14ubuntu1+6. The C library and math library are included in the package **libnewlib-arm-none-eabi**, with version number 2.1.0-3. Important libraries are missing, to create a tool chain for the Freescale board, which are taken from the Sourcery CodeBench [Gra15] to fill this gap:

- The **armv5te** folder, located in `/usr/lib/arm-none-eabi/lib/armv5te`

- The startup object files in the folder `~/imx28/startup`
- The register header files from the folder `~/imx28/registers`
- The linker script for Freescale i.MX233 EVK `~/imx28/imx28evk-ram.ld`
- The debug output method:
  - `~/imx28/debug_uart.c`
  - `~/imx28/debug_uart.h`
- The following files, object files and libraries:
  - `~/imx28/arm-names.inc`
  - `~/imx28/crtbegin.o`
  - `~/imx28/crtend.o`
  - `~/imx28/crti.o`
  - `~/imx28/crtn.o`
  - `~/imx28/libcs3.a`
  - `~/imx28/libcs3arm.a`
  - `~/imx28/libcs3hosted.a`
  - `~/imx28/libcs3lpc21xx.a`
  - `~/imx28/libcs3unhhosted.a`
  - `~/imx28/libg.a`
  - `~/imx28/libm.a`
  - `~/imx28/libstdc++.a`
  - `~/imx28/libsupc++.a`
  - `~/imx28/specs.o`

4.5.2 Compilation

GCC

To cross compile C programs with the GCC, the `arm-none-eabi-gcc` command is used. In listening A.4, the parameters for the command can be found from line 24 to line 30 [Fou15]:

- **FILES_TO_COMPILE**: Contains a string of file names.
- **-w**: Inhibit all warning messages
- **-T** `<script>`: The `<script>` is used as linker script.
- **PATH_TO_PRECOMBILED_IMX28**: This is the path to the shared object files, which are precompiled for each combination of compiler, optimization flag and fixed register.

- **-Xlinker**: Allows to forward options to the linker.
- **-nostartfiles**: The standard system startup files are not used by the linker.
- **-marm**: Enables ARM options.
- **-lm**: Link the math library.
- **ADDITIONAL_BUILD_PARAMETERS**: The `-ffixed-reg` flag is set via this shell variable.

CLANG

The package `clang-3.6` is used for the compilation process, but linking is done by the GNU linker, since the LLVM linker is not capable of using the provided linker script. The command to link the object files can be found from line 36 to line 43 of the listening A.5 Note that a patch file has to be applied, since some built-in macros are optimized for GCC and have to be changed in the file: `/usr/arm-none-eabi/include/machine/_default_types.h` [Nel15], otherwise unknown type name errors are thrown by the compiler. In Listing A.5, the build command can be found from line 28 to line 32 [Tea15e]:

- **-target** `<triple>`: The target architecture can be defined by a triple, with the format `<arch><sub>-<vendor>-<sys>-<abi>`.
  In this work, the triple `armv5te-none-eabi` is used to specify the system, by informing the compiler that an embedded-application binary interface is used with no system. Furthermore the ARMv5TE architecture is specified.
- **-mcpu** `<cpu-name>`: `arm926ej-s` is used as CPU name.

4.5.3 Compile to ASM

To enable assembler mutations the `-S` flag of the GCC and CLANG compilers is used to compile source code to assembler code. In Listing A.6 the `COMPILER_STRING` contains the command line instruction for the compilation process. CLANG needs the explixit inclusion of `/usr/lib/arm-none-eabi/include/` and `/usr/lib/gcc/arm-none-eabi/4.8.2/include/` to work properly.

4.5.4 Precompilation of Libraries and Shared Resources

Too speed up the process of compilation, common methods and libraries are precompiled. The number of precompiled versions is high, since each combination of optimization level, fixed register and compiler generates different versions. In Figure 4.8, an overview of the directory structure is given, where the libraries for the command `gcc -O0 -ffixed-r3` are selected. It is not possible to compile the startup assembler files with the CLANG compiler, so the GCC compiler was used for this directory. Furthermore it is not possible, to use the `-ffixed-register` flag with the CLANG compiler, so register avoidance is limited to GCC.

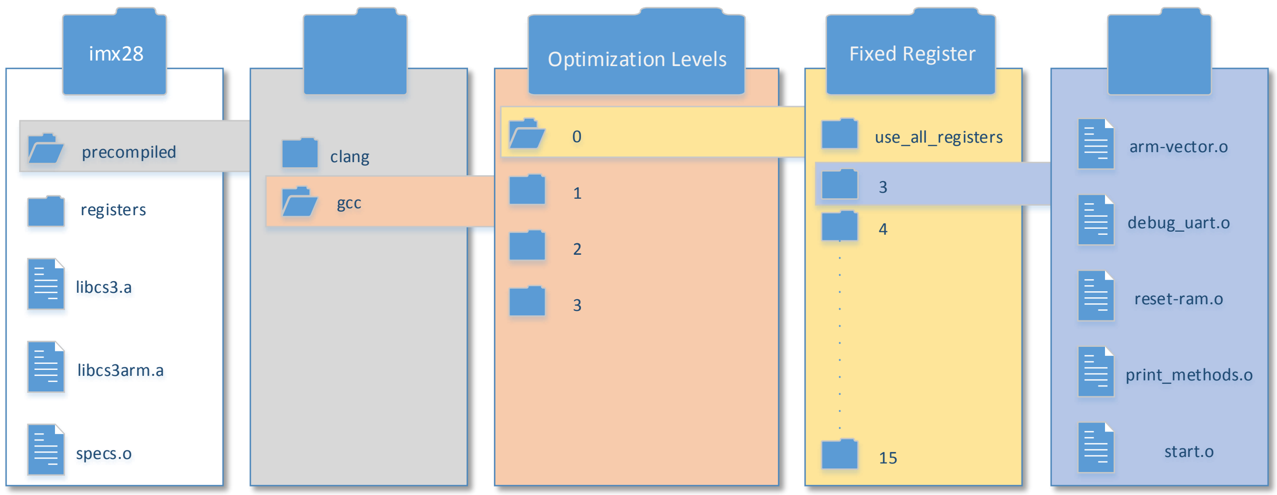


Figure 4.8: The folder structure of the precompiled libraries and common methods.

4.6 Restrictions

- It is not possible to compile the imx28 startup assembler files with CLANG so GCC was used for the files in the directory `~/imx28/startup/`.
- The C++ program used for c-ast mutation must be called within a loop, since excessive AST transformations can result in program termination.
- The function `strlenOwn(portCHAR *string)` in `/imx28/debug_uart.c` is used to avoid external libraries in combination with register avoidance strategies.
- In `imx28/print_methods.c` output functions are collected to avoid library usages.
- Precompiled versions of the imx28 library only exist for each pair of (Optimization Level, Fixed Register) for GCC. Combinations of fixed registers are not used at the moment, so it is not possible to use precompiled libraries when more than one register is blocked.
- CLANG does not support the `-ffixed-reg` flag since it requires additional LLVM backend support [Tea15b].
- Parallel simulation is not supported at the moment.

4.7 Installation

The following parts are necessary for the diversification chain to work properly:

- FIES: A fault injection tool based on Qemu [Sch15a].
- The LLVM compiler infrastructure 3.4 revision 182049

- Python 2.7.6
- The `gcc-arm-none-eabi` package with version number 4.8.2-14ubuntu1+6.
- Additional libraries from the Sourcery CodeBench as described in Chapter 4.5.

Chapter 5

Results

In this chapter the results of this thesis are presented. The diverse compiling approach is compared to unsound randomization methods regarding fault detection. Furthermore, the diverse compiling approach is extended with unsound randomization methods to improve the results. Fault recovery experiments are shown with promising results. Additionally, the performance of creating software mutations is presented.

5.1 Output Methods

The output functionality on embedded systems is very restricted when no operating system is used. `snprintf` can be used to print a formatted string into a buffer. Altough, this approach induce the the import of `<stdio.h>`. Since precompiled libraries can not be diversified, basic output functionality is provided in the files `~/imx28/print_methods.h` and `~/imx28/print_methods.c`. The exact knowledge of the size of different data types on the target system is compulsory. For this reason `print_methods.h` contains definitions of the most common data types and their size in bytes for the imx28 board.

Data Type	**Size in Bytes**
char	1
int	4
long	4
long long	8
float	4
double	8
long double	8
int*	4

Table 5.1: Size in bytes of the most common data types.

5.2 Test Programs

Fifteen different programs out of the categories **Automotive and Industrial Control**, **Network** and **Telecommunications** are used for simulation and mutation. Many programs are taken from the MiBench, which is „a free, commercially representative embedded benchmark suite“ [GRE+01]. To enable fault injection, the modification presented in Chapter 4.4.1 are introduced to every test program. Furthermore comments are removed from the files which are mutated.

5.2.1 Automotive and Industrial Control

The MiBench [GRE+01] offers algorithms which are frequently used on embedded processors in embedded control systems.

Basicmath

Two programs are used for solving a cubic polynomial and for calculating the integer square root to simulate basic math algorithms without hardware support.

- **Cubic**: A cubic polynomial with the form $f(x) = ax^3 + bx^2 + cx + d$ is solved by providing the variables a, b, c and d. Used data types are `double` and `long double`. According to Table 5.1, both datatypes are the same size, eight bytes. The library functions `pow`, `sqrt`, `fabs`, `acos` and `cos` are used within this program.
- **Integer Square Root**: A number x of type `unsigned long` is taken to calculate $\lfloor \sqrt{x} \times 2^{16} \rfloor$ by using solely bit shift operators without multiplications or divisions.

Sorting Algorithms

The three different sorting algorithms are capable of sorting an array of integer numbers. These algorithms are not part of the MiBench.

- **Quicksort**: A quicksort implementation ([Com15]) is taken from the apple open source collection [Inc15]. The implemented source mutation (see Section 4.2.2) makes use of a CLANG based formatting tool ([Tea15d]) to ease the comparisons of mutations. Therefore, some definitions have to be changed to put up with the tool. The core algorithm remains unchanged.
- **Merge Sort**: The merge sort implementation from [Cpr15a] is used, where only the call to `sizeof(int)` is replaced with the defined number `SIZE_OF_INT` from `print_methods.h` (see Table 5.1) to avoid the import of libraries.
- **Heap Sort**: The unmodified heap sort source code from [Cpr15b] is taken.

Bit Counter

Eight different implementations of bit counters taken from MiBench [GRE+01] are used to test different bit manipulation approaches. The output functionality is replaced, so no additional libraries are used. The sixth and seventh implementation are adopted to support negative values.

5.2.2 Network

Algorithms often used on embedded processors in networks are used in this section.

Dijkstra

The source code is taken from the MiBench [GRE+01] and modified by changing the output methods since many `printf` statements are used. Furthermore `NULL` is replaced by `(void *)0` to avoid the import of libraries. The input consists of a adjacency matrix provided by a two dimensional integer array with 50×50 entries. No additional libraries are used by this program.

5.2.3 Telecommunications

To find the frequencies of a given input signal Fast Fourier Transformations are often used in the telecommunication category.

Fast Fourier Transform

A polynomial function is used for the input of the Fast Fourier Transformation algorithm taken from the MiBench [GRE+01]. Only the output functionality is adopted to allow the usage on the imx28 board. The last part is difficult, since the `snprintf` function delivers different output when compiled with `-O0` and `-O3`. Therefore the bit representation of the floating point numbers are used for comparison because even build in functions like `floorf` give different results depending on the optimization level. The output function `writeUnsignedToOutput` located in `~/imx28/print_methods.c` is used to print the bit representation coded as unsigned value to a buffer.

5.3 Fault Detection

In Chapter 2.2.1 *M-out-of-N* architectures are discussed. Throughout this experiment, a 1oo2 architecture is used as presented in [HRIK15]. All fifteen programs listed in Chapter 5.2 are mutated as described in Chapter 5.5. The following single cell FFMs are used as presented in Chapter 2.5.1: Stuck-at faults, Incorrect read faults, Write disturb faults, Read disturb faults, Deceptive read faults.

In Figure 5.11 the percentage of detected faults can be seen. The registers 0 - 15 are used in combination with the above described FFMs and different inputs, giving a total number of 3600 fault injection experiments. For the experiment three different binaries are used for each of the test programs: `GO0`, `MGO0` and `GO3` (see Table 5.2).

Binary Name	**Compiler**	**Flags**	**Mutations**
GO0	GCC	None	None
GO3	GCC	`O3`	None
MGO0	GCC	None	Source
MAGO0	GCC	None	ASM
MSAGO0	GCC	None	Source, ASM
GO3F03	GCC	`O3`, `ffixed-r3`	None
CO0	CLANG	None	None
CO3	CLANG	`O3`	None
MCO0	CLANG	None	Source

Table 5.2: Shortcuts of binaries which are used in the diagrams throughout this chapter compiled with different flags and mutations.

Shortcut	**Program Full Name**
BC1 - BC8	Different bit counter variants
HS	Heap sort
BMSR	Basic math square root
MS	Merge sort
QS	Quick sort
D	Dijkstra
BMC	Basic math cubic
FFT	Fast fourier transformation

Table 5.3: Shortcuts of test programs which are used throughout this chapter.

To allow the measurement of the impact on the fault detection, the following categories of outputs are considered:

- **Crash**: An execution leads to a timeout, segmentation fault, or bad RAM pointer.
- **Output**: An execution that does not crash gives an output D_i which differs from the reference output (Byzantine fault).
- **Golden**: An output is called golden G when the fault is masked.

When the output of two binaries ($Out(B_1) = r_1$ and $Out(B_2) = r_2$) is compared, the following scenarios **improve the fault detection** [HRIK15]:

- $D_i \neq D_j$: The execution does not lead to a crash and gives different outputs r_1 and r_2 with $r_1 = D_i \wedge r_2 = D_j \wedge r_1 \neq G \wedge r_2 \neq G$.
- $r_1 = C \wedge r_2 \neq C$: The same fault gives different results.
- $r_1 = G \wedge r_2 \neq G$: The fault is masked by one execution.
- $r_2 = G \wedge r_1 \neq G$: The fault is masked by one execution.

The **fault detection is not improved** when:

- $r_1 = r_2 = G$: The fault is masked by both execution.
- $r_1 = D_i \wedge r_2 = D_i$: The execution does not lead to a crash but gives the same output D_i with $D_i \neq G$.
- $r_1 = C \wedge r_2 = C$: Both executions lead to a crash.

5.3.1 RAM Faults

In this chapter the effects of RAM address decoder faults and RAM memory cell faults are discussed concerning fault detection. For this reason the used fault injection framework is modified as discussed in Chapter 4.4.2. Only RAM addresses which are actually accessed by the binaries are tested. Therefore each of the binaries is executed with different inputs and without introduced faults to detect the accessed RAM addresses. Out of this list, 250 RAM addresses are selected randomly. To simulate permanent stuck at faults a mask is chosen randomly to select one of the eight bits. The chosen bit is then set to zero or to one, according to a randomly chosen Boolean value. A RAM fault is a triple consisting of a RAM address, a mask and the bit to set. All in all 112500 fault injections are performed to evaluate the presented approaches.

In Table 5.4 the percentages of masked RAM faults for each tested compiler combination can be seen. The proportion of masked faults between binaries using the `-O0` optimization flag and the single binary using the `-O3` flag is similar to the results of instruction decoder faults (see Table 5.6). The selection of the introduced RAM faults is one explanation for this outcome. RAM addresses are chosen randomly from one of the binaries. So it seems, that the accessed RAM addresses of the binary with the `-O3` flag are different than the accessed RAM addresses of the other binaries.

The binary with assembler mutations (`MSAGO0`) and source mutations masks only 70 % of all introduced faults, while the not mutated binary (`GO0`) is capable of masking 81 % of the introduced faults. The different memory behavior can be used for fault detection. This can be seen in Figure 5.3 and in Figure 5.4 which show the detected faults in percent for each of the $\binom{5}{2} = 10$ different binary versions. According to the results shown in Figure 5.3 and in Figure 5.3 it is possible to improve the `GO0/GO3` combination by 8% for RAM register cell faults and by 3% for RAM address decoder faults using a binary with assembler and source mutations `MSAGO/GO3`.

	GO0	**MGO0**	**MAGO0**	**MSAGO0**	**GO3**
Register Cell Faults	81%	77%	77%	71%	93%
Address Decoder Faults	92%	92%	90%	89%	95%

Table 5.4: Masked RAM faults listed for all binary categories.

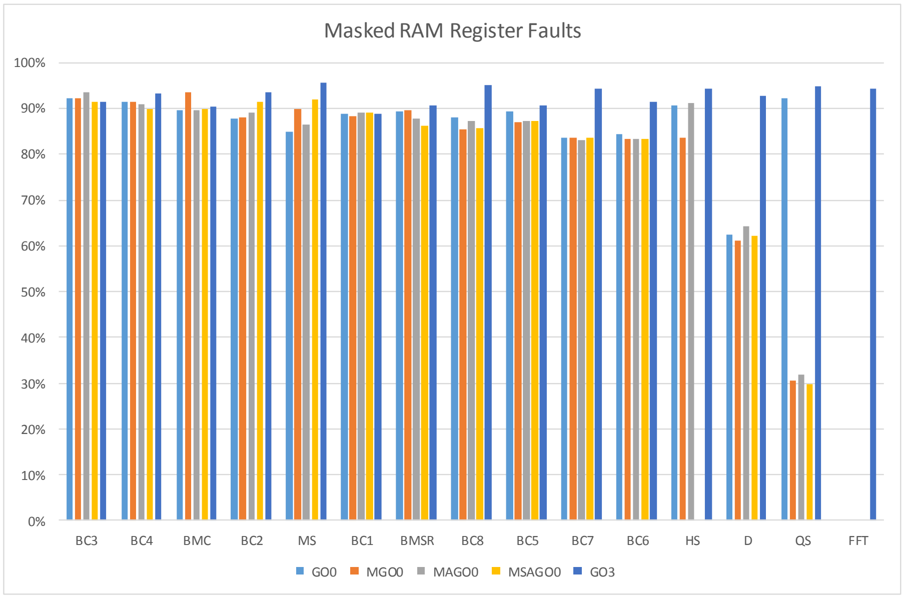


Figure 5.1: The possibility that a RAM register cell fault is masked grouped by the five approaches.

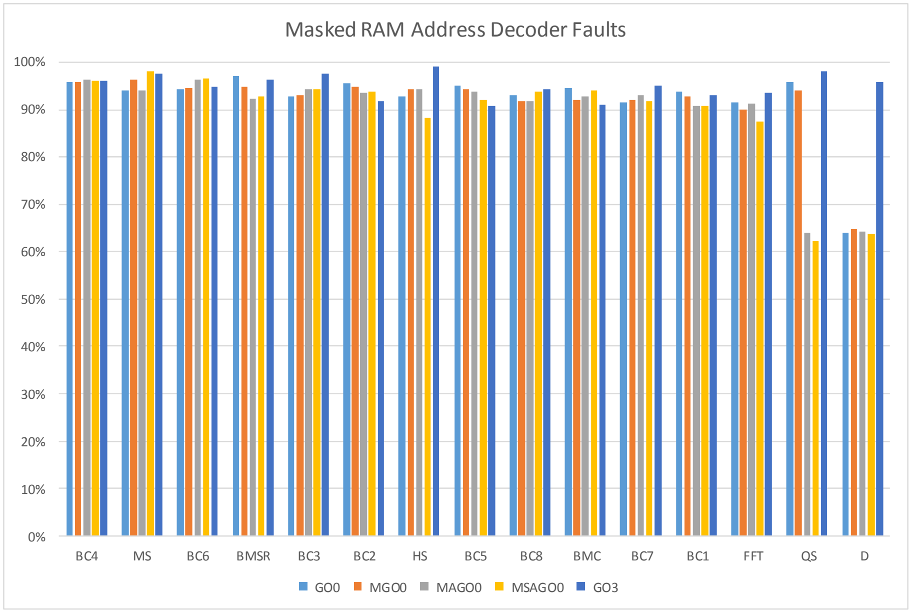


Figure 5.2: The possibility that a RAM address decoder fault is masked grouped by the five approaches.

