Clemens Terasa

A Valgrind-based Soft Error Injection Tool for SIHFT Evaluations

27. März 2013

supervised by:
Prof. Dr. Sibylle Schupp

Hamburg University of Technology (TUHH)
Technische Universität Hamburg-Harburg
Institute for Software Systems
21073 Hamburg
Erklärung der Selbstständigkeit

Hiermit versichere ich, die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie die Zitate deutlich kenntlich gemacht zu haben.

Hamburg, den 27. März 2013  Clemens Terasa
## Contents

1 Introduction .......................... 1  
   1.1 Motivation .......................... 1  
   1.2 Problem description ................. 1

2 Software Implemented Hardware Fault Tolerance .......................... 3  
   2.1 Fault Tolerance ....................... 3  
   2.2 Soft Errors .......................... 4  
      2.2.1 Hardening in hardware ............. 6  
      2.2.2 Hardening in software ............. 7  
   2.3 Software Implemented Hardware Fault Tolerance (SIHFT) ................. 8  
      2.3.1 Data hardening ..................... 8  
      2.3.2 Control flow hardening ............. 9  
      2.3.3 Instruction hardening ............. 10  
   2.4 Abstractions of SIHFT implementation .................. 12

3 Fault Injection .......................... 15  
   3.1 Software quality ...................... 15  
   3.2 Fault injection is testing ............ 17  
   3.3 Abstraction level ..................... 18  
   3.4 Target of injections ................... 19

4 FITIn .................................. 21  
   4.1 Binary Instrumentation ............... 21  
   4.2 Valgrind ............................. 22  
      4.2.1 Valgrind overview .................. 22  
      4.2.2 Valgrind API basics ............... 22  
   4.3 FITIn overview ........................ 25  
   4.4 Target preparation and source code annotation .................. 26  
   4.5 Monitoring ........................... 28  
   4.6 Injection mode ......................... 29  
   4.7 Limits of FITIn and possible improvements .................. 31

5 Experiments ............................ 33  
   5.1 Basic considerations ................. 33  
   5.2 Dhrystone ............................ 34  
      5.2.1 Overview and history ............... 34  
      5.2.2 Customizations ..................... 35  
      5.2.3 Integration of Flipsafe ............. 36
1 Introduction

1.1 Motivation

Large amounts of technology today rely on computers and highly integrated circuits (IC). Integrated chips are used in almost every application from embedded devices like smart phones over satellites and motorized vehicles to large computer server farms. Most of these systems consist of multiple components, which all have their own ICs and each single component may be essential for a system to work. Especially in critical applications like aeronautics or motorized vehicles fault tolerance is highly important as it is crucial for safety.

Fault tolerance in systems can be achieved in different ways and at different scope. For example in aeronautics, passenger planes are equipped with more than one engine to cope with a possible failure of one. Having more than one critical component, thus redundancy of components, is a standard way of achieving fault tolerance, because it is less likely that all these components fail at the same time.

The same idea of redundancy is applicable on computing devices. To improve the fault tolerance of server storage systems with multiple hard disks are used. However, not only the long term data storage may fail but also the processor or volatile memory may get corrupted.

Computing devices can be tested beforehand for errors but might still fail during usage. The most problematic failures are failures that seem to emerge randomly and do not break the system hardware permanently. In micro systems these errors are called soft errors.

Soft errors may emerge in memory cells like SRAM and DRAM and mostly have physical causes. Cosmic rays or overheating may cause the bits in a memory cell to switch their states temporarily or permanently. Such errors are called soft errors. If only one bit flips it is called a Single Event Upset (SEU) if multiple bits are affected this is called a Multiple Bit Error (MBE).

Because of these errors, critical computer systems have to be designed to be tolerant against this kind of faults. To reduce the effect of soft errors, systems may be hardened. This can be done on different levels. To be more robust specially designed hardware with shielded packages and custom ICs may be used. This, of course, is rather costly compared to commercial off-the-shelf (COTS) hardware.

1.2 Problem description

Hardening can also be done in software, running on COTS hardware, and is relatively cheap to use compared to other hardening methods. Software Implemented Hardware
Fault Tolerance (SIHFT) techniques are employed for this purpose.

The quality of SIHFT techniques is generally evaluated through simulations. In the case of soft errors, the hardware errors have to be simulated using the technique of fault injection. Injecting faults into the hardware directly would be rather complicated as special hardware would be necessary.

This work introduces a new software tool, FITIn, to inject faults into a binary executable at run-time. It is a tool that uses the dynamic binary instrumentation framework Valgrind. FITIn was designed and implemented with SIHFT technique evaluations in mind, but may also be useful for other types of fault injection experiments. The simulated errors are single bit flips in the main memory, but the tool may be extended to inject more error types into different kinds of storage units.

FITIn uses a source code annotation API, which helps Valgrind to specify possible injection targets. Further, FITIn implements an algorithm that uses the binary instrumentation functions provided by Valgrind to monitor memory load operations and inject single bit faults. The tool currently only supports the AMD64 architecture with the Linux operating system. Other systems may work but are not targeted.

To evaluate FITIn a customized Dhrystone benchmark is used. The customizations consist of source code annotations necessary for FITIn, elimination of run-time dependencies, and the integration of Flipsafe, a type-based SIHFT library. Two experiment setups will be described and the results presented.

In the following SHIFT will be introduced in Chapter 2. Therefore the categorization into the software development domain and taxonomy has to be defined. After this the technique of fault injection will be presented in Chapter 3. In Chapter 4 follows the main part of this work, the tool description. It consists of the Valgrind description, an architectural overview, and implementation details of the FITIn. Chapter 5 proposes possible experiment setups and gives examples of conducted experiments.
2 Software Implemented Hardware Fault Tolerance

In this chapter, the technique of Software Implemented Hardware Fault Tolerance (SIHFT) will be introduced. We first discuss context of faults and errors as well as different hardening methods and abstraction of SIHFT techniques. The categorization relies on the definitions made by Avižienis et al. [2].

2.1 Fault Tolerance

To discuss the term Fault Tolerance it is necessary to introduce an abstract view of a system. A system consists of one or multiple components and interacts with its environment. The environment can be other systems, the physical outer world, or a user. The components themselves can also be seen as systems of their own.

This abstraction allows a modular view on complex systems. The common approach is to break a system into smaller parts, which are usually easier to understand than the whole system [3]. In the following, the term system refers to a computer system although same the discussion applies to all kinds of systems.

Usually a system serves a purpose, which is somehow specified, and is called a system function. In other words, the system function is the way a system is supposed to work. In industry, these functions are often formally written down in a functional specification document.

Further an implemented system has a behavior which should reassemble the system function. The behavior describes what the implementation actually does. It is not always the case that the system behaves according to its system function. A system that does not behave according to the functional specification has a system failure, or simply failure.

The cause of failures are called faults. Many different kinds of faults exist. Examples include intentional and unintentional faults, faults in the specification or implementation, and faults according to abstraction. The most obvious failure is the system outage. Depending on the system, outages can reach from planned to even catastrophic. While the outage of a consumer product like a TV is non-critical, the failure of an airplane during flight has to be prevented at all costs.

If a failure happens, the part of the system that leads to this is called an error. Although the overall system does not behave according to the functional specification, only a subpart may be affected by an error. Other parts might still behave consistent with a system function. It is obvious that it is important to reduce the number of errors in a system to get an overall system that behaves according to its specification. To that end we have to reduce the influence of faults on the system.
The reduction of fault influences can be done in various ways. **Fault prevention**
tries to avoid faults in the first place. In the design phase this may mean that the
designer chooses one material over another to prevent breakage. Though it is a very
important strategy it is not always applicable in the real world. Some faults may not
be always prevented like failures due to physical damage.

A system that can deal with faults and still performs its function is **fault tolerant**.
Fault tolerance does not mean that the whole system behavior in case of an error is
the same as in the error-less case. It rather means that the critical functions of a
system still perform according to the specification.

Two strategies of fault tolerance can be distinguished. **Error detection** is used to
recognize an error when it emerged. After detection the system can react and deal
with the fault in a manner that suits the needs. As an example we could think of a
thermometer sensor in a car. Although it is not critical to the use of the car, it may
be important to know when this part has a malfunction. The car driver can then be
informed on the instrument panel.

In some cases a fault can be not only detected but also the system can **recover**.
Fault recovery deals with the fault in a way that the system can perform its function.
Recovery is often achieved with redundancy of components. As an example a passenger
airplane has multiple engines. This enables the system 'airplane' to deal with the
outage of one engine.

In Figure 2.1 the fault tolerance taxonomy according to Avižienis et al. is shown. It consists of the error detection and the recovery. The recovery itself can
be categorized in two branches, one that deals with the actual error and one that
handles the error's cause.

![Fault Tolerance Taxonomy](image-url)

**Figure 2.1:** Fault tolerance overview

It should be noted that the terminology of faults and errors is not always consistent
in the literature. The results of faults depend highly on the system itself. Usually the
faults are said to be **benign** if the system continues to work and no error is detectable,
**corrupted** if the system continues to work but the error is changing behavior, and
**catastrophic** or byzantine if the system crashes.

### 2.2 Soft Errors

In preparation of the discussion about Soft Errors (SE) we will first need to categorize
this kind of error. In the previous section the classification of faults and errors was
discussed and also the idea of fault-tolerant systems was introduced.

As an example let us assume we have an integrated circuit with memory units.
These memory units may be registers or main memory. If one of the memory units is
hit by cosmic rays, it may change its internal charge and so its logical state, although
it was not intended. This state change is called a soft error. An unintended change of state of one memory unit may lead to faulty behavior.

To discuss the properties of soft errors a finer classification of faults is necessary. Following Avižienis [2] a fault can be categorized by the phase of occurrence, the system boundaries, the phenomenological cause, the dimension, the objective, the intent, the capability, and the persistence. Figure 2.2 shows this taxonomy in a tree-like structure.

One basic classification of a soft error is its dimension as a hardware fault. In the example above the fault emerges in memory units. Of course these faults can be also observed in software as one is able to read out the erroneous memory unit, but the root cause is a hardware part.

Soft errors are, phenomenologically seen, natural faults. Thus these can also be considered as external faults because the root cause is not inside the system. Although the technology is part of the root cause of these errors, these errors should not be considered human-made.

Soft errors occur in the operational phase. Although they may be avoided with different hardware design in the development phase, the fault itself emerges during operation.

The persistence of soft errors is transient. In fact, these errors are in literature also often called transient faults [36]. In contrast to permanent faults, transient faults are to be volatile. That means that we cannot prove the existence of such a fault.
because it may occur or may not occur. If an error is not reproducible it is also harder to argue if the countermeasures work.

As a consequence, the statistical behavior of such an error has to be considered. In the case of soft errors the statistical analysis might get fairly complex. In case of cosmic rays as root cause we have to consider the energy and the density of the subatomic particles. Further, the position of the circuit in space, the position on the IC, the shielding, and its material have to be considered. We will instead use a fairly simple model for the simulations in Chapter 4.

From the software point of view the capability of a soft error is rather accidental than due to incompetence. It seems that the logical state of a memory unit changed without any action of software so it is not the software that leads to the fault. From the perspective of a hardware engineer the chances of soft errors are stochastic events, which should be prevented.

Because of their properties transient faults are non-deterministic. System designers can not predict if or when soft errors occur. As stated before, the occurrence is a stochastic event. But computer systems can be built in such way that the effects of the fault may not lead to an overall system failure. The aim is to build countermeasures that provide hardening against this kind of fault.

