technische universität
dortmund

Master-Thesis

Emulator-based Fuzzing of Operating-system State-transition Graphs

Alwin Berger

11. Juli 2022

Betreuer:
Prof. Dr.-Ing. Peter Ulbrich (Gutachter)
Dr.-Ing. Alexander Lochmann (Gutachter)
Simon Schuster, M.Sc. (Friedrich-Alexander-Universität Erlangen-Nürnberg)
Dr.-Ing. Peter Wägemann (Friedrich-Alexander-Universität Erlangen-Nürnberg)

Technische Universität Dortmund
Fakultät für Informatik
System Software Group (LS-12)
https://sys.cs.tu-dortmund.de

Abstract

The *worst-case response time* (WCRT) of tasks in a real-time system is a crucial property of the system. Traditionally it is determined by analyzing the *worst-case execution times* (WCET) of all tasks and system functions and combining them in a very pessimistic way [1]. While recent static timing analysis methods have started incorporating the state of the system into the analysis to analyze the whole system at once, they still suffer from scalability issues and overestimation [1]. Apart from static timing analysis, there is *measurement-based timing analysis* (MBTA), which measures the *worst observed execution time* (WOET) [2]. Genetic algorithms are one technique MBTAs have successfully used to generate inputs with long execution times [3], [4]. This thesis leverages another genetic algorithm, coverage-based fuzzing, which is very successful in security focussed testing [5], [6]. It is a grey-box approach that leverages information about a target's control flow to drive a genetic algorithm to maximize code coverage. This thesis uses a fuzzer guided by the state transitions of a real-time system to generate inputs to the system which result in high execution times.

Contents

Chapter 1

Introduction

1.1 Motivation

The worst-case execution time of a program and the worst-case response time of a task in a real-time system are crucial to timeliness guarantees of real-time tasks [1]. State-of-the-art static timing analysis can be used to construct a relatively tight safe upper bound for a task's WCET [2], while the WCRT of a task in a *real-time operating system* (RTOS) that schedules multiple tasks is difficult to estimate [7]. A decoupled analysis of the task and RTOS code can lead to very pessimistic estimates for the tasks WCRT. Progress towards a global timing analysis has been made in recent years by integrating the flow between possible system states into the timing analysis, but pessimism is still an issue[1]. Since the worst-case input for a system is usually unknown it is also non-trivial to even quantify the pessimism. Apart from static timing analysis, there is also dynamic timing analysis, which executes a program and observes its execution time. While a dynamic timing analysis by itself only derives lower bounds for the WCET it does have the advantage of providing a witness input for its worst observed execution time. Those witnesses can be useful during the design phase of real-time systems [3] and allow estimating how pessimistic a result from a static timing analyzer is. For these reasons, this thesis explores a dynamic testing method that searches for the worst-case inputs to a real-time system with multiple interdependent tasks. The technique chosen as the base for this is coverage-guided fuzzing, a technique generating test cases based on a genetic algorithm guided by code coverage [5], [8]. Fuzzing is already a very popular dynamic analysis method in security research. For this thesis, it is adapted to work with coverage in the state transition graph of an RTOS, with additional state-aware optimizations added. To my knowledge, this thesis presents the first approach to apply the grey-box techniques of fuzzing to generating inputs which maximize the whole-system execution-/response-time for an RTOS with multiple tasks.

1.2 Research questions

This thesis aims to answer the following main questions: Are coverage-guided fuzzing techniques effective in discovering system inputs with extremely long execution times? Assuming a static analysis yields upper bounds on the number of executions for each edge between *basic blocks* (BB) in the code. Can these frequencies guide the fuzzer towards witness inputs that match these frequencies and therefore are worst-case inputs? Can the coverage guidance be improved by extending it to a *state transition graph* (STG) of the system, in analogy to state transition graphs used by static timing analyzers? Additionally, some more technical sub-problems need to be solved. For example: How could the system state be extracted during the execution of a system and in what way are they integrated into a state transition graph.

1.3 Structure

This thesis uses multiple established concepts such as fuzzing, RTOS and emulation. An overview of each is presented in Chapter 2. The existing related work around whole system timing analysis and fuzzing is examined in Chapter 3. Chapter 4 details the problem domain and puts the research questions and methods used in context. Chapters 5 and 6 focus on the technique developed for this thesis in abstract and implementation respectively. A test scenario for the technique and evaluation of the results can be found in Chapter 7. Chapter 8 summarizes the results and discusses future research directions for the concepts of this thesis.

Chapter 2

Background

This thesis combines multiple techniques into a novel approach for generating inputs close to the worst-case input of a real-time system. The most important technique is fuzz testing (fuzzing for short), which is used to generate test inputs for programs. Those test inputs are then passed to a real-time system on an emulated machine. During execution, the RTOS is traced and a system state graph is constructed. The following sections explain those basic techniques in more detail.

2.1 Fuzzing

Fuzzing is the process of testing programs for crashes, bugs and vulnerabilities using a large number of randomly generated inputs. A simple setup for fuzzing consists of a target program, an optional monitor to extract runtime information from the target, a generator for test cases and a detector for the bugs [8]. The way fuzzers extract and use information allows for classifying them into different classes, detailed in the next paragraph.

2.1.1 Classification of fuzzers

Fuzzers are classified as black-box, white-box or grey-box depending on their insight into the target. What follows is a short description of the capabilities of each, based on the literature review by Liang, Pei, Jia, *et al.* [8].

Black-box fuzzers operate without any knowledge of the target. Instead, they blindly apply rules, such as mutating a known valid input or generating semi-valid inputs. Such mutations can be simple byte or bit operations (e.g. copy, remove, flip bits). Generators can use grammars or other target-specific methods to generate inputs that are valid enough to pass the initial checks of the target program.

White-box fuzzers by contrast have full knowledge of the target's logic. They typically use dynamic symbolic execution (known as concolic execution) to explore new paths through a program systematically. They do this by recording the conditionals an execution has taken

and collecting the constraints of the conditionals using symbolic execution. By negating some conditionals and solving for their conjunction using a *satisfiability modulo theories* (SMT) solver, they generate inputs with different paths to maximize the desired coverage criteria (like statement-, condition- or even path-coverage) over the *control flow graph* (CFG).
Grey-box fuzzing is in a middle ground between both. It collects information such as code coverage at runtime and uses this information during mutation to achieve its goal faster. Code coverage can be measured and used by the fuzzer to guide the generation of new cases. A detailed example of a fuzzer using this technique can be found in the next section. Another grey-box technique is taint analysis, which traces the data flow of input fields and allows more directed mutations.

2.1.2 Influential example: AFL

A popular and influential example for a grey-box fuzzer is *American Fuzzy Lop* (AFL) [5]. It uses a simple genetic algorithm guided by edge coverage, which the authors summarize as the following steps [5]:

1. Load user-supplied initial test cases into the queue
2. Take the next input file from the queue
3. Attempt to trim the test case to the smallest size that doesn't alter the measured behavior of the program
4. If any of the generated mutations resulted in a new state transition recorded by the instrumentation, add mutated output as a new entry in the queue
5. Go to 2

It acquires information about the code coverage by including instrumentation in the target at compile time. Alternatively and more relevant to this thesis it allows using an instrumented version of the popular QEMU emulator to extract coverage information from an unmodified binary at runtime [9]. The collected information roughly equates to counters on the transition between basic blocks in a control flow graph.
The rest of the thesis refers to the components of a fuzzer in the terms used by the LibAFL [10], which is an AFL-inspired library for building modular fuzzers:

- Target: The program under testing
- Input: Some data used as input to the target, sometimes referred to as test case when combined with metadata about the resulting execution
- Executor: The environment the target code is run in, may reset the target after each run

- Observer: A component that extracts information about the target from the executor
- Feedback: Some function that uses the observers and an internal state to classify an input as interesting or not
- Metadata: Information collected about an input's execution
- Corpus: Collection of interesting inputs for further analysis, including some prioritization algorithm
- Mutator: Applies a mutation to an input, optionally uses observed information from the target
- Mutation stage: Collection of mutators with optional logic for choosing which ones to apply
- Objective: Determines if an input reached the fuzzing goal
- Solutions: Collection of inputs that reach the fuzzing goal

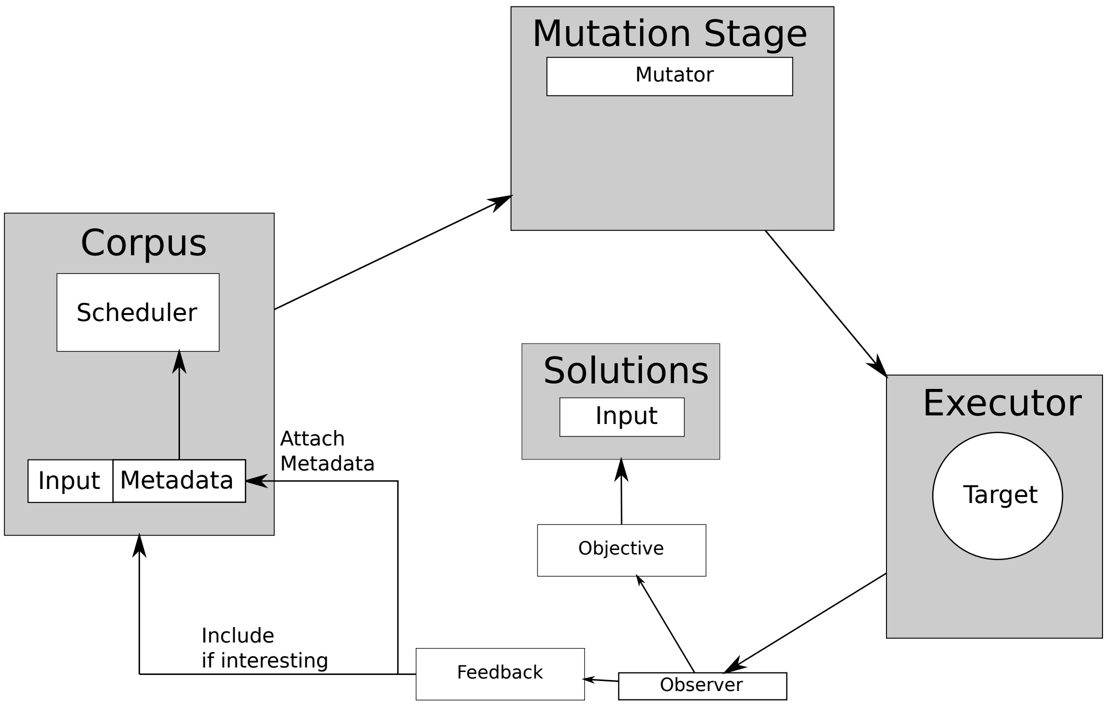


Figure 2.1: Abstract overview of LibAFL-based fuzzer. Arrows indicate information flow. Overall a test case is selected, mutated into a new input, executed, observed, rated and potentially added to the corpus or solutions.