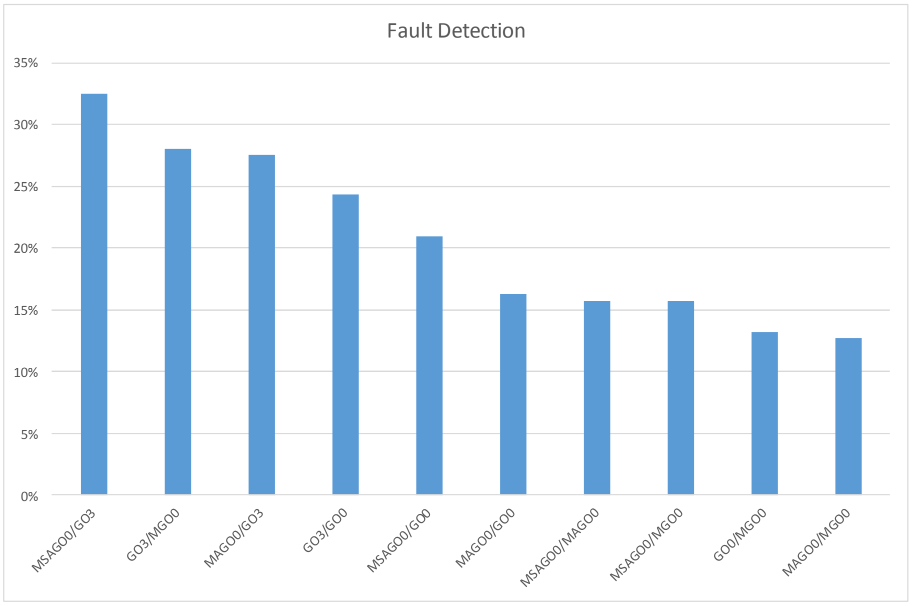


Figure 5.3: The possibility that a RAM register cell fault is detected grouped by all compilation combinations.

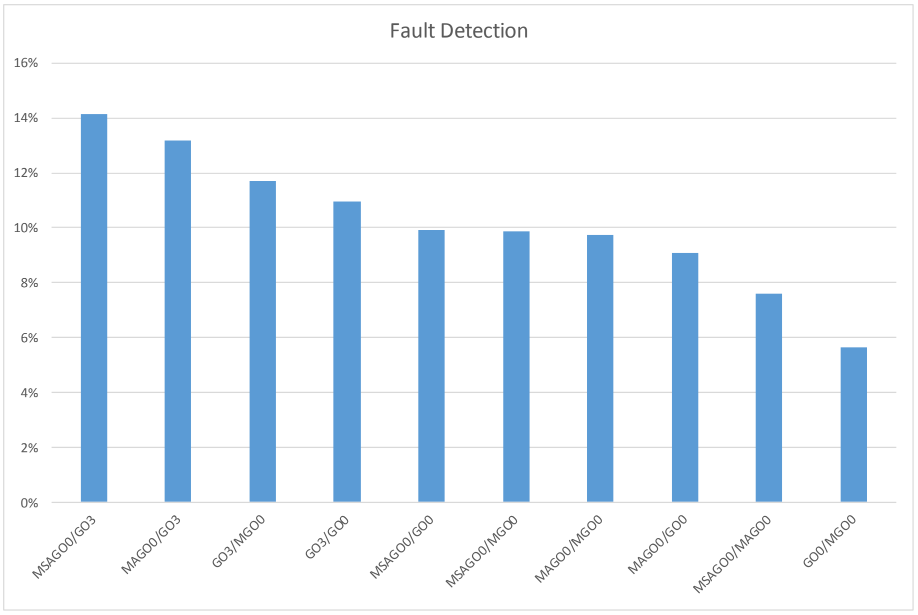


Figure 5.4: The possibility that a RAM address decoder fault is detected grouped by all compilation combinations.

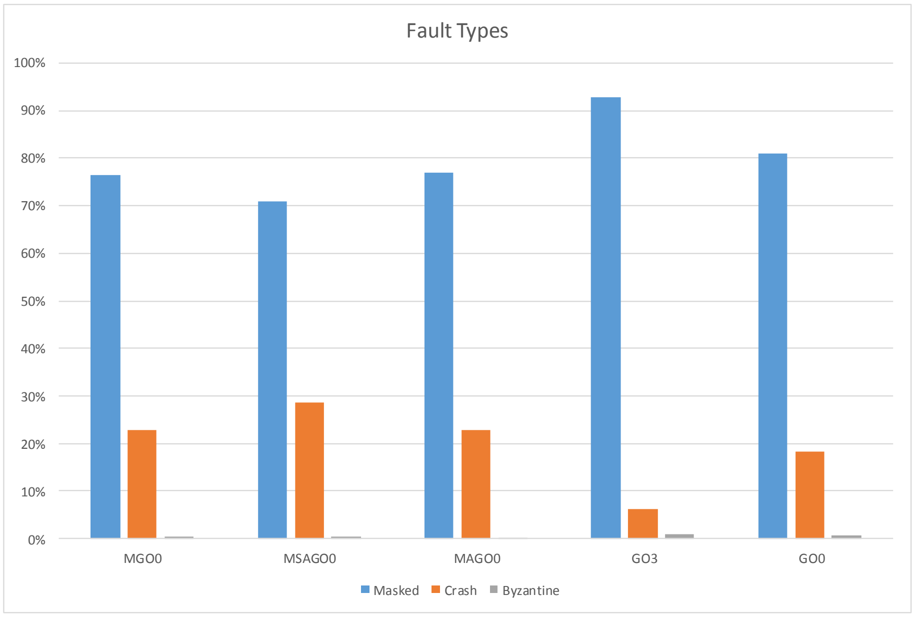


Figure 5.5: Proportion of different effects of RAM faults.

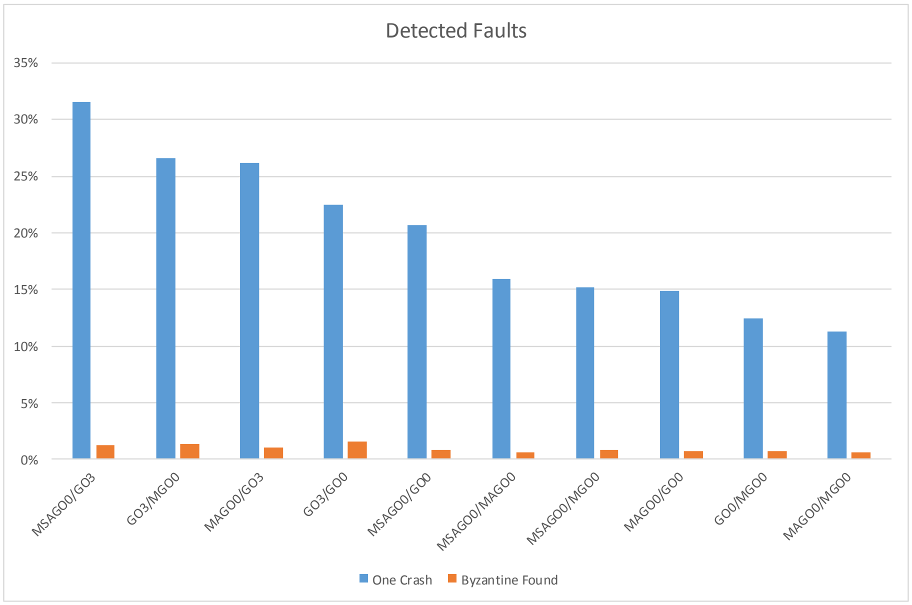


Figure 5.6: Proportion of detected RAM register cell faults.

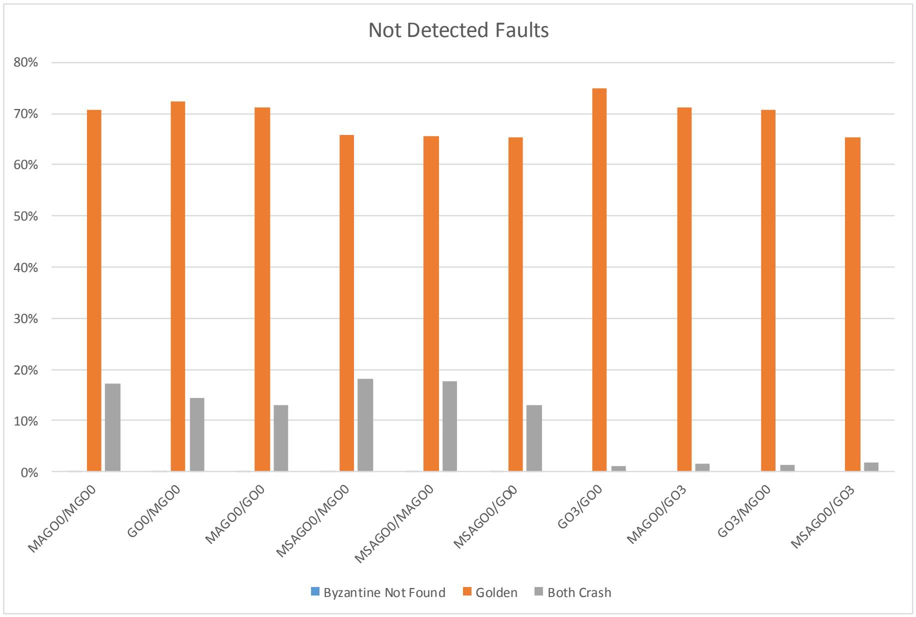


Figure 5.7: Proportion of not detected RAM register cell faults.

5.3.2 Register Faults

The results in this section are created by using source mutations and assembler mutations as described in Section 5.5. Mutations which are 100 steps away from the original version are created. Figure 5.9 show the percentage of masked faults for each register from 0 to 15 and for the three different binary categories `MGO0`, `MAGO0`, `MSAGO0`, `GO0` and `GO3`. It can be seen that the masked register faults are not equally distributed among the registers. In Figure 5.10 the percentage of masked register faults is shown for each test program. It can be concluded that the usage of the `-O3` flag in combination with the GCC compiler makes the resulting binary more vulnerable to register faults than the non optimized version (created with the flag `-O0`). Overall 18000 faults are injected to generate the results.

MAGO0	**MGO0**	**MSAGO0**	**GO0**	**GO3**
42%	42%	41%	42%	13%

Table 5.5: Masked register faults listed for all binary categories.

In Figure 5.8 the detected faults in percent can be seen evaluated for all registers, programs and input data. The pair `MSAGO0/GO3` detects nearly 64% of all injected register faults. This number is equal to the detected faults using `GO0/GO3`. In Figure 5.15 it can be seen that the detection rate is slightly different for each of the registers. The usage of assembler mutations without the `-O3` flag enables to detect 10% of the injected faults, while source mutations are able to find 8%. Merging register and source mutations allows to detect 13 % of all injected register faults.

The fault detection in absolute numbers for different fault modes can be seen in Figure 5.13 and in Figure 5.14 for the two combinations `MSAGO0/GO3` and `GO0/GO3`. Figure 5.16 shows that the influence of the used input values is negligibly.

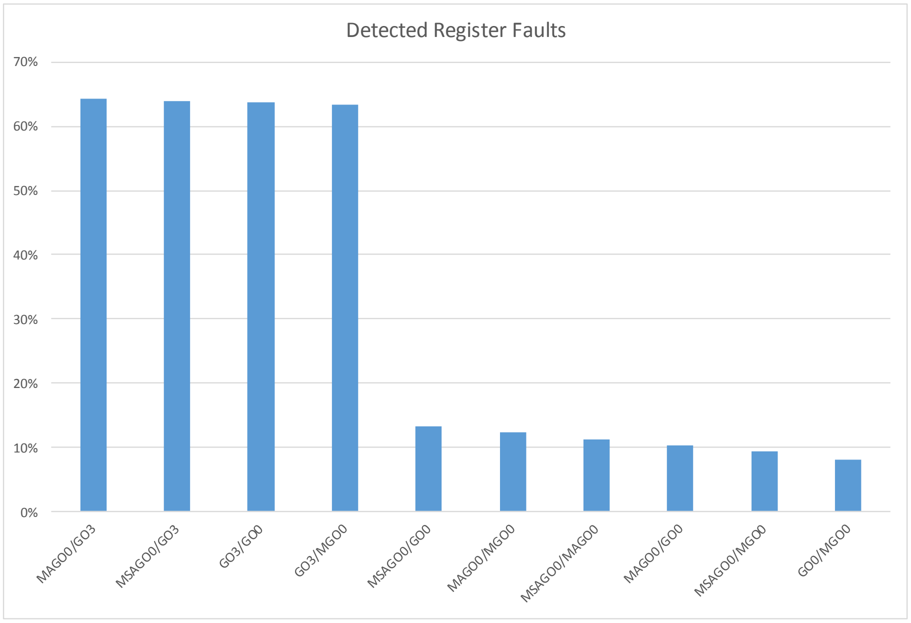


Figure 5.8: Detected register faults listed for all binary categories.

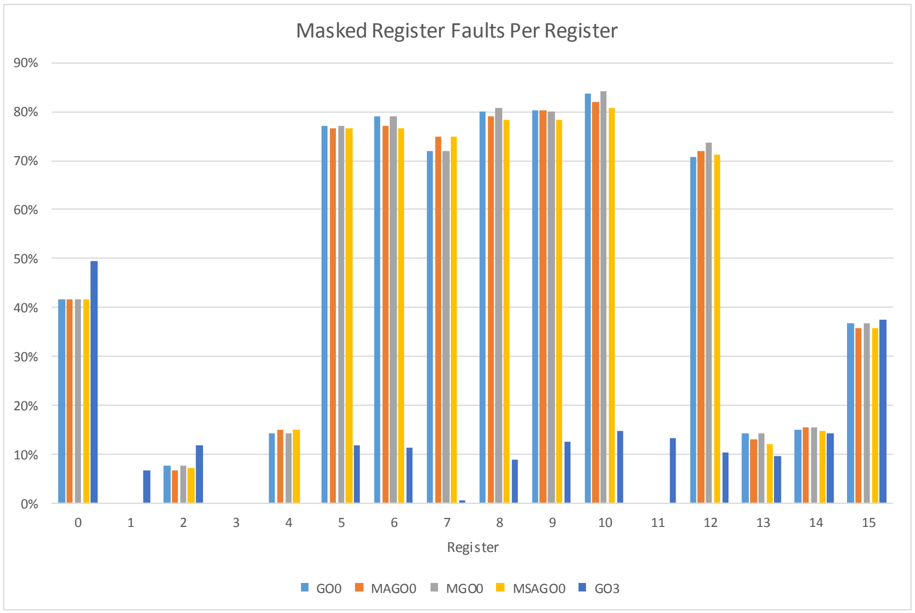


Figure 5.9: Masked register faults listed for each register from 0 to 15 and for the different binary categories.

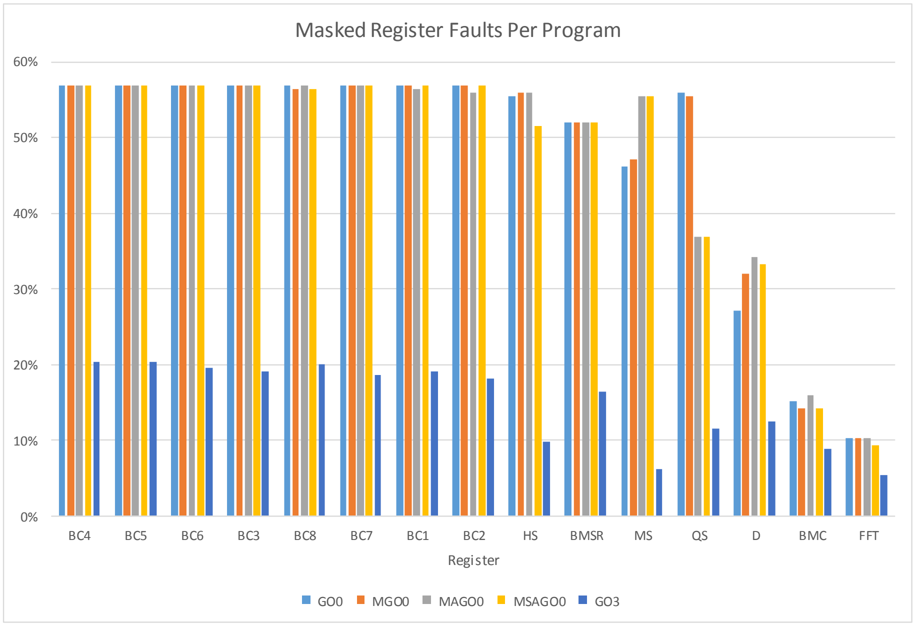


Figure 5.10: Masked register faults listed for each test program for the different binary categories.

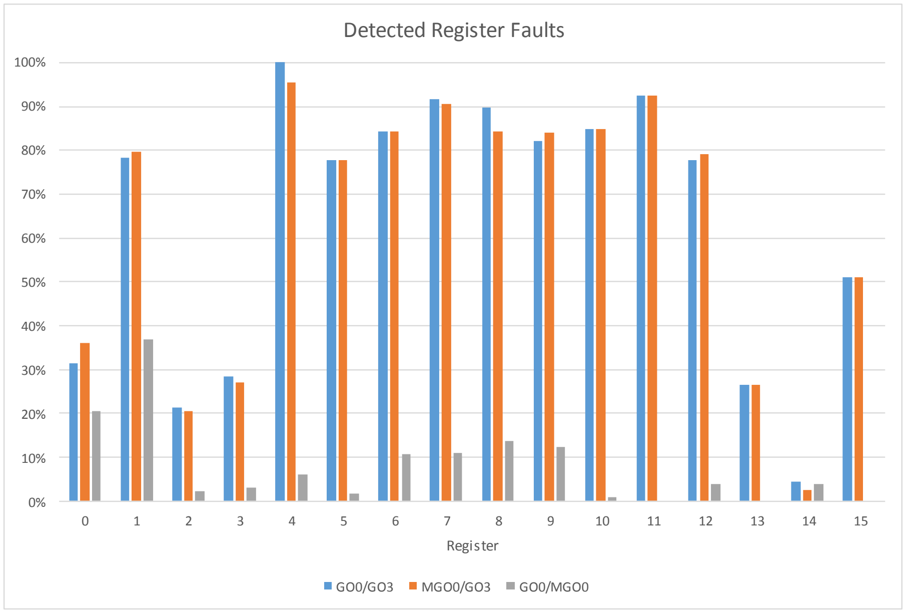


Figure 5.11: Source mutation: Detected register faults in percent for each register.