2.2.1 Hardening in hardware

Because of the non-determinism, the natural cause, and the transient property of a soft error, it cannot be removed but only avoided. This avoidance can be done in the hardware design phase, where properties of the circuit are chosen. These properties may include different structure sizes, materials, or shielding. Still it is only a technique of fault avoidance, not of fault removal. Rather the chance of an occurrence can be lowered so that the chance of a transient fault is negligible for the usual use case of the hardware. This increase of robustness is called hardening.

The standard low-level hardware development like the IC development procedure can be seen as a hardening strategy. The challenges of hardware designed lie often in the conflict between performance and reliability. An example of this is the voltage setting of an integrated circuit. On the one hand a hardware designer may want to lower the voltages to consume less power. On the other hand lower voltages may lead to state errors in the switching transistors, which is a reduction of reliability.

Special hardened hardware solutions are also possible. Not only the integrated parts of a computing systems may contribute to the robustness, also the packaging and shielding of the device have an impact. One major influence for soft errors are the rays that hit the computer chip. If the amount of particles can be reduced the danger of transient faults is lowered. With this in mind one might design specially shielded versions of computer chips.

Another influence is thermal heat. When an IC is overheating over the specified limit the guaranteed properties of the semiconductors are not met anymore. As a result, the logical states may differ from the expected results. A simple solution would be the cooling of the IC, which is done anyway to protect the IC from permanent damage.
When those rather passive strategies are not sufficient, the internal IC logic must be modified to behave in a fault-tolerant way. The basic fault tolerance strategy is to add redundancy. This might be done in several ways and in several dimensions. A simple protection is doubling. The hardware designers might choose to double the memory units and registers. The contents can then be the compared against each other. If an error occurs in one of the memory units the comparison fails and it can be reacted on.

Alternatively also tripling or error correcting codes implemented in hardware might be used to harden the data. This increases the used space of the IC because of added memory and logic. Also, the energy consumption will rise as more calculations have to be conducted.

All these strategies drive the overall system costs. Generally specialized hardware is more expensive than COTS due to the smaller number of produced units. Moreover, using off-the-shelf hardware enables the use of well established tools which come with it as well as the experience of a larger developer community.

Because of that cost factors there is the wish to replace special hardware with cheaper COTS modules when possible. Even in industries that traditionally use specialized hardware COTS products are considered as possible alternatives [27].

2.2.2 Hardening in software

As a result of the soft error avoidance done in hardware, it is usually assumed by the software developers that soft errors do not occur. Especially the operations done on the Central Processing Unit (CPU) and main memory are considered to be free of errors. Thus the data the main memory and CPU contains is not affected and there are no corruptions of the program flow. Given that assumption, we will look at different hardening strategies done in software.

One example of error avoidance done in software is a mechanism that deals with data corruption on permanent memory like a flash memory drive. Because of the underlying technology memory cells of flash memory wear out. This means that after many write circles these cells lose their ability to save data correctly.

The software answer to the problem of worn-out memory cells is called wear leveling. Wear leveling is a mechanism that maps memory addresses to different physical memory units at each write cycle. With this distribution scheme it is less likely that cells will be often rewritten and thus might increase the fault tolerance of the system.

Although this scheme is done in software, the user of the device does not need to know any details. As a common approach in software design the feature of wear leveling is designed transparently to the user of the memory. The user just uses the addresses, the mapping to the physical memory is done outside the user’s programming domain.
This abstraction is important for different reasons:

- Ease of use.
- Compatibility to different memory architectures.
- Interchangeability of hardware and software.

Today, wear leveling is done in the COTS hardware (but still in software) and no special support must be introduced to the system\[29\].

Another usage of hardening in software concerns the communication. A common approach for fault tolerance measures in communication are Error Detecting and/or Correcting Codes\[5\]. With ECCs it is possible to correct some of the faults that might occur during the communication phase.

Providing means for detecting and correcting faults is an advantage of digital communications over the traditional analog techniques, which require hardware solutions.

In the following we will focus on soft errors and assume that the occurrence of a transient error is possible. As a consequence the software designer cannot rely on the fact that memory cannot be changed without a software interaction. Further the assumption is not true anymore that all calculations and data transfers work correctly and have deterministic results.

### 2.3 Software Implemented Hardware Fault Tolerance (SIHFT)

The term 'Software Implemented Hardware Fault Tolerance' describes software-based techniques to deal with faults of the hardware. Techniques of SIHFT are a subset of general fault tolerance techniques. Thus they may provide fault tolerance functions, error detection, and recovery. This increase in fault tolerance might on the other hand lead to performance regressions like lowered speed or increased code size.

We have seen in the previous section that it is possible to prevent the system from errors that emerge during run-time and are permanent. We also explained that the general assumption of an error-free CPU and main memory is not valid. With soft errors in the CPU and main memory other countermeasures than the previously described ones have to be taken to provide fault tolerance. In particular we will discuss the data hardening of the main memory, the hardening of the control flow, and the hardening of instructions. All this needs to be checked during run-time in the field.

#### 2.3.1 Data hardening

The data that is targeted by SIHFT techniques are main memory, CPU cache, and register contents. Usually these units are not protected in cheap off-the-shelf hardware.

Common error detection methods include EDDI\[30\], ED\(^4\)I\[29\] and SWIFT\[34\]. Typically data hardening is based on redundant data and instructions at the instruction level. As an example of these methods EDDI will be explained further.
2.3 Software Implemented Hardware Fault Tolerance (SIHFT)

The Error Detection by Duplicated Instructions (EDDI) technique improves the robustness of a computer program using redundant instructions to provide SIHFT. Instructions will be executed twice, and compared afterwards. Consider the `ADD` operation in Listings 2.1 and 2.2 [30].

```
1 ADD R3, R1, R2 ; R3 ← R1 + R2
2 ADD R23, R21, R22 ; RI
3 BNE R3, R23, erHandl ; CI
```

Listing 2.1: Add instruction without EDDI  

Listing 2.2: Add instruction with EDDI

In both cases the registers R1 and R2 are added and stored in register R3. This is done in both versions. But the right version introduces another `ADD` instruction, which is performed on separate registers. These should contain the same (mirrored) information as the registers before. To check its validity a `BNE` instruction is introduced at the end, which may jump to an error handler function `erHandl`. If a soft error occurs in one of the data storing registers, the comparison will most likely evaluate to false and jump to the error handler. Otherwise the next instruction would be executed.

The EDDI technique is fairly simple, but adds overhead. It uses the original instruction, adds an redundant instruction, and compares. Furthermore it requires additional registers.

However, the EDDI technique is capable of reducing the effect of faults as the error handler may recover the error or repeat the erroneous computation. Of course there are still windows of vulnerabilities. For example the comparison might fail, although the inputs are equal. Still EDDI is a data hardening concept that improves the data safety.

2.3.2 Control flow hardening

Computer programs do not usually have a single execution path. Instead, the order of instructions will depend on input data and decisions done at run-time. All these topics rely on control flow.

The control flow denotes the sequence of executed instructions. The sequence may be altered by conditional statements or jump instructions. To visualize control flow usually Control Flow Graphs (CFG) are used.

Control flow graphs consist of basic blocks as vertices and the statement flow as edges. Basic blocks are a series of statements that are always executed together. We see that the flow of executed statements is no chain and that the flow depends on conditional decisions.

---

1 BNE: Branch if Not Equal
For a working program it is necessary that the control flow works as expected and deterministically. It should be impossible to execute a basic block without following the defined flow.

For instance in the CFG presented in Figure 2.3 it should be impossible to execute the basic block of the code lines 7 and 8 right after the basic block of line 3. To prevent 'illegal flow' control flow hardening can be used.

Control flow hardening describes techniques to prevent the control flow from getting corrupted. It ensures that the basic blocks are being executed in the correct order. There are several approaches to achieve control flow hardening [28][29]. Usually it is more complex than pure data hardening.

The correct execution order involves the test for jumps to the correct instruction. As these jumps may be derived from data, hardening of data also helps with the problem of control flow faults. However, in general it is necessary to check for valid start and endpoints of jump instructions. A deeper insight into control flow hardening beyond scope of this work, as the main contribution, FITIn, covers data manipulation only. Control flow fault injection may be a topic of further research and development of FITIn.

### 2.3.3 Instruction hardening

Another kind of hardening is instruction hardening. It is similar to data hardening, because instructions are encoded into binary data and stored similar to data. A major
difference, however, is that instructions are interpreted by the CPU and executed. The following part explains instruction processing using the Von Neumann architecture without going into detail, and then provides an overview of instruction hardening.

![Figure 2.4: Von Neumann architecture](image)

The Von Neumann architecture is shown in Figure 2.4. One basic idea of the Von Neumann architecture is the usage of one addressable memory for data and instructions. The instruction processing unit is in charge of fetching the correct program instruction from the memory, decoding it and fetching the operands. The arithmetic logical unit (ALU) executes the instruction and stores the result.

Instruction processing, also called instruction cycle, of a Von Neumann computer is presented in Figure 2.5. For the discussion it is not necessary to present the instruction cycle in detail. The main point is that the processing of instructions is itself a non-trivial task. It is assumed that the programmer has no influence on this step, as it is done merely by hardware. An exception might be micro-programmable architectures, which, however, are beyond the scope of this work.

![Figure 2.5: Instruction processing of a Von Neumann architecture](image)

Given this kind of architecture one might ask, what may happen in case of failures to the memory similar to those presented in Section 2.3.1? What happens to the
Let us suppose the memory is vulnerable to SEU effects and the main memory is corrupted. Because the instructions are stored in the memory similarly to data the instructions may have the same error effects as the data. The erroneous instruction will be processed according to the instruction processing (Figure 2.5). Yet, the input of the fault becomes observable when the decoding phase is executed. If the instruction has binary errors, say flipped bits, the decoding phase will most likely produce a wrong command, which is executed afterwards. This error will most likely have an influence on the whole program.

Although instruction hardening, in particular the prevention of erroneous effects caused by faulty instructions, may be critical for a real-world program, it is also harder to achieve purely in software. This is due to the fact that the instruction processing is done in hardware. Thus, there is no purely software-based hardening scheme known to the author that executes instruction hardening with SIHFT.

### 2.4 Abstractions of SIHFT implementation

Software solutions can get fairly complex. Therefore the different parts are encapsulated into abstraction layers.

The classical layer model shown in Figure 2.6 describes usage dependencies. Each layer provides functionality for the layer above and each layer uses the functions the layer below provides. The lowest layer is the hardware, which is in general not modifiable by software.

![Figure 2.6: Classical software layer model](image)

Although the figure shows the classical software architecture model, not all layers are necessary. Small embedded solutions, for example, may lack the operating system level, and may not have a dynamically usable user space. Still the following discussion holds with respect to error propagation, and the possibilities of prevention.

If a soft error emerges in hardware, the error will propagate through all layers and might cause system failure. SIHFT techniques are aimed to prevent fault propagation. As it is not trivial to analyze how the corrupted data may propagate, it is a good strategy to prevent the error as soon as possible. However, it is sufficient to prevent the error at the first usage of the faulty part of the program. A faulty part that is never used does not need to be corrected. But as the hardware is generally not modifiable and usually given beforehand, the software solution needs to handle soft
errors at the lowest layer available. Possibly the fault correction can be done at the driver or Operating System (OS) level. But a modification at these levels is not always possible or a rather complex work.

Another strategy may be to harden all binary programs running on the CPU including the user space programs and the OS. This can be achieved using a specialized compiler, which introduces the hardening at the instruction level. Several strategies of such system were proposed, including EDDI [30], EDI [29], and SWIFT [34].