Figure 2.1 shows how these components combine into a LibAFL-based fuzzer. A test case is selected from the corpus by some scheduling algorithm, the input is mutated using different mutators, injected in the target and executed. The results of the execution are observed, and the input is rated using feedbacks. It maybe considered interesting and added to the corpus, or it may even fit an objective and be added to the collection of solutions.

2.2 Emulation

Emulators are used for simulating a specific machine for software to run on. Which details of the target machine are simulated differ between emulators for different use-cases. QEMU is a popular general-purpose emulator (and hypervisor) with support for many CPU architectures and devices [9]. It is relatively performant but does not simulate hardware details like the memory hierarchy or processor pipeline. Other tools, such as PTLsim [11] allow accurate simulation of all aspects of CPUs (x86-64 in this case). While this would give very accurate execution time measurements, this thesis describes an approach using QEMU instead, as it integrates well with the fuzzing loop.
As mentioned in the previous section emulators may be used in fuzzers to instrument unmodified binary programs. This allows for avoiding probe effects from instrumentation inside the target code, which is relevant when targeting execution time. In case of AFL using QEMU this works by exploiting the way QEMU translates target instructions [5]. QEMU collects instructions into so-called *translation blocks* (TB), which consist of branch-free sequences of instructions [9]. These roughly correspond to basic blocks in the binaries control flow graph, which allows a fuzzer like AFL to easily trace CFG edges when TBs are executed. QEMU also allows counting the number of instructions executed, which serves as the measurement of execution time in this thesis.
QEMU supports emulating complete machines (referred to as softmmu or system-mode) or running the target application as a user space program while translating all system calls (referred to as user-mode). The work presented in this thesis modifies existing fuzzing instrumentation to use system-mode.

2.3 Real-Time Operating Systems

Real-time systems have to meet timing requirements, which means they need to complete their tasks within a deadline and respond to external events quickly [12]. The strictness of this requirement subdivides real-time systems into hard and soft ones. Hard ones need to meet the requirements in every instance, soft ones only need to meet them most of the time [12]. That is because a deadline miss in a hard real-time system can result in a catastrophic failure, while a deadline miss in a soft real-time system only degrades or nullifies the usefulness of the result [13].

A real-time operating system is specialized operating system for use in real-time systems. They usually host multiple tasks concurrently, which need to be scheduled in a way that ensures that all deadlines are met. The common recurrent task model assumes tasks are released periodically and need to meet deadlines relative to their time of release. Once tasks are released they can run, get preempted or block until they terminate, which needs to happen before their deadline [13]. When running for enough periods a system like this schedules the tasks in a repeating pattern. This is the hyperperiod, which is the least common multiple of the periods [14].
Scheduling under this scenario can be done using multiple algorithms, which are out of the scope of this overview. For simplicity, this thesis assumes the RTOS uses fixed-priority, preemptive scheduling, which means each task gets assigned a priority during the system design phase using some schedulability analysis. During runtime, tasks of lower priority can be preempted by higher priority ones [12].
Apart from their common focus on task scheduling, current RTOSes have varied features and use cases, which put them on a wide scale of complexity. On the complex side, a few systems provide features and APIs similar to general-purpose operating systems, such as process isolation using virtual address spaces and filesystems. [12]. There are even multiple efforts to enable real-time capabilities in Linux[1], mainly by allowing the kernel to be preempted [15]. On the other end of the spectrum are small kernels, typically running in the same address space with their tasks [12]. An example of this is FreeRTOS [16], which is a bare-bones kernel with a small API. It uses priority-based preemptive scheduling and features different synchronization mechanisms between tasks, such as semaphores, mutexes (with priority inheritance), queues and notifications. While it supports memory protection on some platforms it usually uses a single address space [12], [16].
The differences in design between single and multiple virtual address spaces influence how a task interacts with the OS. Tasks isolated by virtual address spaces in *general purpose operating systems* (GPOS) usually use system calls, which causes switching to a privileged execution mode with access to the whole memory space [15]. In single address space systems like FreeRTOS on the other hand kernel functions may be called directly without necessarily causing a mode change [16].

2.4 System States

CFGs capture the logic of a function. They are directed graphs consisting of blocks of linear code (basic blocks or BB for short) and are commonly used for static analysis of programs. Edges between two basic blocks express a possible flow of execution from one to the next [17]. The graph of a function has only one entry point. A whole program can be expressed by an *interprocedural control-flow graph* (ICFG), which contains all basic blocks

[1]https://kernel.org/

of its function's CFGs [18].
This concept was extended by Dietrich, Hoffmann, and Lohmann [18] into a *global control-flow graph* (GCFG) for systems with multiple tasks. The authors note that each scheduler might switch from a BB in one task to a BB in another task. This can be expressed in a GCFG, where two BBs are connected by an edge if and only if they may be executed directly after each other on real hardware. To construct the GCFG the authors partitioned BB into larger *atomic basic blocks* (ABB), which are regions in the CFGs with a single entry and exit BB.
The ABBs can be used to define a system state, which consists of the currently running ABB, the status of all tasks in the system and more information, such as the status of shared resources. These system states form a state transition graph, which contains all possible system states and transitions between them. The authors enumerated this graph using abstract interpretation.

2.5 Worst-case timing analysis

Hard real-time systems need to satisfy timing constraints. The Verification of the constraints requires safe upper bounds for the worst-case execution time of the running programs. The WCET of a program can be estimated using timing analysis, either static, measurement-based or a hybrid of both. Static timing analysis overestimates the WCET to derive safe upper bounds, while dynamic methods generally underestimate it and as such are not suitable for deriving safe bounds [2].

Static timing analysis

Static timing analysis usually follows a multi-stage process to analyze all possible control flows in combination with a hardware model to derive execution time bounds [2], [19], [20]. Figure 2.2 depicts how this process typically works [21]. It starts by constructing the control flow graph of a program. Next up, ranges of values for local variables and registers can be analyzed, which is optional but useful for the next step to determine loop bounds. This can be achieved using abstract interpretation. The next mandatory step is control flow analysis, which produces flow facts about the transitions in the CFG. Afterward, the processor behavior is analyzed, by using a model of the hardware to analyze possible states of the processor and caches to derive execution-time bounds per instruction. The bounds are based on possible execution histories leading to them. The final stage combines the flow facts with the timing bounds from the hardware model. The common technique for performing this calculation is called *implicit path enumeration technique* (IPET), first introduced by Li and Malik [22]. It combines the flow facts and time bounds into arithmetic constraints. These are used to maximize the sum of the product of execution count and time of basic blocks using *integer linear programming* (ILP)[2], [22]. The result is a safe upper bound for the WCET.

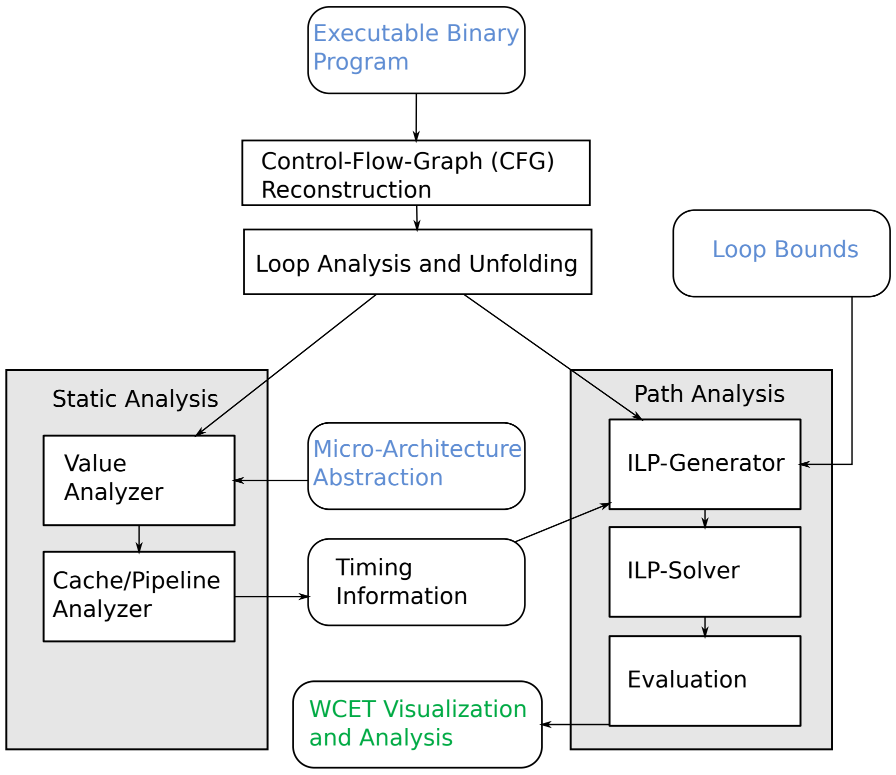

Executable Binary Program
Control-Flow-Graph (CFG) Reconstruction
Loop Analysis and Unfolding
Loop Bounds
Static Analysis
Value Analyzer
Cache/Pipeline Analyzer
Micro-Architecture Abstraction
Timing Information
WCET Visualization and Analysis
Path Analysis
ILP-Generator
ILP-Solver
Evaluation

Figure 2.2: The static analysis WCET pipeline. Inputs consist of the binary program, loop bounds and an abstraction of the target micro-architecture. Figure based on [21].

Measurement-based timing analysis

Measurement-based methods execute the program on hardware or accurate simulators to measure the execution time [2]. This is often used for end-to-end tests with a subset of all inputs. Such dynamic tests produce a worst observed execution time, which is generally not a safe bound for the WCET unless the worst-case input is in the set of test cases. Static and measurement-based methods can also be combined into hybrid methods, which produce tighter estimates compared to static analysis of the micro architecture, but remain safe bounds [2], [23].