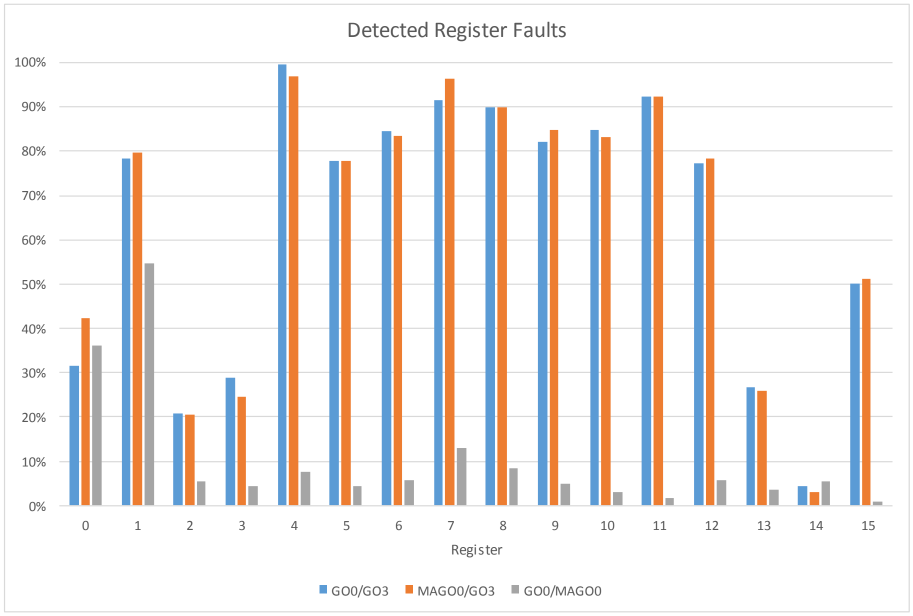


Figure 5.12: Assembler mutation: Detected register faults in percent for each register.

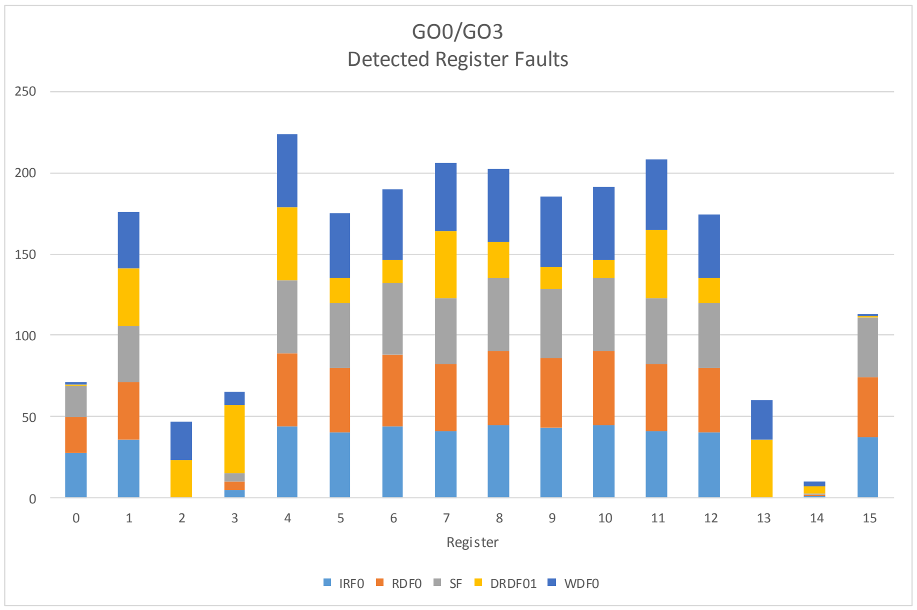


Figure 5.13: GO0/GO3: Detected register faults in absolute number for each register and different fault modes.

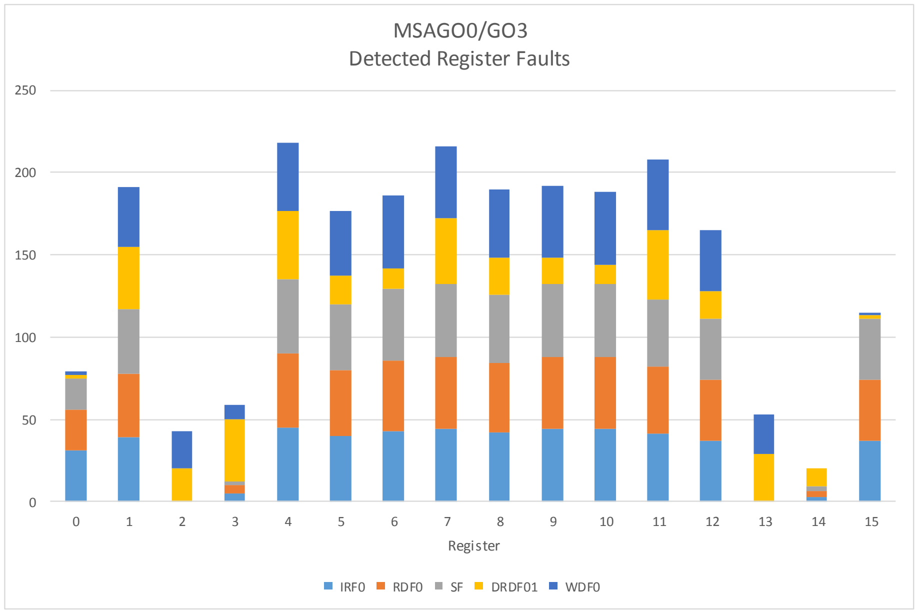


Figure 5.14: MSAGO0/GO3: Detected register faults in absolute number for each register and different fault modes.

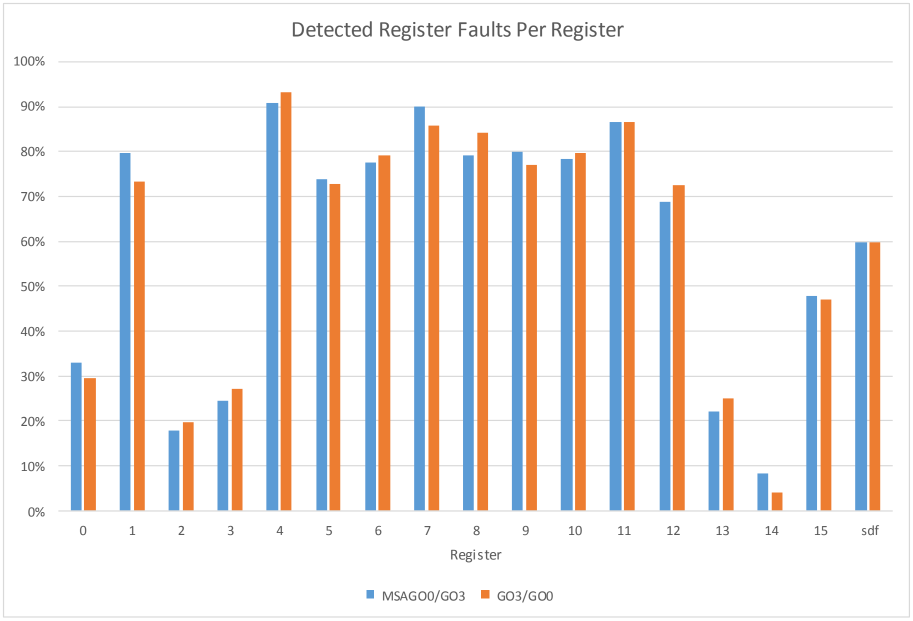


Figure 5.15: Source and assembler mutation: Detected register faults in percent for each register.

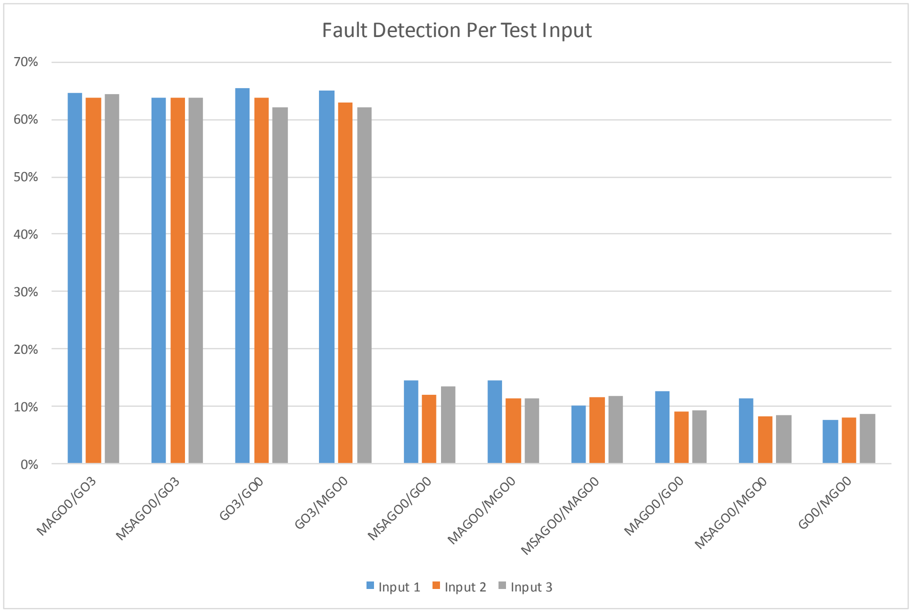


Figure 5.16: Detected register faults in percent for each register per input.

In Figure 5.17 the fault effects for the different binaries (see Table 5.2) can be seen. The binary `GO3` is not capable to mask as many register faults as the other approaches. All other approaches give very similar results. This is one reason for the promising fault detection results when `GO3` is combined with a `GO0` binary (see Table 5.8).

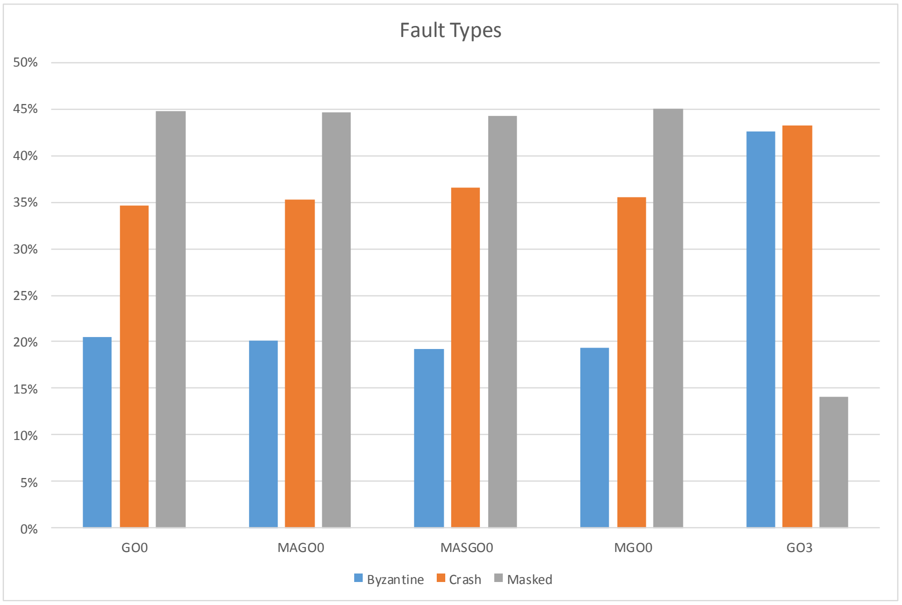


Figure 5.17: Proportion of different effects of register faults.

In Figure 5.18 the detected faults are distinguished according to the output. If both binaries are executed without a crash this is called a Byzantine fault. When the output is different, the fault is detected. In Figure 5.18 this is called *Byzantine Found*. An existing hardware fault can also be detected if one binary crashes while the other terminates correctly. In Figure 5.18 this is called *One Crash*.

Not detected faults can be found in Figure 5.19. Byzantine faults can not be detected if the output of the two binaries is the same. This is called *Byzantine Not Found* in Figure 5.19. When both binaries mask the fault, this is called *Golden* in Figure 5.19. If both binaries crash the introduced diversity is not needed, so the fault is not marked as detected throughout this chapter.

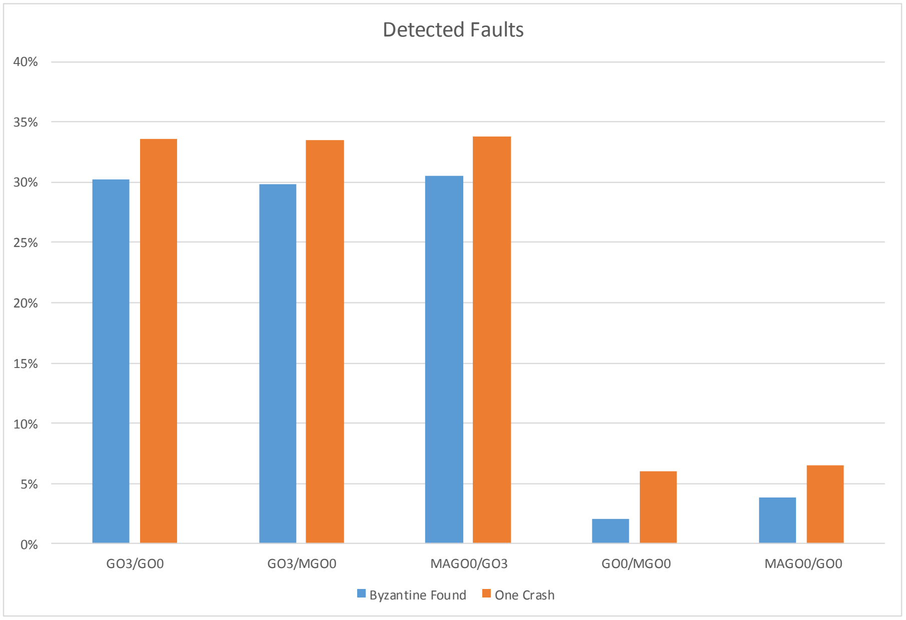


Figure 5.18: Proportion of detected faults for different compilation pairs.

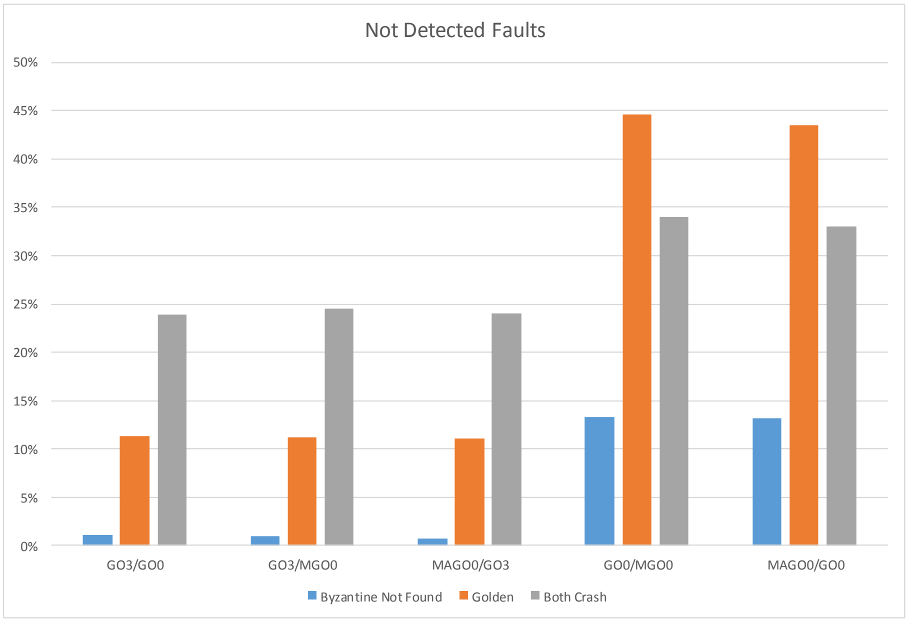


Figure 5.19: Proportion of not detected faults for different compilation pairs.

5.3.3 Instruction Decoder Faults

As discussed in Chapter 2.5.2 and specified in Chapter 4.4.4 NOP instructions are used to simulate inactive instruction decoder faults. To each tested program a set I consisting of 150 different faulty instructions is introduced. The used binaries are `GO0`, `GO3`, `MGO0`, `MAGO0`, and `MASGO0`. The ten most frequent instructions from each binary are added to I. The other instructions are selected randomly from one of the used binaries. Altogether 33750 faults are injected.

In Figure 5.20 the masked instruction decoder faults are given for each program and Table 5.6 gives the absolute results. Compared to Table 5.5 where masked register faults are listed, `GO3` has the most masked faults. The selection of the introduced instruction decoder faults is one explanation for this outcome. Instructions are chosen randomly from one of the binaries. So it seems, that the used instructions of `GO3` are different than the used instructions by the other binaries. The diversity between `GO0`-versions and the `GO3` version is better than the diversion between the mutated versions. This can be seen in Figure 5.21 which shows the detected faults in percent for each of the $\binom{5}{2} = 10$ different binary versions. However it is possible to improve the `GO0/GO3` combination by 4% using a binary with assembler mutations `MSAGO/GO3`.

GO0	**MGO0**	**MAGO0**	**MSAGO0**	**GO3**
85%	85%	83%	83%	92%

Table 5.6: Masked instruction decoder faults listed for all binary categories.

Figure 5.20 shows that all compilation combinations which include `GO3` give the best results. The combination `MAGO0/GO3` which is created with ASM mutations detects about 37.01 % of all injected instruction decoder faults, whereas `GO0/GO3` detects about 33.16 %. The ASM mutation improves the result by 3.85 %.

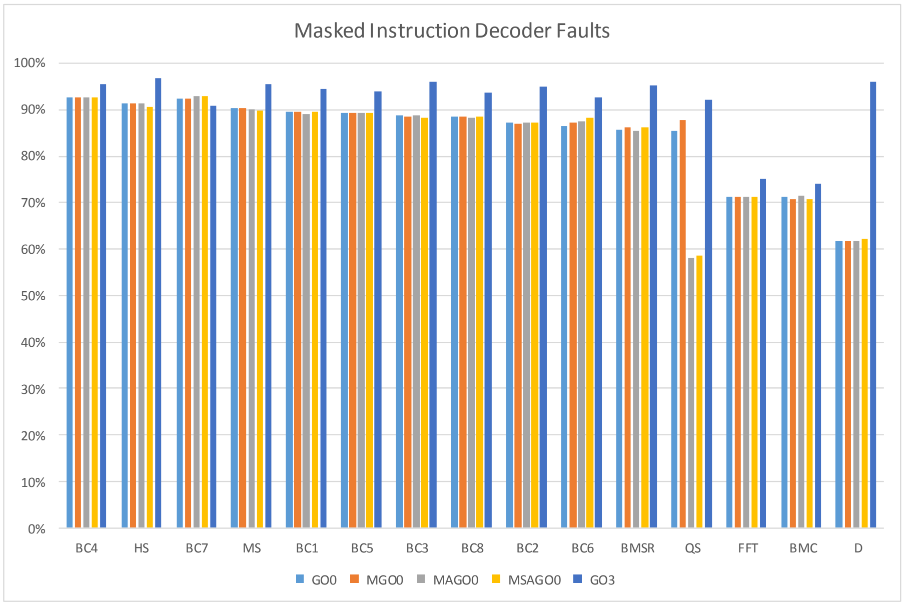


Figure 5.20: The possibility that a fault is masked grouped by the five approaches.