Though the former proposals introduce hardening features, these techniques also introduce the dependency on a specialized compiler. A specialized compiler usually works for one architecture and OS only, thus the usage of a specialized compiler provides only non-portable solutions. In contrast, a goal of software engineering is to provide portable solutions, instead of isolated and non-portable ones.

Another approach, proposed by Munkby, puts the SIHFT methods into the more abstract high-level source code [24]. By this it is more portable than the compiler only version. It can be ported to different operating systems as well as used with different compilers. This approach can be used additionally to the compiler-based techniques to get another level of hardened software.

The source code approach allows various kinds of hardening algorithms that have different performance properties. An example of this approach may be found in the Flipsafe library created by Munkby at the Hamburg University of Technology (TUHH). It includes several hardened types for boolean variables.
3 Fault Injection

This chapter introduces the technique of **Fault Injection**. It is a common reliability testing approach, used to evaluate fault tolerance systems. First the terms of software quality and testing are introduced, then the the technique fault injection is discussed.

### 3.1 Software quality

Every technical system serves a function. This is also true for computing systems as described in the previous chapter. The assessment of quality is important to help with the system development. Verification and Validation may be used to get measures about the systems quality.

"Verification [is the] process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. [...] Validation [is the] process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements."[17]

Verification is the attempt of proving that the product does what it is supposed to do according to the specification. Validation on the other side tries to show if the specification complies to the requirements. These two concepts, also often called V&V, have a different view on a similar problem. It is often said that verification deals with the question "Do we build the product right?" while validation asks "Do we build the right product?"[3].

V&V for software is a topic in its own right and can not be fully featured in this work. However, an overview of verification will be given to set a context for software testing. The methods used by verification may be **Formal Verification** or **Testing**.

Formal verification uses formal methods, like logical reasoning and proof systems, to mathematically proof the correctness of software. Based on axioms, formal specifications have to be met, to build up a reasoning framework. The result of formal verification is the mathematical prove, that with given axioms the software is correct.

Usually such proof is a strong result but not easy to get for real software[10]. Again, the topic of formal verification is beyond scope of this work. A simpler way of validation is testing.

Many definitions of testing can be found in the literature. One definition says:

"Testing is the process of executing a program with the intent of finding errors."[25]
This definition has two main aspects. The first implication is that the program to be tested is actually executed. Execution of code is not necessary in formal verification, as this can be done statically without running the program. But it does not mean that the test is executed in the real productive environment. Testing with a real productive is not always possible and generally not wanted due to emerging errors. Then the real environment is merely simulated by the test environment.

The second idea of test definition above is the "intent of finding errors". This concept complies with the famous citation of Dijkstra that testing shows the presence of errors but not the absence \cite{33}. Testing is therefore fundamentally weaker than formal verification. It is generally not possible to prove the correctness of software using tests. In contrast, tests compare an outcome of a test run against an expected outcome. The idea is that a system behaves in the same way under similar conditions. If the conditions, however, are not similar, the outcome may differ from the tested cases.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{test_overview.png}
\caption{Simple overview of the software testing process}
\end{figure}

To look further into testing a common and simplified test driven-development approach is shown in Figure 3.1. Let us assume the aim of the development is to develop a sorting algorithm in a more complex program. The input is a list of elements that is to be sorted. At the beginning the problem is stated. The problem in this example is the sorting of a list. This problem should be solved using software mechanisms. The software is described in the specification and according to the specification. So the test parameters can be derived from the specification. In the example the test parameter checks if the software produces a sorted version of an input set.

Having the test parameters, the actual software can be developed. The product of the software development can then be verified against the test parameters. Usually the software does not meet the test requirements in the first iteration. In our example the first iteration might output an sorted list that misses elements from the input list. Then the software will be developed further until it meets the test requirements. In the end the software should output a sorted version of the input list.

Although errors may be found, tests also have limits. Testing shifts the problem of proving the correctness of software towards the problem of fulfilling tests. The
development might get misdirected to comply to the test, rather than implementing the correct product.

Another problem are the tests itself. Weak spots in the test, like under-specification or missing test cases, reduce the significance of tests. This problem is shown with the following unit test example of a \textit{C} programming language (C) code fragment in Listing 3.1.

```c
bool stringIsEmpty(char *string)
{
    return (string[0] == '\0');
}
```

\textbf{Listing 3.1:} Simple string emptiness check function

The \texttt{stringIsEmpty} function checks if the given string is an empty \texttt{C} string by comparing the first character against the null character.

A simple test suite could look like the Listing 3.2.

```c
assert(stringIsEmpty("") == true);
assert(stringIsEmpty("Some sort of string") == false);
```

\textbf{Listing 3.2:} Test suite for stringIsEmpty

Though it looks like the suite works for the tested strings, there are inputs that are not covered by the test cases. The important \texttt{NULL} pointer input case is not handled by the test suite. In fact the program would (hopefully) crash trying to dereference the \texttt{NULL} pointer.

This example demonstrates the problem with testing: A test is only as good as its test suite. Choosing a good test suite is a complex task of its own.

Although testing is a complex task in itself it is important in today’s software development where formal verification is not applicable. With software tests errors might be found already at the development phase that can then be fixed early, when tests are conducted already during this phase. This reduces the costs of fixing software compared to fixing systems already used in the field. This thesis describes in Chapter 4 a new software test tool using fault injection, with originally the purpose of evaluating SIHFT techniques.

\section*{3.2 Fault injection is testing}

The technique of fault injection can be considered a form of testing. This work proposes a fault injection scheme that simulates soft errors. To be able to talk about fault injection we need a definition first.

Fault injection is the method of intentionally introducing faults to a system in order to test the systems behavior under the influence of this faults.

To understand a system’s behavior during a fault the simplest way is to simulate this fault. The simulation and injection of a fault is, of course, only done during development and test phases and not in the field, because faults may have unexpected effects and the results could be catastrophic.
As with all test methods, the test parameters have to be determined. The following questions arise:

- Which part is to be tested?
- How to conduct the test?
- Is the test sufficient for its purpose?

The following parts will answer those questions for the case of soft errors.

### 3.3 Abstraction level

To determine the part to be tested we need a view on the whole system under consideration. Although usually only a subpart of a bigger system, software can be seen as a complex system of its own.

The software system itself can be broken down into subparts. A simplified list of different levels of abstraction is given in Table 3.1 which is also related to the earlier discussions about Figure 2.6. The table can be read from button to top, beginning with the least abstracted layer and increasing the abstraction. Each layer depends on the lower layer and abstracts it to provide functionality to the layer above.

<table>
<thead>
<tr>
<th>Layer</th>
<th>Injection method</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 Virtual machine (VM)</td>
<td>Modified VM</td>
</tr>
<tr>
<td>3 Operating System (OS)</td>
<td>Special OS functions</td>
</tr>
<tr>
<td>2 Binary assembly code</td>
<td>Static injection, dynamic injection</td>
</tr>
<tr>
<td>1 Executing hardware</td>
<td>Bus errors, debug interface</td>
</tr>
</tbody>
</table>

**Table 3.1: Abstraction hierarchy and injection methods**

The executing hardware is considered as the least abstracted layer. From the software point of view there is no way to change the given hardware, so there is generally no way of changing its behavior. But there are still possibilities to inject faults the system. A prominent example are bus error simulations. To inject faults into the system at the lowest level one needs special hardware [1], usually debug interfaces that reveal hardware internals to the developers.

The next higher layer is the binary assembly code. It is the binary code that runs directly on the hardware. Assembly code consists of the instructions and data that are needed to execute the program.

Injecting faults at this level means changing the instructions or data. The properties of the faults injected depend on the purpose of the injection tests. The two contents of the instructions, the operations and the data, have to be considered separately. Injection into the assembly code means adding, modifying, or deleting instructions or their operands.

There are two methods to inject into assembly code, in regard to the injection time. While static injection modifies the binary code prior to its execution, the dynamic injection does this at run-time. The proposed tool in this thesis uses an approach
that employs both strategies. This abstraction layer was chosen for FITIn to perform the fault injections as it is independent enough from the hardware and OS.

Continuing the layer discussion, the next layer is the OS layer. An operating system usually abstracts hardware interfaces and provides facilities like memory and resource management and process handling. All these parts are subject to faults. Simulating these failures operating systems may provide fault injection methods. The Linux kernel for instance supports fault injection of different kernel functions \[22\] (like memory allocation faults).

In this work the OS level is not considered to be a useful injection layer. As the target is to simulate transient hardware errors the OS layer is considered too abstract. Also, using this layer would require deep insight into the chosen OS and would be OS-specific.

The same argument holds for the virtual machine layer. A virtual machine interprets and runs software on a virtual system, which itself runs on an OS. These systems are also called run-time systems \[32\]. The most famous VM is probably the Java Virtual Machine (JVM) \[9\]. The purpose of the VM is to make the program as independent as possible from the underlying layers. With this approach only the VM needs to be system-dependent, the software that runs on the VM does not.

This abstraction makes it hard to examine the exact behavior of a program propagating though all the layers down to the hardware. Beyond that, fault injection at VM level is only suitable for programs running on those VMs.

We have seen that a software system consists of different layers. To inject simulated soft errors it is necessary to have access to low level code.

### 3.4 Target of injections

As the method and the system layer now have been determined the next step is to discuss the target of injection, that is, the unit in which the soft errors emerge. A list of the possible targets is shown in Table 3.2 containing the hardware units, their purposes, and examples.

<table>
<thead>
<tr>
<th>Hardware unit</th>
<th>Purpose</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-volatile memory</td>
<td>Long-term data storage</td>
<td>Hard drives, Flash memory</td>
</tr>
<tr>
<td>Volatile memory</td>
<td>Mid-term data storage</td>
<td>SRAM, cache</td>
</tr>
<tr>
<td>Registers</td>
<td>Short-term storage</td>
<td>CPU registers</td>
</tr>
<tr>
<td>Flags</td>
<td>Temporary states</td>
<td>CPU binary flags</td>
</tr>
<tr>
<td>Buses</td>
<td>Communication</td>
<td>Memory bus</td>
</tr>
</tbody>
</table>

**Table 3.2:** Abstraction hierarchy and injection methods

The non-volatile memory is in general comparatively slow. Data written to this kind of memory is kept while the system itself is off. At first sight one might expect that the probability of soft errors is high here, as the data is vulnerable over a longer period. But because of the technology used, like magnetic hard drives or NAND flash memory, soft errors seldomly emerge at this level. Also, because of wearing
effects of the hardware, and the relatively slow access, Error Correcting Code (ECC) algorithms can be used.

A more interesting target for fault injection is the volatile memory. The mid-term data storage is more vulnerable to soft errors than the non-volatile memory, again because of the technology used. SRAM cells, for example, may change their states due to minimal changes of electrical charge. These memories types may also be hardened using ECC, although it is usually not done on cheap COTS hardware. The proposed fault injection tool injects faults to this type of memory.

The registers of a CPU are also valid candidates for fault injection. Technologically similar to volatile memory, these are also similarly vulnerable to transient hardware faults. Registers are very architecture-dependent. The sets of registers may vary largely between the architectures, which is not the case for the previously mentioned targets. Also, not all registers can be read and written arbitrarily. This means that a software-based fault injection tool can not modify every single register. The FITIn fault injection program currently does not support injection into CPU registers, but an extension may be possible.

Similar to the registers are the CPU flags. These are binary memory cells that hold state information. For soft error simulations flags may be of interest, because flags are often used to manipulate the control flow. As with registers, however, flags may not be arbitrarily set. Thus CPU flags are beyond scope of the fault injection done in FITIn.