Chapter 3

Related Work

The approach of this thesis leverages coverage-guided fuzzing to generate inputs that approximate the worst-case (in terms of execution time) for a real-time system with multiple concurrent tasks. While to my knowledge there is no previous literature on this combination of techniques and goal, it sits in an intersection between worst-case timing analysis and fuzz testing, two well-studied fields.
As referenced in Chapter 2 determining the WCET for single programs is well studied using static and measurement-based timing analysis. The static analysis pipeline as described in that chapter is common amongst many static timing analyzers, aiT for example, which is a commercial tool to analyze single applications from a binary level [21]. Some following paragraphs focus on extensions to the basic technique which either extend the analysis to a wider scope or extend it to derive tighter bounds.

Measurement-based timing analysis

Measurement-based timing analysis executes a subset of possible inputs and estimates the WCET from the worst observed execution time. It alone does not deliver safe upper bounds for the WCET. Generating tests that optimize for execution time can still produce close estimates and be useful during the design phase of a system, as pointed out by Mueller and Wegener [3] in 2001. The authors propose a genetic algorithm, which works by selecting the longest running test cases for a program, recombining their parameters and randomly applying mutations to some inputs. Khan and Bate [4] proposed an extended genetic algorithm, jointly optimizing some combinations of execution time, branch mispredictions, data cache misses and instruction cache misses. The authors note that they found no optimal set of optimization criteria for every case, while simply using a wide range of criteria leads to bad results as well.

Hybrid methods

One option to improve upon the standard static WCET estimation is to combine it with measurements on hardware or simulators into a hybrid method. One such example of a hybrid method is TimeWeaver, which combines static analysis for values and paths with measurements of short code snippets on real hardware, instead of pessimistic estimates from abstract hardware models [23].

System-state aware analysis

WCET analysis is mostly limited to uninterrupted executions of tasks, while the worst-case response time of real-time tasks in a larger real-time system is often more relevant and complex[1], [2]. The WCRT includes overheads from the real-time operating system and interferences by other tasks. While this can be analyzed by combining the WCETs of multiple tasks with the kernel overhead, a separate analysis of the application and kernel can lead to very pessimistic results [1]. One method to address this pessimism was developed by Dietrich, Wägemann, Ulbrich, *et al.* [1], which incorporates the state transition graph (as explained in Chapter 2) into an IPET definition. It achieves this by using flow facts about the system's state transition graph. It can even include the effects of interrupts, as long as they are bounded by a minimum inter-arrival time. As a downside of the STG focus its analysis considers paths that become infeasible through the thread-local control flow [1]. It is also based on the assumptions that not only all objects (tasks, resources and *interrupt service routines* (ISR)) are known in advance, but also the call locations and arguments to all functions interacting with them. These assumptions limit the approach to applications where this information can be determined statically.
A later work by [20] uses annotation of the OS on a source level to aid the static analysis in analyzing more generic RTOSes, where necessary information can not be derived from the global control flow alone. The approach uses the annotations during the path-analysis step of the common static timing analysis pipeline. It consists of multiple layers itself. First uses system facts, which include general knowledge about the whole system, such as the priorities of the tasks. These get propagated into the source annotations, which are used to address three main challenges [24]: Annotate indirect function calls, use state-dependent annotation to eliminate infeasible paths based on the system state and lastly, use application knowledge and system facts for bound computations. The last layer is the WCET analysis, which constructs the STG, derives additional system facts from it, and uses the annotation layer to derive control-flow facts describing a given system state. From there on out an IPET-based analysis is carried out. Another later work by the same authors introduces a more expressive annotation language, that features custom expressions for better context-sensitive analysis [25].

Multitasking overhead

Operating systems with multiple tasks have to deal with multiple sources of overhead, regardless of the scheduling algorithm. These sources include preemption or migration, executing the scheduling algorithm and context switching between tasks [26]. In particular costs for preemption and migration are difficult to predict, because their costs increase with the working-set size of the tasks. The working set is the collection of memory segments recently used by the task [27]. Migration of a task means migrating execution to a different core on a multi-threading machine. The costs incurred by preemption and migration are dominated by the time it takes to re-populate the cache, and thus increase with the size of the working set [26]. Research into migration found that simply copying the whole cache content over to the new cache is inadequate to address this cost, but capturing the access behavior of a task to prefetch data into the new cache significantly reduces migration costs [28]. Since the effect caused by preemption effectively increases the WCET of tasks in a real-time system, its effects on the schedulability analysis of a task set in a real-time system have been studied [29]. The resulting analysis techniques determine the schedulability under some algorithm (e.g. earliest deadline first), by assuming an additional component to each task's WCET, based on its actively used working set and the total number of segments it loads into the cache [29]. This schedulability analysis is more advanced than assuming a fixed overhead for all preemptions, but is still relatively pessimistic and not system-state aware in the sense that it takes the interprocedural control flow into account for the individual WCETs.
Further research focuses on searching for ideal migration points over the lifetime of a task, which allow for good overall schedulability and low overhead. This is important because the active working-set is not constant during a task's lifetime and assuming a set of fixed migration points reduces pessimism about the working-set size [30]. Ideal migration-point candidates should be very granular to migrate at the times required for scheduling while minimizing the active working set at that point [31]. The active working set can be determined using compile-time analysis [32]. This is especially important in *non-uniform memory architectures* (NUMA), because the cost of loading the working-set to the target depends on the architecture and between which cores the migration is performed [32]. In uniform memory architectures by contrast the access latencies are more predictable.
Analysis of preemption has to be treated differently, mostly because effects of a NUMA are different [32], yet the active working-set size and its development over time are still relevant.

Advanced fuzzing

In the area of security meanwhile, testing software for bugs using automated test generation (called fuzzing) is very common. One popular technique for efficient fuzzing is called coverage-based fuzzing, in which coverage information of generated test cases is used in a genetic algorithm to maximize the test coverage and therefore bug frequency [8]. The popular fuzzer ALF is an example using this technique [5]. It serves as the base for numerous extended fuzzers specialized for different targets and using different heuristics. One of them is kernel-AFL (kAFL) [33], which is specialized in fuzzing kernels, instead of user-space programs. kAFL executes its target under a hypervisor and uses Intel's Processor Trace technology to trace the control flow in a low-overhead and target-independent way. The authors identify the non-determinism introduced by interrupts and the state fullness of the system as obstacles to coverage-guided fuzzing of kernels. They address the non-determinism of interrupts by filtering out code transitions caused by them. Nyx is a fuzzer targeting hypervisors by running them nested inside a hypervisor with extremely quick snapshot restoration [34]. According to the authors, it achieves multiple thousands of restores per second on commodity hardware and can also be used to fuzz systems including kernels and applications. The snapshot-based rests eliminate differences in the state between runs and allow very complicated targets. A different, yet important part of the fuzzing process is the selection of mutators. The authors of the MOPT mutation scheduler noted, that the efficiency of AFL's default mutators differs between test cases and introduced a scheduler that measures the efficiency of each mutator and derives a probability distribution for choosing the next mutator to apply [35]. The approach of this thesis combines techniques from the areas mentioned above into a fuzzer guided by system state coverage to generate inputs maximizing the worst observed execution time.

Chapter 4

Problem Analysis

As described in Chapter 3, the problem of finding the WCET of an application in isolation is well explored using static and measurement timing analysis. As pointed out by Schneider [7] in 2002, analyzing the WCET of RTOS services and applications in isolation is not very effective. On the RTOS side lacking information about call parameters and context for example lead to pessimistic analysis. From the application perspective, the presence of multitasking raises issues wherever memory or other resources are shared because values for loop bounds and branches might be changed by interfering tasks, which again is dependent on the scheduler [7]. Since then static analyzers have started integrating aspects of the RTOS into a whole system WCRT analysis [1]. At first by tailoring the OS to be suited for a static analysis based on the state transitions and later to use user-provided source annotations to derive the necessary control flow facts [1], [24], [25]. To my knowledge measurement-based approaches currently do not leverage system awareness to optimize their searches.

A disadvantage of these static analysis methods is poor scalability due to the large number of states which need to be considered, even if many of them are infeasible in practice [1]. Measurement-based methods only encounter a small subset of the feasible states and have to optimize their search. This thesis focuses on measurement-based methods due to their better scaling potential. As mentioned in Chapter 3, genetic algorithms have been successfully used for searching long-executing inputs in the past [3], [4]. A potential problem for such an algorithm in a whole system setting is to explore the possible control flows sufficiently using a limited number of inputs per generation. In other contexts, such as finding runtime errors, dynamic testing using coverage-guided fuzzing is very popular and successful [6]. It uses a genetic algorithm to efficiently cover all possible branches in the control flow and keep the relevant inputs in its corpus and should therefore be better suited to explore a system-wide control flow than a black-box approach while being easier to implement than white-box techniques. This raises the main questions that this thesis aims to address. The first is whether coverage-guided fuzzing techniques are also effective in discovering the worst-case inputs of real-time systems. The other is if such an approach can be improved by considering the global control flow between multiple tasks and using their coverage to guide the fuzzer. A third, less central question of this thesis is whether execution counts derived from a theoretical IPET-based static analyzer could be used as an alternative guiding mechanism for the fuzzer to produce a witness input for the statically determined edges.

As the question about the global control flow implies, this thesis focuses on execution time differences caused by the global control flow between tasks and interrupts. Hardware effects are disregarded. More details about the system model used for this thesis can be found in Section 5.1. Chapter 5 describes the fuzzing concept in detail, while Chapter 6 describe an implementation of these concepts. An evaluation of the implementation in terms of the guiding questions can be found in Chapter 7.

Chapter 5

Concept

This chapter describes how system-aware coverage-based fuzzing can be used to address the problems described in Chapter 4, namely finding an input close to the WCRT of a small real-time system. It describes the methods and reasoning for them only in the abstract, implementation details can be found in Chapter 6. The first section of this chapter is dedicated to the model of the system used in this thesis and the implementation of its approaches.