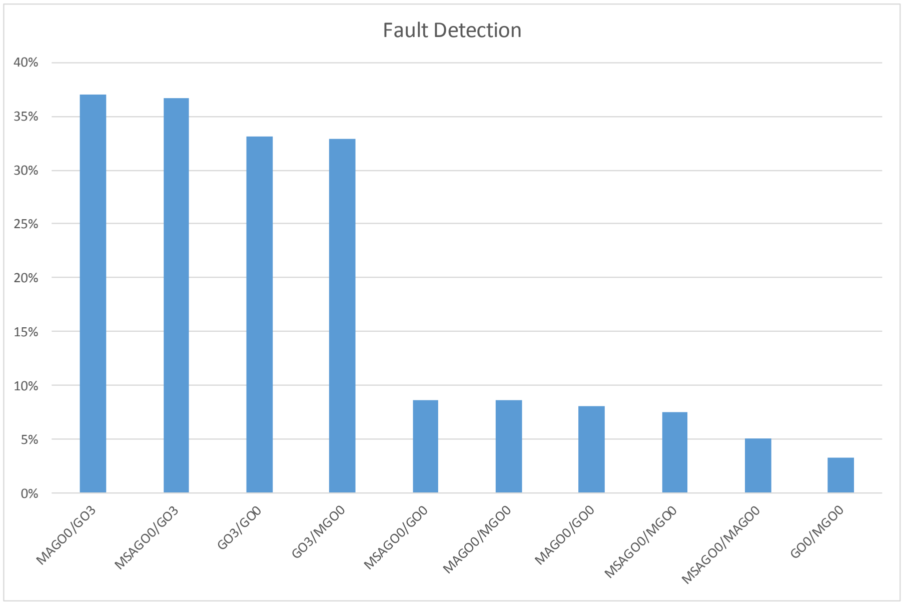


Figure 5.21: The proportion of detected instruction decoder faults by the different combinations of the five compilations.

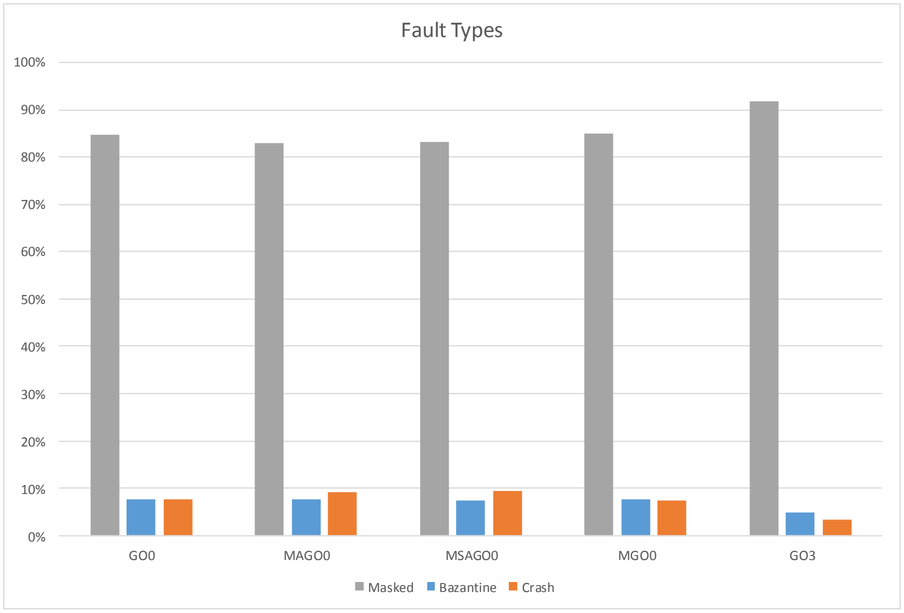


Figure 5.22: Proportion of different effects of instruction decoder faults.

In Figure 5.23 the detected faults are distinguished according to the output. If both binaries are executed without a crash this is called a Byzantine fault. When the output is different, the fault is detected. In Figure 5.18 this is called *Byzantine Found*. An existing hardware fault can also be detected if one binary crashes while the other terminates correctly. In Figure 5.18 this is called *One Crash*.

Not detected faults can be found in Figure 5.19. Byzantine faults can not be detected if the output of the two binaries is the same. This is called *Byzantine Not Found* in Figure 5.19. When both binaries mask the fault, this is called *Golden* in Figure 5.24. If both binaries crash the introduced diversity is not needed, so the fault is not marked as detected throughout this chapter. In most of the cases the fault is masked by both binaries.

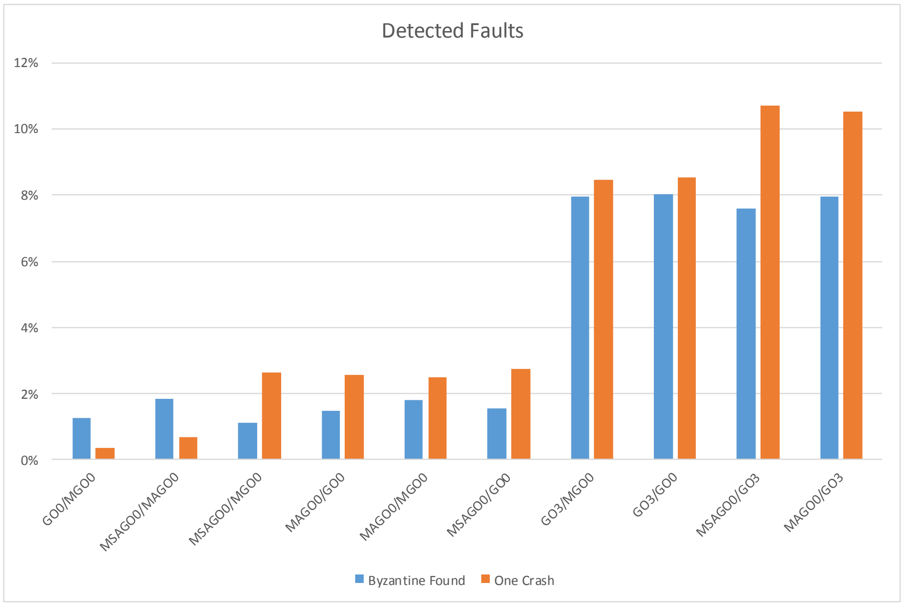


Figure 5.23: The proportion of detected instruction decoder faults by the different combinations of the five compilations.

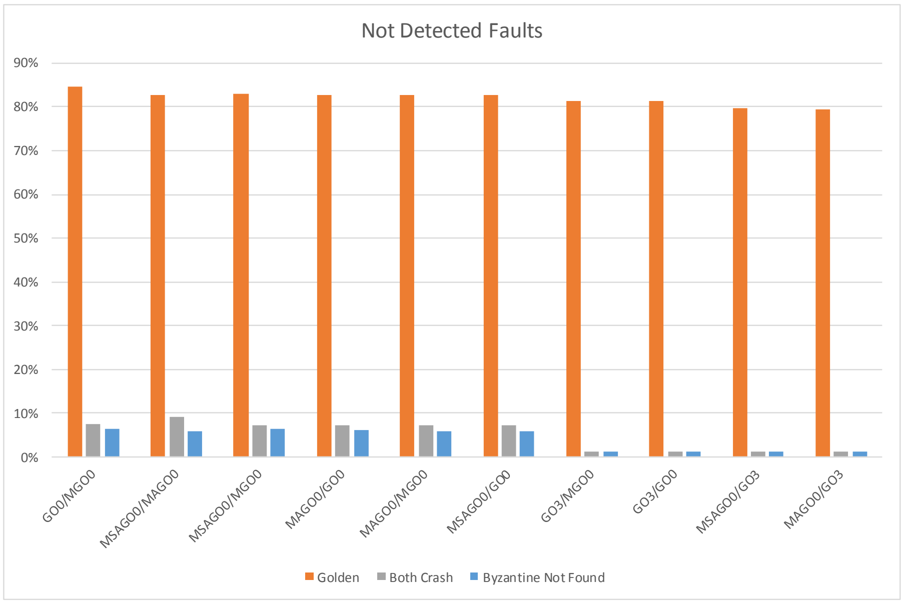


Figure 5.24: The proportion of not detected instruction decoder faults by the different combinations of the five compilations.

5.3.4 Fixed Register Flag

In this section the influence the usage of the fixed register flag is investigated for a 1oo2 system and the GCC compiler. The combination `GO0/GO3F03` improves the detection rate by nearly 5 % (see Table 5.7) when compared to `GO0/GO3`. The binary `GO3F03` is compiled with the GCC compiler using the flags `-O3` and `-ffixed-r3`.

GO0/GO3	**GO0/GO3F03**
61.75%	66.31%

Table 5.7: Detected register faults listed for all binary categories.

In Figure 5.26 the detection rate for each register can be seen. Over 87 % of all register faults which are introduced to register three are detected by using the combination `GO0/GO3F03`. This is the influence of the flag `-ffixed-r3`. In Figure 5.25 the masked faults per each register are listed. The binary `GO3F03` masks over 87 % of all register faults which are introduced to register three while the other binaries are not able to mask even a single fault. Therefore all faults which are masked by `GO3F03` can be detected by the combination `GO0/GO3F03` since the output differs.

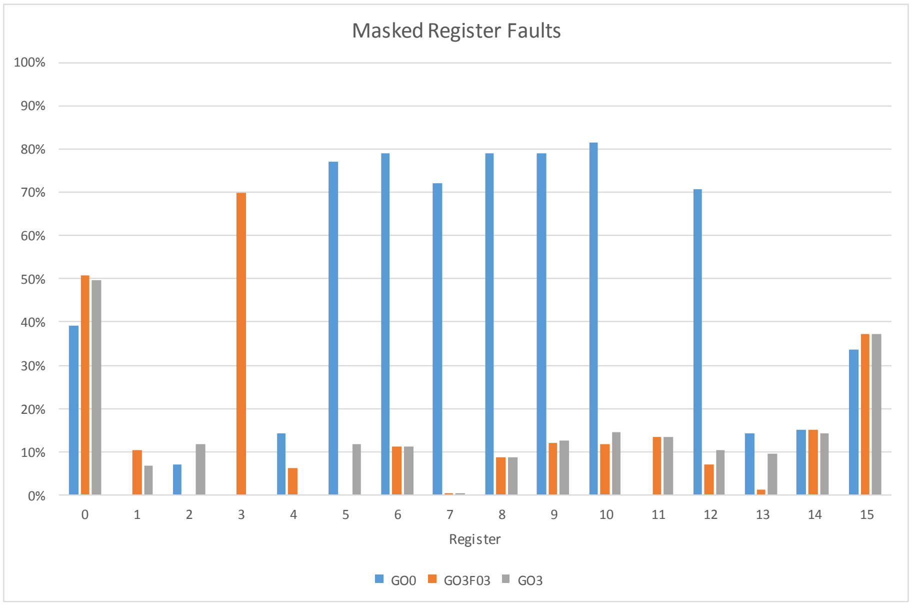


Figure 5.25: The proportion of masked register faults per register.

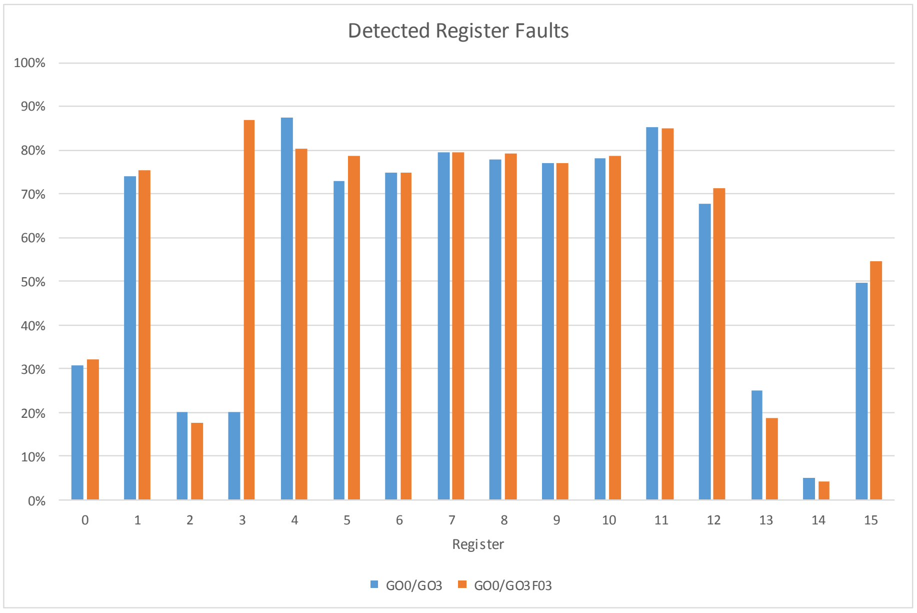


Figure 5.26: The proportion of detected register faults per register.

In order to analyze the impact of the fixed register flag on the overall fault detection the Clang compiler is added to the set of used compilers. This results in the three binaries: `CO0`, `CO3` and `MCO0` (see Table 5.2). The usage of assembler mutations in combination with the Clang compiler is neglected and part of further work (see Chapter 6.1). In Figure 5.27 it can be seen that the top four combinations include the `GO3FO3` binary along with source or assembler mutations. The best combination without mutations is `GO3FO3/GO0`.

The influence of the mutation depth on the fault detection rate for register faults can be seen in Figure 5.28 for source mutations and in Figure 5.29 for ASM mutations. The simulation was carried out for the program `BC1`. It can be seen that the results are very flaky and non-monotonic. Therefore, it can be concluded that the silent evaluation function defined in Chapter 3.6.2 is not suited to generate optimal solutions in the context of register faults.

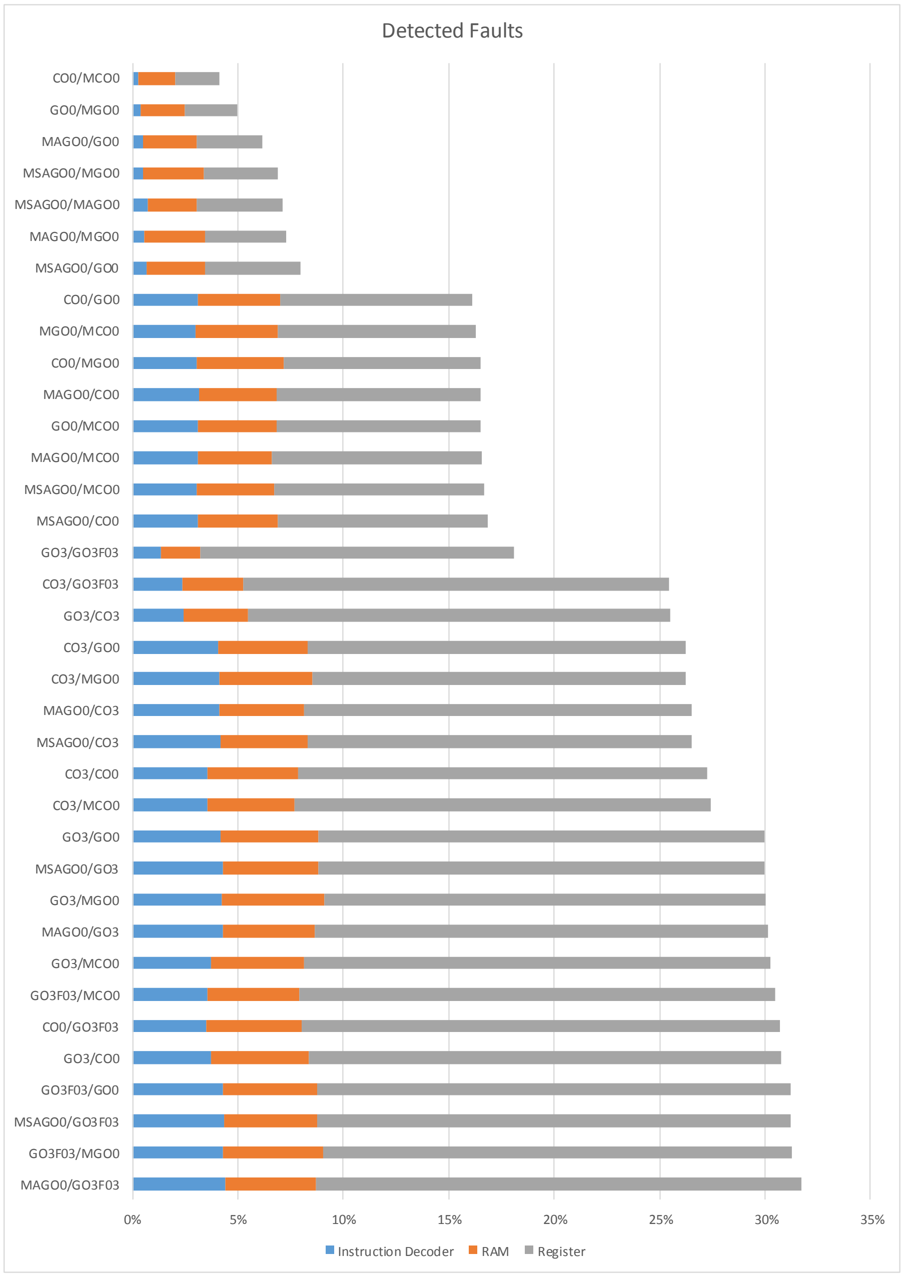


Figure 5.27: The proportion of detected faults for each binary combination.

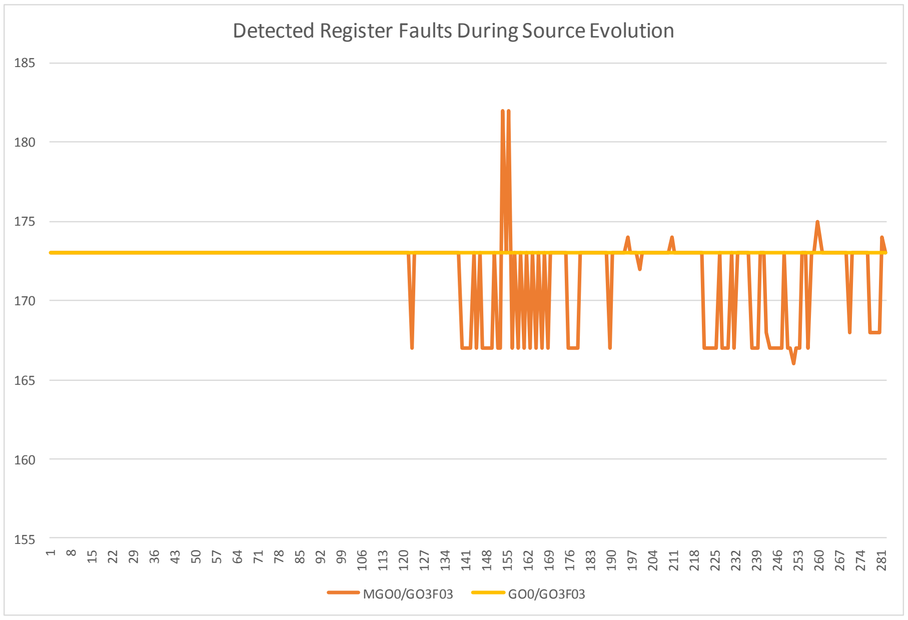


Figure 5.28: The absolute number of detected register faults during source evolution.