The last target that is considered is the bus. Busses are used to transport the data between the components of a computer system. During the transport soft errors may emerge. Depending on the bus system, these errors may be corrected by ECC systems or with other special-purpose encodings. Busses are also beyond the scope of this work, as there is mostly no way to inject errors using a software-driven fault injection approach.
4 FITIn

In this chapter the main contribution of this work is presented. It introduces a new tool for fault injection experiments, called FITIn. The name FITIn stands for Fault Injection Tool using binary Instrumentation. FITIn was designed, implemented, and tested in the context of this master’s thesis. The tool provides mechanisms to conduct experiments for different SIHFT techniques and makes it possible to compare the results. As FITIn uses the binary instrumentation framework Valgrind, the term binary instrumentation is explained first. Then the specifics of Valgrind are presented, including the underlying language VEX. After that the FITIn tool is presented, including the design, implementation, and annotation interface to C programs.

4.1 Binary Instrumentation

In order to conduct fault injection tests on binary programs a possibility is needed to modify the binary code. The typical work flow for getting a binary file from C code consists of the following process:

- Development of an abstract algorithm
- Writing and developing C program source code
- Compiling and linking of the source code to binary executables
- Execution on a computer system

The target of this work was to get a tool that injects faults into binary programs. The faults are therefore injected in the output of the compiling and linking process. The technique of inserting code into binary programs is called binary instrumentation. With instrumentation, binary code can be analyzed and then even modified according to different needs. The modified version can then be executed. Through the modifications, statistics can be gathered, to monitor run-time behavior at a low layer of abstraction.

As an example let us assume a monitoring program that analyzes the use of the arithmetic unit of the CPU. Further, the modified program counts the arithmetic instructions of the binary program, but must not interfere with the original behavior of the program to be monitored.

Such demands are not trivial to meet. Memory allocations may not interfere with each other, binary code may not overlap in the memory, and the inner state of the processor should behave the same as without monitoring. Binary instrumentation is a complex topic in its own right and cannot be described in full detail in this thesis.
Binary instrumentation can be further divided into static binary instrumentation and dynamic binary instrumentation. The difference between the two lies in the time of modification. Static binary instrumentation modifies the program prior run-time and executes the modified version after modification [20]. In contrast, dynamic instrumentation analyzes the program at run-time and also applies the modifications at run-time. The latter approach has the advantage that the run-time behavior can be analyzed and used for further instrumentation. On the other hand, it is usually much more complicated to dynamically instrument code.

4.2 Valgrind

Valgrind is a dynamic binary instrumentation framework. This work proposes FITIn, a fault injection tool that is built on top of Valgrind. Valgrind can be used to analyze program properties of binary programs at run-time.

4.2.1 Valgrind overview

Valgrind is best known for the memcheck tool, a tool that detects program anomalies concerning memory, which might direct to a program fault. It is most helpful for debugging purposes and optimization. But Valgrind itself is more than memcheck, it is a whole instrumentation framework, containing different tools for analysis and debugging purposes. It also provides functionality that makes it possible to develop custom binary instrumentation tools. The framework hides much complexity of the binary from the user while providing highly flexible programming interfaces.

Since Valgrind is a free open source project under the [GNU General Public License (GPL)] its sources can be downloaded and modified freely. Thus, not only the interfaces to the framework can be reviewed, but also the underlying code itself. Furthermore, Valgrind is a multi-architecture framework. It runs under Linux, OSX, and Android on different hardware architectures including Intel x86, AMD64, and PPC [38].

The official Valgrind tools are used by commercial and open source projects of different size. Examples are the web browser Firefox, the database engine MySQL, and the OpenOffice.org office suite.

For a proper understanding of the FITIn tool a further look into the inner workings of Valgrind is necessary.

4.2.2 Valgrind API basics

This section describes the basic ideas and functions of Valgrind that are utilized by every Valgrind tool. To get an overview how the tool works, Figure [4.1] shows a block diagram of the Valgrind instrumentation process.

The Valgrind instrumentation process begins with the binary code. The code is usually compiled using a high-level compiler. Although this is not absolutely necessary, Valgrind uses patterns that the compilers employ to recognize functionalities. The executables may also contain further information, like debug information, which helps to identify and map the code to the original source code.
The binary is then fed into the Valgrind framework. Valgrind reads the binary code and decodes it. The binary code will then be represented in an intermediate representation (IR) by the Valgrind-specific VEX interpreter.

Using this IR, the tools are able to perform their tasks. The code is modified and run on a simulated machine, the guest state in Valgrind terminology. The guest state consists of the same properties the real CPU has. It holds the whole register file, more precisely, all registers and flags of the simulated CPU. Memory access is specially handled through the simulated CPU, to separate the memory of the client program from the memory of Valgrind and the Valgrind tool. This avoids accidentally overriding one of those memory regions.

After the source was decoded by VEX and processed by the Valgrind tool it is executed. The source is therefore reassembled to binary code with the modifications the tool introduced.

Once a piece of code is processed by VEX and the Valgrind tool, it will not be redirected to the tool again. All modifications have to be made on the first iteration. State variables may be used to trigger instructions at a later iteration.

It should be obvious that the use of Valgrind does not come without costs. The Valgrind team estimates 10 times slower execution time for a minimal tool that actually does no modifications but only uses Valgrind parsing. Additional memory will be used for both Valgrind and the tool.

**VEX**

The binary interpreter VEX is a basic part of Valgrind. It translates binary code to an intermediate representation that the Valgrind tools can process. An overview of the language will be given to lay the foundation for an understanding of the inner workings of FITIn.

Valgrind provides the IR for basic blocks to the tools. Basic blocks consist of multiple operations. Usually they contain 1 to 50 operations, which correspond to the original assembly instructions. Basic blocks are single entry and multiple exit blocks.
The IR has a fine granularity representation of the assembler code. The grammar of the VEX IR can be found in Appendix A. Each part of an assembly instruction is represented by a series of elementary operations. The IR consists of statements and expressions. Statements have side effects, they usually save data, while expressions are free of side effects.

The following example of a VEX representation comes from the Valgrind documentation, which can be found in the Valgrind source code. Listing 4.1 shows the intermediate representation of the x86 add operation "addl %eax, %ebx".

```plaintext
1 ------ IMark(0x24F275, 7, 0) ------
2 t3 = GET:I32(0) # get %eax, a 32-bit integer
3 t2 = GET:I32(12) # get %ebx, a 32-bit integer
4 t1 = Add32(t3, t2) # addl
5 PUT(0) = t1 # put %eax
```

Listing 4.1: Valgrind VEX representation of an addl operation with comments

The first line shows a special statement not directly represented by the instruction to be simulated. The IMark statement signals the beginning of an operation. It holds the address of the binary operation, the length in bytes and an offset, which is always zero on the considered x86 and AMD64 platforms.

In the second line the content of the eax register is read and stored into the temporary variable t3. We see that this is a 32 bit integer that is loaded. The index 0 determines the index in the register file of the simulated CPU.

The IR uses the static single assignment (SSA) form [8]. This assures that temporary variables are written only once. It is therefore easier to do some analyses like def-use chains. It further simplifies the binary instrumentation, as it is known which origin the content of a variable has. No double assignments have to be considered for one basic block.

The following, second line works similarly. It reads the ebx register (with the index 12) and stores it to the temporary variable t2. The third line executes the actual addition. It is conducted using a 32 bit representation and the result will be stored in the temporary variable t1. The same temporary will then be used to write the value back to the eax register. This concludes the add operation. After that another operation starting with an IMark can begin.

The above example was a short sample of a list of IR statements, internally called IRStmts. These statements typically contain expressions, which are called IRExprs. The IRExprs can themselves contain further IRExpr expressions.

To reduce the complexity for parsing long chains of IRExpr expressions Valgrind ensures that the IR is flattened. This means that all expressions contained in another expression are "atoms" that is, they are either temporaries or constant expressions. For the example from above of an unflattened statement is given in Listing 4.2, also described in the VEX documentation.

```plaintext
1 PUT(0) = Add32(GET:I32(0), GET:I32(12))
```

Listing 4.2: Unflattened VEX expression

We can see that the Add32 expression contains GET expressions, which themselves
contain constant expressions. With the condition of flattened statements, instrumenta-
tion will get simpler as no deep expression trees have to be evaluated.

This short introduction cannot present all aspects of VEX and the IR. Yet, it
suffices for the following discussion.

4.3 FITIn overview

The actual novelty of this work is the Valgrind tool FITIn. It is an software-based
fault injection tool, which uses the dynamic binary instrumentation capabilities of
Valgrind to inject errors into binary programs at run-time. The purpose is the
simulation of soft errors, to evaluate SIHFT techniques. Therefore the functionality is
oriented at similar evaluation descriptions found in literature \[1,30\]. A short overview
will be given before the detailed description follows in the next sections. The program
that is the subject of FITIn will hereafter be called target program or simply target.

To conduct fault injection experiments it is necessary to prepare the target program.
This preparation is done using source code annotation. The annotation has descriptive
character and defines which variables are under surveillance. Annotations do not
have any functionality in the running program. It is also possible to extend the
functionality of FITIn to have more feature rich annotations. Up to now, only the
addition and removal of monitored variables is supported.

The basic functionality of FITIn is separated in the two steps that are commonly
carried out with fault injection experiments. The first stage is called the golden run.
It does not modify or change any behavior of the target, instead it collects data to
be used after the stage finishes. This is done using monitoring.

The tool counts the load operations of the memory locations under surveillance.
The information which memory location is to be observed comes from the annotated
source code.

After the golden run has finished FITIn outputs a result file of the data it collected.
This data can then be used to conduct further experiments. With the limitation of
deterministic target programs, the golden run will always behave the same, therefore
always return the same output (given the same input). Further, it is the same
behavior the program would have if run without FITIn.

The other step is the injection run step. With the knowledge of the program
behavior in the fault-less case, faults may be injected. Faults currently processed are
single event upsets resulting in single bit flips. The faults are inserted before load
operations. For that reason the load operations were counted in the golden run.

Both the load operation that will be used for injection and the bit that will be
modified can be specified as Valgrind parameter. With this input the injection run
will execute similarly to the golden run.

It counts the load operations of the data that is monitored. If the injection point is
reached, the monitored data will be modified, and the run continues. Because of the
modification the behavior of the target may differ from the behavior of the golden
run.
The actual outcome highly depends on the target program. As explained in Section 2.3, the outcome might reach from 'undetectable' to program crash.

4.4 Target preparation and source code annotation

To use the FITIn tool, the target program has to be prepared. This has to be done at source code level in the C programming language [18]. The tested programs were compiled with the GNU Compiler Collection’s C and C++ (gcc/g++) compilers [11]. As FITIn is only able to monitor and inject to memory locations, not registers, it is necessary to ensure that variables are always stored in memory. This is not the case with the default configuration of the compilers, as these already involve some optimizations. To ensure the correct working, use the -O0 option of gcc.

Additionally to the compiler options, the source code of the target needs annotations. This section explains the step of annotating the target source code using a sample program. The sample program is presented in Listing 4.3. The program is written in C++ [37] and it is simple enough to demonstrate the annotation process. It outputs the string 'Value of variable i:' sixteen times appended with the value of the variable i.

```cpp
#include <iostream>
using namespace std;

int main(int argc, char *argv[]) {
    unsigned int i = 0;
    for (i = 0; i < 16; i++)
        cout << "Value of variable i: " << i << endl;
    return 0;
}
```

Listing 4.3: Simple C++ sample program

The aim now is to register the variable i for monitoring with FITIn. This is done using the annotations features of the tool.

