technische universität
dortmund

Masterarbeit

Fast User-Mode Fault Injection with FAIL*

Philipp Koppenstein

December 2, 2020

Betreuer:
Dr.-Ing. Horst Schirmeier
Prof. Dr. Jian-Jia Chen

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl 12
Arbeitsgruppe Eingebettete Systemsoftware
`https://ess.cs.tu-dortmund.de`

Abstract

Hardware faults induced by radiation can lead to catastrophic failures in safety-critical systems. Mitigation of this via hardware fault tolerance is expensive and slow, so the alternative of software-implemented hardware fault tolerance needs to be considered. It is necessary to test software-implemented hardware fault tolerance implementations to evaluate their effects on reliability. The Fault Injection (FI) tool FAIL* is designed to assist with this, but for large FI campaigns, the currently implemented backend can lead to long runtimes on the scale of hours or days. An additional limitation is that it is only possible to inject faults into full-system images and not into user-space binaries.

In this thesis, we extend the FI tool FAIL* in order to achieve greater simulation speed, a wider variety of target platforms, and the support for user-space binaries in addition to system-mode simulation. To achieve these goals, we evaluate suitable candidates for a new backend for FAIL* and choose the one which fits best. We describe the implementation details and evaluate whether fault injection in user-mode is feasible for testing software fault tolerance mechanisms with FAIL*. We also consider support for multiple architectures and exemplarily implement it for x86 and ARM both respectively in 32 and 64 bit. Additionally, we benchmark the performance of this new backend extensively.

Contents

1 Introduction

Feature sizes of integrated circuits shrink continuously and are predicted to do so in the future [28]. This leads, together with also reduced voltages, to a growing susceptibility towards hardware faults, caused by ionizing radiation, aging, or thermal effects [19].

Hardware faults manifest themselves in multiple forms. Permanent faults stay forever, intermittent faults disappear and reappear again, and transient faults appear once and disappear again. Transient faults, in particular, are the most common as shown by Iyer and Rossetti [29] and often show as bit flips in the memory or registers; they are mostly radiation-induced. Recent developments in space weather also led to increased cosmic radiation levels since the 1950s and point to possible increases in the next decade or two [41], exasperating the transient fault problem further.

Safety-critical systems are not immune to hardware faults, and measures for mitigation, therefore, need to be taken in order to minimize risks for lives or equipment. These systems need to achieve a certain level of fault tolerance. Fault tolerance denotes the ability of a system to operate unimpeded even if faults are present.

One approach to achieve fault tolerance is hardware fault tolerance, which is realized, for example, by redundancy in chip designs or error-correcting code memory. However, this can be prohibitively expensive for some applications, such as automotive applications or applications in other price sensitive products. As a result, more cost-efficient measures need to be considered.

Software implemented hardware fault tolerance (SIHFT) offers a potential solution, as it runs on commercial off-the-shelf (COTS) hardware and thus can be more cost-effective. To implement this, careful software design is needed with suitable mitigation strategies.

1.1 Motivation

For SIHFT, it is critical to evaluate the deployed measures to know which fault tolerance measures are effective. Especially in the case of software fault tolerance, it can be important to quantify how much specific fault tolerance measures decrease risk or whether they are worthwhile at all. It is often not obvious for the developer which data structures and program modules are most critical for fault tolerance and are worth protecting.

To ease the development of fault tolerant systems, a continuous fault tolerance assessment process can be valuable [44]. Such a process keeps developers informed about changes in fault tolerance metrics over the development process's span. FAIL* [43] can be a tool that provides those metrics. FAIL* is capable of surveying the whole fault space through fault injection (FI) and thus provides more accurate results than competing tools. To keep exploration of the whole fault space affordable and within a reasonable timeframe, the FI experiments' high execution speed is crucial. The currently employed backend in FAIL* based on the Bochs emulator [33] lacks with regard to execution speed when compared to industry-leading emulators.

An additional limitation of the current backend is the fact that only system software can be tested. But oftentimes, programs are not developed as a complete operating system but only as a binary that can be executed, for example, in GNU/Linux (user-space binary). These user-space binaries ease development compared to needing to build a system image to test the deployed fault tolerance measures. As of now, FAIL* does not support analyzing such binaries.

1.2 Goals

With an eye to the two main limitations of the current backend we discussed in the previous section, we aim in this thesis to analyze existing emulators for their potential to speed up FAIL*'s FI campaigns significantly.

FAIL* provides a way to hide details about the underlying hardware or simulator details through an abstraction layer. Experiments use the abstraction layer to interact with the underlying backend. This architecture of FAIL* allows for an introduction of a new backend so that experiments can seamlessly use the new one.

To analyze their suitability as the new backend, the emulators would be evaluated whether they better fill the role of the Bochs emulator. Criteria for this will be, for example, speed and ease of integration.

Very importantly, the new backend needs to have the ability to enable FI in user-space binaries, from now on called user-mode FI. This would enable easier adoption of fault injection tools for testing fault tolerance measures, as the need to build a system image would no longer exist.

The new backend then needs to be designed in such a way as to support the new user-mode so that it integrates into FAIL*.

The resulting implementation shall be evaluated with large benchmark setups and compared to other existing back ends. Additionally, it shall be examined whether user-mode FI is suitable for testing fault tolerance measures.

1.3 Outline

Initially, chapter 2 introduces the necessary foundations for understanding the rest of the thesis.

Afterwards, the problem at hand is analyzed in chapter 3, and the requirements for the design are synthesized.

With these established chapter 4 introduces potential candidates for a new FAIL* backend and discusses the choice of one of them.

Chapter 5 then describes the implementation of the chosen backend candidate and the integration into FAIL*.

Subsequently, chapter 6 does an extensive evaluation of the newly implemented backend with special attention given to the user-mode.

Chapter 7 gives an overview of related works; in particular, previous fault injection tools with QEMU are regarded.

And at last, chapter 8 gives a summary and outlook.

2 Foundations

This chapter introduces fault injection and why the need for it exists. Furthermore, FAIL*, the main framework upon which this thesis is based, is presented.

2.1 Faults

We aim to define hardware faults and their different types in this section.

To visualize hardware faults and their consequences, a hardware or software system can be divided into layers. Each layer can only observe the behavior of the layer directly below and is thus dependent on it. [44]

A *fault* can, for example, be an energetic particle hitting a transistor. For the fault to have a consequence, the fault needs to be *activated*, which would mean in the context of the previous example that this impacts the transistor. For instance, it switches slightly differently, or a bit is flipped.

After activation, the fault is called an *error*, which is one layer above the fault. This can then be *externalized* which would be called *failure*.

If the fault propagates through all of the layers, the user is experiencing some disturbance. This can, for example, manifest in an infinite loop or *Silent Data Corruption* (SDC). An SDC means an alteration of the program's output that the user gets to see, without any indication, that the output may be wrong.

This propagation of the fault through the layers can be potentially stopped at any layer. Any mechanism that stops the externalization of an error caused by an underlying fault is called a *fault tolerance mechanism*.

Faults differ in their lifetime; *transient faults* (or soft faults) are typically only present for a very short period of time. They disappear and do not come back. These are different from *intermittent faults*, which disappear and then at some point later appear again. Often these are "early indicators of impending permanent faults" [38]. "Permanent faults," as the name suggests, only appear once and stay forever. The component has a defect at this point. In this thesis, only transient faults are considered, particularly in the form of bit flips.

A fault can occur at any *point in time* at any *point in the hardware*. A point in time is typically the smallest feasible timestep supported, for example, a CPU cycle. A point in the hardware would be, for example, a bit in a memory cell or a CPU register. These two dimensions build the *fault space*, which is visualized in figure 2.2.

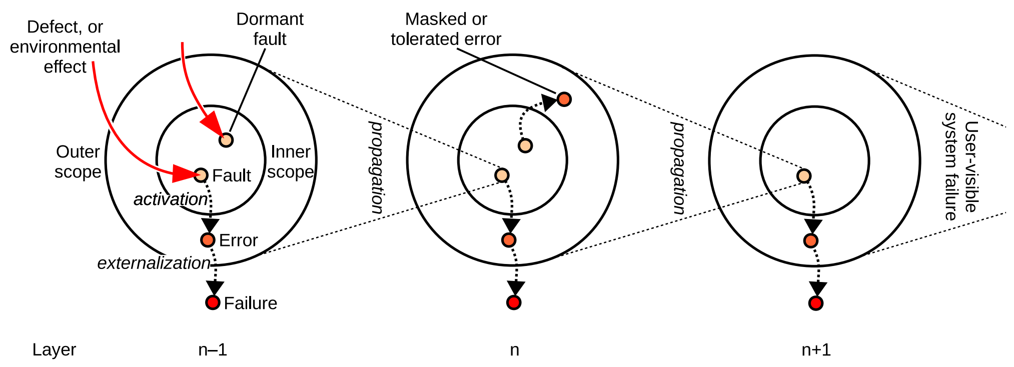


Figure 2.1: This figure illustrates the propagation of faults from defects or environmental effects up to user-visible system failures. Note that a fault needs to be externalized multiple times until the user is impacted. [44].

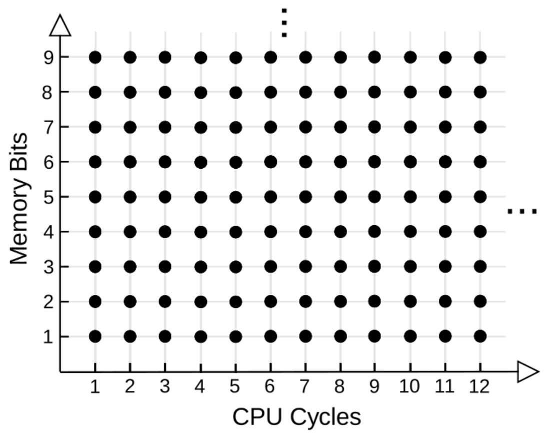


Figure 2.2: The fault space consisting of two dimensions. The one dimension being any point in the memory or the hardware and the other being any point in time, here measured in CPU cycles. [44].

2.2 Fault Injection Techniques

Several Techniques have been developed in the past to inject hardware faults artificially. Such techniques aim to mimic the effects of a naturally occurring cause (e.g., alpha or neutron radiation or a disturbance of the power supply) but drastically increase the rate with which faults occur, such that effects are visible for small sample sizes.

There are several quality criteria for evaluating different fault injection techniques [47].

The first criterion is *repeatability*. Repeatability is the ability to inject a specific fault and obtain the same result.

Controllability is the ability to control when and where a fault is injected.

Intrusiveness is the level of unintended impact on the target system. High intrusiveness can lead to a *probe effect* [14, 42], which manifests itself in the way that a different result is obtained, while, if the fault were to occur on the real hardware, it would not lead to an alteration of the result.

Observability describes the capability to inject a fault and observe and measure the ensuing effects.

Reachability is the degree to which all possible aspects of a processor's state can be altered.

With these quality criteria, we can now turn to several different fault injection techniques and assess them.

2.2.1 Hardware-Based Fault Injection

One of the most straightforward ways to increase the occurrence of soft errors is to amplify the natural causes for them. One way would be to expose the hardware to heavy-ion radiation [25, 31]. Another way would be to disturb the power supply [36].

These techniques fall short in many criteria, the only advantage being no intrusiveness and perfect reachability. Another disadvantage is the high cost for every experiment, especially in the case of radiation exposure [50] [32].

A third way to inject faults is to install probes to the pins of CPU chips, which, for example, the FI tools RIFLE [34] and MESSALINE [2] do. Test access ports can also be used by FI tools to inject faults, for example, GOOFI-2 [47], Xception [10], and Fidalgo et al. [21]. These techniques lead to better repeatability and controllability, even more so if using the test access port. However, the test access port leads to reduced injection speeds, and both techniques suffer from limited reachability.

A significant disadvantage for most of the hardware-based fault injection techniques is that they require specialized hardware setups, which is, of course, expensive. Only the FI via test access port is possible with COTS hardware while still satisfying the controllability and repeatability necessary for a detailed post-injection analysis.

2.2.2 Software-Implemented Fault Injection

Another class of techniques is software-implemented Fault Injection (SWIFI).

One of those techniques is *pre-runtime* SWIFI, where the software or data is modified before it is run. This is done by the FI tools GOOFI [1] and FUCHS [23]. An advantage is the largely unimpacted execution speed, but this technique has a limited reachability of injectable state and the potential for a "probe effect" (high intrusiveness). Another drawback is a long roundtrip when the injection location changes.

The next technique *runtime* SWIFI mitigates this. Runtime SWIFI injects faults into the running system by leveraging exceptions or debugging features of the CPU. FIAT [4], FERRARI [30], Xception [10] and GOOFI-2 [47] use runtime SWIFI. Although the long roundtrip is no longer present, nothing is done to mitigate the probe effect and limited reachability.

An argument for using either form of SWIFI is that there is no need for specialized hardware and, thus, is affordable.

2.2.3 Simulation-Based Fault Injection

Simulation-based FI [6], [32] injects faults into simulated hardware. There is a tradeoff with the simulators between speed and simulation accuracy.

On the one side, there are FI tools for low-level hardware models as VERIFY [46] and MAFALDA [3], and on the other side, there are FI tools, which use slightly inaccurate simulators or even emulators such as F-SEFI [24] and Qinject [22].[1] These inject faults with the QEMU [5] emulator.

The slowdown encountered is often very significant, even orders of magnitude for low-level simulators. The big benefit is very good reachability of injectable state and low intrusiveness, both advantages over SWIFI. Also, controllability, repeatability, and observability are all on a high level but limited by the detail level of the used simulator or emulator.

A functionality, which is impossible with SWIFI but possible with Simulation-based FI, is the use of checkpoints, which can help to mitigate slower speeds [7]. Simulation-based FI has the same benefit to SWIFI of running on COTS hardware and is also more cost-effective than hardware-based FI.

2.3 FAIL*

In this section, we introduce the fault injection tool FAIL*. First, we list which of the FI techniques are supported by FAIL* and some capabilities, until we introduce some terminology used by it. After this, we take a closer look at the architecture of FAIL*.

2.3.1 FI Technique Support and Capabilities

FAIL* supports three of the FI techniques mentioned above, simulation-based FI and a hybrid technique between SWIFI and test access port based FI.

[1]The terms simulator and emulator are used interchangeably in this thesis. The difference between both generally lies in the accuracy of the simulation. An emulator does not simulate as accurately as a simulator. This difference is of no concern here; both are used in the context of simulation-based Fault Injection.

This thesis focuses on simulation-based FI. Currently, three different simulator backends are supported in FAIL*. Bochs is the best-supported backend, and QEMU and gem5 are implemented in a rudimentary state. FAIL* is different from many other FI tools, as most only sample small parts of the fault space [43], while FAIL* can cover the whole fault space of the target application. To achieve this, FAIL* leverages massive parallelization and advanced fault-space pruning techniques.

Moreover, FAIL* can offer fine-grained post-injection analyses on the level of single CPU instructions, variables or, high-level program code lines. In contrast to most FI tools in existence, FAIL* can quantitatively estimate specific applications' hardware fault tolerance.

2.3.2 FI Experiments, Outcomes, and Campaigns

A fault injection *experiment* consists of injecting a fault at runtime into the hardware or simulated hardware that runs the analyzed software. The *outcome* of this injection is recorded at the end of the experiment.

There are multiple types of outcomes; FAIL* considers these in particular:

- No Effect: There was no measurable difference in the outcome of the analyzed software. The time constraints were satisfied, and the output was the same as in the case without the fault.
- Detected: The fault was detected, and appropriate measures were taken.
- Silent Data Corruption: An alteration of the program's output that the user gets to see, without any indication that the output may be wrong.
- Timeout: The fault led to an increase in runtime, which was so large that the given time constraints were not satisfied.

The *golden run* is the first run of the program without any fault injection. It serves the function of tracing all memory accesses and instructions executed. Additionally, the golden run traces the output of a program, for example, the output over a serial port. A fault injection *campaign* consists of multiple fault injection experiments. From the golden run, the fault injection campaign generates experiments for different points in the fault space to inject a fault into and runs the experiments.

2.3.3 Architecture

FAIL* is divided into the *plumbing* and the *assesment-cycle layers.*

The plumbing layer abstracts away the specific backend or execution environment in use, such as a simulator, and is organized in a client/server architecture.