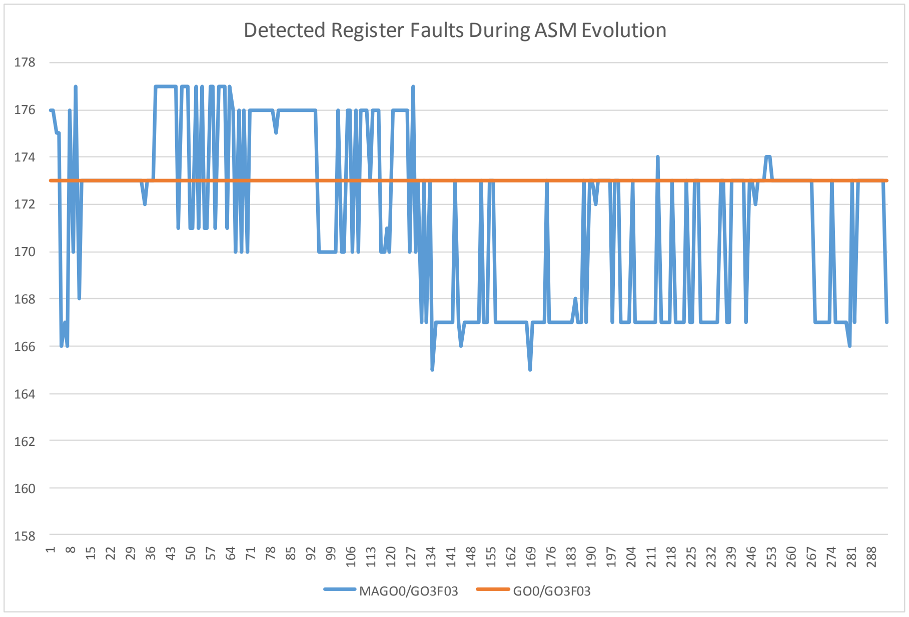


Figure 5.29: The absolute number of detected register faults during ASM evolution.

5.4 Fault Recovery

As discussed in Chapter 2.1 the two phases of hardware fault tolerance are fault detection and fault recovery. In this chapter a special form of fault recovery is evaluated where diverse compiling is used to mask a given hardware fault. The experiments showed that unsound randomization techniques do not improve the recovery rate. Therefore unsound randomization is neglected in this chapter. The GCC compiler with the $-O0$ flag is used to generate the reference binary. First, 150 different faults for each test program are collected which are not masked by the reference binary. This faults consist of

- 50 CPU register cell faults,
- 50 instruction decoder faults, and
- 50 RAM faults.

The CPU register cell faults are specified by the triple consisting of the CPU register, a fault mode and a bit mask. The distinct set of instruction decoder faults is chosen randomly. To select 50 different RAM faults the approach as presented in Chapter 4.4.2 is used to retrieve a set of actually accessed RAM addresses during execution. A RAM fault is specified by a 4-tuple consisting of an address, a mask a bit to set and a target. Address decoder faults and register cell faults are considered as target.

The usage of different compilers and register flags is evaluated concerning hardware fault recovery. A binary can be specified by:

- **Compiler**: `GCC` or `CLANG`
- **Source mutations**: list of used mutations
- **Assembler mutations**: list of used assembler mutations
- **Fixed register flags**: list of registers which should be avoided
- **Optimization flags**: list of optimization flags.

To recover from a fault, different combinations from the above parameters are used to generate binaries. Note that the `CLANG` compiler can not be combined with the fixed register flag. Added together 72 different combinations can be used, to mask a given hardware fault. According to Table 5.8 100 % of all injected RAM faults are recovered. More than 90 % of all injected instruction decoder faults and over 46 % of all injected register faults can be masked using one of the 72 different binaries, specified above.

RAM	Register	Instruction Decoder
100.00%	47.6%	91.33%

Table 5.8: Proportion of faults which can be masked.

In Figure 5.30 the result for each of the test programs can be seen. The recover rate for instruction decoder faults and register fault is strongly related to the number of used external libraries by the test program. The programs `BMC`, `QS` and `FFT` are dependent on libraries which can not be recompiled. When the hardware fault can not be masked by the libraries there is no way to recover from that fault.

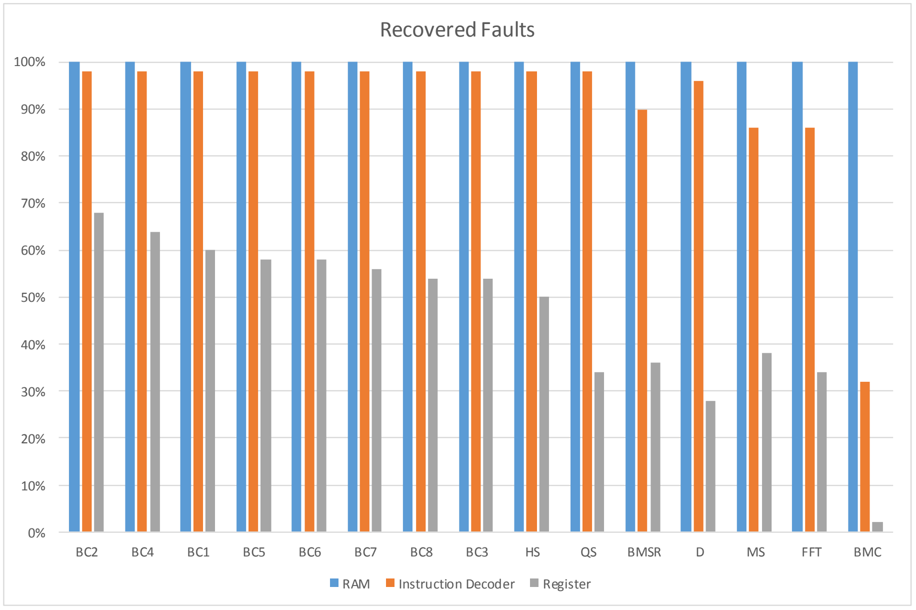


Figure 5.30: The proportion of recovered faults listed per fault type and program.

The absolute number of not recovered register faults can be seen in Figure 5.31. The values are not equally distributed. One reason for the observed behavior might be the special purpose of the registers from register 11 to register 15. The frame pointer, instruction pointer, stack pointer, linking register and program counter are located in these registers. Another reason is related to the probability that a certain register is used by external libraries.

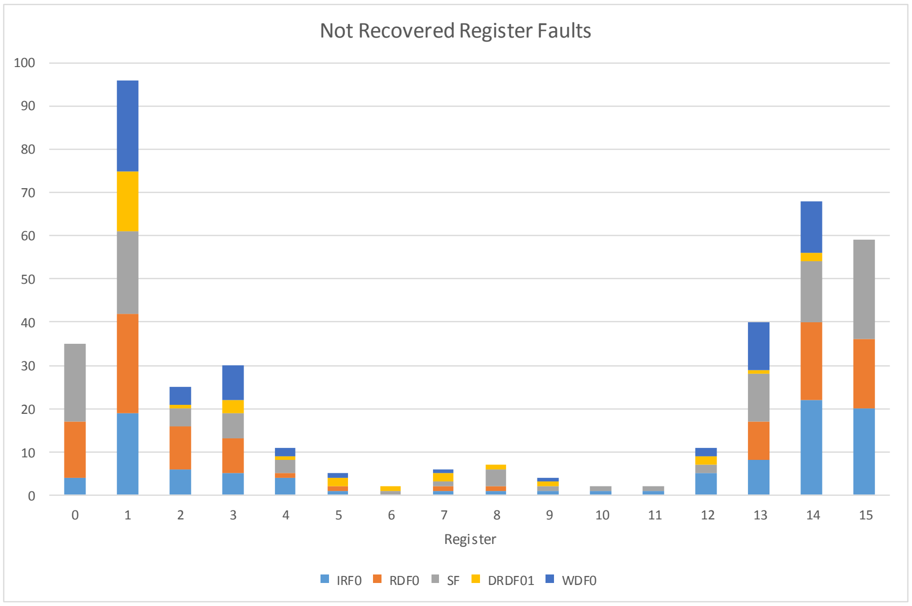


Figure 5.31: The absolute number of not recovered register faults listed for each fault mode.

A selection of instruction decoder faults which are not masked is shown in Table 5.9. Only two instructions are not maskable in different programs: `bx lr` and `push {r4, r5, lr}`. The `bx` instruction is used to branch to the address specified in the register `lr`.

Occurrences	Code	Name	Registers
14	`e12fff1e`	`bx`	`lr`
2	`e92d4030`	`push`	r4, r5, lr
1	`e8bd4ff8`	`pop`	`{r3, r4, r5, r6, r7, r8, r9, sl, fp, lr}`
1	`e8bd4ff0`	`pop`	`{r4, r5, r6, r7, r8, r9, sl, fp, lr}`
1	`e8bd40f8`	`pop`	`{r3, r4, r5, r6, r7, lr}`

Table 5.9: A selection of not recovered instruction decoder faults.

In Figure 5.32 the binaries are listed which are able to mask the fault which is introduced to the binary GO0. The results show that each binary seems to be good to mask on specific fault type. For example binary `GO1` masks most of the introduced RAM faults, whereas the binary `GO1FO1` scores highest for masking instruction decoder faults. Register faults can be masked using the GCC compiler along with the fixed register flag and optimization flag: GOXFXX. This information can be used to create a very efficient N-out-of-M architecture for hardware fault detection.

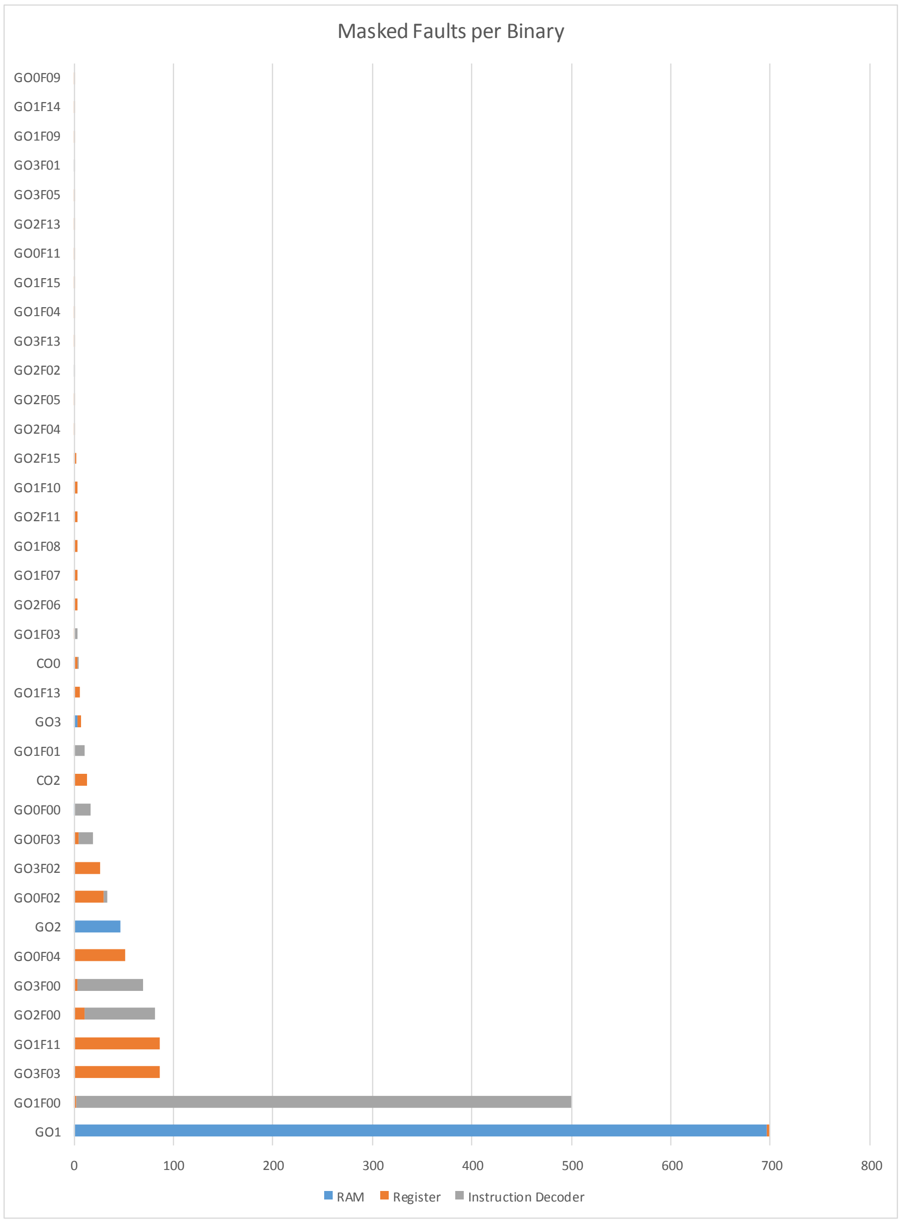


Figure 5.32: The absolute number of masked faults listed for all binaries and fault types.

5.4.1 Common RAM Addresses

The experiment presented in Table 5.8 shows that 100% of all injected RAM faults which are not masked by the reference binary (GO0) can be masked by one of the 72 binaries which are created using diverse compiling. Therefore, a second experiment is used where only RAM addresses are taken into account which are accessed by all of the 72 binaries and by the reference binary. In Figure 5.33 the absolute number of RAM addresses which are accessed by all of the 73 binaries listed for each program. It can be seen that the number of accessed RAM addresses is not commonly distributed. The programs BMC and FFT make heavy use of the libraries *math.h* and *stdlib.h*, which can be one explanation for the many common accessed RAM addresses.

In Figure 5.34 the results of an experiment are shown where 50 distinct RAM faults for the reference binary (GO0) are taken from the pool of common accessed RAM addresses. However, it was not possible to find 50 distinct RAM faults for all of the tested programs. Only the programs D, FFT and BMC make use of the intended amount of injected faults. As described in Section 5.3.1, a RAM fault can be specified as a triple consisting of a RAM address, a mask and the bit to set. For each of the tested programs 1000 iterations are used to find RAM faults which lead to a crash on the reference binary. All in all, 99.12% of all injected RAM faults can be masked with one of the 72 binaries which are created using diverse compiling.

To recover from the injected RAM faults, the set of generated binaries is iterated until a binary is found which masks the given fault. The traversal of the set is always the same. In Figure 5.35 the number of masked RAM faults are listed for the binaries which are generated with diverse compiling using different compiler parameters.

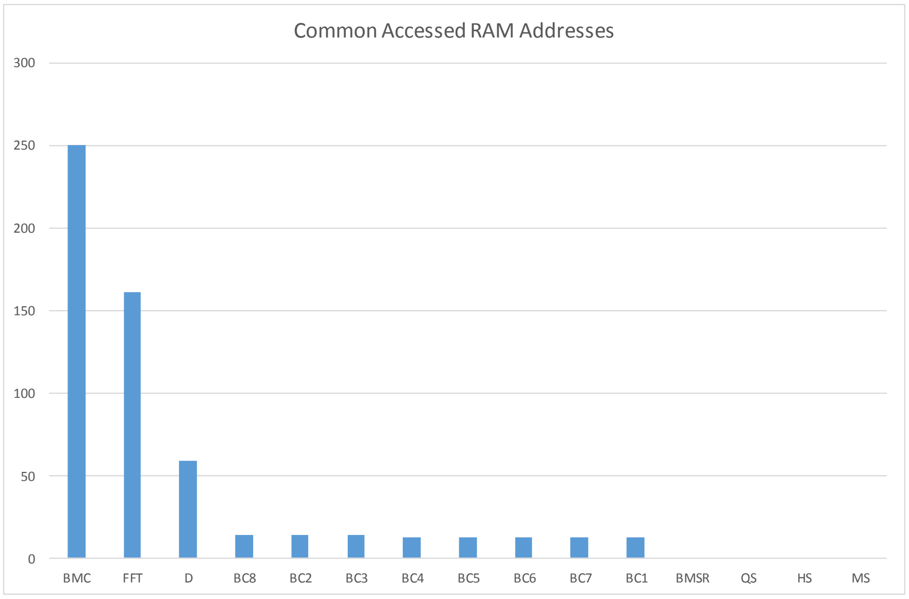


Figure 5.33: The absolute number of RAM addresses which are accessed by all of the 73 binaries listed for each program.

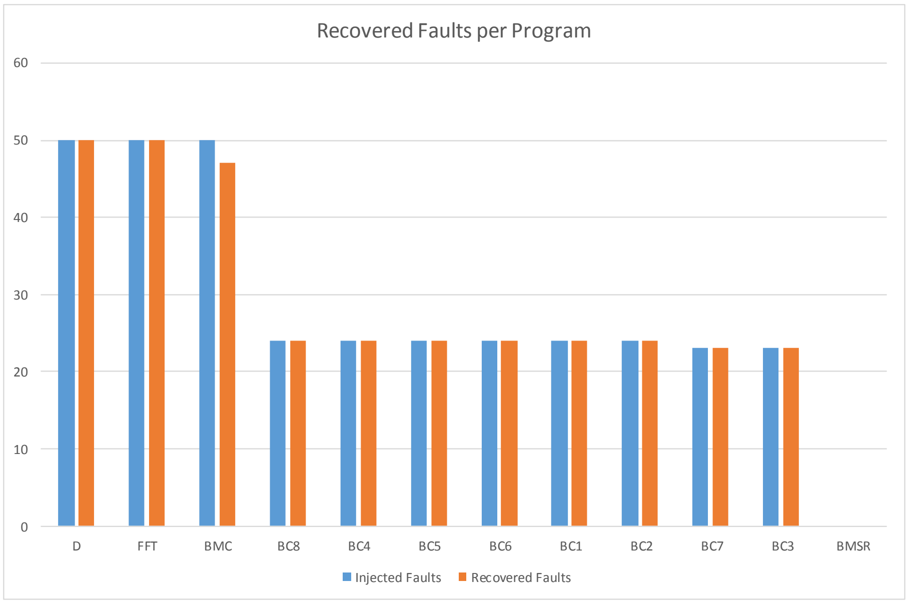


Figure 5.34: The absolute number of injected RAM faults is compared with the number of masked faults for each of the tested programs.

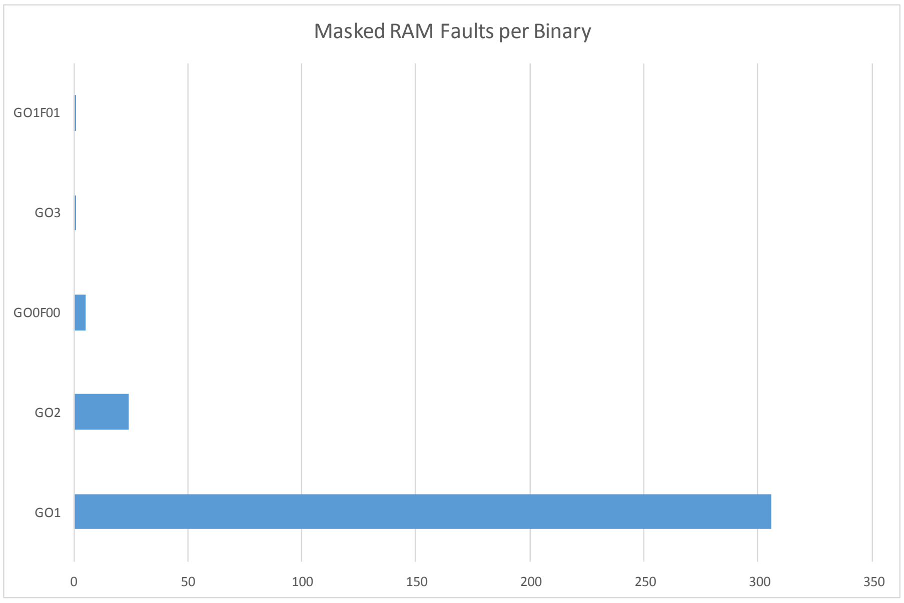


Figure 5.35: The absolute number of masked RAM faults per binary.