The annotation is realized using the client request mechanism of Valgrind. With client requests it is possible for target programs to provide data to a Valgrind tools using special preprocessor macros in C/C++ code. The macros are defined in the valgrind.h file of the Valgrind code. These are

- VALGRIND_DO_CLIENT_REQUEST()
- VALGRIND_DO_CLIENT_REQUEST_STMT()
- VALGRIND_DO_CLIENT_REQUEST_EXPR()

The macros are defined to produce special code, which can be recognized by the
Valgrind interpreter. The code inserted by that macro does not change the behavior of the underlying algorithm of the target program. It is a pattern that is recognizable by Valgrind. If the code is not run with Valgrind the code does not affect the program logic. The macro code increases the code by 6 integer instructions, and thus will have a low performance impact in most situations [39]. Every client request has a unique identifier, derived from the tool’s unique identifier. The tool’s identifier is derived from a pair of characters that come from the tool’s name.

The annotations used by FITIn are presented in Section B.1. These all are preprocessor macros consisting of the previously mentioned client request. In the following the FITIN_MONITOR_VARIABLE and FITIN_UNMONITOR_VARIABLE will be used to instruct the FITIn tool to register and unregister a variable for monitoring in the tool. To add these to the sample code, the header file fitin.h needs to be included in the project. Suppose this file can be found in the directory hierarchy at "../valgrind/include/valgrind/FITIn.h".

Listing 4.4 shows the code with the added annotation commands. To add the variable i to the list of variables to be monitored we need to add the macro FITIN_MONITOR_VARIABLE. At best this is done right after the declaration of the variable since this ensures that all read operations will be recognized.

```cpp
#include <iostream>
#include "../valgrind/include/valgrind/FITIn.h"
using namespace std;

int main(int argc, char *argv[]) {
    unsigned int i = 0;
    FITIn_MONITOR_VARIABLE(i);
    for (i = 0; i < 16; i++)
        cout << "Value of variable i: " << i << endl;
    FITIn_UNMONITOR_VARIABLE(i);
    return 0;
}
```

Listing 4.4: The sample program with code annotations

The inclusion can be seen in line 2, and the monitoring annotation in line 9. At a closer view another addition can be seen in the code. The FITIN_UNMONITOR_VARIABLE command in line 13 is used to remove the variable from the list of monitored variables. Removal is necessary, because Valgrind and FITIn cannot in all cases ensure that the location of the variable will not be used in the future by other functions. It is in

1 The official documentation uses the wording 'handles magically' for the process.
2 This approach presents no strong uniqueness, as it does not prevent one from using an identifier more than once. However, for the small number of Valgrind tools, given the care of the programmers, this solution is sufficient.
general not possible with FITIn and Valgrind to precisely detect the end of functions. Thus, it may be unclear to which function the monitored variable memory belongs. With the removal at the end of a function it can be ensured that the memory will not be monitored after the return of the function.

Now that the code is annotated it can be compiled. As FITIn can monitor memory only and not registers, it is important to ensure that the program will be compiled without any optimizations. This limits the performance of the program, but ensures a working FITIn. The compiler used for in the development is the C++ compiler of the GNU Compiler Collection in the version 4.7.2[11].

4.5 Monitoring

This section explains in more detail the step of monitoring. It is the crucial part of FITIn as both stages, the golden run and the injection stage, use it. The aim is to find and index memory load operations. The index is used in the injection phase as an identifier of the target memory for the injection.

The procedure for the monitoring of one memory variable is as follows.

1. Register the memory location for monitoring
2. Count every memory read operation of this memory location
3. Unregister the memory location

This process is done in FITIn with the help of the Valgrind application programming interface (API). Let us assume the program from above is used as an input for the golden run stage of Valgrind using FITIn. The registration of the variables are initiated by the annotations explained before. These code annotations will be recognized by the Valgrind interpreter and a FITIn call back function will be executed. It recognizes the specific type of annotations, for example, if it is a registering or an unregistering annotation, and handles it accordingly.

In the sample code above, the `FITIN_MONITOR_VARIABLE` will be recognized. The memory address of the associated variable will be added to the list of monitored memory locations. Having a list of these memory locations, FITIn is able to monitor every load operations of the list entries.

To actually monitor the load operations the code needs to be instrumented. As described in Section 4.2.2 this is done using the VEX intermediate representation. At first the load operations have to be found in the target code.

FITIn uses the VEX representation supplied by Valgrind. VEX IR is returned as a list of `IRStmt` statements. As presented before `IRStmt` statements consist of `IRExpr` expressions. The tool analyzes every statement and searches for `IRExpr` expressions with the type of `Load` expressions.

Then the `Load` expressions are processed. As stated in the grammar, they consist of an `IRType` type that holds information about the size of the memory to be loaded.

1 In the sources of Valgrind a discussion about this topic can be found in the folder `docs/internals`.
the endianess, and the address. The address is part also an \texttt{IRExpr} expression. This contained expression will be a \texttt{Temp} temporary or a \texttt{Const} constant expression due to the flattened condition that Valgrind ensures.

Before the Load expression is added to the list of Statements to be executed, another \texttt{IRStmt} is placed before the Load. This statement is of the type \texttt{IRDirty}. With \texttt{IRDirty} statements the tool can inject callback functions to own code into the target program. Inside this callback functions, memory and CPU states of the target program can be read and modified. The FITIn function that is registered before every Load expression is called \texttt{preLoadHelper}.

The \texttt{preLoadHelper} helper function will then be called before a memory location will be loaded, independently from the fact, if the memory address is in the list of monitored variables. The list entries of monitored variables cannot be compared to the load address outside the helper function, because the list belongs to the FITIn tool.

The \texttt{preLoadHelper} takes the load address as an argument as well as the size of the memory location to be loaded by the following Load expression. The helper function serves several tasks. The first task is the count of all load operations. Using the counter as an index we can identify every load operation by this index. This is not an important measure per se, but it helps to relate the monitored load operations to the overall loaded memory. The other task is the counting of the monitored variables. Therefore the monitored variable list is traversed to find the entry of the specified memory address. In the case of a search hit, the counter of the corresponding entry is incremented. If it is not found nothing is counted.

The helper function is always called before a load operation is hit. It may be obvious that this has a great impact on the speed of the program. This is a limit of this type of injection.

The incrementation of the counters in the list of monitored variables is done as long as the memory location is marked as valid in the list of monitored addresses. If this is not the case anymore, for example because of a \texttt{FITIN_UNMONITOR_VARIABLE} command, the load counter will not increase anymore.

If the memory should be counted again, another \texttt{FITIN_MONITOR_VARIABLE} annotation is necessary to add the same variable again to the list. When the end of the program is reached, the tool will output statistics about the run: the overall number of load operations, the cumulated number of monitored load operations as well as the number of monitored bytes loaded and the executed instructions. In a fault injection experiment, the next step would be to use the tool to inject errors to the program. Information gained from the golden run, especially the number of monitored load operations, can be used to get an upper bound for the injection point.

4.6 Injection mode

The injection mode supports the injection of single bit errors to the target program. Therefore, the load statement has to be given at which the injection should happen as well as the position of the bit. This is done by passing the arguments
"--mod-load-time" and "--mod-bit" to FITIn.

The injection mode works similarly to the monitoring mode. The basic behavior is
monitoring the load operations of the annotated memory locations. Internally similar
actions happen as in the monitoring case: the annotated code produces entries in the
monitored location list, which serve as targets of a load counter. But additionally to
the incrementation of load counters, the overall load counter of monitored locations
will be compared with the value given with the "--mod-load-time" parameter. When
the load operation is reached that was input to FITIn, the memory location will be
modified prior to the load operation.

Because the "--mod-bit" might be out of bounds of the memory location, the
bit flip will be brought to the correct bounds. The main idea is to use the modulo
operation. Assuming the memory location has the length of \( l_b \) in bits and \( i_b \) denotes
the bit that should be modified then \( e_b \) denotes the real bit that will be modified,
according to the following formula, \( e_b = i_b \mod l_b \), in which \( \mod \) denotes the modulo
function \([43]\).

If the memory location is a multi-byte memory location, the bit has to be placed
in the correct byte, at the correct position. This can be achieved using the simple
equation \( e_B/8 = e_B + \frac{e_r}{8} \). In this equation \( e_B \) denotes the offset of bytes of the given
memory address and \( e_r \) gives the position in the byte.

Let us discuss a simple example. Assume the memory location that is the target
of injection has the hexadecimal address \( a = 0x000800 \) and the size in bytes of
2, which leads to a bit count of \( l_b = 16 \) bits. Further assume the injection bit
is given at position \( i_b = 30 \). Because \( i_b > e_b \) the injected bit \( (e_b) \) will be set to
\( e_b = i_b \mod l_b = 30 \mod 16 = 14 \). So the 14th bit will be modified. But as this bit
is not located in the first byte (it is bigger than 8), the byte offset and the final bit
offset in this byte need to be calculated. Using the formulas from above we get

\[
e_b/8 = e_B + \frac{e_r}{8} \tag{4.1}
\]

\[
14/8 = 1 + \frac{6}{8} \tag{4.2}
\]

So the offset \( e_B = 1 \) and real error offset \( e_r = 6 \) can be used to inject the bit flip.

The offset is added to the memory address \( a \) and the fault is then injected at this
position. This can be achieved using the binary exclusive OR (XOR) function \([43]\).

\[
\text{XORing has to be applied to the value at the injection address with a bit pattern, where all bits are zero except the bit to be flipped. The resulting pseudo code, similar to C code, dealing with the default types } \text{char} \text{ and } \text{char star}, \text{ is like,}
\]

\[
*(a + e_B) = *(a + e_B) \text{ \^ (1 << } e_r)\text{;}
\]

For the example, we obtain

\[
*(0x000800 + 1) = *(0x000800 + 1) \text{ \^ (1 << 6);}\]

which can be further simplified to

\[
*(0x000801) \text{ \^=} (1 \text{ \^} 6);\]
which is a fairly easy expression. After the injection is conducted, the number of total injections is incremented. Right now only one injection per run is allowed and tested. This constraint might easily be extended, but was out of consideration in this early design for simplicity.

Continuing execution with the injected memory location, the program might behave differently than in the golden run, without any injection. It is likely that a modified value might lead to a modified control flow, or to corrupted pointers, which now point to other memory locations, than before. This might alter the program behavior significantly. As stated in Section 2.1 the outcome may be undetectable, benign, or even lead to a program crash.

One important point is the termination of the target program. Injection to the binary data may lead to infinite loops, which have to be terminated somehow in an experiment. Infinite loops may be prevented using another parameter supplied to FITIn. The \texttt{--inst\_limit} parameter allows setting a fixed limit of instructions to be executed. Once this limit is reached the target program will be terminated by FITIn.

To have some information about the run-time, in particular the number of instructions executed without the injection, information from the golden run can be used. The golden run not only monitors the annotated memory locations but also counts the number of executed instructions. This number can be used as a rough measure to determine a reasonable value for the \texttt{--inst\_limit} parameter of the target program during an injection experiment. As an example, the number of instructions of the golden run and additional 10\% may be used. The 10\% are by no means scientifically grounded, they are rather an empirical measure.

4.7 Limits of FITIn and possible improvements

FITIn is, at this point of development, merely a proof of concept for fault injection with the dynamic binary instrumentation framework Valgrind. It was neither the intention to write a universal tool for all fault injection purposes nor is it likely that such universal tool exists at all, due to the various aspects of fault injection experiments (see Chapter 3).