Many clients run the experiments in parallel, and the server assigns work to the clients and records results in a MySQL database. More specifically, the server runs user-defined *campaigns* and generates experiment parameters for the clients. The client then runs the experiment in its execution-environment abstraction layer, using a user-defined *experiment* procedure. The plumbing layer is the layer with complete control for implementing new experiment- and campaign- procedures. “This layer is primarily aimed at researchers working on FI techniques and optimizations themselves” [44], and the layer on which this thesis focuses.

The plumbing layer exposes an API, upon which the assessment-cycle layer builds. Therefore, the assessment-cycle layer is less flexible and powerful but more straightforward to use. The assessment-cycle layer is used to introduce single-bit flips in CPU registers and main memory. “In contrast to the plumbing layer, the assessment-cycle layer primarily aims at the developers and users of SIHFT mechanisms” [44], and is therefore of not much interest in this thesis as the goal is *not* to introduce new SIHFT mechanisms, so changes to this layer are minimal.

The plumbing layer consists of a client and a server-side, which is shown in figure 2.3. The server side is of little interest; it is sufficient to know that the server distributes job parameters to the clients and collects the results from them.

On the client-side is a so-called FAIL* *client instance*, or in short fail-client, which uses a simulator or emulator and implements an abstraction on top of it, which can run jobs from given job parameters, so basically an extended simulator or emulator.

A FAIL* client instance can be divided into three larger components, user-defined experiment, the execution-environment abstraction, and the backend consisting of a modified simulator or emulator. The user-defined experiment procedure is started in the beginning in parallel to the simulator as a coroutine. Both run at mutually exclusive times; each must give back control to each other at certain events. A visualization of this control flow can be seen in figure 2.4.

For the user-defined experiment procedure to orchestrate the target backends, the *Execution-Environment Abstraction* (EEA) is the common interface. At the heart of the EEA is the *SimulatorController* class. The experiment procedure calls methods from this class to communicate with the EEA. Additionally, the EEA has classes for specific features, for example, accessing memory via the *MemoryManager* class, accessing CPU registers with the RegisterManager class and the *ListenerManager* class. The *ListenerManager* class provides means for registering listeners for different types of events such as: Reaching specific program instructions (similar to breakpoints), access to specific memory addresses[2], CPU exceptions, external interrupts, serial I/O, and the passing of specific amounts of backend time. Furthermore, the EEA provides meta information on the target backends, for example, the number of CPUs.

[2]These are often called watchpoints.

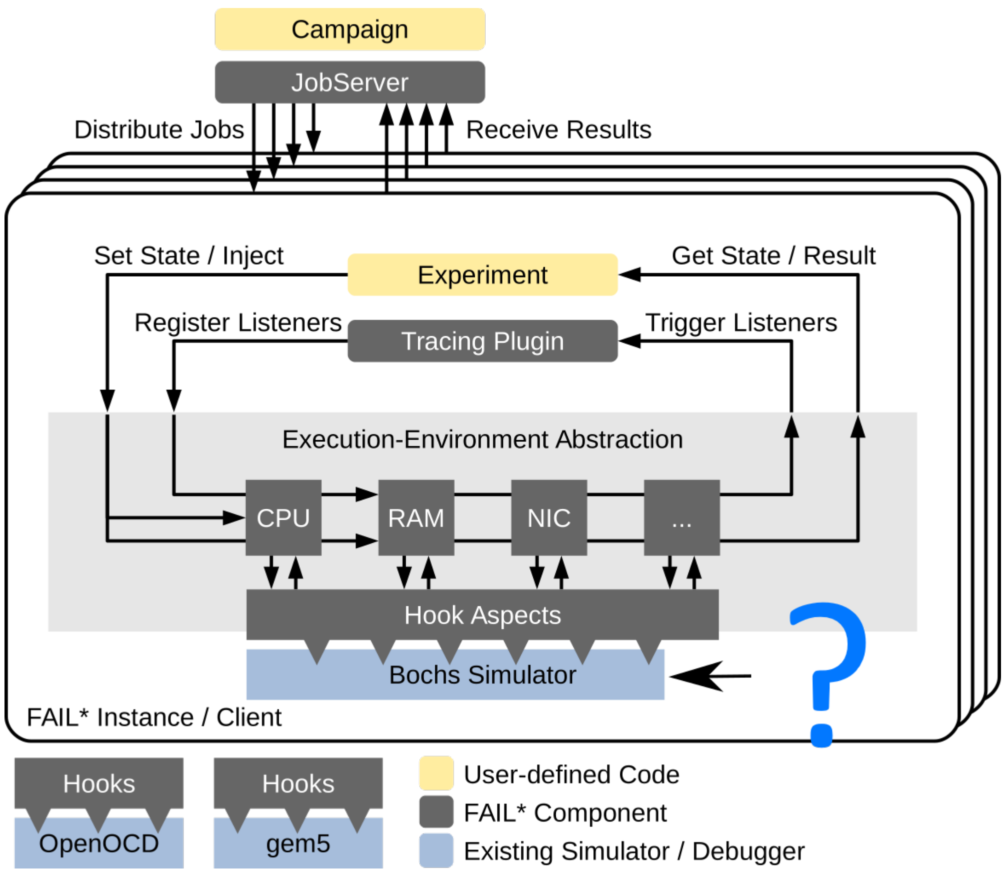


Figure 2.3: Shows an overview of the FAIL* architecture. The client instances receive jobs from the server, which runs a campaign. Then the client instance runs the experiment and injects faults with the use of the EEA, which hooks into a backend. The backends are Bochs, gem5, OpenOCD, and the question mark symbolizes the new backend which we will introduce later in this thesis. Adapted from [44].

A target backend is implemented through the specialization of the EEA classes, primarily via the SimulatorController and Manager classes.

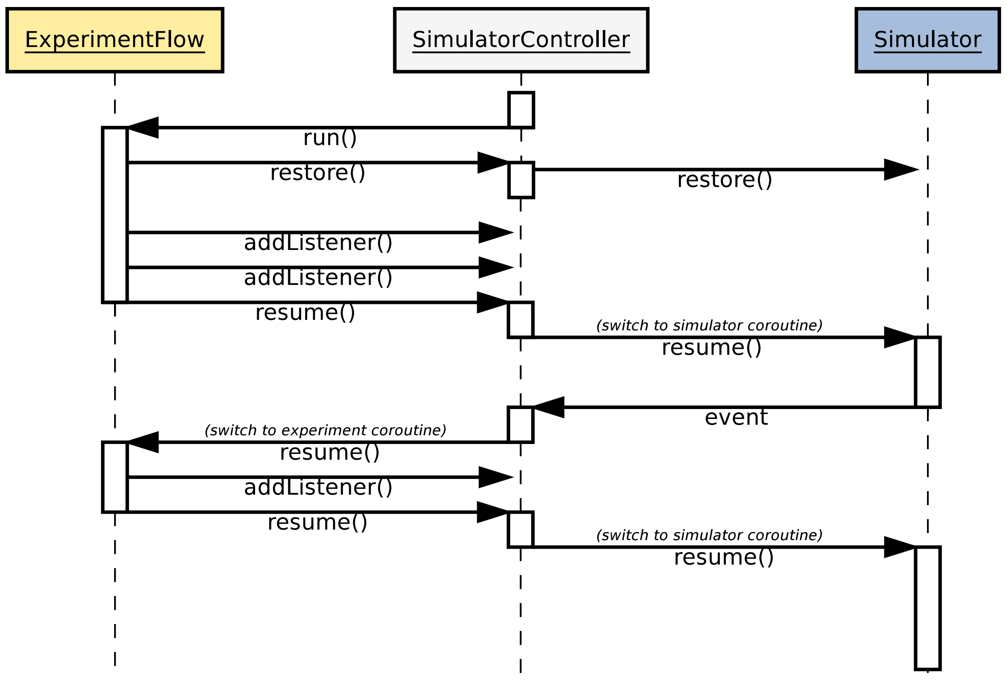


Figure 2.4: This depicts the control flow of FAIL*. The SimulatorController runs the ExperimentFlow and the ExperimentFlow interacts with the Simulator in the backend. The simulator is in a separate coroutine from the SimulatorController and the ExperimentFlow. [44].

2.4 Candidates for the New Backend

In this section, we first introduce the main reason for the speed difference between the new candidates for extending the backend and, after that, introduce the candidates. First, Valgrind is introduced, and then QEMU.

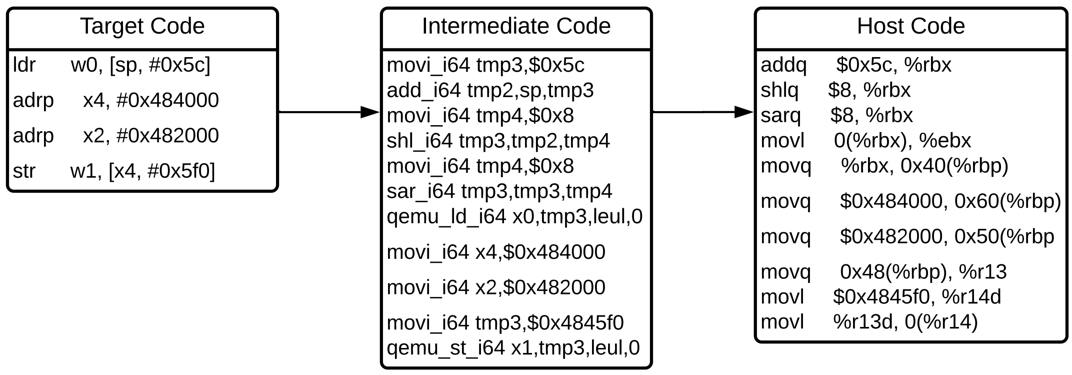
2.4.1 Dynamic Binary Translation

A difference between the newly introduced candidates and the Bochs emulator is that the new candidates utilize dynamic binary translation while Bochs uses a simple interpreter loop.

A dynamic binary translator performs a runtime conversion of target CPU instructions to host instructions and saves the resulting binary code in a translation cache [5]. The target instruction set can be the same as the host instruction set but this is not necessarily the case.

It has the advantage over an interpreter loop of only needing to fetch and decode instructions once, if the cache is big enough. This vastly increases the speeds of dynamic binary translators over simple interpreters like Bochs if instructions are executed more than once, for example, in a loop, which is the main reason why the candidates are considered at all.

An example of dynamic binary translation can be seen in figure 2.5.



```
ldr     w0, [sp, #0x5c]
adrp     x4, #0x484000
adrp     x2, #0x482000
str     w1, [x4, #0x5f0]
```

```
movi_i64 tmp3,$0x5c
add_i64 tmp2,sp,tmp3
movi_i64 tmp4,$0x8
shl_i64 tmp3,tmp2,tmp4
movi_i64 tmp4,$0x8
sar_i64 tmp3,tmp3,tmp4
qemu_ld_i64 x0,tmp3,leul,0
movi_i64 x4,$0x484000
movi_i64 x2,$0x482000
movi_i64 tmp3,$0x4845f0
qemu_st_i64 x1,tmp3,leul,0
```

```
addq     $0x5c, %rbx
shlq     $8, %rbx
sarq     $8, %rbx
movl      0(%rbx), %ebx
movq      %rbx, 0x40(%rbp)
movq      $0x484000, 0x60(%rbp)
movq      $0x482000, 0x50(%rbp
movq      0x48(%rbp), %r13
movl     $0x4845f0, %r14d
movl     %r13d, 0(%r14)
```

Figure 2.5: This shows an example of a dynamic binary translation from aarch64 instructions to x86-64 instructions. We translated the instructions with QEMU. Some dynamic binary translators do not need the intermediate step and translate directly from target to host code.

2.4.2 Valgrind

This subsection describes Valgrind [39], a dynamic binary instrumentation framework for writing dynamic binary analysis (DBA) tools.

Valgrind is unique in its capability of using *shadow values*, which are essentially values that describe every memory and register value. The most well known DBA tool implemented with Valgrind is Memcheck [45], a tool that can detect undefined value errors. To do this, it leverages the shadow values to track the definedness of values.

To support the shadow values, Valgrind needs to access large amounts of analysis data and update them in irregular patterns. Because of this, Valgrind runs comparatively slow but is useful to implement tools that are difficult to impossible to implement with other DBI frameworks.

Valgrind is entirely focused on dynamic binary analysis in user-mode emulation; it is not suited for full system emulation. Valgrind uses an architecture-neutral intermediate representation in which the translation units are *superblocks*, which are single-entry, multiple-exit code fragments. This leads to a rather large amount of supported platforms, which include: x86, x86-64, PPC32, PPC64, PPC64LE, S390X, ARM, ARM64, MIPS32, MIPS64 on GNU/Linux. Some of these architectures are supported in other operating systems as well (Details [49]).

2.4.3 QEMU

This subsection introduces QEMU, the Quick EMUlator.

QEMU [5] is a machine emulator, that supports many platforms (Alpha, ARM, CRIS, HPPA, x86, x86-64, LatticeMico32, 68K, MicroBlaze, MIPS, Moxie, Nios2, OpenRISC, Power, PowerPC, RISC-V, SH4, Sparc, s390x, TileGX, TriCore, Unicore32,Xtensa) and operating systems. It does full system emulation as well as user-mode emulation. Dynamic binary translation allows running a GNU/Linux or BSD program on multiple machine architectures, even those the program was not compiled for, e.g., an ARM program on an x86-64 machine.

QEMU is developed with execution speed in mind as a primary goal. It prefers speed over emulation accuracy and thus outperforms many other comparable emulators [5]. It is not a cycle-accurate emulator, but it is good enough to run the major operating systems like GNU/Linux, Windows, or macOS. QEMU also supports hardware virtualization, mostly via KVM. However, for this thesis, hardware virtualization is not used; instead, the Tiny Code Generator (TCG) is used. The TCG is a binary translation engine that translates guest machine code to an intermediate representation and then back to machine code for the host system. It is considerably more comfortable and versatile to develop with the TCG in comparison to hardware virtualization; additionally, hardware virtualization is supported only for full system emulation and not for the user-mode emulation.

2.5 QEMU Foundations

We now take an in-depth look into the architecture and implementation of QEMU as this will prove necessary to understand the further design and implementation. Particular emphasis is placed on the differences between user-space emulation and system emulation.

2.5.1 Implementation

QEMU consists of multiple different subsystems, among others of the Tiny Code Generator (TCG), which translates target code to intermediate code and then to host code, the dynamic binary translation.

This translation is done for *translation blocks* (TBs). A TB is a part of a basic block. A basic block is a block of code only ended by a branching instruction. The basic blocks can not be of arbitrary length, so they are split up into translation blocks. Every TB consists of up to 65 535 instructions. A basic block is the same as a TB, if and only if a basic block consists of no more than 65 535 instructions. Figure 2.6 shows the translation and execution process of QEMU.

First, an instruction is fetched and translated into an intermediate instruction. This is repeated until either a branching instruction is encountered, or 65 535 instructions are translated. The resulting TB then consists of the translated host code by the TCG and additional information of which a rough summary can be seen in figure 2.7. This TB is then stored in a pysically indexed cache, and after this, it is executed. After the execution of a translation block is complete, the program counter is examined, and the cache is searched for a TB starting with this instruction. The next TB can either be found in the cache, or else the next TB needs to be translated. This next TB is then linked to the previously executed TB so that the next time the execution does not need to be interrupted for the search of a new TB but can instead just run the TBs directly after another without the overhead. This linking of TBs is called *TB chaining* and is one of the fundamental reasons for the speed of QEMU. The TBs can only be chained to TBs which are on the same page. QEMU does this, together with the physically indexed caches, to avoid invalidating TBs when the MMU mappings change. Thus it should be avoided to disable TB chaining so as not to impact performance dramatically. We found a slowdown by an order of magnitude in the case of a loop that was executed a substantial amount of times.