5.5 Mutation Performance

To evaluate the mutational performance the evolution Algorithm 2 is used with the following parameters to generate mutations which are 100 steps away from the original version:

	Source	**Assembler**
Mutation depth	100	100
Mutations per level	2	2
Mutations per iteration	20	8

Table 5.10: The used parameters for the source and assembler evolution algorithm.

The only difference concerning the parameters from Table 5.10 is the number for mutations per iteration. This is because of the success rate of a random mutation. Assembler mutations are more likely to be valid.

All fifteen programs listed in Chapter 5.2 are mutated using the same parameters. The results are gathered on an Intel Core i5-2500 CPU with 3.30 GHz and 8 GB of RAM. The host system is an Ubuntu distribution with version number 14.04.2 running in a virtual machine (Oracle Virtual Box 4.3.28) with 3 GB assigned RAM and no processor speed restriction.

	Source	**Assembler**
Percentage of valid mutations	9.24%	36.83%
Applied mutations	50060	30256
Successful mutations	4810	11144
Percentage of successful linkings	22.34 %	96.63 %
Percentage of successful tests	35.02 %	38.58 %
Percentage of equal mutations	12.28 %	1.21 %

Table 5.11: Number of applied mutations and success rates gathered by the evolution algorithm by generating mutations which are 100 steps away from the original version for fifteen different programs. Additionally for the assembler statistics a source mutated version of each of the programs is mutated.

5.5.1 Source Mutation

According to Table 5.11 the success rate of a random mutation that does not change the semantic of a program according to the test suite is 6.86%. In [SFF+14] the success rate is given with 30 %. One explanation for the huge differences might be the different program representation, although it is stated that „the results hold across all classes of programs, for mutating at both the source code and assembly instructions levels...“. Another explanation might be the used source mutation approach. In this thesis a C-AST mutation tool based on CLANG [Sch15b] is used, whereas the tool in[SFF$^+$14] is based on CIL [NMRW02a]. Note that the usage of the strict evolution approach (Chapter 4.2.4) only changes the result by 0.94 %.

Most of the erroneous mutations are filtered out during compilation. Only 22.34 % of all mutations compile without errors. When a mutation compiles the probability is higher than 35 % that the mutation is valid in respect to the test suite. The probability that a generated valid mutation is equal to a previously generated mutation is 12.28 %.

5.5.2 ASM Mutation

With a success rate of 36.83% the assembler mutation is much more efficient than the source mutation approach. In [SFF$^+$14] the average success rate for assembler mutations are given with 39.6% which is very close to the results presented in Table 5.11. It can be seen that nearly 97% of all mutations pass the process of linking. The linker does not check the code since this is the task of the previous compile process. This is one of the main deviations between source mutation and assembler mutation. Although there is no source check the percentage of successful tests of assembler mutation is nearly as high as the percentage of successful tests of source mutations. It seems easier to find neutral variants on assembler level so the percentage of equal mutations is only about 1.2 % which is much lower then the 12.28% of source mutations.

5.5.3 Runtime Performance

In Table 5.12 it can be seen that for both approaches most of the time is used for compilation and testing. The time needed for equality checks and mutating can be neglected. Especially the assembler mutations are very fast since no external program has to be started. Concerning source mutations, only 22 % of all random mutations are tested so the absolute time for testing per mutation would be much higher than the absolute time for compilation. Since there are no compiler checks when assembler code is mutated nearly all mutations are tested as binaries. This explains that assembler mutations use more time for testing than the source mutation approach. Regardless of the used approach, the time for testing is strongly dependent on the used test suite and program. All in all mutations on assembler level are much faster than mutations on source level when AST transformations are applied.

	Source	**Assembler**
Total time	11.38 hours	7.35 hours
Time for compilation/linking	2.64 hours	1.91 hours
Time for testing	4.44 hours	5.26 hours
Time for mutating	18.3 minutes	0.16 minutes
Time for predecessor checks	8.88 minutes	11.53 minutes
Average time for successful mutation	11.93 seconds	4.75 seconds

Table 5.12: Absolute and average times gathered by the evolution algorithm by generating mutations which are 100 steps away from the original version for fifteen different programs.

Figure 5.36 and Figure 5.37 lists the used time for the evolution process per program.

The test program for calculating the integer square root (BMSR) (see Section 5.2.1) is the slowest program when ASM mutations are applied, but the fastest program when source mutations are used. On the other side the program for fast Fourier transformations (FFT) is among the fastest programs concering ASM mutations but very slow, when source mutations are applied. FFT is a relatively big program which uses many imports and floating point operations. Source mutations are very slow when applied to FFT due to the complexity of the program, but assembler mutations can be found very fast.

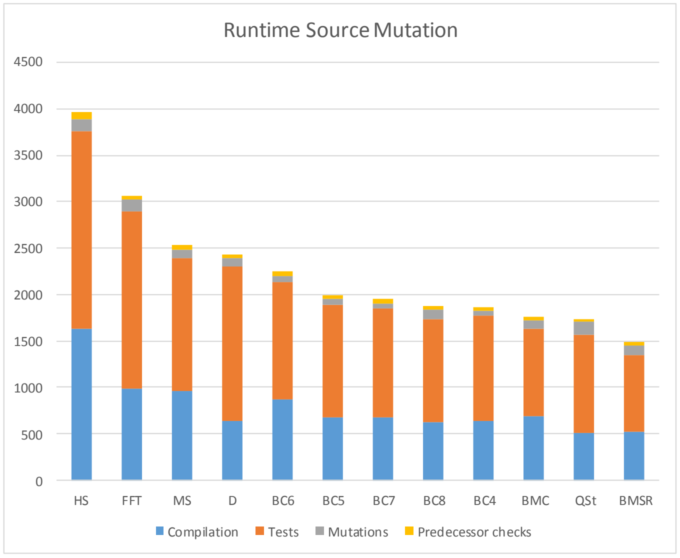


Figure 5.36: Source evolution times listed for each test program.

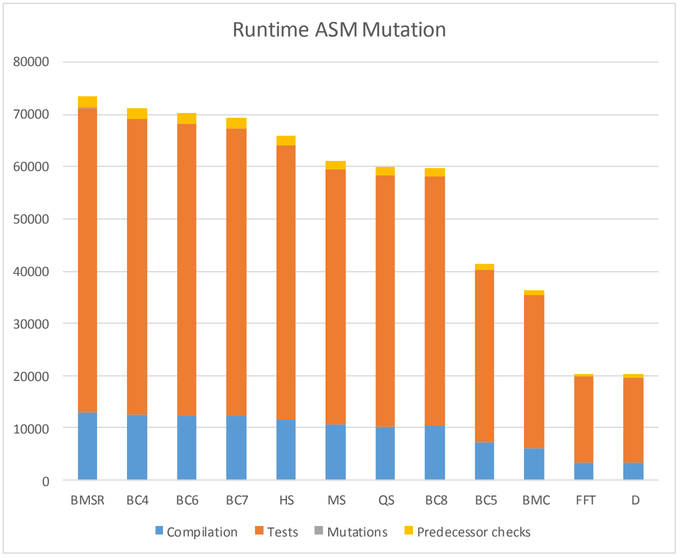


Figure 5.37: Assembler evolution times listed for each test program.

Chapter 6

Conclusion

This thesis provided insights into techniques for automated software diversity with a focus on unsound randomization techniques in the field of hardware fault tolerance. The classification of different approaches for automated software diversity was explained and exemplified by different methods taken from the literature. Furthermore, the most important fault injection frameworks were discussed and classified.

The term fault tolerance was discussed and categorized into fault detection and fault recovery. Methods for both categorizations were implemented and evaluated. For this reason a QEMU based fault injection framework (FIES) was used and modified. FIES was used to simulate a Freescale i.MX28 EVK development board and to inject permanent register faults, instruction decoder faults and RAM faults. Moreover, different modifications were introduced to allow the input selection for the tested programs during execution and to enable the logging of accessed RAM addresses. The mutational robustness of software was utilized to implement an unsound randomization technique. For this reason, a method to mutate the AST of C source code was implemented using LLVM. This method is based on the Software Evolution Library. Furthermore, assembler code was mutated using Python.

In order to enable the generation of automated test results, a framework for automated software diversity was implemented to offer methods to create mutations of programs which can be validated against a test suite. An evolution algorithm simplifies the process of generating mutations which are many steps away from the original file. For this reason, a compiler toolchain was created with Linux as the host system to support the GCC compiler and the CLANG compiler along with different compiler flags. This allows the comparison of unsound randomization techniques with diverse compiling techniques. Even combinations of both approaches can be produced with promising results concerning hardware fault tolerance.

For the evaluation, fifteen different programs out of the categories Automotive and Industrial Control, Network and Telecommunications were used. Results for fault detection and fault recovery were presented. A 1oo2 system was used for fault detection. The results showed that the diverse compiling approach is much more efficient for fault detection compared to pure unsound randomization methods. Therefore, unsound randomization methods were combined with the diverse compiling approach by using source and assembler mutations. It was possible to improve the fault detection rate concerning RAM faults and instruction decoder faults using source mutations and assembler mutations which are

100 steps away from the original version. It was not possible to improve the detection rate of register faults with unsound randomization techniques. However, the results showed that the fixed register flag can be used to improve the diverse compiling approach for register fault detection.

Furthermore, the evaluation pointed out that diverse compiling can be used for the recovery from hardware faults where the technique is very efficient concerning RAM faults, with 99 % of all introduced faults being masked. Over 91 % of all introduced instruction decoder faults were maskable and it was possible to mask over 46 % of the introduced register faults. The results showed that unsound randomization techniques are not suited to recover from permanent register faults.

For safety-critical systems it is very important that the programs are semantically correct. However, unsound randomization methods can modify programs so that they are not semantically equivalent to the original program. Furthermore, the introduced mutations obfuscate the source code and have a bad influence on the readability. Therefore, the next step would be to analyze the parts of the introduced changes which have an influence on hardware fault detection. Based on this knowledge algorithms can be created which do not change the semantic of the program and keep a certain amount of readability.

6.1 Further Work

In this section, some of the most important improvements for the used unsound randomization approach are presented.

- **Fine grained usage of compiler flags**: In this thesis the flags `-O0` to `-O3` in combination with the fixed register flag `-ffixed-r` is used for diverse compiling. Since the optimization flags consist of many different compiler flags which are extended with flags whenever the optimization level is increased, many combinations could be created by using the fine grained flags directly. Over 90 different flags are supported by the GCC compiler giving many different combination possibilities.

- **Optimization of mutation time**: The code concerning mutations can be optimized. Currently some parts are redundant to make the code more readable.

- **Extend the NooM architecture**: In this thesis a 1oo2 architecture is used for hardware fault detection. The usage of different mutations could raise the proportion of detected faults.

- **Precompiled standard library with specified flags**: Since external libraries cannot be mutated or influenced with optimization flags, the usage of precompiled versions of the standard library would boost the fault detection for all presented methods. Especially register fault recovery would be much easier.

- **Source code analysis for mutations**: The success rate and quality of source code mutations is dependent on many things. It is in the nature of mutations, that the mutated program consists solely of source code, that already existed in the original version. The example provided in Listing A.10 is semantically equivalent to the program in Listing A.11 but the quality of the generated mutations differs. The usage of CIL based source mutation might address this problem [NMRW02b].

Appendix A

Diversification Chain Source

A.1 Python

Listing A.1: Python module with functions for diverse compilation, evolution algorithms and over all diversificaton methods

```
import ShellWrapper
import Mutator
import random

mutationFunctionMapper = {"Source": Mutator.mutateSourceFile,
                          "ASM": Mutator.mutateAssemblerFile}

COMPILERS = ["GCC", "CLANG"]
OPTIMIZATION_FLAGS = ["-O0", "-O1", "-O2", "-O3"]
FIXED_REGISTERS = ["-ffixed-r3", "-ffixed-r4", "-ffixed-r5"]


def compileDiverse(pathToFile, pathToDestinationDirectory, listOfMutations):
    for optimization in OPTIMIZATION_FLAGS:
        for register in FIXED_REGISTERS:
            print "Compile GCC with " + optimization + " " + register
            Mutator.compileSourceToBinary(listOfMutations, pathToFile, pathToDestinationDirectory,
                                          "GCC",
                                          [optimization],
                                          [register])
        for compiler in COMPILERS:
            print "Compile " + compiler + " with " + optimization
            Mutator.compileSourceToBinary(listOfMutations, pathToFile, pathToDestinationDirectory,
                                          compiler,
                                          [optimization],
                                          [])


def diversifyAll(pathToFile, numberOfRandomMutations, maximalMutationDepth,
                 minimalNumberOfMutationsPerLevel,
                 pathToDestinationDirectory):
    numberOfRandomMutations = str(numberOfRandomMutations)

    # Diverse Compile without mutations
    compileDiverse(pathToFile, pathToDestinationDirectory, [])
    # Mutator.compileSourceToBinary([],pathToFile,pathToDestinationDirectory,"GCC",["-O0"],[])

    # Create source mutations
    evolutionAlgorithm(pathToFile, numberOfRandomMutations, maximalMutationDepth,
                       minimalNumberOfMutationsPerLevel,
                       "Source")

    mutationToDiverseCompile = [
        ShellWrapper.getListOfMutationsFromSpecifiedMutationDepth(pathToFile,
                                                                  str(maximalMutationDepth))[0]]

    # Diverse Compile with source mutations
    print "Compile source mutations"
    compileDiverse(pathToFile, pathToDestinationDirectory, mutationToDiverseCompile)

    # Compile to ASM
    print "Compile to ASM"
    pathToASMFile = Mutator.compileToASM(mutationToDiverseCompile, pathToFile, "GCC", ["-O0"], [])

    # Create ASM mutations
    print "Create assembler mutations"
    evolutionAlgorithm(pathToASMFile, numberOfRandomMutations, maximalMutationDepth,
                       minimalNumberOfMutationsPerLevel,
                       "ASM")

    # Compile ASM mutations
    mutationToCompile = [
        ShellWrapper.getListOfMutationsFromSpecifiedMutationDepth(pathToASMFile,
                                                                   str(maximalMutationDepth))[0]]

    print "Compile ASM"
    Mutator.compileASMToBinary(mutationToCompile, pathToASMFile, pathToDestinationDirectory)


def evolutionAlgorithm(pathToFile, numberOfRandomMutations, maximalMutationDepth,
                       minimalNumberOfMutationsPerLevel,
                       mutationType):
    numberOfRandomMutations = str(numberOfRandomMutations)
    while True:
        print "Calculate mutation depth"
        mutationDepth = ShellWrapper.getHighestMutationDepth(pathToFile)
        listOfMutations = []
        if mutationDepth == None:
            listOfMutations = [pathToFile]
        else:
            listOfMutations = ShellWrapper.getListOfMutationsFromSpecifiedMutationDepth(
                pathToFile, str(mutationDepth))

        if mutationDepth > maximalMutationDepth:
            mutationDepth = maximalMutationDepth

        if mutationDepth == maximalMutationDepth:
            if len(listOfMutations) >= minimalNumberOfMutationsPerLevel:
                print "Evolution finished"
                break

        if mutationDepth != None and len(listOfMutations) < minimalNumberOfMutationsPerLevel:
            mutationDepth -= 1
            if mutationDepth == 0:
                listOfMutations = [pathToFile]
            else:
                listOfMutations = ShellWrapper.getListOfMutationsFromSpecifiedMutationDepth(
                    pathToFile,
                    str(mutationDepth))

        numberOfMutations = len(listOfMutations)
        if numberOfMutations == 0:
            print "listOfMutation has no content!"
            exit()

        indexOfFileToMutate = random.randrange(numberOfMutations)
        fileToMutate = listOfMutations[indexOfFileToMutate]

        print "Mutation depth is " + str(mutationDepth)
        mutationFunctionMapper[mutationType](fileToMutate, numberOfRandomMutations)
```

Listing A.2: Python module with functions for generating fault libraries and run binaries

```
import ShellWrapper
import BinaryInformation

import sys
import pickle

FAULT_LIBRARY_DIRECTORY = "/home/bernhardsp/diversity_chain/qemu_fault_libs"
GENERATED_FAULT_LIB = "/home/bernhardsp/diversity_chain/qemu_fault_libs/generated_fault_lib.xml"

def generateFaultLibInstructionDecoderNOP(instructionToReplace):
    parameters = str()
    parameters += ("-e s/\${address}/" + instructionToReplace + "/ ")
    templateFile = FAULT_LIBRARY_DIRECTORY + "/instruction_decoder_nop.xml"
    ShellWrapper.generateFaultLib(templateFile, parameters, GENERATED_FAULT_LIB)
    return GENERATED_FAULT_LIB

def generateFaultLibRegisterPermanent(address, mask):
    parameters = str()
    parameters += ("-e s/\${address}/" + address + "/ ")
    parameters += ("-e s/\${mask}/" + mask + "/")
    templateFile = FAULT_LIBRARY_DIRECTORY + "/register_permanent_template.xml"
    ShellWrapper.generateFaultLib(templateFile, parameters, GENERATED_FAULT_LIB)
    return GENERATED_FAULT_LIB

def generateFaultLibNoErrors():
    return "/home/bernhardsp/diversity_chain/qemu_fault_libs/no_error.xml"

def loadDirectory(pathToDirectory):
    binaryInformationFileList = ShellWrapper.getBinaryInformationFilesFromDirectory(
        pathToDirectory)
    listOfBinaries = []

    for binaryInformationFile in binaryInformationFileList:
        binaryInformation = pickle.load(open(binaryInformationFile, 'rb'))
        listOfBinaries.append(binaryInformation)
    return listOfBinaries

def runBinary(pathToDirectory, binaryInformation, pathToFaultLib, runDirectory, outputFileName):
    ShellWrapper.runInstance(pathToDirectory + "/" + binaryInformation["binaryName"],
                             runDirectory, pathToFaultLib,
                             outputFileName)
```