FITIn is at this stage rather a basis for further development. In the following we will discuss a list of improvements.

1. Fault injection into CPU registers
2. Different injection methods (not only at memory load operations)
3. Different user interface/target assessment
4. Help using static analyses
5. Additional architectures/OSes
6. Performance
7. Limits of software

The first item in the list deals with register manipulations of the central processor unit. Valgrind allows access to the processor registers, similar to the access of memory. However, the unsolved question is how to decide when to inject faults and which register to choose. The source code annotation is not usable for this as it is not generally possible to know the binary output at this program abstraction level. Also the problem of choosing the instrumentation method is unsolved. The performance penalty of instrumenting every register access seems too high for a usable fault injection tool.

A similar problem is addressed in the second list item. The injection target may be all kinds of operations. One might consider to manipulate jump instructions to simulate control flow faults. Such experiments deal with other kinds of faults, and are therefore not considered in this work.

Using a different user interface or a different way to choose targets may be another topic for improvement of FITIn. Currently FITIn uses source code annotations and parameters as input to choose the injection points. However, dynamic instrumentation provides the possibility to use an interactive user interface as well. Run-time information may be used to decide when and where to inject faults dynamically.

Although the dynamic instrumentation may improve analyses using run-time data, the FITIn tool itself may be improved using data from static analyses. For example, static analyses may generate data about data flow like def-use chains, which may enable FITIn to have more sophisticated injection algorithms. Also the post injection processing may be supported using static analyses. The combination of static analysis and dynamic fault injection is a topic for further research.

Because FITIn is a tool for research additional features like the support of multiple architectures and performance optimizations were not considered. FITIn is not tested to run using a different OS than LINUX or using a different processor, although it might work. In the current design FITIn is not optimized for performance either. Performance may become an issue if bigger target programs are used or different injections schemes are implemented.

Some of the limitations are due to the fact that the tool is a software tool itself. This means that all limitations of software apply here also. So is it impossible to get access to hardware components that have no interfaces to the software domain. Shadow registers or special buffers, which hold information not accessible to software, are examples of unreachable device features. This limitation is not as critical as it might sound at first. The idea is that information that cannot manifest itself in software, is not considered to be of interest.
5 Experiments

The goal of FITIn is to conduct experiments that involve single event upsets in form of bit flips. This chapter describes a possible experiment configuration and reports the results from a fault injection experiment where FITIn was used to evaluate the Flipsafe SIHFT library. The evaluation was conducted using an integer based benchmark, Dhrystone.

5.1 Basic considerations

In [35] the proceedings for such experiments are laid out. They identify 4 key properties of an experiment:

- Repeatability
- Controllability
- Intrusiveness
- Observability
- Reachability

In the following these will be described in more detail. One of the most important property of scientific work is repeatability. It says that an experiment should be described in a way that with the same configuration the same results will be achieved. Repeatability is not always possible, as some experiments involve randomness. However, in computer science, also random experiments can be made repeatable, for example, experiments based on pseudo random number generators [19]. Such experiments are also known as Monte Carlo experiments [21].

Randomness is also a topic for the controllability of an experiment. The ability to control the configuration, and by this all varying aspects, is crucial for an experiment. Let us assume we had no control how, when, and where a bit would flip in a fault injection experiment. In this case fewer conclusion can be drawn from the results, as the input parameter is not known, and no relation between the fault and the experiment’s outcome can be formulated.

Additionally an experiment configuration for fault injection should be intrusive. Intrusiveness denotes the ability of changing data that is the target of the experiment. The current implementation of FITIn currently only supports the injection into main memory. While it lacks more options, this is sufficient for bit flips in main memory.

Another important factor is observability. It should be obvious that for every experiment the results should be somehow observable. In the case of fault injection,
the faults could manifest themselves somehow in the output or behavior of the program. However, there may be benign faults, which do not lead to different behaviors. Such cases should of course be observable too. The observability is the reason to introduce the golden run: The golden run presents the error free behavior, which acts as reference to compare the injected variants to.

The last topic of the experiment property list is denoted reachability. A fault injection experiment should ensure that the part of the code that is the target of injection is really reachable by the control flow. Thus, the code which leads to the injection should be called at some time. This fact is ensured by several conditions. First it is ensured using dynamic instead of static fault injection. The injection code is only instrumented and executed when the code is reached. By this FITIn ensures that, if an injection was done, the code was reachable. Secondly reachability is ensured by the injections before the load operations. This guarantees that the modified value is read after the injection and thus it is reached.

5.2 Dhrystone

One aspect of repeatable experiments is the specification of the object or target of the experiment. In this work a customized version of the commonly used benchmark Dhrystone is used [31] [6]. The customizations mostly concern code refactoring, which enables the usage with SIHFT technologies and FITIn annotations, than behavioral changes of the benchmark functions themselves. This benchmark will not be presented in detail, however a short overview will be given and the customizations will be discussed.

5.2.1 Overview and history

Dhrystone is a synthetic performance benchmark developed by Reinhold Weicker in 1984 [40]. Its aim is to measure the integer computation performance of a computer system. The name Dhrystone is an analogy to the floating point based benchmark Whetstone. Originally constructed in the programming language Ada, the benchmark is also available in the C language. There are 2 versions of this benchmark, with the second being a revised version of the first [41]. For the following benchmarks the version 2.1 will be used.

The Dhrystone benchmark consists of different integer and string operations, which are executed in a loop. It neither performs floating point operations, nor system calls or IO functions [31]. To measure the integer performance on a system, the execution time is measured. The results are commonly presented as Dhrystones per second or million instructions per second (MIPS).

Dhrystone consists of 12 functions called in a loop. The functions manipulate integer variables, standard C strings, and integer arrays. No floating point calculations are made in the benchmark.

The loop of Dhrystone is controlled dynamically. The number of loop iterations is chosen at run-time. This scheme prevents the benchmark from finishing too early.
The benchmark, however, should run for a while to get a more exact performance result.

### 5.2.2 Customizations

In the following the changes made to Dhrystone will be discussed. The reasons of choosing Dhrystone was its exploitation of integer operations, its simplicity, and the wide usage of this benchmark. It is, however, designed as performance benchmark. Some of its properties therefore did not serve well the requirements needed for an evaluation program for FITIn. The dynamic number of loops of the benchmark functions is removed and the possibility is added to use Flipsafe SHIFT types.

For conducting experiments with FITIn, a deterministic program is needed that has no run-time dependencies. Even more strict, the program should act as a pure function: It should only depend on its inputs and not on the machine state. Since the original Dhrystone does not have the function property, customizations were needed.

One important customization is the removal of the time measurement. Time measurement introduces non-deterministic machine state dependencies, and different behavior at each run. Also the time measurement was not scope of this test, as its originating purpose, measuring performance, is not the aim of the program running with FITIn. Thus, the time measurement dependencies could be safely removed.

Another customization concerns the possibility of a program parameter supplied at start-up. The parameter is supposed to be a positive integer value. It is used to specify the number of runs of the Dhrystone loop. As said before, the benchmark consists of a loop executing several integer and string functions. In the original version the number of iterations of this loop is changed dynamically, to have a minimum of runs, but enough runs to get a proper time measurement. Because the time measurement is not in the scope of the fault injection experiment, but controllability was, the number of runs was made constant. This number can now be supplied from outside as start parameter of the program. If the loop number is not supplied by a parameter, the loop number is set to 50 by default.

Only one behavioral change was made to one of the Dhrystone functions. The function Proc_4 was modified to support error propagation. The customized function is presented in Listing 5.1.
Listing 5.1: Modified version of Dhrystone’s Proc_4 function

Ignoring the annotations used for FITIn, the only change was made in line 8, and commented in line 7. The original version used a binary OR function to provide the new value for Bool_Glob. The problem is the error propagation in the fault case in an fault injection experiment and the use of the OR function.

By analyzing the program manually, it was found, that both variables, Bool_Glob and Bool_Loc, should always evaluate to true. In memory both variables contain a non-zero value, more precisely a binary 1 at the least significant bit. Using the binary OR with an operand being non zero, the result, stored in Bool_Glob, would become non zero either.

Let us now assume, one variable is zero due to a fault. Still the OR function would create a non zero value and the error would not propagate. With the addition of the logical AND this would not be the case, as both need to be non zero. Thus there exists the possibility to propagate the error further. This change was done to support better observability, as the output of an injection run may now contain values that are results of error propagation.

Additionally to the change discussed before, one can see annotations in Listing 5.1 specifically in lines 4 and 11. To enable useful testing with FITIn these annotations are necessary. Looking slightly different from those presented earlier in Section 4.4 and referenced in Appendix B, these are just redefinitions of the macros of the described ones. The purpose is that it is now simpler to include separately the registration of monitored variables for boolean an integer variables. However, the integer versions of the macros are not used in the Dhrystone, as the experiment is limited to injections into boolean variables only.

5.2.3 Integration of Flipsafe

The original purpose of FITIn is the evaluation of SIHFT techniques. Accordingly the Flipsafe library was embedded into the Dhrystone benchmark to evaluate FITIn and to show the design serves its purpose. As mentioned in Section 2.3 Flipsafe is an SIHFT technique, which uses types provided in source code to establish protection against bit faults.
While the original Dhrystone is pure C, the Flipsafe library is written in C++. To be able to use C++ and varying type definitions also the type definitions were customized with the help of `typedef` type redefinitions and macros. By default this customization has no effect the program itself, but provides the possibility to exchange the type easily at compile time. In the simulations one can obtain different versions of the program, one with ‘ordinary’ types and several others using SIHFT safety types.

Also the dependencies are added to build Dhrystone with Flipsafe support. To choose a Flipsafe boolean variable type the customized Dhrystone program can be compiled with specific compiler flags. This is done in the `Makefile` used to build the program. Flipsafe types can be used when compiled with the define statement `FLIPSAFE` added as compiler flag. To specify the type used to protect variables additional parameters may be supplied. These parameters are listed in Table 5.1.

<table>
<thead>
<tr>
<th>Define name</th>
<th>SIHFT type</th>
</tr>
</thead>
<tbody>
<tr>
<td>BCBOOL</td>
<td>Bit-counting boolean</td>
</tr>
<tr>
<td>CSBOOL</td>
<td>Improved C-style boolean</td>
</tr>
<tr>
<td>EPBOOL</td>
<td>Even partition boolean</td>
</tr>
<tr>
<td>RMBOOL</td>
<td>Rotation mask boolean</td>
</tr>
<tr>
<td>SHBOOL</td>
<td>Shift mask boolean</td>
</tr>
<tr>
<td>DUP</td>
<td>Duplicated integer</td>
</tr>
<tr>
<td>TRI</td>
<td>Triple integer</td>
</tr>
<tr>
<td>TRUMP</td>
<td>Integers using TRUMP</td>
</tr>
</tbody>
</table>

Table 5.1: Description of defines to enable the Flipsafe type with gcc

With this design it is possible to choose between different combinations of SIHFT boolean and integer types.

5.3 Experiment setup

The following section describes a possible setup for a SIHFT experiment using FITIn and Flipsafe. With the preliminaries described before, a trial may be set up to evaluate the performance of the Flipsafe library as well as to check the possibilities of FITIn. The experiment will conduct a golden run, without any injections, and its output will be compared against injection experiment outputs.

In this work two possible setups will be discussed: the random bit flip and the full sweep. However, these are not the only possible setups, as experiments may need special configurations depending on their purpose. To plan and execute FITIn a bash script is provided [12]. An overview of the basic experiment work flow is given in Figure 5.1.