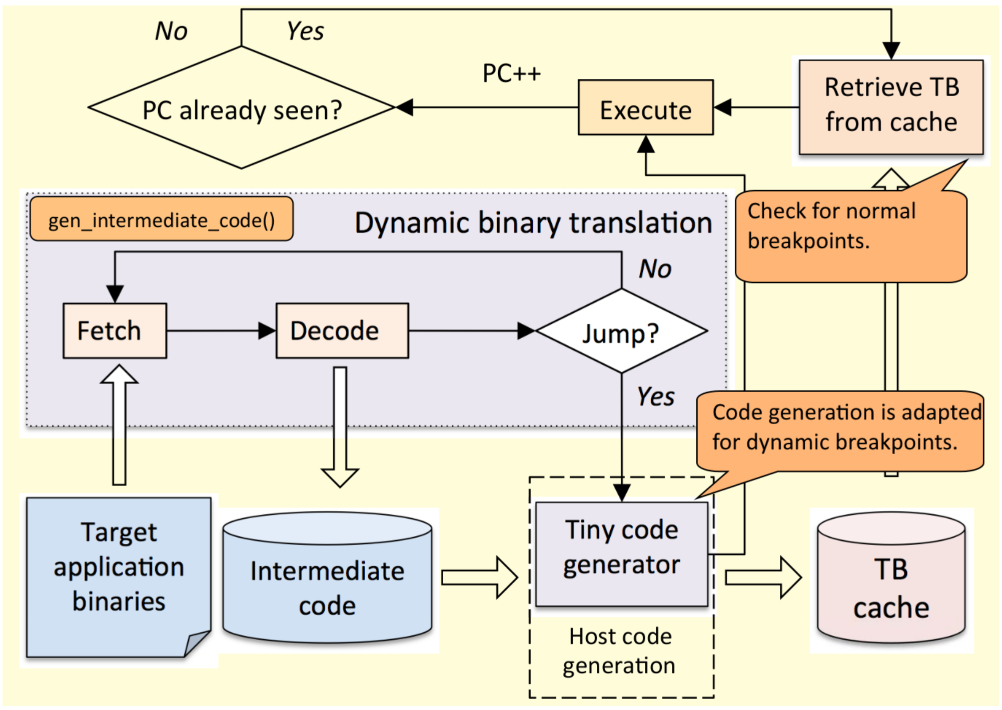
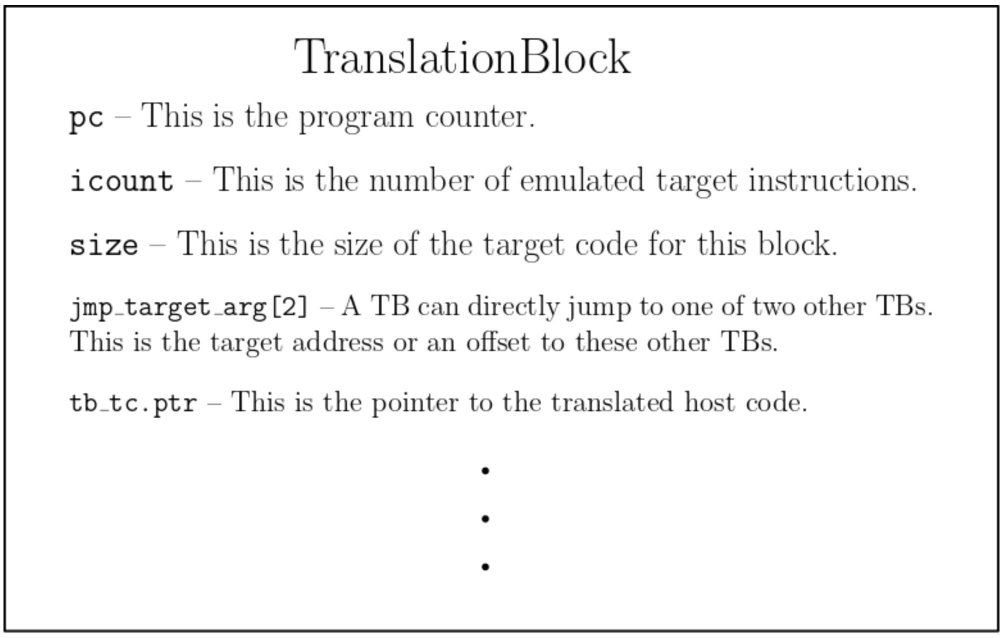


Figure 2.6: This shows the Dynamic Binary Translation from QEMU. Right before the execution of a TB it is checked for breakpoints and a callback is issued if it is hit to allow for fault injections. The intermediate code is adapted for the TCG to allow for counting every instruction. Adapted from [20].



TranslationBlock

`pc` – This is the program counter.

`icount` – This is the number of emulated target instructions.

`size` – This is the size of the target code for this block.

`jmp_target_arg[2]` – A TB can directly jump to one of two other TBs. This is the target address or an offset to these other TBs.

`tb_tc.ptr` – This is the pointer to the translated host code.

•
•
•

Figure 2.7: This figure depicts a translation block in QEMU. The TB consists of a number of translated target instructions. The variable `pc` is the address of the first target instruction. It is not the address of the translated host code. This is stored in the variable `tb_tc.ptr`. Size and number of target instructions are saved in the variables `size` and `icount`. Additionally, the target addresses of other translation blocks, which this TB can directly jump to, are saved in `jmp_target_arg[2]`.

2.5.2 Execution Flow

The execution flow of QEMU begins like every C program with its `main()` function. In the main function, QEMU is initialized in the function `qemu_init()`, where the virtual CPU (vCPU) threads are started. With the parameter `-smp`, the amount of vCPUs can be controlled. After `qemu_init()`, the main function does handle, for example, timers, but the main functionality now lies in the vCPU threads.

A vCPU thread begins with the function `qemu_tcg_cpu_thread_fn()` which handles exceptions which stop the vCPU and calls `tcg_cpu_exec()`. This function waits for exclusive operations and calls the next function `cpu_exec()`. `cpu_exec()` is the main execution loop for a vCPU. In every iteration of the loop it handles exceptions in the function `cpu_handle_interrupt()` or returns if the exception can not be handled here. Then in another loop inside the main execution loop, interrupts are handled by the function `cpu_handle_interrupt()`. If the interrupt could be handled, the procedure described in the subsection above starts. The function `tb_find()` is called which tries to find a TB in the translation cache, or if it finds no cached TB, calls the function `tb_gen_code()` which does the translation from target to intermediate and then to host code. If chaining is possible the function `tb_add_jump()` chains the last TB and the next. After that `tb_find` returns the next TB. Then this TB is executed in the function `cpu_loop_exec_tb()`. After this, host and virtual clocks are aligned, and the main execution loop iterates again. This Execution Flow can be seen in figure 2.8.

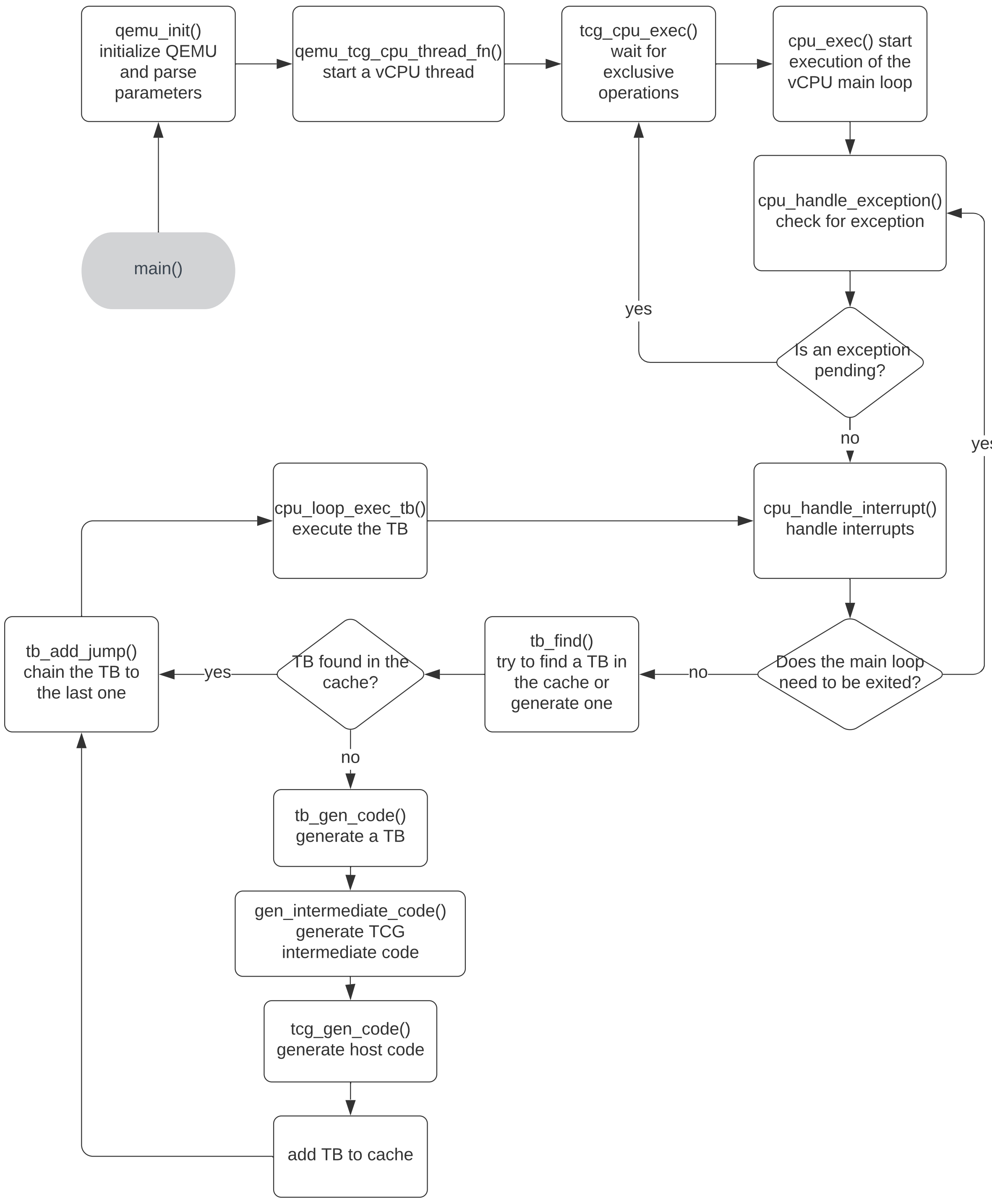

qemu_init() initialize QEMU and parse parameters
qemu_tcg_cpu_thread_fn() start a vCPU thread
tcg_cpu_exec() wait for exclusive operations
cpu_exec() start execution of the vCPU main loop
cpu_handle_exception() check for exception
main()
yes
Is an exception pending?
no
yes
cpu_loop_exec_tb() execute the TB
cpu_handle_interrupt() handle interrupts
tb_add_jump() chain the TB to the last one
yes
TB found in the cache?
tb_find() try to find a TB in the cache or generate one
no
Does the main loop need to be exited?
no
tb_gen_code() generate a TB
gen_intermediate_code() generate TCG intermediate code
tcg_gen_code() generate host code
add TB to cache

Figure 2.8: QEMU execution flow diagram.

2.5.3 Translation Block Chaining

To understand why it is so important for the execution speed that the TBs get chained, the procedure is now illustrated in more detail in figure 2.9. When the main loop starts execution (in `cpu_loop_exec_tb()`), it can not directly jump to the generated host code of the TB that is to be executed, it must take the detour over the prologue. The processor is initialized by the prologue for the execution of the generated host code, and only after the initialization is complete, the jump to the generated host code occurs. After a TB is finished executing, it is either chained to another TB, in which case the execution can continue, or it is not. In the latter case, the epilogue is called. The epilogue restores the normal state, for example, restores callee-saved registers saved in the prologue, and enters the main loop again. The usage of epilogue and prologue is necessary but introduces an additional overhead that can be avoided if the TBs are chained.

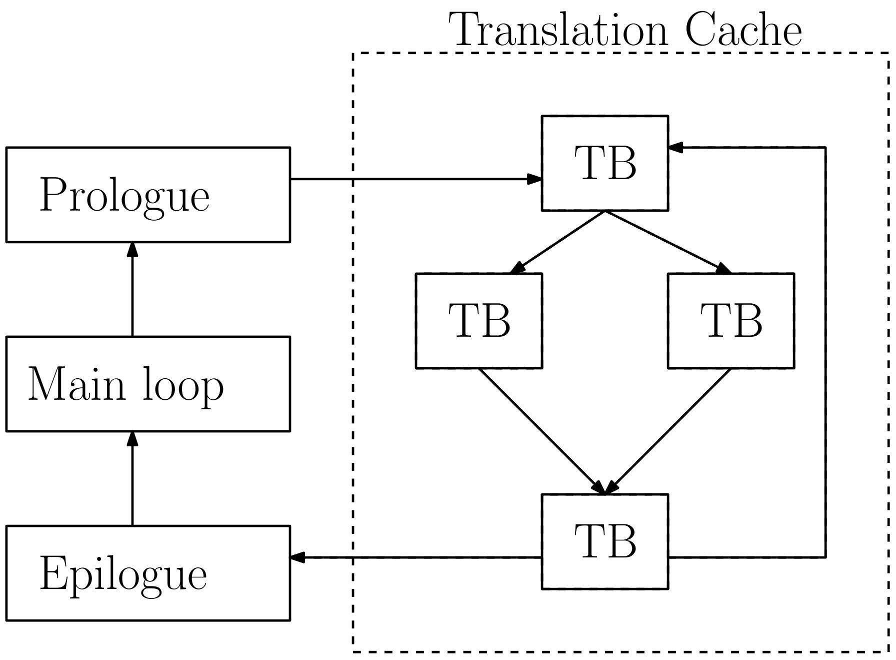


Figure 2.9: Chaining of QEMU translation blocks. If a TB is chained to another TB, the execution can continue, even in loops. If the TB is not chained, a return to the main loop is necessary via the epilogue. Execution can then later begin again via the prologue. The arrows denote chains.

2.5.4 Differences between User-Mode and System Emulation

Here a few differences between QEMU user-mode emulation and system emulation are described. The QEMU user-mode and system emulation are quite different in many aspects but have some big parts in common, too. This section tries to bring to light some of these aspects.

The first big difference between user-mode and system emulation is the absence of interrupts[3], as they are encountered on various CPU models. The handling of those is unnecessary for user-mode emulation and is consequently not compiled into the emulator. However, there is a mechanism similar to interrupts, Unix signals.

Signals play a similar role to interrupts, the difference being that the interrupts are handled by the CPU or kernel, and that signals are typically handled by a user-space process. For example, the x86 exception "divide-by-zero error" would be issued by the CPU, handled by the kernel, and would lead to a "SIGFPE" signal, at least in the case of GNU/Linux. Another analogy between an exception and a signal would be the "general protection fault" x86 exception and the signal SIGSEGV, which both mean a memory access violation. QEMU passes some signals directly from the host to the emulated program and generates others itself, for example, SIGFPE.

The memory management unit is simulated by the QEMU system emulation, virtual vs. physical address translation is implemented as well as a translation lookaside buffer. In the user-mode, this is not necessary. Here simply a region of memory is allocated, and in our observations, the host addresses mostly did not differ from the addresses the emulated program uses.[4]

In system emulation QEMU provides multiple different timers called `QEMUTimers` [9]. These timers allow calling a given function after some specified amount of time has elapsed. These timers can run against different clocks:

- The virtual clock. This clock runs at a high resolution and only when the emulator is running.
- The real-time clock. This clock runs at a resolution of 1000hz and runs even if the emulator is not running.
- The host clock. This clock is nearly the same as the real-time clock but reacts to changes to the system clock.

QEMU does not provide any of these timers in user-mode emulation.

[3]or traps or exceptions

[4]Our understanding of the source code is that the addresses are only translated in case of an address space conflict

The next feature only supported in system emulation is the ability to take live snapshots of the emulator. Snapshots images have to be saved in a specific `QCOW2` or `QED` image format. This can be but does not need to be the original image. Snapshots of raw images are also supported.

Another difference between user-mode and system emulation are syscalls. Syscalls do not leave the system emulator, as the operating system itself resides in it. In user-mode emulation, however, a special mechanism is needed to handle syscalls. For example, a binary compiled for a little-endian architecture must be able to use syscalls, even when a host system is a little-endian machine. Therefore QEMU includes a generic system call translator. This translator translates the system call parameters and handles issues with different endianness and differences between 32- and 64-bit architectures.

2.6 Summary

This chapter explained the foundations, which are necessary for further understanding.

First faults are explained and how they need to be externalized in order for the user to be impacted. Additionally, we introduced the fault space and different FI techniques, such as hardware-based FI, software-implemented FI, and simulation-based FI.