5.1 System Model

This thesis focuses on execution time differences caused by the global control flow between tasks and interrupts, as well as scheduling while disregarding hardware effects. The model used for this purpose assumes an RTOS (the implementation uses a library OS) with periodic and aperiodic, fixed priority, preemptive tasks with task notifications and message queue, in addition to interrupt handling. It is running on an emulator which does not simulate hardware effects such as caches and execution pipelines. The emulated machine is a uniprocessor system. The system runs until reaching the end of the first hyperperiod, which marks the WCRT of the system's main task. For simplicity's sake this thesis makes some assumptions about the effects of an interrupt:
To keep the input simple and things like minimum inter-arrival time out of scope, this thesis assumes only one kind of incoming IRQ with a minimum inter-arrival time longer than the hyperperiod of the system. Since the approach in this thesis is a state-aware fuzzer it is necessary to ensure the effects of interrupts are visible in the state transition graph. Therefore, an additional assumption is made, which is that the only effect of the interrupt handler on the system state is to unblock/enable an aperiodic task, which would then show up as a state transition when the task runs. To limit the growth of the STG one last assumption is made, which is that the priority of the interrupt-activated task is relatively low compared to the other tasks. This limits the number of branches in the STG

caused by different interrupt times since only a few tasks can be preempted because of it. While this last assumption is not necessary for the concepts described in this thesis to work, it is advantageous to them. No further assumptions about the effects of interrupts are made.
Under this hardware model, the main timing effects caused by interrupts are the overhead of interrupt handling, scheduling and context switching, as well as the execution time of the newly activated task, if it causes a priority inversion. A priority inversion occurs if a middle priority task (activated by the interrupt in this case) preempts a low priority task which is blocking a high-priority task [12].

5.2 Fuzzing for execution time

The general setup and software used for this thesis originate from fuzzers used for finding runtime errors and security bugs. Fortunately the common fuzzing loop (see Figure 2.1) is very similar to existing genetic execution time maximization methods discussed in Chapter 3 [3]. In both cases, inputs need to be executed, rated, mutated and prioritized in some way. Figure 2.1 shows an abstract overview of a generic fuzzer setup. The main difference is the rating and prioritization, which translates to the feedback and corpus scheduler components. Common fuzzers keep inputs in their corpus for at least as long as they cover a unique code path, instead of only keeping a fixed number of inputs per generation. This is connected with the rating since the feedback in coverage-based fuzzers decides not only on a fitness function but also whether a new edge is covered.
The most straightforward approach to create a coverage-based fuzzer for execution time is to change the feedback and scheduler. The feedback has to reward inputs that increase the number of times a control flow edge is executed, not just when an edge is executed for the first time. Also, the scheduler needs to prioritize inputs with high execution times in some way when selecting the next input from the corpus. Choosing only the top-performing one from the corpus would drastically limit the choices and increase the risk of getting stuck in a local maximum. An alternative is to prioritize all inputs which reach the highest execution time amongst some subsets of inputs. For example one subset for each edge in the CFG (see Chapter 6 for details [36]). Figure 5.1 shows an overview of this concept, where the changed components are highlighted when compared to a basic fuzzer. The new components are a feedback rewarding increases in the execution count of edges, and a scheduler that prioritizes the test cases with the highest execution time amongst any set of inputs that cover a specific CFG edge each.
The approach of this thesis also extends beyond the coverage information typically used, which are execution counts (or just one bit) of the edges between basic blocks in the CFG. For this purpose, the execution times and a measured system state transition coverage are combined to rate inputs for inclusion in the corpus and prioritization of inputs from the

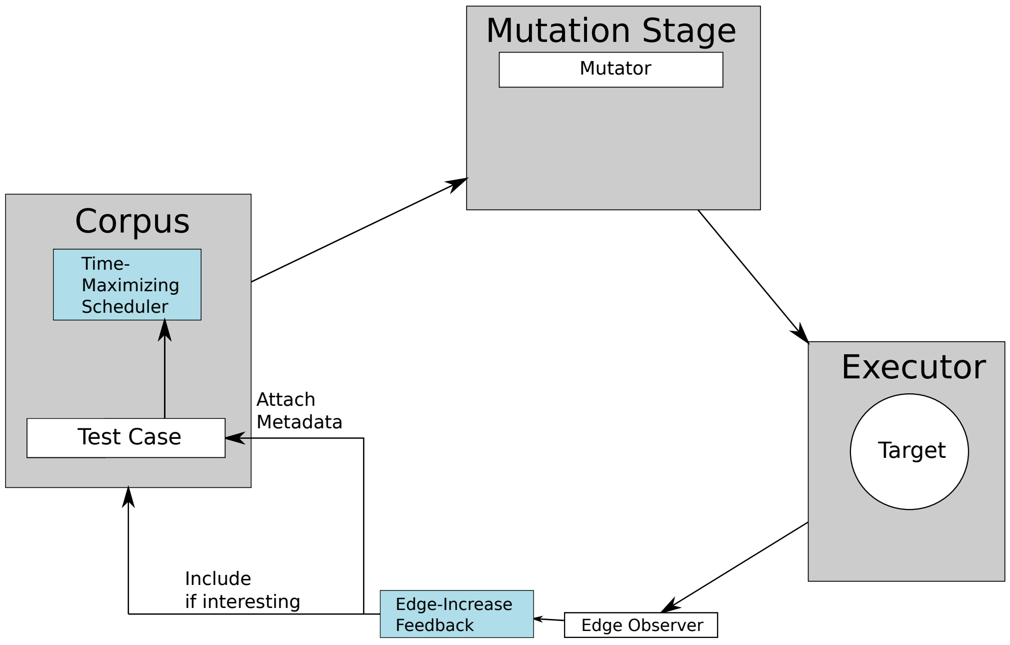


Figure 5.1: Abstract overview of LibAFL-based fuzzer which maximizes execution time. Arrows indicate information flow. Blue highlights are execution time focussed changes compared to a generic AFL-like fuzzer. The feedback has to reward all increasing edge-counts and the scheduler has to prioritize long-running cases.

corpus. For the feedback, this means including an input to the corpus if it increases the execution time of a system-state path or a component of it. The scheduler is preferring inputs that trigger the worst execution time for a path or component of one. More details about how system state transitions are used can be found in the next section.

5.3 Utilizing the state transition graph

As described in Chapter 2, CFGs can be extended to a graph of system states. This section describes how the necessary information is captured and utilized, as well as the effects of interrupts on the STG.

Tracing state transitions

The concepts of atomic basic blocks, system states and state transitions were described in Chapter 2. Unlike the concepts introduced by static analysis approaches, the approach of this thesis does not construct the CFG or the complete STG statically. The lack of CFG means ABBs are also not known to the analysis and a different method for distinguishing states is needed. For the purpose of this thesis, the execution is split at the context switches between tasks. Such a switch can be caused by calling an RTOS routine or a timer or other asynchronous event. At those points the following information about the state and previous block gets collected:

1. Current task's name
2. Current task's priority
3. Current task's base priority
4. Current task's number of mutexes held
5. Current task's task notification state and value
6. Info 1-5 for each task in the ready state
7. PC before last jump/system call, which is an approximation for the ABB boundaries

Each resulting block of execution thus contains only the execution of the one active task and RTOS kernel routines. Figure 5.2 depicts a fuzzer in which the executor hooks into

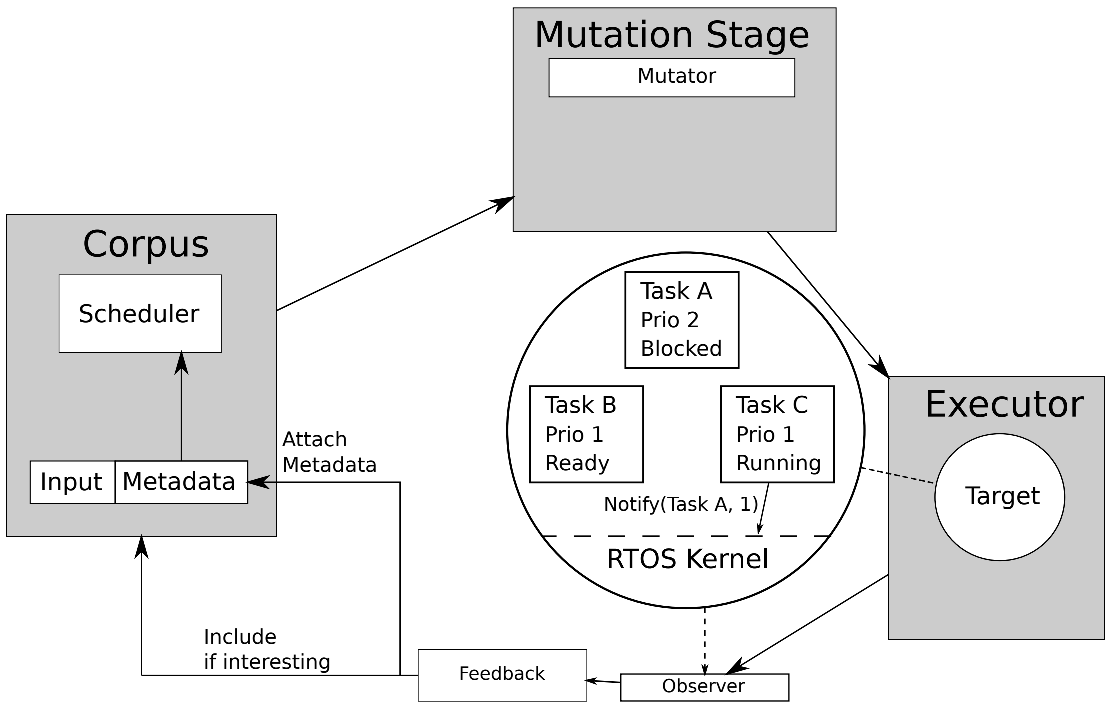


Figure 5.2: Abstract overview of a fuzzer using system-state information. The real-time system shown in the middle is in a state where some Task C is running and just sent a notification to a higher priority Task A using the RTOS API. An observer collects those states.

the execution of the target at the moment of a context switch, caused by a notification that unblocks a task of higher priority.

Multiple state traces can be combined into an STG, like the one depicted in Figure 5.3. It

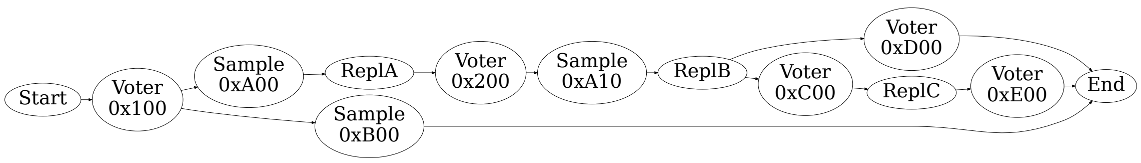


Figure 5.3: Example STG of a small triple modular redundant application. Three paths are possible depending on the availability of data for sampling and disagreement between replicates A and B. Priorities from high to low: Voter, ReplA, ReplB, ReplC, Sampler. PC address annotations below names signify a difference in local control flow, which cause changes in the global control flow.

represents a small application that processes input data using three redundant replicates (*triple modular redundancy* (TMR)), of which the third one is optionally activated when the first two disagree. It can follow one of three paths. Either terminate early during data sampling due to lacking data, finish the run after two replicates or use the third one. Branches in a state block are highlighted by different program counters.