In both experiment setup proposals this scheme is used. The first step is to execute the golden run. It is the mode of FITIn that just monitors the execution to gain
information about the number of load operations. This information may be used to set an end condition for the injection loop. The injection point is chosen and the injection runs will be executed in a loop.

5.3.1 Random bit flip

The aim of the random bit flip is to simulate the random events of cosmic rays hitting the memory cells. Effects on other parts of the system, like CPU registers or busses, are not considered.

The justifications for a random experiment are the simulation of real fault occurrence and speed issues. Transient hardware faults, as explained in Chapter 2, often occur in a random manner. Simulating random behavior thus seems reasonable.

Another argument for Monte Carlo experiments is the reduction of time effort, compared to a complete simulation. Instead of simulating a full fault coverage, as done in the later full sweep experiment, the number of errors injected is lower. Thus, the experiment will finish faster, but with lower precision, because not all fault cases are examined.

Two versions of the Dhrystone benchmark will be compiled: One that is the customized Dhrystone benchmark with the default types, and one Dhrystone version that uses Flipsafe for fault tolerance. In this work only the boolean variants of the fault secured types of Flipsafe will be used. Then a golden run will be executed to verify the behavior and acquire statistics about the 'normal' run without fault injection.

In the following runs faults will be injected randomly into both versions. This is done using the features of FITIn.

With the information of the golden run, the number of monitored load operations
can be determined. This is an upper bound for the injection, that is, the monitored load in which the fault is injected. Also the length of the monitored variables is known, which states an upper bound for the position of the bit to be flipped.

Both simulation variants will take the following inputs for FITIn:

1. A file path where the result files should be stored with iteration number as an index.
2. The load number, or index, when the injection should occur.
3. The position of the bit to be flipped.
4. The file path where the target program was compiled before.

The last helps FITIn with identifying the actual program code of the target. FITIn should not modify standard library or other external code.

One aim is to relate the injections of the 'ordinary' typed Dhrystone with those with the Flipsafe typed Dhrystone. Assuming the Flipsafe version loads a boolean variable as often as the default typed version, it is assumed that a 1-to-1 relation exists. If the load number is the same for the default typed Dhrystone and the Flipsafe version, the equivalent memory is modified. As example assume in both Dhrystone versions, the default typed and the Flipsafe typed, the \( n \)th load is modified. Then both modifications are executed on the memories representing the same variable in the C source code.

However, the number of load operations may be a multiple of load operations in the Flipsafe case compared to the default typed Dhrystone. To have a relation between the injection and the corresponding variable of the two versions the load injection position of the Flipsafe version needs adjustment. Let us assume now that the Flipsafe version has twice the monitored load operations as the normal version. Then the corresponding load index to the load number \( n \) of the default typed Dhryston corresponds with the load number \( 2n \) in the Flipsafe version. The rule is, multiplying the load operations used for Flipsafe means also multiplying the injection load number.

### 5.3.2 Full sweep

A different scenario from before, where faults are injected randomly, is the full sweep. The idea behind the full sweep is that the number of load operations and the sizes of all variables are known and by this all bit may be tested. This is, however, only possible for a small number of load operations.

The overall setup is similar to the previous one for random bit flips. Only the load index and the bit flip positions need to be modified. To iterate over all bits, the number of load operations and the length is needed. Assume the number of loads is \( l \) and the size of the boolean variables in bytes is \( B \). With the common definition that one byte consists of eight bits we get the bit count per boolean with \( b = 8 \cdot B \).

C-like pseudo code of the full sweep is shown in Listing 5.2. It consists of two nested loops, which iterate over all load operations and all bit positions.
5.3.3 Result evaluation

The result of a run is given by the output files produced by the target program and FITIn. In the case of Dhrystone the summary output of Dhrystone itself can be used as result. The Dhrystone summary consists of checks of the valid variable contents. Correct values will create an OK string in the output file, while wrong variable contents create an error string.

With this it is possible to run a difference comparison of one fault injection against the golden run. Deviating fault injection outputs indicate observable faults.

Further the FITIn output files may be used to evaluate the run-time performance. One can compare the golden runs of each program version, the original and the hardened variations. Especially the executed instructions may be of interest, as these are indicators of performance penalties using the secured Flipsafe versions.

5.4 Results

The following experiments were conducted on an Debian Linux system, running on an AMD Phenom II X6 1090T processor. First the results of the random bit flip experiment will be presented and thereafter the full sweep.

5.4.1 Random bit flip

The random bit flip simulates the random occurrence of soft errors. It is not considered a full analysis of a system under the influence of faults, yet it gives a rough impression of the system behavior. Without simulating the whole space of possible bit errors it is considerably faster than the full sweep experiment, but less accurate.

The test was run with 100 injection runs and 50 Dhrystone loops. The test duration was about 6 minutes and 18 seconds. Figure 5.2 shows the results of the bit flip experiment.

It is visible that only a small number of errors can be detected. Even in the original, unsecured version of the Dhrystone benchmark, only 2 faults led to errors. All other injections led to no detectable error and can be considered as benign.
5.4 Results

Except for the version using \texttt{epbool} type, the SIHFT extended targets do not show detectable errors. Thus it can be assumed that all these techniques successfully harden boolean variables against single bit errors. Because the limitation of the fault space investigated by a random experiment a full sweep will presented in the next subsection.

5.4.2 Full sweep

As stated before, the full sweep changed every monitored bit once. Thus this experiment had a longer run time than the random bit flip of about three hours.

In Figure 5.2 the results of the full sweep experiment are show. The original version of the customized Dhrystone program shows a total number of errors of 182. Again errors have been determined by comparing the output files to the output files of the golden run.

All SHIFT techniques, except for the even partition boolean, have a error count of zero. This shows these are working according to the requirements, namely that they mask single bit flips in memory.

However the even partition boolean with 300 errors seems to perform even worse than the original. This is possibly due to an error in the implementation or misusing the \texttt{epbool} type. A quick code review could not show an implementation error. A fault in the FITIn injection scheme is also a possible source of the error.

5.4.3 Other performance measures

As stated in the previous Section 5.3.3 additional measurements may be made. Especially the penalty of using the different SIHFT mechanisms is of interest.
Figure 5.3: Results of the full sweep experiment

Figure 5.4 compares the file size of the different executables. While the original version is about 79 kByte in size, the SIHFT versions are considerably larger. They all have a size of about 987 kByte, almost one MByte, only differing by a few bytes.

Figure 5.4: Comparison of executable file sizes used in the FITIn experiments in bytes

This comparatively high increase is not caused by the additional SIHFT code alone. In addition to the Flipsafe dependency, C++ features are used in the SIHFT versions,
whereas the original Dhrystone is pure C code. Further, the whole Flipsafe library will be compiled into the binaries. This code was compiled without any optimization or inlining code.

To show that Flipsafe itself does not lead the executable grow another trial was made. Code inlining was activated again and no debug information is supplied with the executables. These modifications prevent FITIN to work correctly with these executables, yet they run correctly without fault injection. The file size comparison is shown in Figure 5.5.

The different settings denote the corresponding compiler flags of the gcc compiler. A higher number means more optimization is used. It may be seen that the sizes all are considerably lower than in the previous example shown in Figure 5.4. This is the result of the removal of the debug information. Further, the unoptimized code, compiled with the flag -O0, is bigger than the further optimized versions. In this figure one advantage of Flipsafe is visualized, that the additional code of Flipsafe is not very high compared to the original code. However, it should be noted that only a few variables were hardened. The Dhrystone benchmark is comparably small, and the overhead of system specific initializations and destruction is high for such small programs.

While the binary size grows using the SIHFT technologies, also the actual performance penalty should be considered. In Figure 5.6 the actually executed instructions acquired by FITIn are presented.

It is clearly visible that all SIHFT techniques increase the number of instructions to be executed. This can be expected due to the additional work for detection and recovery of faults. But it is also visible that the SIHFT versions are still in the same order of magnitude considering the number of instructions. Not all variables were
hardened, so the effect on the instruction number is considerably lower than in a fully hardened case, where all variables have a SIHFT representation. The fully secured case is not presented in this work.

A comparison between the SIHFT techniques is shown in Figure 5.7. It shows that the bit counting boolean uses the highest number of instructions while the improved C style boolean uses the least number of additional instructions. Figure 5.7 presents the qualitative overview of the varying instruction numbers.

Figure 5.6: Executed instructions during golden runs

Figure 5.7: Executed instructions during golden runs of SIHFT versions only
5.4 Results

All plots above display results using the customized Dhrystone benchmark. Further examinations are necessary to get a better analysis of the properties and implications of the Flipsafe library. Additional experiment setups using varying programs and experiment configurations are needed, but are beyond the scope of this work. Further research may be done using FITIn as well as other SIHFT techniques. This chapter presented experiment setups for simulations of the SIHFT-secured Flipsafe programs driven by FITIn. The designs of the experiments were explained and the results presented. Further the results were interpreted. The experiments show that it is possible to use FITIn to conduct simulations using fault injection, especially in the domain of SIHFT techniques.
6 Conclusion

This chapter concludes the work. It summarizes the findings and gives an outlook on future development.

6.1 Summary

Chapter 2 provides insight into faults, errors and SIHFT techniques. To simulate SIHFT techniques the faults needed to be specified. For this work the targeted types are soft errors, transient hardware faults.

Different SIHFT approaches were discussed in terms of abstraction and target. It was concluded that a flexible way to harden variables are type based SIHFT techniques as these are platform-independent and mostly compiler-independent. However, SIHFT mechanisms may still fail, when software access is not possible or the failures are beyond the targeted requirement.

Fault injection as a possible test method was presented in Chapter ???. The common approach to do SIHFT evaluation is fault injection. Fault injection simulates a system in the presence of faults. The faults targeted by the SIHFT mechanisms presented are soft errors, so errors which have their origin in hardware faults. Hardware faults led to a low level of abstraction for the fault injection method. Because of the low level binary instrumentation was chosen as a possible technique to conduct fault injection experiments for SIHFT based methods.

In Chapter 4 Valgrind and FITIN, the main contribution of this work were presented. Valgrind, an open source dynamic binary instrumentation framework, was found as a possible software tool enabling fault injection. Using Valgrind it was possible to design and implement FITIn, a new fault injection tool. The aim of FITIn is the simulation of soft errors that emerge in memory.

The basic functionality of FITIn is to monitor memory load operations. FITIn requires the user to annotate the target program in source code first to provide useful functionality. Load operations are counted and the count can then be read from an output file.

Further FITIn allows the flipping of bits as a simulation of single event upsets. This is done similarly to the monitoring. Both functionalities together make it possible to conduct SIHFT experiments.

In the following Chapter 5 two possible experiment setups were discussed. The 'random bit flip' randomly changes one bit per experiment run. On the other hand, the 'full sweep' changes every bit once. The experiments were conducted and its result presented.
6.2 Outlook

The vision of developing FITIn is to get a multifunctional fault injection tool. With the version finished with this work it is not yet the case. Several improvements can be made to make FITIn really powerful.

The first improvements is to get rid of annotations. It should not be necessary to edit the original target source code to conduct experiments. Instead, the debug information of programs should be used and some preinjection (static) analysis may help.

Further the model of faults may be specified more flexible. Right now only single bit flips, once per run are supported. Various bit errors may be implemented such as multiple bit flips, specific error patterns, or multiple random bit errors.

In addition to different fault models the inner workings of the processor may be exploited. The register file may be modified to simulated errors in the CPU. Although Valgrind offers access to a CPU environment, the question is how to choose the injection target, i.e. the specific register, and the controllability of the runs, when to inject faults, has yet to be solved.