In the next section, we presented the FI tool FAIL* and established key terminologies of it such as fault injection campaigns, experiments, the golden run, and the different result types. After establishing these terms, we took a closer look at the Architecture of FAIL*.

The candidates for extending FAIL*, Valgrind, and QEMU are introduced in the next section, and QEMU's internal mechanisms are examined as well.

In the next chapter, the problem on hand is examined in detail, and requirements are synthesized for the following design.

3 Problem analysis

As was shown in the previous chapter by now, there is no possibility of using simple user-mode FI with FAIL*. This constitutes a hindrance to the easy adoption of fault tolerance measurements. Therefore it is the first goal of this thesis to address this. Other problems we identified were potentially long-running FI campaigns and the effort needed for introducing new backends for new architectures. We analyze these problems further in this chapter and synthesize the requirements for the design.

3.1 User-Mode Fault Injection

Currently, if a researcher wants to test a new fault tolerance measure, the researcher needs to

1. build a program that implements some fault tolerance measure,
2. have an operating system ready in which he can start the program,
3. and build an image from the operating system and the program.

This process is time consuming, complicated, and therefore not beginner-friendly at all. Support for user-mode FI would reduce the last two steps to a simple compiling step. This leads to faster iteration times and easier introduction into the evaluation of fault tolerance measures.

3.2 Efficiency

As FAIL* currently uses the Bochs emulator, which only uses a simple interpreter loop, there is probably room for improvement, maybe through the usage of a dynamic binary translator. The current solution probably leads to longer running campaigns than ultimately necessary. This problem is somewhat mitigated by the architecture of FAIL*, as one can use extreme amounts of computing power, in order to mitigate long campaign running times, but this comes at a cost nonetheless.

A faster target backend would still be preferable, especially so if efficient energy use is a concern.

3.3 Ease of Porting to Different Architectures

In FAIL* as of now, every target backend supports exactly one instruction set architecture. For example, Bochs supports i386 and gem5 supports ARM. This is not optimal in terms of implementing new architectures for FAIL*. This approach is wasteful in the regard that every new backend must be adapted for FAIL* completely from scratch. There is nothing that helps with implementing new architectures. Only if a new backend for an already present architecture is implemented, the execution-environment abstraction carries over partly. But, for example, from the x86 Bochs backend, nothing carries over to the ARM gem5 backend.

It would be preferable to have a backend that could be used for multiple architectures, with only minor architecturally dependent parts.

3.4 Requirements for the Design

In this chapter, we synthesize requirements for the design from the previous observations.

1. To simplify fault tolerance measure evaluation the new backend for FAIL* *must* be able to run campaigns in user-mode.
2. It *must* be possible to trace all memory accesses and instructions.
3. It *must* be possible to modify target memory and registers.
4. Breakpoints *must* be supported at specific program counters and after a specific number of instructions are executed.
5. FAIL* *must* be able to run experiments significantly faster than it does currently with the Bochs backend.
6. The new backend *should* be more flexible with regards to the target architecture so that it is more easily portable to new architectures.

3.5 Summary

In this chapter, we analyzed the problems, especially no support for user-mode FI, potentially suboptimal execution speeds, and the difficulty of porting a backend to different architectures. Additionally, we derived the requirements, which will serve as the basis for the design established in the next chapter.

4 Design

In this chapter, we first choose a valid new target backend for FAIL*. We evaluate the two candidates Valgrind and QEMU, in their suitability for the requirements given in the previous chapter, and then we explain why we chose one over the other. After choosing a candidate, aspects of the design of the new backend are described.

4.1 Comparison between Valgrind and QEMU

We now explain the choice of QEMU over Valgrind with regards to the requirements outlined in the Problem Analysis chapter.

The first four requirements of the ability to run in user-mode, memory access tracing, memory modification, and breakpoints are satisfied by both.

The requirement of speed seems to be better satisfied by QEMU. QEMU is considerably faster than VALGRIND, in our experiments, about 2x faster for a bubble sort implementation. A comparison from Bellard between QEMU and Valgrind says QEMU is 1.2x faster [5], but this is from 2005, and QEMU has seen substantially more development effort since then.

The last requirement is that porting to new architectures must be more easily possible. This is in parts satisfied by Valgrind as it supports many architectures and is explicitly designed to implement other dynamic binary analysis tools in contrast to QEMU. The problem is that Valgrind does not support system emulation. QEMU supports even more architectures and supports system emulation too, so this requirement is satisfied better by QEMU.

In total, the support for system emulation and the faster speed outweigh the ease of integration of Valgrind, and accordingly, QEMU is chosen as the new target backend.

4.2 Architecture Overview

With the foundations of QEMU established already in section 2.5 we now provide a bird's eye view of the architecture of the new FAIL* backend. It shows the changes we did to FAIL*, which were needed to integrate the new backend. We largely oriented this on the architecture from different backends and the already present rudimentary QEMU implementation. An overview of the architecture can be seen in figure 4.1.

From a conceptual standpoint, the experiment instruments the parts of the EEA to do all things necessary for the execution of the experiment. This aspect is unchanged from the normal FAIL*. In order to conduct the fault injections, several features need to be present, which we establish now.

The main component of the EEA to control the execution of the experiment is implemented in the `QEMUController` class. At the start, QEMU initializes FAIL* through the `QEMUController`. The `QEMUController` is responsible for calling functions from the backend such as saving and restoring snapshots, adding breakpoints, or adding timers. Additionally, in the user-mode, the `QEMUController` ensures that there is no difference between the stack starting address in the golden run and the later experiments. This could happen if FAIL* were invoked with different environment variables for the experiments and the golden run.

The `QEMUMemoryManager` is responsible for all memory accesses. It interacts with the backend to read from or write to the memory of the target program which QEMU runs; this is used for the actual injection of bit flips.

An additionally needed function is it to read or write registers of the virtual CPU. This is implemented through the architecture-dependent `QEMUarchCPU`, for example, the `QEMUX86CPU`. This encapsulates details about the specific architectures; for example, the program counter can be queried independently of the underlying architecture, as a program counter is present in most architectures. In the case of x86, the EIP register and in the case of ARM, the PC register would be returned. Reads to architecture-specific registers, however, are not designed in an architecture-independent way.

Many experiments capture the serial output to get results. This is, however, not possible in user-mode since GNU/Linux user-space programs typically do not use the serial output. Most programs instead use the standard output (stdout) and standard error (stderr) to emit their results. Therefore a new plugin for FAIL* is conceptualized, which has the capability of capturing the stdout. The plugin is called `StdoutLogger` and is designed in a similar way to the `SerialOutputLogger`. It uses a new listener, the `StdoutListener` which is triggered by system calls which print to stdout or stderr.

Timers are implemented in the backend; they callback to the EEA when they expire. Additionally, it is necessary to provide a way to trace memory accesses, which is implemented through the QEMU plugin system. This will be discussed in the next section in more detail.

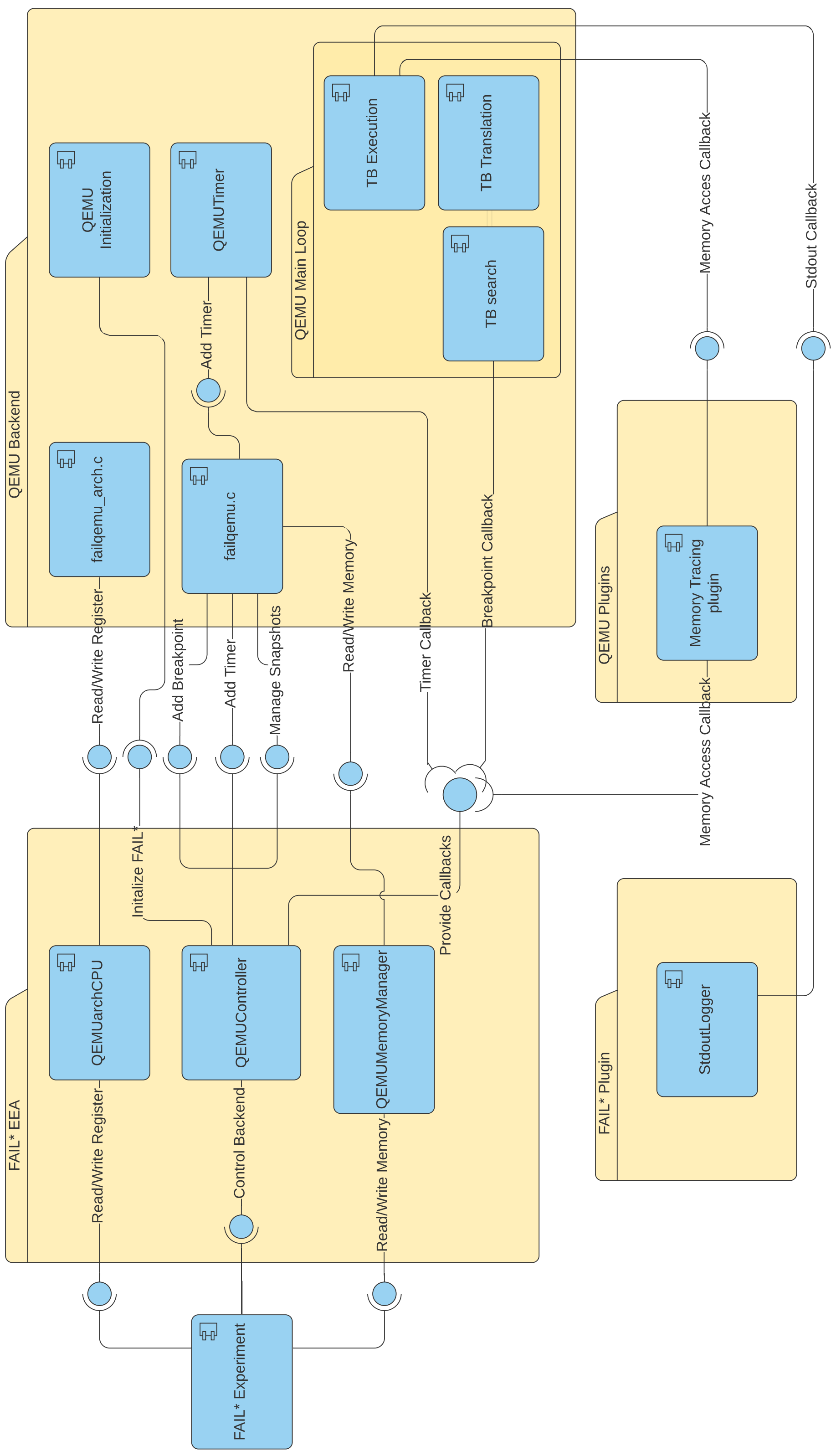


Figure 4.1: Component diagram of FAIL* together with the new QEMU backend. The StdoutLogger is replaced by the SerialOutputLogger for full system emulation.

4.3 Memory Access Tracing and Watchpoints

This section explains how memory accesses are traced with the new QEMU backend and additionally how watchpoints are derived from this.

Since version 4.2, QEMU provides a plugin system for the TCG. Plugins can passively monitor every instruction and memory access. To do this, they subscribe to events, which happen during the execution or translation, for example, events such as memory accesses. The plugin is then notified if an event is encountered. Plugins are not able to change any aspect of the system state; instead, they are limited to be mere observers. Plugins are shared libraries that are dynamically linked to QEMU at runtime.

For FAIL*, a new plugin was implemented for memory access tracing. It registers callbacks at the initialization at the beginning, which are called upon memory accesses.

Watchpoints are special breakpoints, which can stop execution whenever memory at a specific address is read or written to. Watchpoints can be derived rather easily with the already present memory access tracing. As every memory access leads to a callback, it is checked whether a watchpoint is set for this memory address, and if it is, the watchpoint is triggered.

4.4 Instruction Counting

In this section, we discuss the concrete design for target instruction counting and breakpoints. Target instruction counting is important to implement breakpoints, which trigger after a specific amount of instructions passed. For the remainder of the thesis, every time we speak of instruction counting, target instruction counting is actually meant.

The first method we propose to use for instruction counting is to disable the chaining of TBs and then add up instruction counts of each TB before they are executed. This seems to have the advantage of being easy to implement, but this has a few tedious corner cases which complicate the implementation. For example, it can happen that after a TB started executing, it first checks whether an exception is pending and aborts immediately if it is, so the instructions, which we want to count, are not executed at all. In this case, the counter needs to be decreased again, but this can lead to other issues that must be handled too. This leads to increased complexity and a more error-prone implementation. Also, the disabling of TB chaining comes with a hefty performance hit. In our observations, this can lead to an order of magnitude slower execution speed.

The second method we designed avoids counting instructions before TBs are executed; instead, the TBs themselves are augmented so that they count instructions executed while running. This ensures that the instruction count is only increased if the instructions are actually executed, in contrast to the first method. Additionally this method does not depend on entering `tb_find()` for every executed TB, therefore TB chaining can stay enabled.

We ultimately chose the second method first and foremost because the performance hit from disabling TB chaining is to big to be tolerated.[1]

4.5 Breakpoints

This section considers two variants for implementing breakpoints, which trigger after a specific amount of instructions are passed. In the following, we call these breakpoints *dynamic breakpoints* to distinguish them from breakpoints that trigger a specific program counter. After this, we discuss two methods for implementing normal breakpoints, which trigger at a specific program counter.

The first method for implementing dynamic breakpoints we describe is achieved through altering an already present QEMU feature.

QEMU has a feature called “icount”, which guarantees that only a specific number of instructions are executed every 2^N nanoseconds. It utilizes essentially dynamic breakpoints for this, but these are only supported in the system mode since they work only in conjunction with the QEMUTimers, which are not present in the user-mode. To do this, every virtual CPU in QEMU has a budget that saves the number of instructions the CPU can execute until the budget is increased again after some amount of time has passed. So the first thing each TB does is check whether the CPU still has the budget remaining to execute all instructions in the TB. If it has enough and no interrupt is pending, the budget is decreased by the number of instructions in the TB. If it has not, the TB does not resume execution but instead returns to the CPU main loop.

This “icount” feature is not really useful for FAIL*, as it does not support user-mode, so it is completely taken over for a different purpose. The budget now gets a different meaning; it now means the number of instructions until the next dynamic breakpoint. If the execution now returns to the main execution loop without an interrupt pending, we know that a dynamic breakpoint resides inside the current TB. This TB is then split[2] to ensure that the instruction, which triggers the breakpoint, begins a new TB and can be caught before it is executed.

The other method for dynamic breakpoints is very similar to the second method for instruction counting. For this, TB chaining is disabled, and before every TB is executed, it is checked whether a dynamic breakpoint is present inside this TB.

Regarding normal breakpoints, the first possible method we considered were the breakpoints QEMU provides, which are used with GDB, for example. The downside is that only 4 breakpoints at a time are supported.

[1]This is evaluated and shown in section 6.4

[2]This splitting is explained further in section 4.7.

The second approach for implementing breakpoints we considered works in a similar way to the first method for instruction counting. Before the execution of a TB, it is checked whether a breakpoint needs to be activated or whether splitting is needed.

Additionally, setting a breakpoint requires invalidating a potentially previously generated TB because the TB could already be chained to another TB. This would lead to missing the TB since the breakpoints are only checked for before the execution runs and not while it runs.

We chose the second approach because this method is not limited in the number of breakpoints.

4.6 Timers

For injecting a fault or aborting an experiment, as it takes too long, stopping the execution after a certain time is necessary. For this, timers are a needed element which we design in this section.

As detailed in section 2.5.4 QEMU provides the QEMUTimers to callback to a given function. For the full system emulation, this suffices as the way to implement a timer. More specifically, a timer against the virtual clock is chosen as the best option, as it is undesirable to have the clock running while no target instructions are being executed.

For the user-mode, this solution is not feasible as the QEMUTimers do not exist here. The solution we designed was creating a new thread that sleeps for the time specified and then issues a callback. This has the disadvantage of running even while the execution is not running, but for the benchmarks at hand, execution time always dominated the overall runtime anyway, so this is only a minor drawback. A possible remedy would be to measure, for example, the time needed for translating new TBs and increasing the time slept accordingly, but we decided against this as the difference is very small.