In STGs used in this thesis, different paths with the same prefix and suffix are never merged (until the common end state), which means the directed graph is almost a tree. This was done for simplicity's sake and because none of the mutators proposed in the next paragraph would benefit from merging branches. Apart from constructing a graph, it is also possible to collect and use the traces (paths in the STG) of the states as is for coverage.

Maximizing execution time using the STG

To maximize the total execution time it is necessary to trigger the worst possible task activation pattern and thus path through the system state graph, as well as maximize each task's execution time within the confines of that path. The approach of this thesis uses coverage-guided fuzzing to explore the complete graph and maximize each path's execution time. The necessary state information is gathered from the emulator running the target as described in the previous paragraph. As mentioned in the previous section, the collected information is used by multiple heuristics for rating, mutating and scheduling of inputs:

The rating is performed by a feedback function that identifies which part of the system state graph an input covers, and based on the execution time decides if the input should be included in the corpus. This coverage over the graph may be interpreted in different ways, analogous to path and statement coverage in classical CFGs. The simplest method is to treat each complete system state path as one category for coverage and update the corpus whenever the worst observed execution time for a category increases. This does not require the graph to be present as a data structure, since a hash over the trace is sufficient to distinguish different paths. The other, more granular approach treats each system state

as a category and updates the corpus if even just one block has an increased execution time.
The common algorithm for selecting preferred inputs from the corpus for mutation (as used in LibAFL [36]) works by selecting an input that has the worst observed execution time of all inputs in one of the coverage categories. This can be achieved with either complete path traces or single states in the graph. Both the feedback and corpus prioritization are based on existing coverage-guided fuzzers, which would take each unique CFG edge as such a category [36].

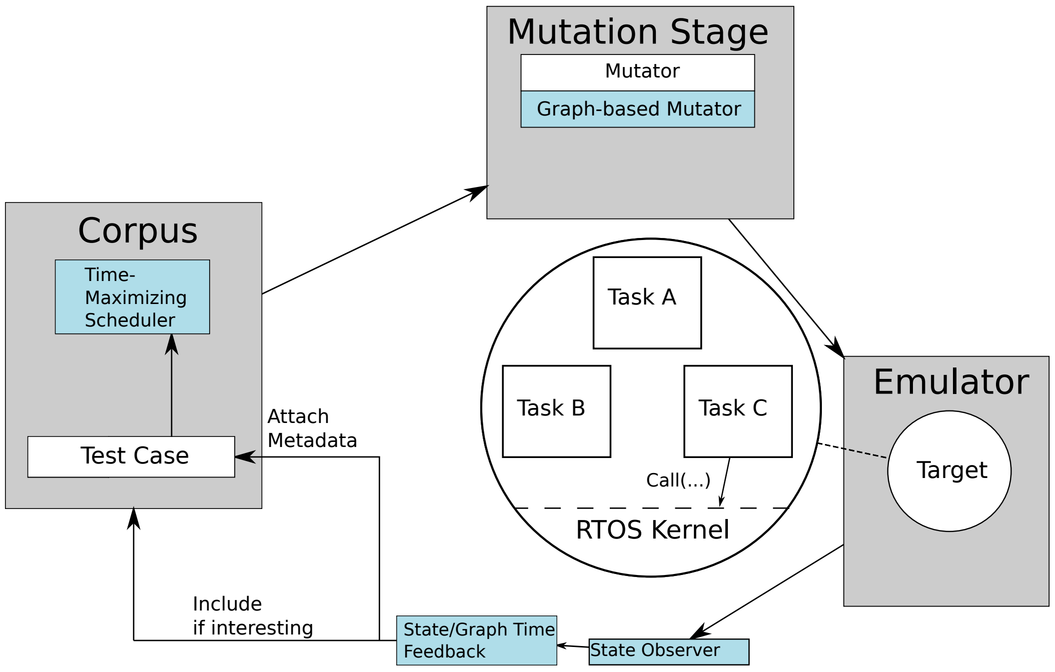


Figure 5.4: A System-State aware fuzzer to maximize execution time. Arrows indicate information flow. Differences to Figure 5.1 are highlighted in blue.

The last component of the fuzzer which can use the information from the system state graph is the mutation stage. This thesis explores three different mutators specifically:
The simplest one randomly replaces one input snippet or a complete suffix. Another mutator manipulates when an interrupt will fire and causes an asynchronous event to occur. Details on interrupts and their effects follow in the next section. The most advanced proposed mutator combines snippets from different inputs on the same trace to maximize the execution time of the whole trace. This mutator in particular is very powerful under certain assumptions. The first assumption is that an input to the system is read in small chunks over the course of the execution and the progress can be tracked. This tracking of which input was read during which part of the execution trace enables a primitive form

of taint analysis over the system state graph. This allows more targeted changes in the state path by only manipulating the inputs up to a point of interest. It becomes yet more powerful under the second assumption, that the behavior of a state block is largely dependent on the last part of the input read. If this assumption is met it allows freely choosing the worst input snippets from all inputs within the same state path. While the first assumption is largely dependent on the input model of the system, the second one is very strict and can not be guaranteed generally.
Figure 5.4 shows an overview of a fuzzer using either the graph or state paths as coverage. Compared to the concept based on CFG-edges this one needs a specialized observer to record the states of an execution. The new feedback function and scheduler are based on either state-path or graph coverage, as described. The graph-based mutators are placed in the list of possible mutations which can be applied.

5.4 Fuzzing for Interrupts

Real-time systems may need to handle and respond to some *interrupt requests* (IRQ) from external sources, be it from sensors or communication for example. Therefore, the executor of the fuzzer needs the ability to activate the virtual IRQs of the target at certain times. The straightforward way to include this in the fuzzing process is to include the time of the interrupts into the input as an offset from the virtual system start.
In addition, to support interrupts in the executor it is also desirable to make other components of the fuzzer aware of the inputs for interrupt times. In particular, the mutators based on system state information need to be aware of which part of the input is reserved for the interrupt time. The mutator described in the last section, which combines parts of the input, needs to be aware that the time inputs are consumed immediately. Additional mutators can be constructed for the time input. The simplest one just shifts the time of the interrupt activation into another state block in the graph, which would ensure the value is meaningful and also helps explore all possible placements. Since hardware effects are disregarded (see Section 5.1), shifting the interrupt this way has only a limited effect on the preemption overhead, but the activated task can still interact with the global control flow in arbitrary ways.

5.5 Fuzzing with known edge counts

Apart from exploring and discovering long-running inputs on its own, a fuzzer can in theory be used to search a witness input for the result of a static timing analysis. This section assumes that the number of executions of each CFG edge for the worst case is available as an input for fuzzing, to generate inputs with similar numbers.
A straightforward genetic algorithm exploiting this assumption works by determining the

difference between the set of observed and given edges and only selecting the k most similar (for some k) inputs for the next generation. One method to calculate the difference between the sets is a variation of the mean square error method. Assuming o_{ij} is the number of times the edge between basic blocks $i \in B$ to $j \in B$ was executed and t_{ij} is the same for the set of edges from the worst-case analysis, the fitness can be defined as $fitness(o, t) = 1 \div \sum_{i \in B} \sum_{j \in B} (o_{ij} - t_{ij})^2$.
Translating this algorithm to a fuzzer is not as straightforward. Since feedbacks in fuzzers only determine whether to include an input to the corpus, in this case, the feedback needs to include an input only if it has a fitness greater than all previous inputs. For selecting the next input to mutate it is desirable to have multiple options to avoid getting stuck in a local maximum. This can be solved using a corpus that weights the inputs according to their fitness and chooses randomly, thereby preferring multiple of the fittest inputs. Such a scheduler was not implemented for this thesis, but a simple queue was chosen instead. Also, due to the lack of a suitable static analysis method to determine the targets this use case was only evaluated using the edges of the known worst-case input to a program. Since the prototype for this concept only consists of the described feedback function, it needs no further elaboration in Chapter 6. Overall this novel use case is not the main focus of this thesis but might become relevant in future works if compatible static analyzers are available.

5.6 Resume

The previous sections present two coverage-based fuzzers aiming to maximize the execution time of their targets. The simpler one uses a variation of the AFL's feedback function with a corpus scheduler aiming to maximize execution time. It is shown in Figure 5.1. A second, more complex fuzzer is shown in Figure 5.4. Compared to the first one it traces the states of the execution of an input and adds a scheduler and mutations to make use of this additional information. Depending on the chosen configuration it can either use coverage over complete state paths or individual states in the STG. All fuzzers can optionally let the executor inject interrupts into the target, which simulates an important part of real systems.

Chapter 6

Implementation

This chapter focuses on the implementation of the concepts explained in Chapter 5. The implementation consists of an emulator running a target and the fuzzer around the emulator controlling it as part of its fuzzing loop. The last major component is the target for the fuzzer, which consists of an RTOS with multiple tasks forming an application. The first section presents the components and their interaction, while the other sections detail how they are implemented.