Another possible improvement is the exploitation of the dynamic behavior of Valgrind. The dynamic instrumentation and injection is an advantage over static fault injection as the injection can be made during runtime, including runtime information in the injection decision. A runtime specific interactive mode may be implemented, where the user can dynamically specify the time of injection during runtime.
Bibliography


[17] IEEE Standard for System and Software Verification and Validation. Institute of Electrical and Electronics Engineers (IEEE), 2012. DOI: 10.1109/IEEESTD.2012.6204026


List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Fault tolerance overview</td>
<td>4</td>
</tr>
<tr>
<td>2.2</td>
<td>Taxonomy of faults according [2]</td>
<td>5</td>
</tr>
<tr>
<td>2.3</td>
<td>Control flow graph example</td>
<td>10</td>
</tr>
<tr>
<td>2.4</td>
<td>Von Neumann architecture [7]</td>
<td>11</td>
</tr>
<tr>
<td>2.5</td>
<td>Instruction processing of a Von Neumann architecture [14]</td>
<td>11</td>
</tr>
<tr>
<td>2.6</td>
<td>Classical software layer model</td>
<td>12</td>
</tr>
<tr>
<td>3.1</td>
<td>Simple overview of the software testing process</td>
<td>16</td>
</tr>
<tr>
<td>4.1</td>
<td>An overview of the Valgrind instrumentation process</td>
<td>23</td>
</tr>
<tr>
<td>5.1</td>
<td>Overview of the experiment setup</td>
<td>38</td>
</tr>
<tr>
<td>5.2</td>
<td>Results of the random bit flip experiment</td>
<td>41</td>
</tr>
<tr>
<td>5.3</td>
<td>Results of the full sweep experiment</td>
<td>42</td>
</tr>
<tr>
<td>5.4</td>
<td>Comparison of executable file sizes used in the FITIn experiments in bytes</td>
<td>42</td>
</tr>
<tr>
<td>5.5</td>
<td>Comparison of optimized executable file sizes in bytes</td>
<td>43</td>
</tr>
<tr>
<td>5.6</td>
<td>Executed instructions during golden runs</td>
<td>44</td>
</tr>
<tr>
<td>5.7</td>
<td>Executed instructions during golden runs of SIHFT versions only</td>
<td>44</td>
</tr>
</tbody>
</table>
List of Tables

3.1 Abstraction hierarchy and injection methods . . . . . . . . . . . . . 18
3.2 Abstraction hierarchy and injection methods . . . . . . . . . . . . . 19
5.1 Description of defines to enable the Flipsafe type with gcc . . . . . . . 37
B.1 List of FITIn command-line parameters . . . . . . . . . . . . . . . 60
A VEX IR

Valgrind provides a tool libVEX, which converts binary program code to an intermediate representation (IR). In the following we will use VEX IR, VEX and IR synonymously. This representation can be analyzed and modified by the Valgrind tools. This Appendix provides the grammar of the VEX IR in a simplified form. LibVex is actively developed as is Valgrind and may change.

A.1 Language description

The IR describes operations of a binary program. The following definitions are taken from [23] and adapted to have a more complete list of definitions. It is the Backus-Naur for representation of the IR language.

The IR is strongly typed. If modified by a tool, the IR will be checked for typing errors by libVEX.

\[
\begin{align*}
\text{IRStmt} & : = \text{NoOp} \\
& \mid \text{IMark of Addr64 * Int * UChar} \\
& \mid \text{AbiHint of IRExpr * Int * IRExpr} \\
& \mid \text{Put of Int * IRExpr} \\
& \mid \text{PutI of IRRegArray * IRExpr * Int * IRExpr} \\
& \mid \text{Tmp of IRTemp * IRExpr} \\
& \mid \text{Store of IREndness * IRExpr * IRExpr} \\
& \mid \text{CAS of IRCAS} \\
& \mid \text{LLSC of IREndness * IRTemp * IRExpr * IRExpr} \\
& \mid \text{Dirty of IRDirty} \\
& \mid \text{MBE of IRMBusEvent} \\
& \mid \text{Exit of IRExpr * IRConst * IRJumpKind * Int}
\end{align*}
\]

\[
\begin{align*}
\text{IRExpr} & : = \text{Binder of Int} \\
& \mid \text{Get of Int * IRType} \\
& \mid \text{GetI of IRRegArray * IRExpr * Int} \\
& \mid \text{Tmp of IRTemp} \\
& \mid \text{Qop of IROp * IRExpr * IRExpr * IRExpr * IRExpr} \\
& \mid \text{Triop of IROp * IRExpr * IRExpr * IRExpr * IRExpr * IRExpr} \\
& \mid \text{Binop of IROp * IRExpr * IRExpr} \\
& \mid \text{Unop of IROp * IRExpr} \\
& \mid \text{Load of IREndness * IRType * IRExpr} \\
& \mid \text{Const of IRConst} \\
& \mid \text{CCall of IRCallee * IRType * IRExprVec}
\end{align*}
\]
| MuxOX of IRExpr * IRExpr * IRExpr

IRMBusEvent ::= Imbe_Fence | Imbe_CancelReservation

IRExprVec ::= IRExpr | IRExpr * IRExprVec

IREndness ::= LittleEndian | BigEndian

IRRegArray ::= Int * IRType * Int

IRTemp ::= UInt

IRCAS ::= IRTemp * IRTemp * IREndness * IRExpr * IRExpr
* IRExpr * IRExpr * IRExpr

IRConst ::= Bool | UChar | UShort | Uint | ULong | Float | Double

IRJumpKind ::= Ijk_Boring | Ijk_Call | Ijk_Ret
| Ijk_ClientReq | Ijk_Yield
| Ijk_EmWarn | Ijk_EmFail | Ijk_NoDecode | Ijk_MapFail | Ijk_TInval
| Ijk_NoRedir | Ijk_SigTRAP | Ijk_SigSEGV | Ijk_SigBUS | Ijk_Sys_syscall
| Ijk_Sys_int32 | Ijk_Sys_int128 | Ijk_Sys_int129 | Ijk_Sys_int130
| Ijk_Sys_sysenter

IRTType ::= Ity_INVALID | Ity_I1 | Ity_I8 | Ity_I16 | Ity_I32
| Ity_I64 | Ity_I128 | Ity_F32 | Ity_F64 | Ity_D32 | Ity_D64
| Ity_D128 | Ity_F128 | Ity_V128 | Ity_V128
B  FITIn API

This appendix describes some details of FITIn usage. The aim is to have a reference overview of the interfaces needed to program FITIn.

B.1 Annotation description

FITIn uses source code annotations to help identifying possible monitoring targets. The annotations can be used in the target program even if the program is not used by FITIn. Annotations are realized with the Valgrind Client Request mechanism. It allows passing information in an effective manner from the target program to the Valgrind tool. Client Requests do not interfere with the actual program.

The following interfaces are defined in the fitin.h file.

B.1.1 FITIN_MONITOR_VARIABLE(var)

Monitor the given variable for FITIn handling.
Parameters:
• var: Variable to be monitored

B.1.2 FITIN_MONITOR_ARRAY(array, size)

Monitor the array content.
Parameters:
• array: Array to be monitored
• size: Number of elements in the array

B.1.3 FITIN_MONITOR_MEMORY(mem, size)

Monitor the given memory area.
Parameters:
• mem: Start address of the memory area to be monitored
• size: Size of the memory area to be monitored

B.1.4 FITIN_UNMONITOR_VARIABLE(var)

Unregister the given variable for FITIn handling.
Parameters:
• var: Variable to be unregistered
B.1.5 FITIN_UNMONITOR_ARRAY(array, size)

Unregister the monitoring of the array content.
Parameters:
- **array**: Array to be unregistered
- **size**: Number of elements in the array

B.1.6 FITIN_UNMONITOR_MEMORY(mem, size)

Unregister the monitoring of the memory area.
Parameters:
- **mem**: Start address of the memory area to be unregistered
- **size**: Size of the memory area to be unregistered

B.2 Command-line parameters

FITIn, as a tool for Valgrind, uses command-line parameters to work correctly. It is executed using the `--tool=fitin` parameter of Valgrind. The FITIn specific parameters are listed in Table B.1.

<table>
<thead>
<tr>
<th>Command-line parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>--fnname=&lt;name&gt;</code></td>
<td>Only monitor instructions in function <code>&lt;name&gt;</code>. (Default: main)</td>
</tr>
<tr>
<td><code>--include=&lt;dir&gt;</code></td>
<td>Only monitor instructions that have debug information from this directory.</td>
</tr>
<tr>
<td><code>--mod-load-time=&lt;number&gt;</code></td>
<td>Modify at a given load number. (Default: 0)</td>
</tr>
<tr>
<td><code>--mod_bit=&lt;number&gt;</code></td>
<td>Modify the given bit of the target.</td>
</tr>
<tr>
<td><code>--inst_limit=&lt;number&gt;</code></td>
<td>The maximum numbers of instructions to be executed. Use to prevent infinite loops.</td>
</tr>
<tr>
<td>`--golden-run=[yes</td>
<td>no]`</td>
</tr>
</tbody>
</table>

Table B.1: List of FITIn command-line parameters

B.3 Step-by-step tutorial

Let us suppose the Valgrind source code including FITIn is located in the `~/valgrind` directory on a LINUX machine. The customized Valgrind version is built like the following:

1. `cd ~/valgrind`
2. `./autogen.sh`
3. `./configure --prefix=$PWD`
4. `make install`
The Valgrind executables are then located in /valgrind/bin/. In the following example we build a sample program that may be analyzed using FITIn. One may execute the following steps:

1. Create a new folder:

```bash
1 cd ~
2 mkdir testProg
3 cd testProg
```

2. Create a program file `main.c` using your favorite editor. Insert the following source code. This code is already annotated using the `FITIN_MONITOR_VARIABLE` and `FITIN_UNMONITOR_VARIABLE` macros (See lines 9 and 17).

```c
#include <stdio.h>
#include "../valgrind/include/valgrind/fitin.h"

int main (int argc, char *argv[])
{
    unsigned int testVar = 1;
    //Register for monitoring in FITIn
    FITIN_MONITOR_VARIABLE(testVar);
    if(testVar == 1)
        printf("Hello World!\n");
    else
        printf("Fault injected! testVar == %d\n", testVar);
    //Unregister the monitoring
    FITIN_UNMONITOR_VARIABLE(testVar);
    return 0;
}
```

3. Run FITIn in the 'golden run' mode.

```bash
./valgrind/bin/valgrind --tool=fitin --golden-run=yes --
include=$PWD $PWD/testProg
```

The output should look similar to this:
We see that only one memory load operation was monitored (See line 10).

4. Run FITIn in the 'fault injection' mode. The following line will flip the least significant bit before the first load operation of the monitored variable testVar.

```
../valgrind/bin/valgrind --tool=fitin --mod-load-time=1 --mod-bit=0 --include=$PWD $PWD/testProg
```

The output should look similar to this:

```
==23594== FITIn, A simple fault injection tool
==23594== Copyright (C) 2013, and GNU GPL'd, by Clemens Terasa
==23594== Hamburg University of Technology (TUHH)
==23594== Institute for Software Systems
==23594== Using Valgrind-3.8.0.SVN and LibVEX; rerun with -h for copyright info
==23594== Command: /home/user/testProg/testProg
==23594== Fault injected! testVar == 0
==23594==
Totals:
Overall memory loads: 6
Monitored memory loads: 2
Monitored memory bytes load: 8
Instructions executed: 99976
```

We see in line 6 that the fault was successfully injected. The variable testVar is modified by the fault injection scheme.