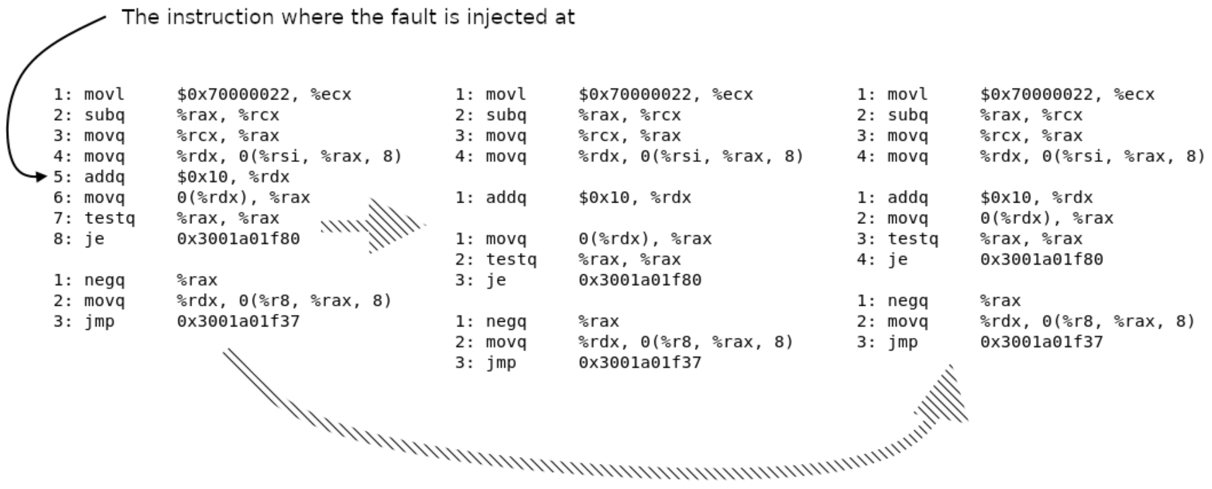
4.7 Varying Sizes for Translation Blocks - or TB Splitting

The nature of QEMU to only execute TBs with no way to interfere while a TB is executed poses a problem for fault injection. If a fault is to be injected at an instruction in the middle or at the end of a TB, this is not immediately possible. Only if an instruction is the first instruction in a TB it is possible to inject a fault right at this instruction.

One obvious solution to this problem is to use the single-stepping mode of QEMU that only allows one instruction per TB. Unfortunately, this incurs a big overhead as bigger TBs are better for performance, and single-stepping is ultimately unnecessary for the vast majority of instructions, which are not affected by fault injections.

In order to solve this, the master's thesis [8] of Bhat adapted the translation so that only the instruction that will be executed directly after the fault injection is limited to be the sole instruction in a TB. This improves the overhead by a lot, but it is possible to do even a little better.

There is no need at all to limit the TB, in which the instruction in question resides, to single-stepping; it is sufficient to guarantee that the instruction is the first instruction of a TB. This is enough to inject faults at any point in the program. A visualization of this splitting can be seen in figure 4.2.



```
1: movl    $0x70000022, %ecx
2: subq    %rax, %rcx
3: movq    %rcx, %rax
4: movq    %rdx, 0(%rsi, %rax, 8)
5: addq    $0x10, %rdx
6: movq    0(%rdx), %rax
7: testq   %rax, %rax
8: je      0x3001a01f80

1: negq    %rax
2: movq    %rdx, 0(%r8, %rax, 8)
3: jmp     0x3001a01f37
```

```
1: movl    $0x70000022, %ecx
2: subq    %rax, %rcx
3: movq    %rcx, %rax
4: movq    %rdx, 0(%rsi, %rax, 8)

1: addq    $0x10, %rdx

1: movq    0(%rdx), %rax
2: testq   %rax, %rax
3: je      0x3001a01f80

1: negq    %rax
2: movq    %rdx, 0(%r8, %rax, 8)
3: jmp     0x3001a01f37
```

```
1: movl    $0x70000022, %ecx
2: subq    %rax, %rcx
3: movq    %rcx, %rax
4: movq    %rdx, 0(%rsi, %rax, 8)

1: addq    $0x10, %rdx
2: movq    0(%rdx), %rax
3: testq   %rax, %rax
4: je      0x3001a01f80

1: negq    %rax
2: movq    %rdx, 0(%r8, %rax, 8)
3: jmp     0x3001a01f37
```

Figure 4.2: Translation block splitting. The left column is without any additional TB splitting at all. The middle column shows the the approach from Bhat which splits the affected instruction into its own TB and the right column shows the improved design.

4.8 Architecture-Dependent Design Aspects

We kept the implementation architecture-independent in the parts where this was possible. This section delves into the design decisions we made for choosing either an architecture-independent approach for a specific component or an architecture-dependent one.

Reading and writing from the Registers of a virtual CPU needed to be conducted in an architecture-dependent way, as the registers differ between architectures.

The second architecture-dependent design element was the recording of traps. These are handled differently for the architecture, and as such, the implementation also needed to do this.

We did design instruction counting in a completely architecture-independent way, as we did change the intermediate representation code of the TCG to achieve this.

Watchpoints, breakpoints, and dynamic breakpoints were also designed to work in an architecture-independent way. As these features did not depend on architecture-specific aspects, this was straightforward to achieve, as a program counter is present for all architectures that QEMU supports. It is a part of its internal architecture-independent CPU state.

Additionally, snapshots and timers are also implemented independently of the architecture. QEMU handles the architectural details of snapshots in the background and does not expose any differences between architectures to carry them out.

Memory access tracing works architecture-independent, too, as the QEMU plugin system does not even provide a way to handle architecture differences.

The translation blocks also exist for every architecture, so there is no reason to introduce architecture-dependent parts for the TB splitting.

Writes or Reads to the memory also pose no problem, as there is a mechanism internally to QEMU to deal with issues that arise here, for example, issues with the host system being a little-endian system and the target architecture being big-endian.

4.9 Summary

To summarize, we did establish the design of the new backend in this chapter. First, we chose QEMU as the new backend, first and foremost to its support for system-mode emulation in contrast to Valgrind. Then we gave an overview of the new backend architecture and explained how we intend to implement memory access tracing and watchpoints. After this, we discussed different approaches for introducing instruction counting and chose to augment TBs in order to achieve it. Additionally, we described the design decisions taken for breakpoints and timers. At last, we discussed an optimization to TB splitting and listed aspects of the design, which could be conducted in an architecture-dependent and -independent manner.

Now with the design introduced, we can describe the implementation in more detail in the next chapter.

5 Implementation

This chapter describes the implementation of the new backend for FAIL*.

5.1 Differences between User-Mode and System Emulation

We describe the implementation differences between user and system-mode emulation in this section. Challenges are described, and countermeasures are taken.

5.1.1 Program Behaviour Differences

We observed differences in the behavior of programs themselves, which may be obvious but are nonetheless not present in system emulation. We depict these differences now.

First, the stack begins at a different address depending on the environment in which the user-space binary is started. At the guest stack's initialization, QEMU puts all environment variables in a memory region directly above the stack. This means if there is an environment with many environment variables set, the stack will begin at a different address, whereas typically, addresses in system emulation do not change depending on the emulator's environment.

Another point that changes the start address of the stack is the length of the program's path, which the program was executed with. This is passed in C programs typically as the first element on the argv array, given as a parameter to the main function. For example, a path "`./a.out`" will have another stack starting address than the path "`/some/random/very/long/path/a.out`". Another difference is that a program can query things about the way in which it is run. For example, some programs test whether their standard output is sent to a terminal or something else, as a pipe or a file.

The chosen solution was to implement a detection for differences in the stack start address between an experiment and the golden run and abort if differences are detected. For this, the stack starting address is saved for the golden run in a file with the filename extension `.state`. The experiment then loads the file and compares the stack starting address, aborting, or continuing accordingly.

Address Space Layout Randomization (ASLR) is a security technique which is useful for preventing the exploitation of memory corruption vulnerabilities [40].

It is designed to randomize addresses between program executions and add a *stackgap*, so that memory addresses are not predictable. This is not tolerable if the memory addresses shall be in any way consistent between the golden run and the later experiments.

To achieve execution in user-mode in a way that the behavior and stack starting address does not differ between the tracing in the golden run and experiments, we found these measures to be effective:

1. To keep the environment variables the same, run both the golden run and the experiments with the command `env -ignore-environment`.
2. Keep the arguments of the program the same and especially the length of the program path.
3. Execute both with the output redirected into a file or both without it.
4. Disable ASLR. For example with the command `echo 0 | sudo tee /proc/sys/kernel/randomize_va_space`.

5.1.2 Implementation Differences

Some Aspects of user-mode and system emulation are handled differently in QEMU. We list some passages to show this.

QEMU is organized in many ways through C macros; the general way in which the QEMU source code differentiates between user-mode code and system mode-code is via the macro `CONFIG_USER_ONLY`. If it is defined, the user-mode is being compiled, and if it is not, the system-mode. Similarly, the macro `TARGET_X86_64` indicates the parts of code which are to be compiled for x86-64 targets; similar macros exist for every target architecture. The newly added functions to QEMU also adhere to this principle. As an example, the differing translation between guest (or target) to host addresses can be seen in listing 5.1. This additionally illustrates some of the complexity incurred by having a memory management unit or not.

Another difference between user-mode and system-mode are snapshots. As snapshots are unsupported in the user-mode, the loading of snapshots is hijacked to instead implement the detection for an invalid stack start address.

```
 1 uint64_t failqemu_guest_to_host(CPUState *cpu, uint64_t virt_addr) {
 2 #ifndef CONFIG_USER_ONLY
 3     // 6 Lines of variable declarations omitted
 4 
 5     cpu_synchronize_state(cpu);
 6 
 7     page_virt_addr = virt_addr & TARGET_PAGE_MASK;
 8     page_phys_addr = cpu_get_phys_page_attrs_debug(cpu,
 9         page_virt_addr, &attrs);
10     /* if no physical page mapped, return an error */
11     if (page_phys_addr == -1)
12     return -1;
13 
14     //21 Lines omitted
15 
16     ram_ptr = ramblock_ptr(block, addr1);
17     return (uint64_t) ram_ptr;
18 #else
19     return (uint64_t) g2h(virt_addr);
20 #endif //CONFIG_USER_ONLY
21 }
```

Figure 5.1: This listing shows the translation from guest to host addresses. The parts from line 3 to 17 are for the system-mode, and only line 19 is for the user-mode, as split by the `#ifndef` directive. As can be seen, the amount of code needed for translating memory addresses in the user-mode is much simplified compared to the system mode. The system emulation first calculates a page virtual address, then a physical address as well. In the omitted part, further components such as an address space and a RAM block are involved.

5.2 Memory Tracing Plugin

We aim to explain the implementation of the memory tracing plugin for QEMU in this section.

To use the TCG plugins QEMU needs to be compiled for them. This can be done by the configure option `configure -enable-plugins`.

To compile a plugin it needs to be compiled as a shared library which exports a `qemu_plugin_version` symbol. QEMU TCG plugins are designed with API stability in mind. The QEMU developers try to keep breaking changes in the api behind version number changes and thus we need to declare which version of the API we intend to use with the `qemu_plugin_version` symbol.

Upon starting QEMU the memory tracing plugin is loaded and the `qemu_plugin_install` function is called. This registers callbacks for plugin events, in this case for memory accesses. The plugin can choose to listen to either read accesses, write accesses or both. The callbacks are registered for every instruction as the translation of a TB finishes. If such an instruction is then executed and does a memory access the callback is called. The plugin then notifies FAIL* of the memory access. Listing 5.2 shows the plugin.

```c
QEMU_PLUGIN_EXPORT int qemu_plugin_version = QEMU_PLUGIN_VERSION;
static enum qemu_plugin_mem_rw rw = QEMU_PLUGIN_MEM_RW;

//Callback function for memory access
static void vcpu_mem(unsigned int cpu_index,
                     qemu_plugin_meminfo_t meminfo,
                     uint64_t vaddr, void *udata) {
    fail_watchpoint_hit(cpu_index, meminfo, vaddr, udata);
}

//Callback function for translation of TBs
static void vcpu_tb_trans(qemu_plugin_id_t id,
                          struct qemu_plugin_tb *tb) {
    //register memory access callback for every instruction
    size_t n = qemu_plugin_tb_n_insns(tb);
    for (size_t i = 0; i < n; i++) {
        struct qemu_plugin_insn *insn;
    insn = qemu_plugin_tb_get_insn(tb, i);
    qemu_plugin_register_vcpu_mem_cb(insn, vcpu_mem,
                              QEMU_PLUGIN_CB_NO_REGS,
                              rw, NULL);
    }
}

//Initialization function
QEMU_PLUGIN_EXPORT int qemu_plugin_install(qemu_plugin_id_t id,
                                           const qemu_info_t *info,
                                           int argc, char **argv) {
    //omitted initialization of read or write tracing from params

    //register callback for TB translation
    qemu_plugin_register_vcpu_tb_trans_cb(id, vcpu_tb_trans);
    return 0;
}
```

Figure 5.2: This listing shows the important parts of the memory tracing plugin.

5.3 Instruction Counting and Dynamic Breakpoints

This section looks at the implementation of instruction counting and dynamic breakpoints in detail.

In order to achieve instruction counting for TBs, which only counts if the TB is run and not aborted at the start, we added a few additional instructions to the intermediate representation code, which is generated for each TB. These instructions load the instruction count from host memory and increase it by the number of target instructions present in the TB.

The dynamic breakpoints are implemented in a somewhat more complicated way than described before, in order to avoid unnecessary branches. As already mentioned in section 4.5 the QEMU feature "ìcount" introduces a budget for every virtual CPU that saves the number of instructions the CPU can execute in a given amount of time. We chose this budget to implement dynamic breakpoints; the budget now means instructions left until the next dynamic breakpoint. The remaining instruction budget resides in the lower 16 bits of a 32 bit signed integer value. The upper 16 bits of the value are supposed to indicate pending interrupts. If the whole value is less than zero after the TB returns to the main execution loop, it indicates an interrupt. If it is still greater than or equal to zero, it means a return was done because of an insufficient instruction budget for the virtual CPU.

This limits dynamic breakpoints to $2^{15}-1$ instructions in the future. To address this, the size of the signed integer value was increased to 64 bits, such that there is a 32 bit signed integer for setting a dynamic breakpoint. This means a dynamic breakpoint can be set up to $2^{31}-1$ instructions into the future.

The adaptation of TBs for instruction counting and dynamic breakpoints can be seen in listing 5.3.

```c
static inline void gen_tb_start(CPUState *cpu)
{
    tcg_ctx->exitreq_label = gen_new_label();
    TCGv_i32 tb_icount_i32 = tcg_temp_new_i32();
    /* We emit a movi with a dummy immediate argument.
     * Keep the insn index of the movi so that we later
     * (when we know the actual insn count) can update
     * the immediate argument with the actual insn count. */
    tcg_gen_movi_i32(tb_icount_i32, 0xdeadbeef);
    icount_start_insn = tcg_last_op();

    TCGv_i64 new_icount = tcg_temp_new_i64();
    TCGv_i64 tb_icount_i64 = tcg_temp_new_i64();
    TCGv_ptr ptr = tcg_const_ptr(&(cpu->dynamic_instruction_counter));

    // Add the insn count of this TB to the overall counter.
    tcg_gen_ld_i64(new_icount, ptr, 0);
    tcg_gen_extu_i32_i64(tb_icount_i64, tb_icount_i32);
    tcg_gen_add_i64(new_icount, new_icount, tb_icount_i64);
    tcg_gen_st_i64(new_icount, ptr, 0);


    // Load the remaining instruction budget for next breakpoint.
    TCGv_i64 count = tcg_temp_local_new_i64();
    tcg_gen_ld_i64(count, cpu_env,
                   offsetof(ArchCPU, neg.icount_decr.b64)
                   - offsetof(ArchCPU, env));

    tcg_gen_sub_i64(count, count, tb_icount_i64);

    /* Check whether a dynamic instruction breakpoint is hit
     * or an interrupt is pending. */
    tcg_gen_brcondi_i64(TCG_COND_LT, count, 0, tcg_ctx->exitreq_label);

    // Save new instructions left until dynamic instruction breakpoint.
    tcg_gen_st32_i64(count, cpu_env,
                     offsetof(ArchCPU, neg.icount_decr.b32.low)
                     - offsetof(ArchCPU, env));
```

Figure 5.3: This listing shows the added instructions to the QEMU intermediate representation code. First, a `movi` instruction with a placeholder value is introduced. This placeholder denounces the number of instructions in the TB. As this number is unknown at the start of generating a new TB, the placeholder is needed, which is replaced at the end of TB generating. Then the global instruction count is increased, and it is checked for breakpoint triggers and pending interrupts.