6.1 Overview

Figure 6.1 shows a detailed overview of the three components and their subcomponents in an STG-based fuzzer. The approach presented in this thesis is implemented using LibAFL, a modular fuzzing library [36]. It provides the overall architecture of observers, feedbacks, corpus and the mutation stage, as well as an interface to a modified version of QEMU to execute target code [9]. The RTOS used in this thesis is FreeRTOS [16].
The version of QEMU with LibAFL's instrumentation is built as a library and exposes additional functions to components called `QemuHelpers`. The example in Figure 6.1 contains three of them. `QemuEdgeHelper` hooks into QEMU's code generation to trace the flow between basic blocks. It is a standard component in LibAFL. `QemuSnapshotHelper` resets the state of the virtual machine after each run. It is specific to the implementation of this thesis, because by default LibAFL works on user-mode emulation instead of fully emulated machines. The last helper is `QemuSysStateHelper`, which is responsible for extracting state information from the target OS at specified times. In the figure, it is triggered at a system call in the target system.
Whenever an execution is complete the helpers expose their potential results to observers for further processing. The `MapObserver` observes a bitmap that gets filled by the callbacks from the `EdgeHelper`. This observer reconstructs them into a hash table of basic block edges with execution counts, which is used in the fuzzing approach that takes known

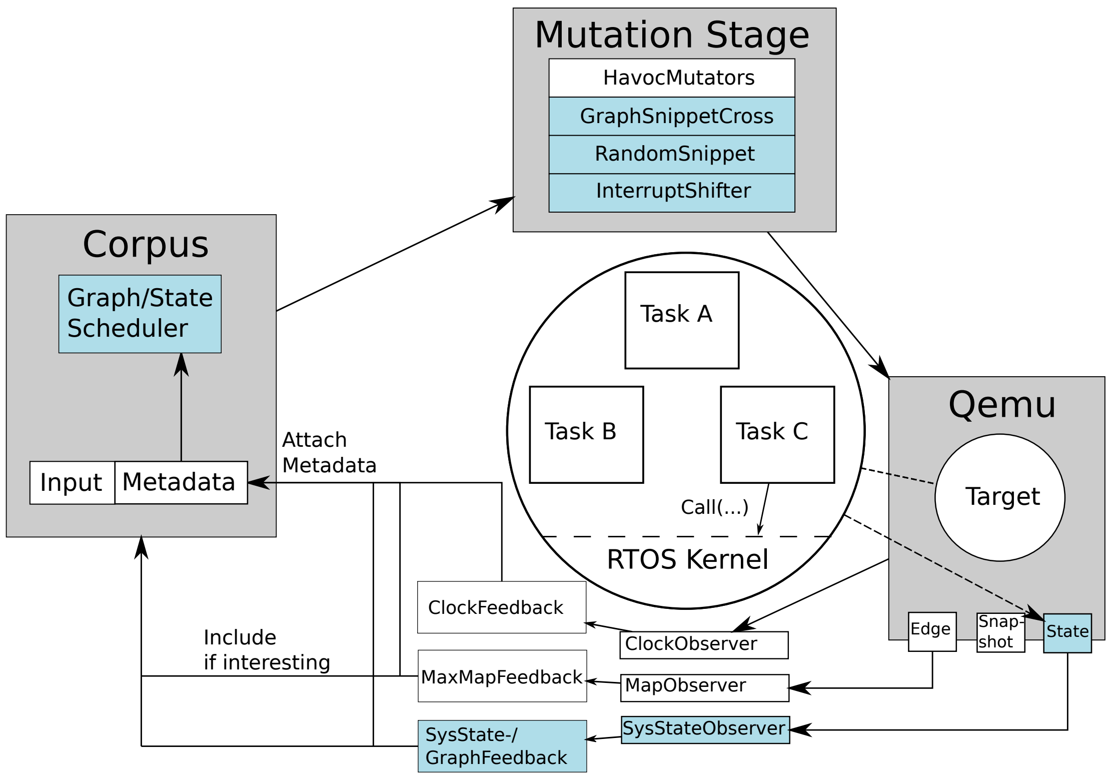


Figure 6.1: A complete STG-based fuzzer for increasing the execution time. Blue components are specific to the tracing, analysis and use of the STG. They either construct and work with the graph or just complete state traces. All non-blue components also form an AFL-like execution time-focussed fuzzer based on CFG edges. In this example, both the feedback mechanisms are used in conjunction. Apart from the blue components, the `ClockObserver` and `QemuSnapshotHelper` required implementation/modification to accommodate QEMU's system-mode.

execution counts as input to compare with this hash table. The `SysStateObserver` takes a series of state dumps produced by the `StateHelper` and processes them into a reduced form. The last observer is the `ClockObserver`, which converts QEMU execution time from instructions executed to nanoseconds and stores the resulting time. This is more precise than LibAFL's default, which has to measure host time because QEMU's user-mode emulation lacks precise measurement.

Once observers have performed their work the feedbacks take their information. Both `MaxMapFeedback` and either `SysStateFeedback` or `GraphFeedback` rate if an input is interesting according to the concepts described in Chapter 5. Along with `ClockFeedback`, they attach certain metadata to the test case if it is considered interesting. Of those feedbacks, `MaxMapFeedback` and `ClockFeedback` are based on default LibAFL components, while the others are created from scratch for the work presented in this thesis.

Once a test case is executed and feedbacks have been processed the next input has to be chosen. This is done by a `GraphCorpusScheduler` or `StateCorpusScheduler` using the

metadata attached by the feedbacks. The schedulers used for this thesis are based on a standard component from LibAFL, with modified prioritization.
Once chosen from the corpus the next input gets mutated in the mutation stage using multiple different mutators. LibAFL contains a default set of mutators that are also used by AFL, called `HavocMutators` [6]. The mutators `GraphSnippetCross`, `RandomSnippet` and `InterruptShifter` are using metadata about state transitions. The choice of mutators to apply is performed by another scheduler already implemented in LibAFL.

6.2 Real-Time Operating System

One approach of this thesis is to estimate the WCRT of real-time systems using the states of a whole system. A small real-time operating system is ideal to achieve this, because it is easier to understand and the state has to capture fewer variables. Its authors describe FreeRTOS as a "market-leading RTOS for small microprocessors", which makes it a valuable target for WCRT analysis tools [16]. It features fixed priority-based preemptive scheduling. The following paragraphs describe how the FreeRTOS-based test application receives input and what constitutes its state.

Input Model

To fuzz the system it needs to receive some input that influences the behavior of the system. In real-world systems, there can be multiple sources of input. For example, sensors can be read synchronously as part of a task. Another example is asynchronous communication over a *universal asynchronous receiver transmitter* (UART) component used for serial communication.
The target system in this implementation is running under emulation. Theoretically, this allows reproducing arbitrary synchronous and asynchronous inputs to the system transparently by emulating the necessary input devices. In practice, the emulation of real input devices for the system is out of scope for this thesis. The target system is built to simulate a sensor reading from values in memory instead, written by the fuzzer. Whenever a task reads input from this memory it increments a counter by the number of bytes read, which allows the instrumentation to track in which states the input was read. Asynchronous communication is simulated by handling a UART interrupt injected by the fuzzer. Details on the interrupt generation and memory access can be found in Section 6.4.

Target-specific instrumentation

The fuzzer needs to gather some information from the RTOS which make up the state of the system. FreeRTOS is very minimal and usually operates in a single address space. Its kernel functions are directly exposed as library functions to the task code. Compared to general-purpose OSs which use system calls and switch in execution mode this complicates the notion of system states and transitions described in Chapter 2. This is because ABBs, which are defined as single entry and exit regions do not necessarily end at the point where the kernel functions are called but might extend inside them. From the implementation side, this raises the issue of where the cutoff point for a state should be and how it can be reliably intercepted by the instrumentation. Some options would be the entry points of all kernel API functions, points where critical regions of the kernel are entered, or the point where task switches occur. For this thesis, the interrupt handler which manages task switches is used, because this single point can easily be intercepted in the emulator. The consequence of choosing this point is, that each block of execution ending in a system state starts with the handler and usually ends after the kernel API code was executed. This has the positive side effect that a critical section just ended, and all data structures are in a consistent state. A disadvantage is that since the last instruction before the task switch handler is always in the kernel it hides which branches in the task were taken and what function call caused the task switch. This issue is addressed by the emulator, which records the outgoing address of each jump from application (task) code to the kernel functions. The necessary separation is performed by injecting markers into the symbol table of the FreeRTOS application during linking. These additional symbols mark the beginning and end of the application codes region. Each jump that originates in the application code is recorded. Section 6.4 describes in detail how the emulator instrumentation intercepts the handler/jumps and which information it gathers from the system.

6.3 Fuzzer

The fuzzer is built using LibAFL, a fully modular library to build fast and scalable fuzzers for custom targets [36]. It also comes with a library of ready-to-use modules for different targets and fuzzing techniques. This modularity and ease of modification is the key factor for choosing it as the framework for this thesis.

Modularity

LibAFL divides a fuzzer into multiple concepts, such as observers, executors, feedbacks, inputs, corpora and mutators [10]. Executors are an abstraction over some software that can run the target. In the case of this thesis, it is a modified version of LibAFL's `QemuExecutor`, which is described in the next section. After an execution observers extract information from the target and feedbacks rate the relevance of the result. If a result is interesting the corresponding input gets added to the corpus of known inputs and eventually gets mutated. A comprehensive overview of modular components used in the implementation is given in Section 6.1.

System State Graph

During the execution of the target, callbacks from `QemuSysStateHelper` are called. One such callback reads the current *task control block* (TCB) and the priority lists of executable tasks from the emulator's memory. Details on the extraction from memory can be found in Section 6.4. After execution of the target, a list of this recorded information gets post-processed by the `SysStateObserver`. The only information preserved from the run is a list of state snapshots containing the following information:

- `start_tick`: Time when the task began executing
- `end_tick`: The time when the snapshot was taken and a different task will enter execution
- `last_pc`: Last position of the program counter in the area of the application code
- `input_counter`: Counter how many bytes of the input have been read up until the end of this block
- `current_task`: TCB of the current task, mainly contains its name, current- and base priority, notification state and number of mutexes held
- `ready_list_after`: Priority sorted list of TCBs which are ready for execution

If the fuzzer uses the `SysStateFeedback`, it computes a hash of the list of state snapshots, excluding the timing-related information. These hashes are looked up in a hash table and if a previous entry is found the latest run will only be marked as interesting if its execution time exceeds the one stored in the hash table, which gets updated in this case. If the hash of the latest run is not found it will be rated as interesting and its execution time and trace length will be saved. If a `GraphFeedback` is used instead of `SysStateFeedback` the decision is more complex. First, this list of state snapshots gets inserted into the system state graph by `GraphFeedback`. The graph consists of nodes that share most of the attributes of the state snapshots, except the input-related properties like the input bytes and times, which are collected as tuples in a list. During the update, the list of states is iterated while the graph is traversed at the same time. Whenever the list contains a state which does not correspond to a current graph node it gets inserted into the graph. Otherwise, the new input information is compared to the list of inputs in the graph node. If the new input shows a property which makes it interesting, such as taking the most/least amount of time or being the longest/shortest input, it gets added to the node. Once the iteration ends and if at least one node was updated or added the input is regarded as interesting and will be added to the corpus.