Listing A.3: Python module with functions for AST source mutating, ASM vector mutating, source to binary compilation, source to ASM compilation and ASM to binary compilation

```
import ShellWrapper
import AssemblerMutation
import pickle

COMPILER_METHODS = {"GCC": ShellWrapper.compileFilesToBinaryGCC,
                    "CLANG": ShellWrapper.compileFilesToBinaryClang}


def loadMutationInformationFile(pathToFile):
    return pickle.load(
        open(pathToFile, 'rb'))


def saveMutationInformationToFile(mutationInformation, pathToFile):
    pickle.dump(mutationInformation, open(pathToFile, 'wb'))


def getPathToImx28Precompiled(compiler, optimizationFlag, fixedRegister):
    path = "/imx28/precompiled/" + compiler.lower() + "/" + optimizationFlag[2:]
    if fixedRegister is None:
        path += "/" + "use_all_registers"
    else:
        path += "/" + fixedRegister[9:]
    return path


def generateMutationInformation(listOfMutatedFiles):
    tempList = []
    for mutatedFile in listOfMutatedFiles:
        mutationDepth = ShellWrapper.getMutationDepthOfFile(mutatedFile)
        tempList.append({"path": mutatedFile, "mutationDepth": mutationDepth})
    return tempList


def compileToASM(listOfMutatedFiles, pathToOriginalMainFile, compiler, listOfOptimizationFlags,
                 listOfFixedRegisters):
    pathToMainFileInTemporaryProject = ShellWrapper.getPathToMutatedFileInTemporaryProject(
        pathToOriginalMainFile,
        False)
    ShellWrapper.generateTemporaryProject(pathToOriginalMainFile)
    for mutatedFile in listOfMutatedFiles:
        ShellWrapper.copyMutationToTemporaryProject(mutatedFile)
    ShellWrapper.compileFilesToASM(pathToMainFileInTemporaryProject,
                                   "_".join(listOfOptimizationFlags) + "_" + "_".join(
                                       listOfFixedRegisters), compiler)

    pathToASMDirectory = ShellWrapper.createNewASMFolder(pathToOriginalMainFile)
    ShellWrapper.copyASMFilesToDirectory(pathToMainFileInTemporaryProject, pathToASMDirectory)
    # Generate mutation information
    tempList = generateMutationInformation(listOfMutatedFiles)
    mutationInformation = {"usedSourceMutations": tempList,
                           "fixedRegisters": listOfFixedRegisters,
                           "optimizationFlags": listOfOptimizationFlags, "compiler": compiler}
    saveMutationInformationToFile(mutationInformation,
                                  pathToASMDirectory + '/mutationInformation.txt')
    return pathToASMDirectory + "/" + ShellWrapper.getFileNameWithoutExtension(
        pathToOriginalMainFile) + ".s"


def compileSourceToBinary(listOfMutatedFiles, pathToOriginalMainFile, pathToDestinationDirectory,
                          compiler,
                          listOfOptimizationFlags, listOfFixedRegisters):
    pathToMainFileInTemporaryProject = ShellWrapper.getPathToMutatedFileInTemporaryProject(
        pathToOriginalMainFile,
        False)
    ShellWrapper.generateTemporaryProject(pathToOriginalMainFile)
    for mutatedFile in listOfMutatedFiles:
        ShellWrapper.copyMutationToTemporaryProject(mutatedFile)
    if len(listOfFixedRegisters) == 0:
        fixedRegister = None
    else:
        fixedRegister = listOfFixedRegisters[0]
    optimizationFlag = listOfOptimizationFlags[0]
    pathToImx28 = getPathToImx28Precompiled(compiler, optimizationFlag, fixedRegister)
    # Compile source files or ASM files
    COMPILER_METHODS[compiler](
        pathToMainFileInTemporaryProject, pathToImx28,
        "␣".join(listOfOptimizationFlags) + "␣" + "␣".join(
            listOfFixedRegisters))
    binaryName = ShellWrapper.copyBinaryToDirectoryAndRenameUniquely(
        pathToMainFileInTemporaryProject,
        pathToDestinationDirectory)
    # Generate mutation information
    tempList = generateMutationInformation(listOfMutatedFiles)
    mutationInformation = {"usedSourceMutations": tempList,
                           "fixedRegisters": listOfFixedRegisters,
                           "optimizationFlags": listOfOptimizationFlags, "compiler": compiler,
                           "binaryName": binaryName}
    saveMutationInformationToFile(mutationInformation,
                                  pathToDestinationDirectory + '/' + binaryName + '.info')


def compileASMToBinary(listOfMutatedFiles, pathToOriginalMainFile, pathToDestinationDirectory):
    pathToMainFileInTemporaryProject = ShellWrapper.getPathToMutatedFileInTemporaryProject(
        pathToOriginalMainFile,
        False)
    pathToOriginalFile = pathToOriginalMainFile
    pathToASMDirectory = ShellWrapper.getPathToASMDirectory(pathToOriginalFile)
    ShellWrapper.generateTemporaryProject(pathToOriginalMainFile)
    for mutatedFile in listOfMutatedFiles:
        ShellWrapper.copyMutationToTemporaryProject(mutatedFile)
    mutationInformationSource = loadMutationInformationFile(
        pathToASMDirectory + '/mutationInformation.txt')
    fixedRegister = mutationInformationSource["fixedRegisters"]
    if len(fixedRegister) == 0:
        fixedRegister = None
    else:
        fixedRegister = fixedRegister[0]
    optimizationFlag = mutationInformationSource["optimizationFlags"][0]
    optimizationFlagsString = "␣".join(mutationInformationSource["optimizationFlags"])
    fixedRegistersString = "␣".join(mutationInformationSource["fixedRegisters"])
    compiler = mutationInformationSource["compiler"]
    pathToImx28 = getPathToImx28Precompiled(compiler, optimizationFlag, fixedRegister)
    ShellWrapper.compileFilesToBinaryGCC(pathToMainFileInTemporaryProject, pathToImx28,
                                         optimizationFlagsString + "␣" + fixedRegistersString)
    binaryName = ShellWrapper.copyBinaryToDirectoryAndRenameUniquely(
        pathToMainFileInTemporaryProject,
        pathToDestinationDirectory)
    # Generate mutation information
    tempList = generateMutationInformation(listOfMutatedFiles)
    mutationInformationSource["usedASMMutations"] = tempList
    mutationInformationSource["binaryName"] = binaryName
    saveMutationInformationToFile(mutationInformationSource,
                                  pathToDestinationDirectory + '/' + binaryName + '.info')


def mutateSourceFile(pathToFile, numberOfMutations, strictMode=True):
    isFileMutation = ShellWrapper.isFileMutation(pathToFile)
    pathToMutatedFileInTemporaryProject = ShellWrapper.getPathToMutatedFileInTemporaryProject(
        pathToFile,
        isFileMutation)
    pathToOriginalFile = pathToFile
    if isFileMutation:
        pathToOriginalFile = ShellWrapper.getPathToOriginalFile(pathToFile)
    ShellWrapper.emptyTempFolder()
    ShellWrapper.generateTemporaryProject(pathToOriginalFile)
    if isFileMutation:
        ShellWrapper.copyMutationToTemporaryProject(pathToFile)
    print "Mutate␣source␣file␣with␣" + numberOfMutations + "␣mutations"
    ShellWrapper.openSubprocessClangMutate(numberOfMutations, pathToMutatedFileInTemporaryProject)
    ShellWrapper.getMutationsFromTempDirectory(pathToMutatedFileInTemporaryProject)
    pathToImx28 = getPathToImx28Precompiled("GCC", "-O0", None)
    print "Compile␣mutations␣with␣GCC␣-O0"
    ShellWrapper.compileMutations(pathToMutatedFileInTemporaryProject, pathToImx28, "-O0")
    numberOfSuccessfulCompilations = ShellWrapper.getNumberOfSuccessfulCompilations(
        pathToOriginalFile)
    print "Number␣of␣successful␣compilations:␣" + numberOfSuccessfulCompilations
    ShellWrapper.generateTestResultsIfNeeded(pathToOriginalFile)
    print "Test␣compiled␣mutations"
    ShellWrapper.testCompiledMutations(pathToMutatedFileInTemporaryProject)
    numberOfPassedTests = ShellWrapper.getNumberOfSuccessfulMutations(
        pathToOriginalFile)
    print "Number␣of␣passed␣tests:␣" + numberOfPassedTests
    if numberOfPassedTests > 0:
        ShellWrapper.formatCodeOfMutations(pathToMutatedFileInTemporaryProject)
    if strictMode:
        print "Check␣predecessors␣for␣equal␣code"
        if isFileMutation:
            ShellWrapper.deleteMutationsIfPredecessorIsEqual(pathToFile)
            numberOfMutationsAfterPredecessorCheck = ShellWrapper.getNumberOfSuccessfulMutations(
                pathToOriginalFile)
            print "Number␣of␣mutations␣\
␣␣␣␣␣␣␣␣␣␣␣␣after␣predecessor␣check:␣" + numberOfMutationsAfterPredecessorCheck
    print "Save␣compiled␣mutations"
    ShellWrapper.saveTestedSourceFiles(pathToFile, isFileMutation)
    numberOfSuccessfulMutations = ShellWrapper.getNumberOfSuccessfulMutations(pathToOriginalFile)
    print "Number␣of␣successful␣mutations:␣" + numberOfSuccessfulMutations


def mutateAssemblerFile(pathToFile, numberOfMutations, strictMode=True):
    isFileMutation = ShellWrapper.isFileMutation(pathToFile)
    pathToMutatedFileInTemporaryProject = ShellWrapper.getPathToMutatedFileInTemporaryProject(
        pathToFile,
        isFileMutation)
    pathToOriginalFile = pathToFile
    if isFileMutation:
        pathToOriginalFile = ShellWrapper.getPathToOriginalFile(pathToFile)
    pathToASMDirectory = ShellWrapper.getPathToASMDirectory(pathToOriginalFile)

    mutationInformation = loadMutationInformationFile(
        pathToASMDirectory + '/mutationInformation.txt')
    fixedRegister = mutationInformation["fixedRegisters"]
    if len(fixedRegister) == 0:
        fixedRegister = None
    else:
        fixedRegister = fixedRegister[0]
    optimizationFlag = mutationInformation["optimizationFlags"][0]
    optimizationFlagsString = "␣".join(mutationInformation["optimizationFlags"])
    fixedRegistersString = "␣".join(mutationInformation["fixedRegisters"])
    compiler = mutationInformation["compiler"]
    pathToImx28 = getPathToImx28Precompiled(compiler, optimizationFlag, fixedRegister)
    ShellWrapper.emptyTempFolder()
    ShellWrapper.generateTemporaryProject(pathToOriginalFile)
    if isFileMutation:
        ShellWrapper.copyMutationToTemporaryProject(pathToFile)
    print "Mutate␣ASM␣file␣with␣" + numberOfMutations + "␣mutations"
    AssemblerMutation.applyMutations(pathToMutatedFileInTemporaryProject, int(numberOfMutations))
    ShellWrapper.getMutationsFromTempDirectory(pathToMutatedFileInTemporaryProject)
    print "Link␣mutations"
    ShellWrapper.compileMutations(pathToMutatedFileInTemporaryProject, pathToImx28,
                                  optimizationFlagsString + "␣" + fixedRegistersString)
    numberOfSuccessfulLinkings = ShellWrapper.getNumberOfSuccessfulCompilations(
        pathToOriginalFile)
    print "Number␣of␣successful␣linkings:␣" + numberOfSuccessfulLinkings
    print "Generating␣Test␣Results"
    ShellWrapper.generateTestResultsIfNeeded(pathToOriginalFile)
    print "Test␣Mutations"
    ShellWrapper.testCompiledMutations(pathToMutatedFileInTemporaryProject)
    numberOfSuccessfulTests = ShellWrapper.getNumberOfSuccessfulMutations(pathToOriginalFile)
    print "Number␣of␣passed␣tests:␣" + numberOfSuccessfulTests
    if strictMode:
        print "Check␣predecessors␣for␣equal␣code"
        if isFileMutation:
            ShellWrapper.deleteMutationsIfPredecessorIsEqual(pathToFile)
            numberOfMutationsAfterPredecessorCheck = ShellWrapper.getNumberOfSuccessfulMutations(
                pathToOriginalFile)
            print "Number␣of␣mutations␣\
␣␣␣␣␣␣␣␣␣␣␣␣after␣predecessor␣check:␣" + numberOfMutationsAfterPredecessorCheck

    print "Save␣Mutations"
    ShellWrapper.saveTestedSourceFiles(pathToFile, isFileMutation)
    numberOfSuccessfulMutations = ShellWrapper.getNumberOfSuccessfulMutations(pathToOriginalFile)
    print "Number␣of␣successful␣mutations:␣" + numberOfSuccessfulMutations
```

A.2 Shell Scripts

Listing A.4: Shell script to cross compile C programs with GCC.

```
#!/bin/sh
############### PARAMETERS ###################
FILE=$1
BUILD_DIRECTORY=$2
PATH_TO_PRECOMPILED_IMX28=$3
ADDITIONAL_BUILD_PARAMETERS="$4"

############### BODY #########################
FILE=$1
```

```
FILE=$(readlink $FILE -f)
DIRECTORY=$(dirname $FILE)

FILE_NAME=${FILE##*/}
FILE_NAME=${FILE_NAME%.*}

cd $DIRECTORY
[ -d $BUILD_DIRECTORY ] && rm -R $BUILD_DIRECTORY

FILES_TO_COMPILE=$(ls -d $PWD/* | grep "\.[cso]$")

mkdir $BUILD_DIRECTORY
cd $BUILD_DIRECTORY

COMMAND="arm-none-eabi-gcc␣-w␣-T␣~/imx28/imx28evk-ram.ld␣$FILES_TO_COMPILE
$PATH_TO_PRECOMPILED_IMX28/print_methods.o␣-mcpu=arm926ej-s␣␣-Xlinker␣-Map=hello.map␣-marm
-nostartfiles␣-lm␣~/imx28/crti.o␣~/imx28/crtend.o␣~/imx28/crtn.o␣~/imx28/crtbegin.o
~/imx28/specs.o␣-I␣~/imx28/␣$PATH_TO_PRECOMPILED_IMX28/debug_uart.o␣-I␣~/imx28/
-I␣~/imx28/registers/␣$PATH_TO_PRECOMPILED_IMX28/start.o
$PATH_TO_PRECOMPILED_IMX28/imx28evk-reset-ram.o␣$PATH_TO_PRECOMPILED_IMX28/arm-vector.o
-o␣$FILE_NAME␣$ADDITIONAL_BUILD_PARAMETERS"

eval $COMMAND

if [ ! -f $FILE_NAME ]
then
        cd ../
        rm -R $BUILD_DIRECTORY
        echo "BUILD␣NOT␣SUCCESSFUL!"
        return 1
fi
echo "BUILD␣SUCCESSFUL!"
return 0
```

Listing A.5: Shell script to cross compile C programs with CLANG.

```
#!/bin/sh
############### PARAMETERS ##################
FILE=$1
BUILD_DIRECTORY=$2
PATH_TO_PRECOMPILED_IMX28=$3
ADDITIONAL_BUILD_PARAMETERS="$4"

############### BODY #######################
FILE=$(readlink $FILE -f)
DIRECTORY=$(dirname $FILE)

FILE_NAME=${FILE##*/}
FILE_NAME=${FILE_NAME%.*}

cd $DIRECTORY
rm -R $BUILD_DIRECTORY

FILES_TO_LINK=$(ls -d $PWD/* | grep "\.[o]$")
FILE_NAMES_TO_COMPILE=$(ls -d $PWD/* | grep "\.[cs]$")

mkdir $BUILD_DIRECTORY
cd $BUILD_DIRECTORY

for FILE_TEMP_FULL_PATH in $FILE_NAMES_TO_COMPILE
do
        FILE_TEMP=${FILE_TEMP_FULL_PATH##*/}
        FILE_TEMP=${FILE_TEMP%.*}
        COMMAND="clang-3.6␣-w␣-c␣-target␣armv5te-none-eabi␣-ffreestanding␣-mcpu=arm926ej-s
␣␣␣␣␣␣␣␣-fmessage-length=0␣-marm␣␣$FILE_TEMP_FULL_PATH␣-I␣~/imx28/
␣␣␣␣␣␣␣␣-I␣/usr/lib/arm-none-eabi/include/
␣␣␣␣␣␣␣␣-I␣/usr/lib/gcc/arm-none-eabi/4.8.2/include/␣$ADDITIONAL_BUILD_PARAMETERS␣"
        eval $COMMAND
        FILES_TO_LINK=$FILES_TO_LINK"␣"$FILE_TEMP".o"
done

echo "Before␣Linking"
COMMAND="arm-none-eabi-gcc␣-w␣-T
~/imx28/imx28evk-ram.ld␣$FILES_TO_LINK␣$PATH_TO_PRECOMPILED_IMX28/print_methods.o
-mcpu=arm926ej-s␣␣-Xlinker␣-Map=hello.map␣-marm␣-nostartfiles␣-lm␣~/imx28/crti.o
~/imx28/crtend.o␣~/imx28/crtn.o␣~/imx28/crtbegin.o␣~/imx28/specs.o
-I␣~/imx28/␣$PATH_TO_PRECOMPILED_IMX28/debug_uart.o␣-I␣~/imx28/␣-I␣~/imx28/registers/
$PATH_TO_PRECOMPILED_IMX28/start.o␣$PATH_TO_PRECOMPILED_IMX28/imx28evk-reset-ram.o
$PATH_TO_PRECOMPILED_IMX28/arm-vector.o␣-o␣$FILE_NAME␣$ADDITIONAL_BUILD_PARAMETERS␣"
eval $COMMAND

if [ ! -f $FILE_NAME ]
then
        cd ../
        rm -R $BUILD_DIRECTORY
        return 1
fi
```

```
return 0
```

Listing A.6: Shell script to cross compile C programs with CLANG and GCC to assembler.

```
#!/bin/bash
############### PARAMETER CHECK #############
FILE=$1
ADDITIONAL_BUILD_PARAMETERS="$2"
COMPILER="$3"
############### BODY ########################
FILE=$(readlink $FILE -f)
DIRECTORY=$(dirname $FILE)
DIRECTORY=$DIRECTORY

FILE_NAME_WITH_EXTENSION=${FILE##*/}
FILE_NAME=${FILE_NAME_WITH_EXTENSION%.*}

BUILD_DIRECTORY="build"

cd $DIRECTORY
mkdir $BUILD_DIRECTORY

FILE_NAMES=$(ls -d $PWD/* | grep "\.c$")

cd $BUILD_DIRECTORY

#################### COMPILE ################
COMPILER_STRING=""
if [ $COMPILER == "GCC" ]; then
        COMPILER_STRING="arm-none-eabi-gcc␣-S␣-mcpu=arm926ej-s␣-fmessage-length=0␣-marm
␣␣␣␣␣␣␣␣--param␣ggc-min-expand=100␣--param␣ggc-min-heapsize=131072␣-I␣~/imx28/␣"
else
        COMPILER_STRING="clang-3.6␣-S␣-target␣armv5te-none-eabi␣-ffreestanding␣-mcpu=arm926ej-s
␣␣␣␣␣␣␣␣-fverbose-asm␣-fmessage-length=0␣-marm␣-I␣~/imx28/␣-I␣/usr/lib/arm-none-eabi/include/
␣␣␣␣␣␣␣␣-I␣/usr/lib/gcc/arm-none-eabi/4.8.2/include/"
fi

for FILE_TEMP_FULL_PATH in $FILE_NAMES
do
        FILE_TEMP=${FILE_TEMP_FULL_PATH##*/}

        COMMAND=$COMPILER_STRING"␣"$FILE_TEMP_FULL_PATH"␣"$ADDITIONAL_BUILD_PARAMETERS
        eval $COMMAND
done
```