5.4 Usage Details

In order to use the new implementation some information about the usage is necessary which we will provide here. First we explain some aspects about the compilation process and then explain how to conduct an examplary golden run and a FI campaign.

In order to compile FAIL* with the QEMU backend the CMake-based configuration needs to be changed. The variables `BUILD_QEMU` and `BUILD_X86_64` need to be enabled in order to build the system-mode backend for the x86_64 architecture. For the user-mode the variable `BUILD_QEMU_USER` needs to be activated instead of `BUILD_QEMU`. Additionally the correct FAIL* plugins need to be activated, either the stdout plugin for the user-mode or the serialoutput plugin for system-mode.

First the golden run needs to be conducted. For this the fail-client compiled for tracing needs to be run with the FAIL* command line parameters given in table 5.1 and the qemu specific parameters listed in table 5.2. The tables are to be read in such a way that all fitting parameters are to be used. For example, all parameters for the user-mode except those which are only for the experiments.

For conducting the FI campaign typically many experiments need to be run. An experiment can be run in much the same way as described for the golden run except the parameters need to be chosen from the tables for the experiment and not the golden run.

As many experiments need to be conducted, we opted to use GNU parrallel [48] to parallelize the execution of the experiments. "GNU Parallel is a command-line tool for Unix-like operating systems which allows the user to execute shell commands in parallel" [48].

Mode	Parameter	Description
System	`state-file=X`	Specifies the name `-X` of the snapshot to be saved inside the `qcow2` image.
User	`state-file=X`	Specifies the file `-X` in which to save the stack starting address.
Both	`start-symbol=X`	Specifies the symbol `-X` at which to start tracing.
Both	`end-symbol=X`	Specifies the symbol `-X` at which to stop tracing.
System	`state-dir=X`	Specifies the name `-X` of the snapshot to be loaded from the `qcow2` image.
User	`state-dir=X`	Specifies the file `-X` from which to load the stack starting address.
Both	`ok-marker=X`	Specifies a symbol `-X` at which to stop the experiment, with the indication, that no error was encountered.
Both	`detected-marker=X`	Specifies a symbol `-X` at which to stop the experiment, with the indication, that an error was detected.
Both	`trap`	Specifies to abort on traps encountered.
Both	`timeout==X`	Specifies a time `-X` in microseconds after which the experiment is aborted.
Both	`serial-file=X`	Specifies the file `-X` in which to save (or load) the serial output or standard output.
Both	`trace-file=X`	Specifies the trace file `-X`.
Both	`elf-file=X`	Specifies the elf file `-X`.

Table 5.1: Command line parameters for FAIL* with the QEMU backend. All options need to be preceded with `-Wf,--`. The yellow highlighted rows (the upper four rows) symbolize golden run specific parameters, while blue highlighted symbolize experiment specific parameters. No color means the parameter applies to both.

Mode	Parameter	Description
System	`-m XM`	Specifies the amount of memory in `-X` Megabytes.
System	`-serial stdio`	Instruments QEMU to redirect the serial output to the standard output. Alternatively `-nographic` can be used if only a terminal is used without a screen.
System	`-drive file=X.qcow2`	Specifies the name X.qcow2 of the `qcow2` system image.
Both	`-plugin X.so,arg=rw`	Specifies the file `-X.so` to be loaded as a TCG plugin. `arg=rw` passes the `rw` as an argument to the plugin.
User	last parameters	Specifies the name of the target program to execute and its parameters.

Table 5.2: Command line parameters for FAIL* with the QEMU backend which are passed to QEMU.

5.5 Summary

We explained some details about the implementation of QEMU as the new backend of FAIL*. With this, differences between user-mode emulation and system emulation were shown, and the arising challenges, such as ASLR were listed and countermeasures presented. After this, elements of the new implementation were presented, most notably the implementation of dynamic breakpoints. We highlighted the changes done to the TB to count instructions in listing 5.3. Additionally, we described how the QEMU plugin system was leveraged for tracing memory accesses. Following this, we explained the changes to the build process, and clarified which command line parameters need to be passed in order to conduct the golden run and the experiments.

At last, every important implementation detail is established, and so the evaluation in the next chapter can begin.

6 Evaluation

We quantitatively evaluate the new backend of FAIL* in this chapter with regards to runtimes and outcomes of FI campaigns.

First, we evaluate a bubble sort implementation with several increasing elements to sort. Then we evaluate a more diverse set of benchmarks and examine asymptotic execution speeds. After this, we discuss some aspects of the boundaries of determinism and then evaluate user-mode FI and compare it to system-mode FI. At last, 64 bit ARM support is shortly discussed.

6.1 Small System Software Benchmark

This section compares the Bochs and QEMU backend on a small system software benchmark. First, two fault space plots are shown to compare the results of fault injections across the two backends. Second, the results' qualities are compared against each other by comparing the raw values for SDCs, timeouts, and traps. After that, the runtimes of the fault injection campaigns are compared, and an average experiment runtime is inspected in detail.

A fault space plot visualizes the different memory areas in a system and what consequences arise from injecting bit flips at specific locations. The fault space plots resulted from a system software implementation of a bubble sort with 24 elements with either Bochs or QEMU as a backend where the timeout was set to 500 ms. The timeouts behave differently for Bochs and QEMU; while Bochs uses a guest time, which depends on how fast instructions are executed on a particular machine, QEMU does use the host clock for this. The whole fault space was covered, which was possible with 2990 experiments. From the start of the main function to the call of the function `FAIL_FINISHED`, which marks the end of the program's execution, there were 4144 instructions. The faults injected always flipped every bit of one whole byte. There is one difference between the images used for QEMU and Bochs; the Bochs image uses the port 0xe9 for serial output, whereas the QEMU image uses the standard port 0x3f8. However, this should not alter the benchmark results, as the instructions themselves stay the same.

In direct comparison, the QEMU fault space plot and the Bochs fault space plot, which can be seen in figure 6.1, look quite similar. The data to sort resides in a global array called `input_data`, which can be seen in the upper part of the diagram from the symbol `input_data` to the top of the plot. If anything of the data to be sorted is changed, the program ends up with SDC with both backends. If the length of the array saved in the variable `input_data_length` is changed, the bubble sort takes longer and sorts even some random parts of the memory. This happens in both cases, and depending on how high the length is after the injection, this either times out or leads to SDC.

The only visible difference in the fault space plots is at RAM address $2.101 \times 10^6 + 175$, where a fault injection leads to a timeout in the case of Bochs and an SDC in the case of QEMU. In this case, QEMU continued pretty much unimpacted and finished in the usual time it takes for a run, while Bochs takes a far longer time and prints a lot of garbage to the serial output. This indicates a similar quality for the QEMU backend as the Bochs backend, as there is only this one difference, which actually stems from a single experiment.

Backend	Number of occurrences			Runtime in s	
	SDC	Timeout	Trap	Trace	Experiments
QEMU	141 433	30 348	10 627	1.61	40.05
BOCHS	137 189	34 328	10 632	3.75	48.01

Table 6.1: Benchmark values with both the QEMU and Bochs backend.

As is apparent in table 6.1 the differences in the number of traps encountered are very similar with both backends, and the difference in timeout results stems entirely from the single experiment at RAM address $2.101 \times 10^6 + 175$. Therefore the results here are not much different.

Other aspects of comparison are the runtimes of the different fault injection campaigns. We measured both campaign runtimes' on an Intel® Core™ i7-9750H CPU, with turned off Intel® Turbo-Boost Technology and the frequency fixed @ 2.60GHz. This CPU has 6 cores and 12 threads, and the benchmarks were done with 12 threads for both backends.

QEMU was 20% faster in regards to the experiments, but QEMU exhibited a significant speedup of 232% for tracing. The time for a complete campaign did not differ much from the time needed for the experiments, so the tracing speedup is rather insignificant for the overall time needed for a campaign in this benchmark.

A typical runtime of a single experiment is now further discussed and can be seen in figure 6.2.

With the Bochs backend, loading a snapshot took on average a time of 138 ms, while with the QEMU backend, loading a snapshot took only 25 ms on average. This is the sole reason why QEMU is faster than Bochs for this benchmark. The average execution times for the 4144 instructions were only on the order of less than 1 ms for both backends and are therefore insignificant for the runtimes at large. The obvious question now immediately becomes where QEMU loses its advantage and why it only comes out ahead 20% over Bochs.

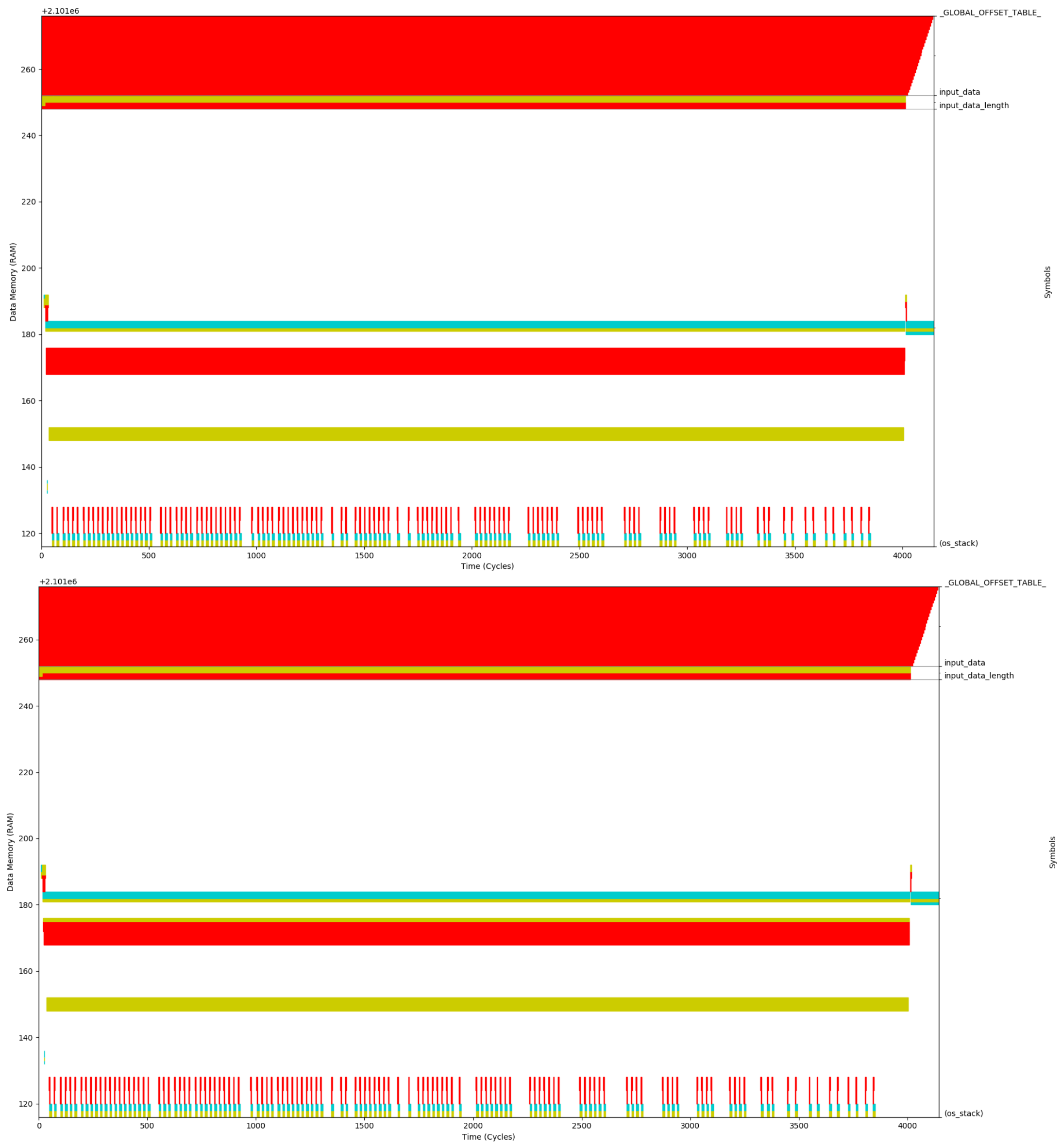


Figure 6.1: Fault-space plot for an x86 FI campaign with QEMU and Bochs as backends. The yellow color depicts timeouts, turquoise depicts traps, white depicts no effect, and red depicts SDC. Note the completely red area from the memory address 168 to 175 in the upper plot which can be seen as a timeout in the lower plot.

The big reason for this is that FAIL* with the QEMU backend can currently only handle one experiment for one execution of a fail-client process. This leads

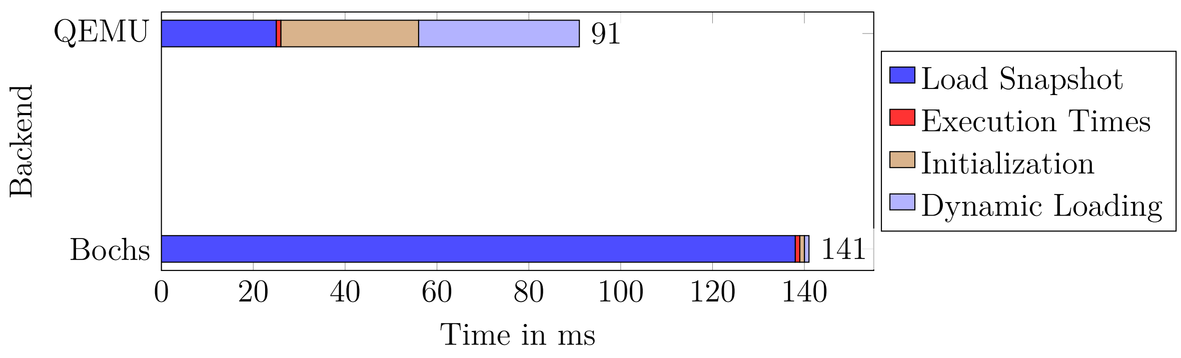


Figure 6.2: Stacked bar plot, which compares different components of overall time needed, for a very small experiment.

to another significant amount of time incurred, which must be calculated as well for QEMU, which is the initialization and dynamic loading of shared libraries. Initialization takes an additional 30 ms in QEMUs case, while this takes less than 1 ms in Bochs' case. Dynamic loading of shared libraries delays program execution on average by an additional 35 ms.

If QEMU as a backend had the capability to run multiple experiments per FAIL* process, significant speedups could be possible, at least for small benchmarks, where loading the snapshot takes the longest amount of time.

6.2 Large System Software Benchmark

The next section compares the Bochs and QEMU backend for FAIL* on a larger system software benchmark. As there is such a large fault space, full fault space coverage is not feasible, and instead, 10 000 samples are taken for each backend.

The benchmark is very similar to the one in the previous section, as this is again a bubble sort implementation with the only difference being 2000 elements instead of 24. As the runtime of bubble sort is $\mathcal{O}(n^2)$, differences in execution times are large. The benchmark runs for 28 883 867 instructions.

Backend	Number of occurrences		
	SDC	Timeout	Trap
QEMU	58 065 013 271	123 180 652	93 851 926
BOCHS	58 053 279 793	158 375 119	58 657 452

Table 6.2: Benchmark values with both the QEMU and Bochs backend.

As can be observed in table 6.2, the SDC counts with the QEMU and Bochs backend vary very little. This can be attributed to sampling inaccuracy (standard deviation 207 064 285). Bochs had 28% more timeouts, which might be explained by a higher execution speed from QEMU. The timeouts seem to have been largely turned into traps as the number of traps encountered was nearly as much higher with the QEMU backend as the number of timeouts was less.

Backend	Runtime in s			
	Trace	Importing	Pruning	Experiments
QEMU	92.5	254.7	13.3	450
QEMU + memtrace plugin	92.5	254.7	13.3	3297
BOCHS	81.0	281.8	13.6	3218

Table 6.3: Benchmark runtimes with both the QEMU and Bochs backend. The times for taking the golden run were called "Tracing". "Importing" means to import the trace into a database, "Pruning" reduces the number of experiments needed, and "Experiments" refers to the time needed for the fail-clients to complete all generated experiments.