Mutation

The three mutations described in Chapter 5 use the metadata of the test cases in combination with the graph generated by a `GraphFeedback`. Particularly the mutator which aims to combine snippets of different runs (`GraphSnippetCross`) needs the information when input was read. The states in the STG track which inputs triggered it and how far the input was read in each case. This number is tracked by a variable that is updated by the target system. By comparing the number of input bytes in a state with the number for its predecessor it is possible to determine which part of the input field was read by the task in this state block. The mutator combines these input snippets by traversing a path through the STG and picking one of these input snippets at each state it visits. The assumption behind this mutator is that a part of the input being read determines the behavior of the application's tasks mainly for the time until the next part of the input is read (as mentioned in Section 5.1). As such combining these snippets which are known to lead to the same paths in the STG allows to form a new input for the same path. This new input may have a longer total execution time than any of the other inputs on the path because it can combine the snippets which cause the longest observed execution time in parts of the state path. This means it is focused on exploiting the information in the STG to generate overall longer executing inputs. The other mutator which uses the snippets (`RandomSnippet`) simply randomizes one snippet in an input to potentially trigger a different branch in the STG or prolong parts of the execution. It is more focused on exploring the STG. The last mutator that uses the STG (`InterruptShifter`) just selects a state block from the trace and sets the interrupt input to trigger some time during that block.
These mutators are added to a list of mutators which also contains a pre-defined list of mutators called `HavocMutators`, provided by LibAFL. This list contains a large collection of byte- and multibyte-operators for swapping, copying, incrementing, deleting and more, as well as cross mutating parts with other inputs. This cross-mutation is different from the graph-based mutator, as that one targets specific inputs per state block in the graph. The collection of mutators gets applied to the inputs by a mutation scheduler, which is implemented in LibAFL and based on MOPT [35], a scheduler that tracks the effectiveness of each mutator to optimize their selection probability.

6.4 Emulation

The default `QemuExecutor` executes user-mode binaries in QEMU's user-mode emulation, while the implementation in this thesis loads a kernel into a QEMU-emulated system. In this case, QEMU is responsible for ensuring the target is in the same state for each execution, loading fuzzing inputs and terminating the execution once the relevant part of the code is done executing. The input handling (apart from interrupts) works mostly the same in both user-mode and system-mode cases, where the input is written to a static array in the target using QEMU's memory-access functions. Resetting the target to the same initial state is usually not necessary in user-mode because the targets are often libraries, which are designed to be executed multiple times without keeping a state between runs. The work presented in this thesis uses QEMU's native snapshot feature to save and restore the entire virtual machine for the system-mode. Terminating the finished execution is handled using breakpoints in both user-mode and system-mode. Additionally, QEMU is responsible to execute callbacks that gather information from the guest's memory.

Guest Memory Access

To gather information from the target it is necessary to read from its memory and parse the data structures for consumption in the fuzzer. The required information is present in the target memory in the form of target-specific data structures. For FreeRTOS those include a reference to the currently active task control block and a priority list of tasks ready for execution. This presents three problems: First, the memory location of the structures has to be found. Second, the structures need to be read from the emulator. Third, the structures need to be interpreted, as they originate from a different architecture and refer to a different address space.
The first problem is solved using the symbols found in the *executable and linking format* (ELF) table of the target's kernel, which contains the addresses of static pointers to all relevant structures. The second problem is solved using QEMU's API, which offers the function `cpu_physical_memory_read` to read a variable amount of memory from the target at a specified virtual address. The third problem is more complicated. The emulated target in the work presented in this thesis was a 32*bit* ARM machine, while the host was a 64*bit* AMD64 machine. Since the size definitions are different, the memory representation of structs is not portable and they can not simply be interpreted by using the original c header files. This problem was solved by an architecture-dependent prototype. It uses bindgen [1] to create rust bindings for the struct definitions and then replaces the types in the resulting definitions with size-equivalent types to the target architecture. Since the alignment of the resulting struct is equivalent, this simple process allows the interpretation of extracted memory as structs. A more generalized solution for this problem would use the struct layout information in the debug information of the target generated by the target compiler and stored in the DWARF format [2]. Furthermore, even if a struct (and each one its fields point) gets translated correctly it's pointers still refer to a different address space and can not be dereferenced. This is solved by inserting the struct and each struct it points to (recursively) into a hash table with its virtual address as the key. This allows

[1]https://github.com/rust-lang/rust-bindgen
[2]https://dwarfstd.org/doc/dwarf_1_1_0.pdf

the host to follow the pointers in the snapshot through hash table lookups instead of direct dereferencing.

Interrupts

As described in Chapter 5, interrupts at pre-determined times are part of the input to the executor. More specifically this implementation assumes no more than one interrupt within the first hyperperiod of the application. This is implemented by reserving at least two bytes at the start of the input which get interpreted as the number of instructions to execute until the interrupt gets fired. Those bytes are not copied into the VM memory. The number of instructions is also offset by the known number of instructions it takes for the RTOS to finish its basic setup and start the application. This cuts down the input space of the interrupt time by eliminating times when interrupts are not enabled yet. Timing the interrupt is done in QEMU using a `ptimer`, which waits for a specified virtual time (a multiple of the virtual clock) to execute a callback function. The callback to raise the interrupt request is dependent on the virtual hardware. This thesis focuses on ARM processors using the *Nested Vectored Interrupt Controller* (NVIC), which supports setting the desired interrupt to pending inside of QEMU. Inside the RTOS an interrupt handler has to be registered for the activated IRQ.

Execution time measurements

The measured execution time in QEMU is taken by counting the executed instructions. This is a simplified model compared to established timing analysis tools. This is unrealistic for advanced processors, as it ignores memory latency and with it the associated effects of caches, as well as effects on the processor's pipeline, such as branch (mis-)predictions. Since the goal of this approach is not to give guaranteed upper bounds but to find inputs that approximate the worst possible path through the global whole system (see 5.1), this level of abstraction is sufficient for a proof of concept. If the fuzzing approach proves its usefulness, switching to an accurate emulator is possible.

Limitations

Due to an unknown reason snapshots that were taken after some execution time always showed the same final execution time after resuming and finishing. Therefore, the overhead of booting the RTOS is included in every run, which takes up the majority of the emulator runtime in small examples. Another limitation of QEMU is that it does not simulate hardware effects of the target platform. This is reflected in the system model this thesis assumes (see Section 5.1), as well as the previous paragraph.

Chapter 7

Evaluation

This chapter details the evaluation of the fuzzer described in previous chapters, with the goal of answering the guiding questions of this thesis. These are: Are coverage-guided fuzzing techniques effective in discovering system inputs with long execution times? Can the coverage guidance be improved by using the STG? Can know CFG-edge counts (e.g. from an IPET-based static analysis) be used to guide the fuzzer to generate a witness input with those edges (i.e. a worst-case input)?
Since there is no prior work available that is directly comparable, this evaluation only uses comparisons amongst the techniques described in Chapter 5. To do this the first section will describe the application used for testing the fuzzer. The following sections will evaluate the performance of multiple fuzzer configurations in two different scenarios, one with an asynchronous event and one without. In an additional last scenario, the fuzzer tries to replicate pre-determined counts of control-flow edges, simulating what a fuzzer could achieve in a hybrid scenario with a hypothetical IPET-based method.

7.1 Evaluation setup

The fuzzing loop was set up using LibAFL[1] version 0.7.1, and its QEMU fork[2]. LibAFL's QEMU instrumentation was extended to allow running in system-mode. This meant adding library hooks for LibAFL to save and reload the virtual machine, for injecting interrupts and setting breakpoints to terminate the target upon finishing execution. Additional instrumentation was added to instrument generation and execution of jump instructions. The instrumentation had to be backported to upstream QEMU[3] version 6.1.1 because a bug[4] in version 6.2.0 prevented saving the virtual machine on certain emulated ARM machines.

[1]https://github.com/AFLplusplus/LibAFL/
[2]https://github.com/AFLplusplus/qemu-libafl-bridge/
[3]https://gitlab.com/qemu-project/qemu
[4]https://gitlab.com/qemu-project/qemu/-/issues/803

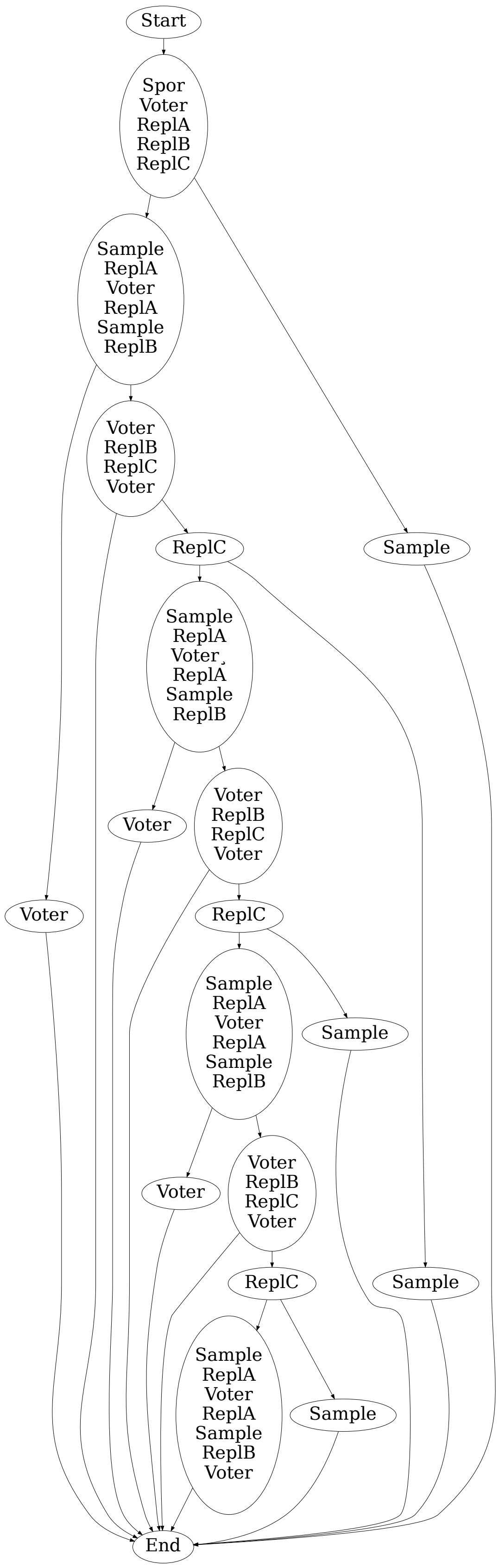

Start
Spor Voter ReplA ReplB ReplC
Sample ReplA Voter ReplA Sample ReplB
Voter ReplB ReplC Voter
ReplC
Sample
Sample ReplA Voter ReplA Sample ReplB
Voter
Voter ReplB ReplC Voter
Voter
ReplC
Sample ReplA Voter ReplA Sample ReplB
Sample
Voter
Voter ReplB ReplC Voter
ReplC
Sample
Sample ReplA Voter ReplA Sample ReplB Voter
Sample
End

Figure 7.1: System-State Graph for the test case, an example of TMR processing of inputs with an optional third replicate. Nodes represent states with their task name shown. Multiple names in one node indicate basic blocks of multiple states. It contains 11 different paths from start to end. 4 of them are early terminations in the Samper, caused by insufficient input. 3 of them end in the Voter after ReplB agrees with ReplA. 4 of them end in the Voter after ReplC resolves the disagreement.