Listing A.7: Start script for the QEMU based fault injection tool.

```
#!/bin/bash
if [ "$#" -ne 1 ]
then
  echo "Usage␣:␣./startTests.sh␣<path-to-config-file>"
exit 1
fi
DATA_COLLECTOR_PATH="data_collector.txt"
CONFIG_FILE=$1

counter=1
while read line
do
  KERNEL_PATH=$(echo "$line\n" | cut -f1 -d,)
  FAULT_LIBRARY_PATH=$(echo "$line\n" | cut -f2 -d ,)
  FILE_INPUT_ID=$(echo "$line\n" | cut -f3 -d ,)

  FAULT_COUNTER_ADDRESS=$(readelf $KERNEL_PATH -s | grep fault_counter)
  FAULT_COUNTER_ADDRESS=$(echo $FAULT_COUNTER_ADDRESS | cut -f2 -d :)
  FAULT_COUNTER_ADDRESS=$(echo $FAULT_COUNTER_ADDRESS | cut -f1 -d ' ')

  SBST_CYCLE_COUNT=$(readelf $KERNEL_PATH -s | grep sbst_cycle_count)
  SBST_CYCLE_COUNT=$(echo $SBST_CYCLE_COUNT | cut -f2 -d :)
  SBST_CYCLE_COUNT=$(echo $SBST_CYCLE_COUNT | cut -f1 -d ' ')

  INPUT_ADDRESS=$(readelf $KERNEL_PATH -s | grep input_data_id)
  INPUT_ADDRESS=$(echo $INPUT_ADDRESS | cut -f2 -d :)
  INPUT_ADDRESS=$(echo $INPUT_ADDRESS | cut -f1 -d ' ')

  ./qemu-system-arm -M imx28evk -m 128 -nographic -semihosting -kernel $KERNEL_PATH  -fi \
  $FAULT_COUNTER_ADDRESS,$FAULT_LIBRARY_PATH,$SBST_CYCLE_COUNT,$FILE_INPUT_ID,$INPUT_ADDRESS
  DATA_COLLECTOR_OUT="data_collector_$counter.txt"
  cat $DATA_COLLECTOR_PATH > $DATA_COLLECTOR_OUT
  counter=$((counter+1))
done < $1
#echo "Fault injection experiment finished"
```

A.3 XML

Listing A.8: A XML file to represent a template for permanent register faults.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<injection>
        <fault>
                <id>1</id>
                <component>REGISTER</component>
                <target>REGISTER CELL</target>
                <mode>WDF0</mode>
                <trigger>ACCESS</trigger>
                <type>PERMANENT</type>
                <params>
                        <address>${address}</address>
                        <mask>${mask}</mask>
                </params>
        </fault>
</injection>
```

Listing A.9: A XML file to represent a template for permanent instruction decoder faults.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<injection>
        <fault>
                <id>1</id>
                <component>CPU</component>
                <target>INSTRUCTION DECODER</target>
                <mode>NEW VALUE</mode>
                <trigger>ACCESS</trigger>
                <type>PERMANENT</type>
                <params>
                        <address>${address}</address>
                        <instruction>0xDEADBEEF</instruction>
                </params>
        </fault>
</injection>
```

A.4 Further Work

Listing A.10: Source code which is hard to mutate.

```c
int main( void ) {
  int column, row, index;
  column = index = row = 0;
  while( index++ < 3 )
    doCalc( column, row );
}
```

Listing A.11: Source code which is easy to mutate.

```c
int main( void ) {
  int column = 0;
  int row = 0;
  int index = 0;

  while( index < 3 ) {
    index = index + 1;
    doCalc( column, row );
  }
}
```

Bibliography

[AAVdG01] Zaid Al-Ars and A Van de Goor. Static and dynamic behavior of memory cell array opens and shorts in embedded DRAMs. In *Proceedings of the conference on Design, automation and test in Europe*, pages 496–503. IEEE Press, 2001.

[AC77] Algirdas Avizienis and Liming Chen. On the implementation of N-version programming for software fault tolerance during execution. *Proc. IEEE COMPSAC*, 77:149–155, 1977.

[AHK15] Tobias Rauter Andrea Höller, Nermin Kajtazovic and Christian Kreiner. Adaptive Automated Software Diversity: Towards Dynamic Hardware Fault Tolerance. 2015.

[AK84] Algirdas Avizienis and John PJ Kelly. Fault tolerance by design diversity: Concepts and experiments. *Computer*, 8(17):67–80, 1984.

[AK88] Paul E Ammann and John C Knight. Data diversity: An approach to software fault tolerance. *Computers, IEEE Transactions on*, 37(4):418–425, 1988.

[Arl90] Jean Arlat. *Validation de la sûreté de fonctionnement par injection de fautes, méthode- mise en oeuvre- application.* PhD thesis, 1990.

[ARM08] ARM Limited. ARM926EJ-S Development Chip Reference Manual. Technical report, ARM Limited, 2008. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0198e/index.html.

[AVFK01] Joakim Aidemark, Jonny Vinter, Peter Folkesson, and Johan Karlsson. Goofi: Generic object-oriented fault injection tool. In *Dependable Systems and Networks, 2001. DSN 2001. International Conference on*, pages 83–88. IEEE, 2001.

[Avi95] Algirdas Avizienis. The methodology of n-version programming. *Software fault tolerance*, 3:23–46, 1995.

[BAFS05] Elena Gabriela Barrantes, David H Ackley, Stephanie Forrest, and Darko Stefanović. Randomized instruction set emulation. *ACM Transactions on Information and System Security (TISSEC)*, 8(1):3–40, 2005.

[BAM14] Benoit Baudry, Simon Allier, and Martin Monperrus. Tailored source code transformations to synthesize computationally diverse program variants. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 149–159. ACM, 2014.

[BDS03] Sandeep Bhatkar, Daniel C DuVarney, and Ron Sekar. Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *USENIX Security*, volume 3, pages 105–120, 2003.

[Bel05] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46, 2005.

[BFM10] Jean-Marie Borello, Éric Filiol, and Ludovic Mé. From the design of a generic metamorphic engine to a black-box classification of antivirus detection techniques. *Journal in computer virology*, 6(3):277–287, 2010.

[BK04] Stephen W Boyd and Angelos D Keromytis. SQLrand: Preventing SQL injection attacks. In *Applied Cryptography and Network Security*, pages 292–302. Springer, 2004.

[BM14] Benoit Baudry and Martin Monperrus. The Multiple Facets of Software Diversity: Recent Developments in Year 2000 and Beyond. *CoRR*, abs/1409.7324, 2014.

[CDSDB13] Bart Coppens, Bjorn De Sutter, and Koen De Bosschere. Protecting your software updates. *Security & Privacy, IEEE*, 11(2):47–54, 2013.

[CEF$^+$06] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-Tuong, and Jason Hiser. N-variant systems: a secretless framework for security through diversity. In *Usenix Security*, volume 6, pages 105–120, 2006.

[CKMT10] Tsong Yueh Chen, Fei-Ching Kuo, Robert G Merkel, and TH Tse. Adaptive random testing: The art of test case diversity. *Journal of Systems and Software*, 83(1):60–66, 2010.

[CMMN12] Christian Collberg, Sam Martin, Jonathan Myers, and Jasvir Nagra. Distributed application tamper detection via continuous software updates. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 319–328. ACM, 2012.

[CMR$^+$01] Pierluigi Civera, Luca Macchiarulo, Maurizio Rebaudengo, M Sonza Reorda, and Massimo Violante. Exploiting FPGA for accelerating fault injection experiments. In *On-Line Testing Workshop, 2001. Proceedings. Seventh International*, pages 9–13. IEEE, 2001.

[Com15] Apple Computer. qsort.c. `http://www.opensource.apple.com/source/xnu/xnu-1456.1.26/bsd/kern/qsort.c`, 2015. [Online; accessed 14-August-2015].

[Cpr15a] Cprogramming.com. Merge Sort in C++. `http://www.cprogramming.com/tutorial/computersciencetheory/merge.html`, 2015. [Online; accessed 14-August-2015].

[Cpr15b] Cprogramming.com. Simple swapping heapsort source code. `http://www.cprogramming.com/snippets/source-code/simple-swapping-heapsort`, 2015. [Online; accessed 14-August-2015].

[DER05] Hyunsook Do, Sebastian G. Elbaum, and Gregg Rothermel. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[dev15] QEMU developers. QEMU Internals. `http://qemu.weilnetz.de/qemu-tech.html`, 2015. [Online; accessed 14-August-2015].

[Dub13] Elena Dubrova. *Fault-tolerant design.* Springer, 2013.

[ES03] Agoston E Eiben and James E Smith. *Introduction to evolutionary computing.* Springer Science & Business Media, 2003.

[Fou15] Free Software Foundation. Options for Linking. `https://gcc.gnu.org/onlinedocs/gcc/Link-Options.html`, 2015. [Online; accessed 14-August-2015].

[Fra10] Michael Franz. E unibus pluram: massive-scale software diversity as a defense mechanism. In *Proceedings of the 2010 workshop on New security paradigms*, pages 7–16. ACM, 2010.

[FS10] Blair Foster and Anil Somayaji. Object-level recombination of commodity applications. In *Proceedings of the 12th annual conference on Genetic and evolutionary computation*, pages 957–964. ACM, 2010.

[FSK98] Peter Folkesson, Sven Svensson, and Johan Karlsson. A comparison of simulation based and scan chain implemented fault injection. In *Fault-Tolerant Computing, 1998. Digest of Papers. Twenty-Eighth Annual International Symposium on*, pages 284–293. IEEE, 1998.

[Gil05] Walter R Gilks. *Markov chain monte carlo.* Wiley Online Library, 2005.

[Gra15] Mentor Graphics. Sourcery CodeBench. `http://www.mentor.com/embedded-software/sourcery-tools/sourcery-codebench/overview`, 2015. [Online; accessed 14-August-2015].

[GRE$^+$01] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, pages 3–14. IEEE, 2001.

[HNL$^+$13] Andrei Homescu, Steven Neisius, Per Larsen, Stefan Brunthaler, and Michael Franz. Profile-guided automated software diversity. In *Code Generation and Optimization (CGO), 2013 IEEE/ACM International Symposium on*, pages 1–11. IEEE, 2013.

[HNTC$^+$12] Jason Hiser, Anh Nguyen-Tuong, Michele Co, Mathew Hall, and Jack W Davidson. ILR: Where'd My Gadgets Go? In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 571–585. IEEE, 2012.

[HRIK15] Andrea Höller, Tobias Rauter, Johannes Iber, and Christian Kreiner. Diverse Compiling for Microprocessor Fault Detection in Temporal Redundant Systems. In *The 13th IEEE International Conference on Dependable Autonomic and Secure Computing*, 2015.

[Inc15] Apple Inc. Apple Open Source. `http://www.opensource.apple.com/`, 2015. [Online; accessed 14-August-2015].

[jac08] The superdiversifier: Peephole individualization for software protection, author=Jacob, Matthias and Jakubowski, Mariusz H and Naldurg, Prasad and Saw, Chit Wei Nick and Venkatesan, Ramarathnam. In *Advances in Information and Computer Security*, pages 100–120. Springer, 2008.

[KBLN04] Miryung Kim, Lawrence Bergman, Tessa Lau, and David Notkin. An ethnographic study of copy and paste programming practices in OOPL. In *Empirical Software Engineering, 2004. ISESE'04. Proceedings. 2004 International Symposium on*, pages 83–92. IEEE, 2004.

[KKA92] Ghani A Kanawati, Nasser A Kanawati, and Jacob A Abraham. FERRARI: A tool for the validation of system dependability properties. In *Fault-Tolerant Computing, 1992. FTCS-22. Digest of Papers., Twenty-Second International Symposium on*, pages 336–344. IEEE, 1992.

[KLD$^+$94] Johan Karlsson, Peter Liden, Peter Dahlgren, Rolf Johansson, and Ulf Gunneflo. Using heavy-ion radiation to validate fault-handling mechanisms. *IEEE micro*, (1):8–11, 1994.

[KML$^+$06] Nektarios Kranitis, Andreas Merentitis, N Laoutaris, George Theodorou, A Paschalis, Dimitris Gizopoulos, and Constantin Halatsis. Optimal periodic testing of intermittent faults in embedded pipelined processor applications. In *Proceedings of the conference on Design, automation and test in Europe: Proceedings*, pages 65–70. European Design and Automation Association, 2006.

[koo15] Kuo Zuo koon. The k out of n system model. `http://www.ewp.rpi.edu/hartford/~ernesto/S2008/SMRE/Papers/Kuo-Zuo-koon.pdf`, 2015. [Online; accessed 14-August-2015].

[LA04] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed* *and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[Lat08a] Chris Lattner. LLVM and CLang: Next Generation Compiler Technology. *The BSD Conference*, pages 1–2, 2008.

[Lat08b] Chris Lattner. LLVM and Clang: Next generation compiler technology. In *The BSD Conference*, pages 1–2, 2008.

[Law96] Kevin P Lawton. Bochs: A portable pc emulator for unix/x. *Linux Journal*, 1996(29es):7, 1996.

[LDE$^+$07] JC LKnight, JW Davidson, D Evans, A Nguyen-Tuong, and C Wang. Genesis: A Framework for Achieving Software Component Diversity. Technical report, DTIC Document, 2007.

[LHBF14] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. SoK: Automated Software Diversity. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP '14, pages 276–291, Washington, DC, USA, 2014. IEEE Computer Society.

[MRMS94] Henrique Madeira, Mário Rela, Francisco Moreira, and João Gabriel Silva. RIFLE: A general purpose pin-level fault injector. In *Dependable ComputingEDCC-1*, pages 197–216. Springer, 1994.

[NB10] Gene Novark and Emery D. Berger. DieHarder: Securing the Heap. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, pages 573–584, New York, NY, USA, 2010. ACM.

[Nel15] Ed Nelson. newlib-no-uint-type-defines.diff. `https://sourceware.org/ml/newlib/2014/msg00082/newlib-no-uint-type-defines.diff`, 2015. [Online; accessed 14-August-2015].

[NMRW02a] G.C. Necula, S. McPeak, S.P. Rahul, and W. Weimer. Cil: An infrastructure for C program analysis and transformation. *International Conference on Compiler Construction*, pages 213–228, 2002.

[NMRW02b] George C Necula, Scott McPeak, Shree P Rahul, and Westley Weimer. CIL: Intermediate language and tools for analysis and transformation of C programs. In *Compiler Construction*, pages 213–228. Springer, 2002.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[OCJ08] Jon Oberheide, Evan Cooke, and Farnam Jahanian. CloudAV: N-Version Antivirus in the Network Cloud. In *USENIX Security Symposium*, pages 91–106, 2008.

[OMM02] Nahmsuk Oh, Subhasish Mitra, and Edward J McCluskey. ED 4 I: error detection by diverse data and duplicated instructions. *Computers, IEEE Transactions on*, 51(2):180–199, 2002.

[PLMK08] Riccardo Poli, William B Langdon, Nicholas F McPhee, and John R Koza. *A field guide to genetic programming.* Lulu. com, 2008.

[PPK12] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 601–615. IEEE, 2012.

[PSS12] Peter Popov, Vladimir Stankovic, and Lorenzo Strigini. An Empirical Study of the Effectiveness of „Forcing“ Diversity Based on a Large Population of Diverse Programs. In *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*, pages 41–50. IEEE, 2012.

[Pul01] Laura L Pullum. *Software fault tolerance techniques and implementation.* Artech House, 2001.

[RPB$^+$01] C Rousselle, Matthias Pflanz, A Behling, T Mohaupt, and H Vierhaus. A register-transfer-level fault simulator for permanent and transient faults in embedded processors. In *date*, page 0811. IEEE, 2001.

[Sch14] Eric Schulte. *Neutral Networks of Real-World Programs and their Application to Automated Software Evolution.* PhD thesis, University of New Mexico, Albuquerque, USA, July 2014. https://cs.unm.edu/ eschulte/dissertation.

[Sch15a] Gerhard Schoenfelder. FIES: A Fault Injection Framework for the Evaluation of Self-Tests. Master's thesis, Graz University of Technology, 2015.

[Sch15b] Eric Schulte. Clang-Mutate: Manipulate C-family ASTs with Clang. `https://github.com/eschulte/clang-mutate`, 2015. [Online; accessed 14-August-2015].

[SDMHR11] Stelios Sidiroglou-Douskos, Sasa Misailovic, Henry Hoffmann, and Martin Rinard. Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 124–134. ACM, 2011.

[Sem11] Freescale Semiconductor. i.MX28 EVK Hardware User's Guide. Technical Report 924-76415, Freescale Semiconductor, 2011.

[SFF$^+$14] Eric Schulte, Zachary P. Fry, Ethan Fast, Westley Weimer, and Stephanie Forrest. Software Mutational Robustness. *Genetic Programming and Evolvable Machines*, 15(3):281–312, September 2014.

[SFW10] Eric Schulte, Stephanie Forrest, and Westley Weimer. Automated Program Repair Through the Evolution of Assembly Code. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 313–316, New York, NY, USA, 2010. ACM.

[STB97] Volkmar Sieh, Oliver Tschache, and Frank Balbach. VERIFY: Evaluation of reliability using VHDL-models with embedded fault descriptions. In *Fault-Tolerant Computing, 1997. FTCS-27. Digest of Papers., Twenty-Seventh Annual International Symposium on*, pages 32–36. IEEE, 1997.

[Tea15a] The Clang Team. Clang 3.8 documentation: LibTooling. `http://clang.llvm.org/docs/LibTooling.html`, 2015. [Online; accessed 14-August-2015].

[Tea15b] The Clang Team. Clang Compiler Users Manual. `http://clang.llvm.org/docs/UsersManual.html`, 2015. [Online; accessed 14-August-2015].

[Tea15c] The Clang Team. Clang::ASTConsumer Class Reference. `http://clang.llvm.org/doxygen/classclang_1_1ASTConsumer.html`, 2015. [Online; accessed 14-August-2015].

[Tea15d] The Clang Team. ClangFormat. `http://clang.llvm.org/docs/ClangFormat.html`, 2015. [Online; accessed 14-August-2015].

[Tea15e] The Clang Team. Cross-compilation using Clang. `http://clang.llvm.org/docs/CrossCompilation.html`, 2015. [Online; accessed 14-August-2015].

[Tea15f] The Clang Team. Introduction to the Clang AST. `http://clang.llvm.org/docs/IntroductionToTheClangAST.html`, 2015. [Online; accessed 14-August-2015].

[Tea15g] The Clang Team. JSON Compilation Database Format Specification. `http://clang.llvm.org/docs/JSONCompilationDatabase.html`, 2015. [Online; accessed 14-August-2015].

[TIJ96] Timothy K Tsai, Ravishankar K Iyer, and Doug Jewitt. An approach towards benchmarking of fault-tolerant commercial systems. In *Fault Tolerant Computing, 1996., Proceedings of Annual Symposium on*, pages 314–323. IEEE, 1996.

[WDHK01] Chenxi Wang, Jack Davidson, Jonathan Hill, and John Knight. Protection of software-based survivability mechanisms. In *Dependable Systems and Networks, 2001. DSN 2001. International Conference on*, pages 193–202. IEEE, 2001.

[WHD$^+$09] Daniel Williams, Wei Hu, Jack W Davidson, Janson D Hiser, John C Knight, and Anh Nguyen-Tuong. Security through diversity: Leveraging virtual machine technology. *Security & Privacy, IEEE*, 7(1):26–33, 2009.

[Wik15] Wikipedia. Translation unit (programming) — Wikipedia, The Free Encyclopedia, 2015. [Online; accessed 1-September-2015].

[WWB03] Rong Wang, Feiyi Wang, and Gregory T Byrd. Design and implementation of Acceptance Monitor for building intrusion tolerant systems. *Software: Practice and Experience*, 33(14):1399–1417, 2003.

[ZAV$^+$04] Haissam Ziade, Rafic A Ayoubi, Raoul Velazco, et al. A survey on fault injection techniques. *Int. Arab J. Inf. Technol.*, 1(2):171–186, 2004.