The first observation that can be made from the benchmark runtimes depicted in table 6.3 is the amount of time spent for importing the traces. The tracing runtime seems almost irrelevant in contrast to the time spent for importing the traces generated by the tracing steps. This problem only intensifies if an even bigger benchmark is chosen, i.e., the time spent importing grows faster than the time spent tracing, irrespective of the backend. Regardless of this, Bochs comes out ahead 12% faster in tracing than QEMU.

Bochs has a tiny edge in experiment runtimes above QEMU with the memory tracing plugin enabled. However, QEMU without the memory tracing plugin is $7.32x$ faster than Bochs, which is a significant reduction in the time needed for the experiments.

The plugin capabilities of tracing memory accesses are only needed in the tracing part of the campaign. The experiments benchmarked did not need to trace memory accesses; therefore, the memory tracing plugin is unnecessary to use and should be avoided.

The explanation for such a large performance hit is the implementation of the memory tracing. Each memory access leads to a callback from optimized TCG code. Every callback needs to call an epilogue first, which saves the execution state, and on reentry, the saved state must be loaded again in the prologue. This amounts to a substantial overhead, as can be observed in the runtimes.

6.3 eCos Benchmarks

This section evaluates the QEMU versus the Bochs backend in a diverse range of algorithms implemented on top of the *e*mbedded *C*onfigurable *o*perating *s*ystem (eCos) [35].

Interrupts were turned off before entry to the main function. Otherwise, the current QEMU backend can not run the experiments in a deterministic manner. The benchmarks were conducted on different hardware than the other benchmarks in this chapter. It was run on four Intel® Xeon® CPU E5-4640 @ 2.50GHz. All in all, there were 32 cores and 64 threads. All values are median values from three runs each. Timeouts were chosen as 3 × the instructions Bochs can execute per second × the respective instruction count for every benchmark.

1. "basicmath" calculates a few basic mathematical calculations like cubic functions, integer square roots, and exclusive ors. It was the largest of the benchmark runs, which shows in the times needed for the tracing. It runs for 71 176 202 instructions.

2. "bitcount" tests different bit manipulation functions. It runs for 4 397 163 instructions.

3. "rijndael" is an implementation of the Rijndael encryption algorithm (also known as AES) [15]. It runs for 1 058 537 instructions.

4. "qsort" is an implementation of the quick sort algorithm [26]. It runs for 4 943 909 instructions.

5. "bfs" is an implementation of breadth-first search from "Introduction to Algorithms" [13]. It runs for 4 994 492 instructions.

6. "swaptions" computes various option prices.[1] It runs for 5 049 367 instructions.

The results of the benchmarks are shown in table 6.4. Overall the results are rather consistent across all benchmarks except for the "basicmath" benchmark. SDCs are very similar between the QEMU and Bochs benchmarks in all of the benchmarks. Traps were noticeably more common with the QEMU backend, but accordingly, there were fewer timeouts with QEMU, again probably because the execution speed is higher.

The "basicmath" benchmark ran slower than the other benchmarks, and therefore it was not sufficient to use the same way for choosing a timeout value as with the other benchmarks, as the time would not be enough for even a single

[1]Options as in financial instruments.

experiment with a fault injection at the beginning. So the timeout was increased by a factor 3 for QEMU in this benchmark. The low execution speed can likely be explained by the high amount of instructions that write to serial output. These instructions lead to a callback and leave the optimized TCG code, which slows down the QEMU backend considerably. The experiments are overall still faster than in the Bochs case, but the difference is not that big as in the other cases.

Backend	Number of occurrences			Runtime in s	
	SDC	Timeout	Trap	Trace	Experiments
basicmath					
Bochs	28 759 836 469	1 257 390 252	8 837 318 280	201.8	1737.0
QEMU	29 785 914 734	1 607 324 351	7 977 310 896	242.4	986.3
bitcount					
Bochs	1 991 219 736	225 745 968	497 725 221	18.1	120.4
QEMU	2 029 482 367	91 191 195	600 714 027	20.4	21.5
rijndael					
Bochs	18 954 792 530	28 320 928	89 008 631	7.9	137.3
QEMU	18 914 354 076	18 206 330	93 054 576	7.7	49.9
qsort					
Bochs	134 782 059 388	355 428 054	692 149 368	19.9	198.3
QEMU	134 464 077 624	299 307 908	804 390 002	21.3	52.3
bfs					
Bochs	655 669 937 904	20 269 922 371	413 671 885	35.0	229.3
QEMU	652 636 645 836	22 614 073 509	551 562 769	25.4	62.6
swaptions					
Bochs	2 885 191 666	658 588 808	584 139 639	18.7	215.3
QEMU	2 939 024 374	539 470 179	628 809 190	21.0	48.6

Table 6.4: Benchmark values with QEMU system- and QEMU user-mode as a backend.

6.4 Asymptotic Speedup in a Best Case

This section compares the Bochs and QEMU backend for use with very high instruction breakpoints. Once again, an implementation of bubble sort was used as the benchmark, the only difference being a much longer array to sort. These are nearly ideal conditions for dynamic binary translation, as the same parts of the code are executed over and over again. It is, therefore, likely to show QEMU in a favorable light. This is intended to estimate the upper bound of the speedups to be realized with the current implementation of the QEMU backend versus the Bochs backend.

The breakpoint was set 182 536 109 796 instructions in the future and was reached after 403 s by the QEMU backend. That corresponds to 456.3 millions of instructions per second (MIPS).

The Bochs backend does not currently support breakpoints more than $2^{32} - 1$ instructions in the future. It is nevertheless possible to extrapolate from a smaller breakpoint. For this, a breakpoint was set at instruction $2^{32} - 1$ for Bochs and was reached after 223 s. That corresponds to 19.3 MIPS, which is 23x slower than QEMU. We provide a visualization of the different backend speeds in figure 6.3.

The QEMU user-mode backend was even faster; it only needed 182 s to reach the breakpoint, which corresponds to 1002.9 MIPS. The QEMU user-mode backend for 64 bit ARM needed 437 s and thus ran at 416.8 MIPS.

As the user-mode binary can be run natively as well, this was done also. The execution of approximately 412 316 859 272 $\pm$ 1% instructions took 60 s. This can then be calculated as 6871.9 MIPS. An unmodified QEMU ran the same 412 316 859 272 $\pm$ 1% instructions in 310 s, which means 1330.1 MIPS.

The performance impact, for this program at least, of the unmodified QEMU can be determined to be the factor $5.1\overline{6}$. FAIL* together with the QEMU user-mode backend then incurs an additional slowdown of factor 1.32, or 6.85 versus the native execution speed.

Additionally, this setup allows us to evaluate the impact of the design decision discussed in section 4.4, which allowed the chaining of TBs. Hence this test was repeated with TB chaining disabled. In user-mode, the execution took 3399 s which leads to 53.7 MIPS, a slowdown of a factor of approximately 18.7. Thus it can be inferred that the design decision in question contributes to a higher execution speed.

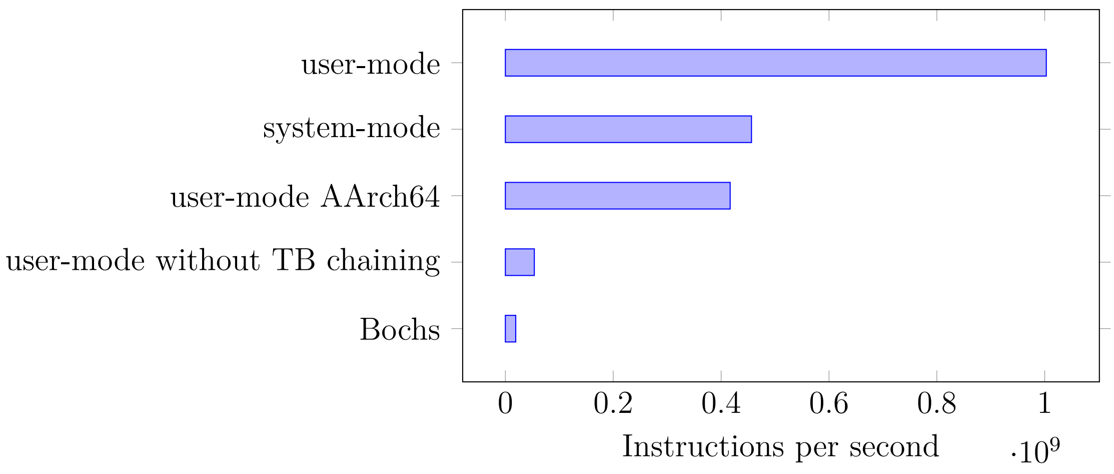


Figure 6.3: Bar chart which compares execution speeds under optimal conditions. All values except for Bochs are values with different versions of the QEMU backend.

6.5 Boundaries of Determinism

This section shows instances in which the current implementation of QEMU as a backend for FAIL* is behaving nondeterministically. Nondeterministic means that some external factor which the current backend can not cope with leads to a different sequence of instructions in the experiment than protocolled in the tracing step, even before injecting. This can happen due to a multitude of factors, which are examined in this section.

The most obvious introduction of nondeterminism in a program is the time. If a program reads the Real-Time Clock (RTC) at specific points in the program and behaves differently depending on the time, then the current backend is unable to provide the time which was present at the time of tracing. Instead, the current RTC is shown.

Interrupts are a problem too, they can happen at unpredictable times, and if an interrupt happens at a different time than in tracing, the program will likely not behave the same. In our observations, the interrupts happened at different instructions in multiple experiments. We hypothesize that those were timer interrupts, although this aspect needs more research.

In user-mode emulation, the results are very much dependent on the environment the program finds itself in. For example, an obvious case would be a program that opens a file. This program behaves differently depending on whether the file exists or not.

Also, the environment must be kept the same between experiments and tracing to ensure determinism, as is mentioned in more detail in chapter 5, section 2.5.4.

6.6 User-Mode Fault Injection

User-space programs have a multitude of advantages over system software, for example, a simpler build process and easier debugging. Is a developer, therefore, better advised to use user-mode, or do the drawbacks outweigh the benefits after all? This section attempts to answer this question.

The program again implements bubble sort with 24 elements similarly as in section 6.1, only slightly modified in order to be usable as a GNU/Linux program and in 64 bit. The program is statically linked, compiled with optimization, and without stack protection[2], and executed without Address Space Layout Randomization. Timeouts were set at 50 ms.

We tried to minimize differences between the full-system and user-space variants of the bubble-sort program, but there are nonetheless some. One is the usage of the GNU C Library (glibc), which was not used in the system software case. The biggest difference between the two is input/output (I/O). In the user-space program, I/O is not done via a serial port; instead, the glibc wrapper function for the write system call was used, as an equivalent to the x86 assembly instruction `OUT`. This leads to a somewhat longer trace, as the write function has a greater overhead than one instruction. This lead to a runtime of 3887 instructions, approximately 7% more than the 3642 in the full system implementation. As in the full system implementation, the whole fault space was covered, but this needed 15% more experiments.

Now the execution speeds, SDCs, timeouts, and trap counts are compared for user-mode and system mode in the upper half of table 6.5.

First, it is apparent that there are about 37% more SDCs in the full system-mode, in contrast to the user-mode. This might be due to the higher instruction count. The trap-result count was about 23% in the user-mode. This hints at a tendency to fail more often in user-mode, which might be explained by the differences between exceptions and signals or stricter memory protection. Timeouts were less pronounced than in the full system case. This is to be expected since the execution speeds of the user-mode are higher.

Execution speeds were rather different in comparison. Especially the tracing is drastically faster in user-mode, 30.4x faster for this benchmark. This stems from a much lower overhead of user-mode binaries. While in the system software, over 60 million instructions were executed to even reach the main function, in the user-mode, only approximately 5000 are needed to reach the main function. The experiments also took less time. This was mostly the fact due to not needing to restore snapshots, which takes 30 ms each in the system software case. In user-mode, it takes only about one ms to reach the main function.

[2]With the compiler options: `-m64 -Wall -g -fno-stack-protector -O -fno-threadsafe-statics -fno-use-cxa-atexit -fno-rtti -fno-exceptions -static -z max-page-size=0x1000 -Wl,-gc-sections`.

Backend	Number of occurrences				Runtime in s	
	SDC	Timeout	Trap	Detected	Trace	Experiments
No detection						
system	85 983	9910	28 711	0	1.52	19.35
user	62 583	6257	35 590	0	0.05	12.13
With detection						
system	1647	10 306	28 829	103 934	1.53	19.30
user	1647	6781	36 340	111 768	0.05	13.01

Table 6.5: Benchmark values with QEMU system- and QEMU user-mode as a backend in a bubble sort with 24 elements.

To estimate if the user-mode is similarly effective in gauging the effect of fault tolerance mechanisms, a simple fault tolerance mechanism is employed in both cases. A sum of the elements is calculated at the beginning of the program and then again at the end. The results of this can be seen in the latter half of table 6.5. The number of experiments and instructions was barely impacted by this, less than 3%.

In the system software case, the mechanism reduced the amount of SDCs down to 1647, down to exactly the same amount as in the user-mode example. This is likely the case due to program parts which are responsible for the introduction of SDCs, being identical.

An additional observation made was that it is important to program in a careful way with regards to the output in user-space programs. If, instead of the write function, the putchar function from glibc is used, the amount of SDC observed is as high as 47 415, which is a drastic difference. This difference can be explained by the big overhead and unnecessary copying of data that occurs in this case but is a hindrance for implementing and testing fault tolerance mechanisms.

The observed results make a compelling argument for the use of the user-mode. The user-mode proved to be sufficient for evaluating the fault tolerance mechanism in question and is likely suited for the evaluation of other such mechanisms as well. The reduced execution time is an added bonus that is helpful in achieving fast iterations in an iterative software developing process.

6.7 64 bit ARM Support

This section presents a simple test of the functionality of the 64 bit ARM implementation. As the ARM implementation was only a side goal, it is the least refined part of the implementation.

The test was once again a user-mode implementation of the bubble sort algorithm, but this time it was cross-compiled for the 64 bit ARM architecture AArch64. It was compiled with optimizations, and the main function consisted of 3542 ticks. This resulted in 918 experiments to be conducted.

This resulted in 194 028 SDCs, 39 timeouts, and 63 269 traps. The tracing took 67 ms, and the experiments took 5.23 s.

The results seem plausible; only the timeouts are a little rare. This would need further investigation in the future.

6.8 Summary

We showed in this chapter that across every benchmark done the QEMU backend was faster compared to the Bochs backend. The speedups ranged from $2x$ to $23x$ from the worst to the best case. Very similar amounts of SDCs were encountered with both backends. Because of the faster execution speed, QEMU did time out less frequently than Bochs. Traps, however, increased in occurrences across the two backends, likely they were converted from timeouts.

The user-mode showed promise for the evaluation of fault tolerance mechanisms. No significant drawbacks over the system emulation were encountered. Additionally, it could be shown that the ARM 64-bit backend is at least functioning.

After this chapter we will now discuss some related works to this thesis.

7 Related Work

This chapter gives an overview of related works which did implement fault injection tools. We give particular emphasis to the works which used QEMU as the base. First, we discuss interface-based approaches and then modifications built on QEMU.

7.1 Interface-Based Approaches

In this section we focus on approaches, that instrument QEMU from the outside through an interface such as the gdb interface.

David and Campbell, together with Chan and Carlyle, developed a method for injecting faults with QEMU, without modifying QEMU itself, called QInject [17]. QInject injects fault using the gdb debugger interface built into QEMU. There are multiple limitations to this approach, however. The gdb interface only allows for basic functionality like setting breakpoints and lacks more advanced capabilities, such as instruction counting.

Chyłek also proposed a fault injection tool built on QEMU, which is called QEMU Fault Injector or QEFI [11]. It uses the monitor and gdb interface of QEMU to instrument fault injections from a supervisor framework. There are multiple different ways QEFI implements to determine the time at which to inject a fault. At random points, at program counter breakpoints, and custom instructions which the target program can invoke to trigger a fault injection itself. Additionally, device-triggered injections are supported, which inject, for example, when interactions with USB devices are observed. There were no benchmarks with regards to execution speeds conducted, but the author warns that “its speed may be decreased”[12].

7.2 QEMU Adaptations

In this section we describe approaches, which built on QEMU itself to inject faults.

David and Campbell examined a self-healing operating system, which can recover from faults and exceptions through various techniques such as code reloading, micro-rebooting or watchdog timer based recovery [16]. To test their operating system, they injected faults with a modified QEMU. The fault injection tool supports ARM targets, is capable of injecting faults into memory and registers, and can raise CPU exceptions. The implementation is relatively small and can be found online [37]. The authors did not provide benchmarks for their implementation, but it is likely that it is with a high overhead because the injector code runs at every executed instruction according to the source.

Di Guglielmo et al. built a fault injection tool based on dynamic binary translation and in particular on QEMU [18]. It leverages the dynamic binary translation aspect of QEMU to inject faults into CPU registers, flags, and instructions at translation time. This was done with the user-mode part of QEMU that is capable of running user-space programs on top of the GNU/Linux operating system. There are, however, limitations of their approach as fault injection into memory is not supported, and only breakpoints at specific program counters are used for determining the point of injection. Additionally, the implemented caching by QEMU of the translated code was disabled, which does not bode well for performance.

Ferraretto and Pravadelli expanded on this and presented an efficient fault injection approach based on QEMU with a focus on emulator performance [20]. They could prevent disabling caching but did need to enable single-stepping, which is also detrimental to QEMUs performance. Injection into memory is also not possible with this approach.

FIES is a fault injection tool built on QEMU by Höller et al. It is designed specifically with software-based self-tests in mind. It supports fault injection into CPU registers and memory. Timer and memory access based triggers for the fault injection are supported. Additionally, program counter breakpoints are supported. However, a shortcoming is that it is not possible to inject a fault at a specific instruction.

A notable optimization in contrast to the work of Ferraretto and Pravadelli was implemented by Bhat in his masters' thesis [8]. Here all instructions executed are counted in a golden run, and a fault can be injected at a specific instruction in the experiment later. In order to avoid single-stepping all instructions, only the precise instruction at which the fault injection should happen is run in a single step. To do this, the corresponding translation block is isolated to a single instruction. This guarantees the possibility to inject at the right instruction. As the implementation is proprietary and no benchmarks are provided, it is impossible to estimate the execution speed accurately, but it is likely on a high level.

7.3 Summary

Multiple approaches for FI with QEMU were brought up; from gdb or monitor based approaches to more advanced approaches which modify QEMU from the ground up. None of the listed approaches, except for the gdb-based approaches, have the capability of conducting FI campaigns in user-mode as well as in full system emulation. However, the gdb-based approaches are limited in their control of the emulator. We overcame this shortcoming in this thesis; it is now possible to conduct such campaigns for both cases. Additionally, we did extensive benchmarks, which were also lacking in the previous works.

After the discussion of the related work, we now get to the closing chapter.

8 Summary and Conclusions

In this chapter we summarize the thesis, and then give an outlook and propose further work.

8.1 Summary

The first and foremost goal of the thesis was to implement a usable user-mode for FAIL*. To achieve this, Valgrind and QEMU were evaluated, and QEMU was chosen for its higher speed and additional support for system emulation.

The implementation of the new backend was done in a largely architecturally independent way, which should help with porting to new architectures. To achieve dynamic instruction counting and dynamic breakpoints with the new QEMU backend, the architecturally independent intermediate representation from QEMU was modified.

The newly implemented backend was benchmarked in a variety of ways. It showed that the new backend could achieve a $23x$ faster execution speed in a best-case scenario if compared to the Bochs backend. The user-mode showed to be approximately twice as fast in this best-case benchmark.

In the general case, large speedups could be realized as well, albeit not to the same degree; around $2x$ to $6x$ speedups were observed for practical benchmarks. In addition, the amount of SDCs encountered was mostly very similar between the two backends, while timeouts were less pronounced with QEMU, likely due to higher execution speeds. Traps were often more pronounced with the QEMU backend, but not in every case.

Additionally, it is noteworthy that evaluating a fault tolerance mechanism with the newly implemented user-mode looked promising. The observed amounts of SDCs before and after applying a fault tolerance mechanism were similar to the system case. Additionally, speeds with the user-mode, especially of the tracing step, were higher across the board.

8.2 Outlook

Something to be considered for future work could be determinism. In the current implementation, interrupts needed to be turned off for some benchmarks, in particular, those using the eCos operating system. QEMU offers a way to introduce more deterministic behavior with the “icount” option. This records, for instance, interrupts. At a later time, it can playback this recording to keep the interrupts exactly as in the first run. The impact on execution time is likely heavy as a big overhead is introduced by this. For example, to achieve a more deterministic time behavior, the execution is limited to a specific amount of instructions executed per second.

An additional option for the future would be to use hardware virtualization with KVM. This would make another big speedup possible. Possibly near-native speeds could be achieved. The drawbacks of this, however, would be large. First, no dynamic translation of target code between architectures would be possible as the TCG is doing this work. This would limit the backend to native images. Secondly, the user-mode would not be supported, and instructions could not be counted. Normal hardware breakpoints would have to suffice.

One more open problem is the handling of parallelism. QEMU can use multiple virtual CPUs, so this could possibly be integrated into FAIL* as well. This will probably worsen the problems with determinism, though, so this aspect would need to be closely watched.

Bibliography

[1] J. Aidemark et al. “GOOFI: generic object-oriented fault injection tool”. In: *2001 International Conference on Dependable Systems and Networks*. 2001, pp. 83–88. DOI: 10.1109/DSN.2001.941394.

[2] J. Arlat et al. “Fault injection for dependability validation: a methodology and some applications”. In: *IEEE Transactions on Software Engineering* 16.2 (1990), pp. 166–182. DOI: 10.1109/32.44380.

[3] J. Arlat, J.-C. Fabre, and M. Rodríguez. “Dependability of COTS microkernel-based systems”. In: *IEEE Transactions on Computers* 51.2 (2002), pp. 138–163. DOI: 10.1109/12.980005.

[4] J. H. Barton et al. “Fault injection experiments using FIAT”. In: *IEEE Transactions on Computers* 39.4 (1990), pp. 575–582. DOI: 10.1109/12.54853.

[5] F. Bellard. “QEMU, a fast and portable dynamic translator.” In: *USENIX Annual Technical Conference, FREENIX Track*. Vol. 41. 2005, p. 46.

[6] A. Benso and P. Prinetto. *Fault injection techniques and tools for embedded systems reliability evaluation*. Vol. 23. Springer Science & Business Media, 2003. ISBN: 9781402075896. DOI: 10.1007/b105828.

[7] L. Berrojo et al. “New techniques for speeding-up fault-injection campaigns”. In: *Proceedings 2002 Design, Automation and Test in Europe Conference and Exhibition*. 2002, pp. 847–852. DOI: 10.1109/DATE.2002.998398.

[8] S. N. Bhat. “Robustness Assessment of Linux by Simulation-Based Physical Fault Injection in System Calls”. MA thesis. Kaiserslautern, Germany: Technische Universität Kaiserslautern, 2018.

[9] A. Bligh. *Changes to QEMU’s timer system*. Aug. 2013 (accessed November 3, 2020). URL: http://blog.alex.org.uk/2013/08/24/changes-to-qemus-timer-system/.

[10] J. Carreira, H. Madeira, and J. G. Silva. “Xception: a technique for the experimental evaluation of dependability in modern computers”. In: *IEEE Transactions on Software Engineering* 24.2 (1998), pp. 125–136. DOI: 10.1109/32.666826.

[11] S. Chyłek. "Emulation based software reliability evaluation and optimization". In: *Przeglad Elektrotechniczny* 90 (Feb. 2014), pp. 121–124. DOI: 10.12915/pe.2014.02.32.

[12] S. Chyłek and M. Goliszewski. "QEMU-BASED FAULT INJECTION FRAMEWORK". In: *Studia Informatica* 33 (Jan. 2012), pp. 25–42.

[13] T. H. Cormen et al. *Introduction to algorithms.* MIT press, 2009. ISBN: 9780262533058.

[14] J. C. Cunha, M. Z. Rela, and J. G. Silva. "Can software implemented fault-injection be used on real-time systems?" In: *European Dependable Computing Conference.* Springer. Berlin, Heidelberg, 1999, pp. 209–226. DOI: 10.1007/3-540-48254-7_15.

[15] J. Daemen and V. Rijmen. *The design of Rijndael: AES — the Advanced Encryption Standard.* Springer-Verlag, 2002, p. 238. ISBN: 3540425802.

[16] F. M. David and R. H. Campbell. "Building a Self-Healing Operating System". In: *Third IEEE International Symposium on Dependable, Autonomic and Secure Computing (DASC 2007).* 2007, pp. 3–10. DOI: 10.1109/DASC.2007.22.

[17] F. M. David et al. *QInject : A Virtual Machine based Fault Injection Framework.* International Conference on Architectural Support for Programming Languages and Operating Systems (Poster Presentation). Mar. 2008.

[18] G. Di Guglielmo et al. "Efficient fault simulation through dynamic binary translation for dependability analysis of embedded software". In: *2013 18th IEEE European Test Symposium (ETS).* 2013, pp. 1–6. DOI: 10.1109/ETS.2013.6569351.

[19] A. Dixit and A. Wood. "The impact of new technology on soft error rates". In: *2011 International Reliability Physics Symposium.* Apr. 2011, 5B.4.1–5B.4.7. DOI: 10.1109/IRPS.2011.5784522.

[20] D. Ferraretto and G. Pravadelli. "Efficient fault injection in QEMU". In: *2015 16th Latin-American Test Symposium, LATS 2015* (May 2015). DOI: 10.1109/LATW.2015.7102401.

[21] A. Fidalgo et al. "Using NEXUS Compliant Debuggers for Real Time Fault Injection on Microprocessors". In: *Proceedings of the 19th Annual Symposium on Integrated Circuits and Systems Design.* SBCCI '06. Ouro Preto, MG, Brazil: Association for Computing Machinery, 2006, pp. 214–219. ISBN: 1595934790. DOI: 10.1145/1150343.1150397.

[22] D. Francis. "Qinject: A virtual machine based fault injection framework". In: *International Conference on Architectural Support for Programming Languages and Operating Systems, 2008.* 2008.

[23] E. Fuchs. “An evaluation of the error detection mechanisms in MARS using software-implemented fault injection”. In: *Dependable Computing — EDCC-2*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 73–90. ISBN: 9783540706779.

[24] Q. Guan et al. “F-SEFI: A Fine-Grained Soft Error Fault Injection Tool for Profiling Application Vulnerability”. In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. 2014, pp. 1245–1254. DOI: 10.1109/IPDPS.2014.128.

[25] U. Gunneflo, J. Karlsson, and J. Torin. “Evaluation of error detection schemes using fault injection by heavy-ion radiation”. In: *1989 The Nineteenth International Symposium on Fault-Tolerant Computing. Digest of Papers*. Los Alamitos, CA, USA: IEEE Computer Society, June 1989, pp. 340, 341, 342, 343, 344, 345, 346, 347. DOI: 10.1109/FTCS.1989.105590.

[26] C. A. R. Hoare. “Quicksort”. In: *The Computer Journal* 5.1 (Jan. 1962), pp. 10–16. ISSN: 0010-4620. DOI: 10.1093/comjnl/5.1.10.

[27] A. Höller et al. “A Virtual Fault Injection Framework for Reliability-Aware Software Development”. In: *2015 IEEE International Conference on Dependable Systems and Networks Workshops*. 2015, pp. 69–74. DOI: 10.1109/DSN-W.2015.16.

[28] “International Roadmap for Devices and Systems, 2020 update More Moore”. In: (2020). URL: https://irds.ieee.org.

[29] R. K. Iyer and D. J. Rossetti. “A Measurement-Based Model for Workload Dependence of CPU Errors”. In: *IEEE Transactions on Computers* C-35.6 (1986), pp. 511–519. DOI: 10.1109/TC.1986.5009428.

[30] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. “FERRARI: a flexible software-based fault and error injection system”. In: *IEEE Transactions on Computers* 44.2 (1995), pp. 248–260. DOI: 10.1109/12.364536.

[31] J. Karlsson et al. “Using heavy-ion radiation to validate fault-handling mechanisms”. In: *IEEE Micro* 14.1 (1994), pp. 8–23. DOI: 10.1109/40.259894.

[32] M. Kooli and G. Di Natale. “A survey on simulation-based fault injection tools for complex systems”. In: *2014 9th IEEE International Conference on Design Technology of Integrated Systems in Nanoscale Era (DTIS)*. 2014, pp. 1–6. DOI: 10.1109/DTIS.2014.6850649.

[33] K. P. Lawton. “Bochs: A Portable PC Emulator for Unix/X”. In: *Linux J.* 1996.29es (Sept. 1996), 7–es. ISSN: 1075-3583.

[34] H. Madeira et al. “RIFLE: A general purpose pin-level fault injector”. In: *Dependable Computing — EDCC-1*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1994, pp. 197–216. ISBN: 9783540487852.

[35] A. Massa. *Embedded Software Development with eCos*. Prentice Hall Professional Technical Reference, 2002. ISBN: 9780130354730.

[36] G. Miremadi and J. Torin. "Evaluating processor-behavior and three error-detection mechanisms using physical fault-injection". In: *IEEE Transactions on Reliability* 44.3 (1995), pp. 441–454. DOI: `10.1109/24.406580`.

[37] *Modified QEMU sources with support for ARM fault injection*. `https://web.archive.org/web/20121022230012/http://choices.cs.uiuc.edu/download.html`. Accessed: 2020-11-21. 2007.

[38] S. Mukherjee. *Architecture Design for Soft Errors*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2008. ISBN: 9780080558325.

[39] N. Nethercote and J. Seward. "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation". In: *SIGPLAN Not.* 42.6 (June 2007), pp. 89–100. ISSN: 0362-1340. DOI: `10.1145/1273442.1250746`.

[40] T. D. Raadt. *Exploit Mitigation Techniques (updated to include random malloc and mmap)*. `http://www.openbsd.org/papers/ven05-deraadt/index.html`. OpenCon 2005 Accessed: 2020-11-29. 2005.

[41] F. Rahmanifard et al. "Galactic Cosmic Radiation in the Interplanetary Space Through a Modern Secular Minimum". In: *Space Weather* 18.9 (2020), e2019SW002428. DOI: `10.1029/2019SW002428`.

[42] M. Rodriguez, A. Albinet, and J. Arlat. "MAFALDA-RT: a tool for dependability assessment of real-time systems". In: *Proceedings International Conference on Dependable Systems and Networks*. 2002, pp. 267–272. DOI: `10.1109/DSN.2002.1028909`.

[43] H. Schirmeier et al. "FAIL*: An Open and Versatile Fault-Injection Framework for the Assessment of Software-Implemented Hardware Fault Tolerance". In: *2015 11th European Dependable Computing Conference (EDCC)*. 2015, pp. 245–255. DOI: `10.1109/EDCC.2015.28`.

[44] H. B. Schirmeier. "Efficient fault-injection-based assessment of software-implemented hardware fault tolerance". PhD thesis. Dortmund, Germany: Technische Universität Dortmund, 2016.

[45] J. Seward and N. Nethercote. "Using Valgrind to Detect Undefined Value Errors with Bit-Precision." In: *USENIX Annual Technical Conference, General Track*. Anaheim, California, Apr. 2005, pp. 17–30.

[46] V. Sieh, O. Tschache, and F. Balbach. "VERIFY: evaluation of reliability using VHDL-models with embedded fault descriptions". In: *Proceedings of IEEE 27th International Symposium on Fault Tolerant Computing*. 1997, pp. 32–36. DOI: `10.1109/FTCS.1997.614074`.

[47] D. Skarin, R. Barbosa, and J. Karlsson. “GOOFI-2: A tool for experimental dependability assessment”. In: *2010 IEEE/IFIP International Conference on Dependable Systems Networks (DSN)*. 2010, pp. 557–562. DOI: `10.1109/DSN.2010.5544265`.

[48] O. Tange. *GNU Parallel 2018*. Ole Tange, Apr. 2018. DOI: `10.5281/zenodo.1146014`.

[49] Valgrind^{TM} Developers. *Supported Platforms*. 2020 (accessed November 2, 2020). URL: `https://www.valgrind.org/info/platforms.html`.

[50] H. Ziade, R. A. Ayoubi, R. Velazco, et al. “A survey on fault injection techniques”. In: *Int. Arab J. Inf. Technol.* 1.2 (2004), pp. 171–186.

List of Figures