The target was based on FreeRTOS[5] version v202111. It was built from the demo[6] application for QEMU's mps2-an385 model, which features an ARM Cortex M3. Details of the application used for testing are described in the next section.
The evaluation of the fuzzers is performed on a quad-core AMD Ryzen™3 2200G with three different fuzzer instances running in parallel. Each fuzzer instance runs the fuzzing loop in a single thread and using one instance of QEMU. While LibAFL allows parallelizing the fuzzing process easily and this was successfully tested with the fuzzers implemented in this thesis, it was not used for the final evaluation. The reason for that is to avoid any synchronization when writing out the execution times and producing a sequence of executions where each one is based on all available information so far, without delays due to synchronization of fuzzing states between instances. Parallel fuzzing would reach the same number of executions quicker, but since multiple single-threaded fuzzers are run in parallel this does not decrease the overall runtime of the evaluation.
The key metric to evaluate the fuzzers is their efficiency in discovering long execution times. It is tracked by the value of the WOET over the number of executions, which is a proxy for time. Since the fuzzing process does not have a stop criterion (in absence of knowing the WCET or its input), all experiments need to have some limit. The time was arbitrarily chosen as one hour, because the fuzzing progress has started to flatten off until then in every case. Over a long enough time, any testing method which allows mutating the input to completely random values would converge to the worst-case, as it can always be randomly selected, even if that is unlikely. So the expectation is for the WOET of every fuzzing method to converge towards the worst case, the speed of this convergence is a result of the fuzzers efficiency.
During fuzzing all execution times get recorded. Each fuzzer starts with the same initial input in the corpus but different seeds for the random number generator for all further actions. Each configuration is run ten times to average the maximum execution time seen up until a certain number of executions. Averaging of multiple results is done because the progress and results of individual runs show a great variance, so the average in addition to the standard deviation is more representative.

7.2 Test Case

The goal for the test case is to be small and present multiple ways to lengthen execution times and reach new states. The test application which was chosen aims to mimic a real-time system that reads a sensor and performs a computation on the value using triple modular redundancy, with the third replicate being only called if the first two disagree. On disagreement of all three replicates, the process will be re-tried three times, including

[5]https://www.freertos.org/index.html
[6]https://www.freertos.org/freertos-on-qemu-mps2-an385-model.html

data sampling. Once the voter receives two agreeing results the vote succeeds and the hyperperiod ends and execution gets halted by the emulator. The idea is that the worst execution time is reached only if the longest possible state trace gets reached and each value processed gets optimized to maximize the duration of the processing in the replicates. This scenario should allow the STG-based fuzzer to explore all paths and combine parts of the worst-performing inputs for each retry into one long-running input. The combination of parts is possible because parts of the input are largely independent, apart from triggering state transitions.
In the application, this concept manifests as a sampler, voter and replicates A, B and C. The sampler reads two bytes of input each time. Faults in replicates A and B are triggered by one of the bytes being divisible by certain numbers. Those numbers change per retry. The work performed by each replicate consists of iterations dependent on the value read. This makes it necessary for the fuzzer to discover multiple *magic numbers* to reach different system state paths which increases the difficulty of finding the worst path. It also necessitates further optimization within a path or state. Figure 7.1 shows the full state transition graph of the system without asynchronous events. Each node represents a state and is labeled by the name of its task. Linear stretches in the graph (basic blocks) have been summarized into nodes with multiple names for readability. It shows 11 different paths through the whole execution. The worst case of the system is reached when each byte in the input has the maximum value which is divisible by whatever number is demanded by replicates A and B to fail. Early terminations from the sampler task are possible if the input is too short. The worst-case inputs are known by construction and lead to the WCET of 375707 without interrupts and 379033 with an interrupt at its worst-case position. The worst-case position for the interrupt is where it activates a task at a point that causes the highest possible scheduling overhead before becoming active. Preemption overhead due to cache effects is not considered in the system model.
Overall this application offers a scenario where the fuzzer needs to first discover the next stage of the retry and then needs to exploit an existing path to reach alternative inputs per stage to reach higher execution times. If an asynchronous interrupt is injected, it will unblock a task without data dependency on the rest of the system. This task has a higher priority than the sampler, but nothing else. This leads to a scenario, where the maximum delay from the interrupt can be caused by triggering it after the sampler is active but before the samplers last activation. The newly activated task cause a priority inversion for the next activation of the sampler and maximum scheduling overhead until then.
One last noteworthy detail about the test case is that most of the execution time (at least 347870 ticks) gets spent on setting up the OS and tasks. For this reason, the comparisons in the following sections are only focused on the execution after the first application task starts executing. Before then no input to the system is read or has any effects on the execution.

7.3 Configurations for comparison

As explained in Chapters 5 and 6, multiple alternatives for each of the key fuzzer components have been developed. Each of them uses a different combination of the components from Section 6.1. This evaluation focuses on comparing the following interesting configurations (common components are not listed):

- `random`: Black-box fuzzer using inputs
- `afl_queue`: `MaxMapFeedback` with a queue-based scheduler
- `afl_mapmax`: `MaxMapFeedback` with a time maximizing scheduler
- `state`: `SysStateFeedback` with `StateCorpusScheduler`
- `state_afl`: Like `state`, additional `MaxMapFeedback`
- `graph`: `GraphFeedback` with `GraphCorpusScheduler`
- `graph_afl`: Like `graph`, additional `MaxMapFeedback`
- `graph_muta`: Like `graph`, but includes mutators utilizing the graph
- `graph_muta_afl`: Combines `graph_afl` and `gaph_muta`
- `known_edges`: A fuzzer minimizing the difference to a set of target edges

7.4 Comparison of Fuzzing Methods

All following comparisons of the different configurations are based on average measurements over ten iterations with one hour each. Comparison graphs have markings for the worst-case execution time and the best execution time on the worst system path. The first set of comparisons assumes no asynchronous interrupts as input.

Figure 7.2 shows a comparison of the three main fuzzing setups. It shows that the AFL- and graph-based fuzzers perform almost the same, while the state-based fuzzer outperforms them and reaches close to the worst-case input. All of them reach relatively close to the worst-case and outperform random fuzzer by a wide margin. The random fuzzer did not even reach the worst possible state trace, as shown in the graph by the fact that it did not reach the lower bound set by the best case input which triggers the worst-case state-trace (referred to as worst trace).

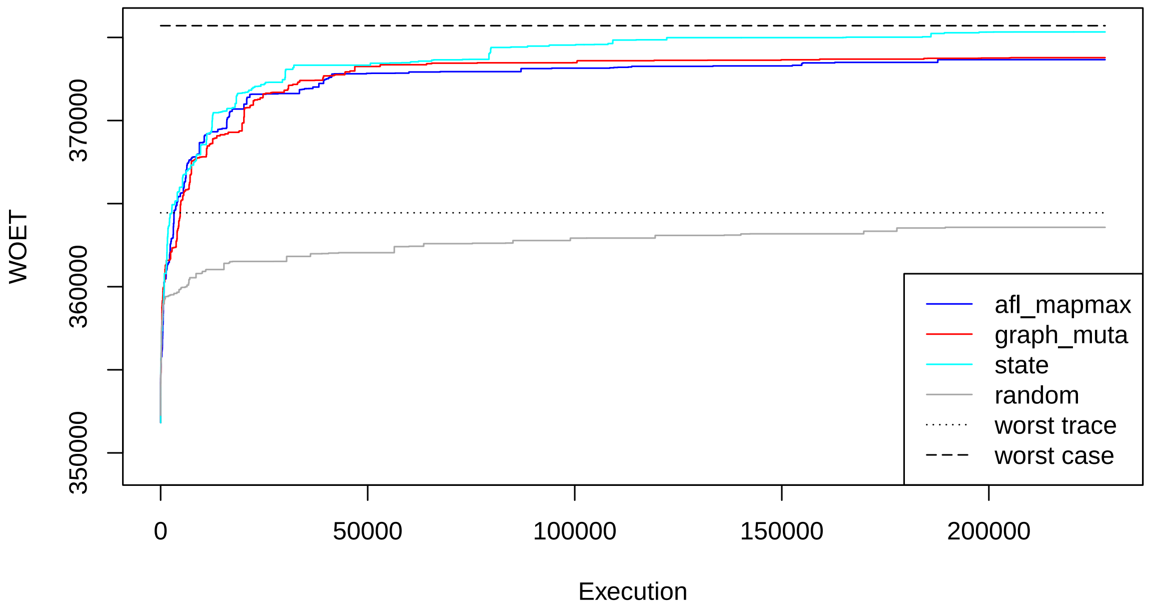


Figure 7.2: Comparison of WOET over time for multiple configurations. The x-axis is the number of inputs executed and the y-axis is the WOET until then. The state-based fuzzer outperforms the AFL- and graph-based ones. All three of them outperform the random fuzzer by a wide margin. The line called `worst trace` marks the execution time of the best-case input for the same state trace as the worst-case input.

Figure 7.3 compares the different configurations of the graph-based fuzzer in detail. It shows that additional feedbacks and mutators did not have a significant impact on the performance. The same can be observed for state-based fuzzers in Figure 7.4, which shows that the AFL-based feedback did not improve the state-based fuzzer. This is confirmed by Table 7.1, as the maximum execution time of all variants is well within the standard deviation of the base technique. The table also shows how close each WOET is to the WCET, while only counting the time since the start of the application code after 347870 ticks. The state-based fuzzer performs best with over 98% of the applications WCET. The lack of effects from the AFL-like feedback can be explained by the corpus size, which will be discussed later in this section. The lack of effect from mutators on the other hand is not as clear. One explanation for `GraphSnippetCross` not performing well is not picking long-running snippets enough and also being limited by the number of available variants to combine. That is because identical states on different branches of the graph are not merged. It would also adversely affect the `RandomSnippet` mutator. The overall observation is that the additional mutators for the graph did not improve the fuzzing performance.

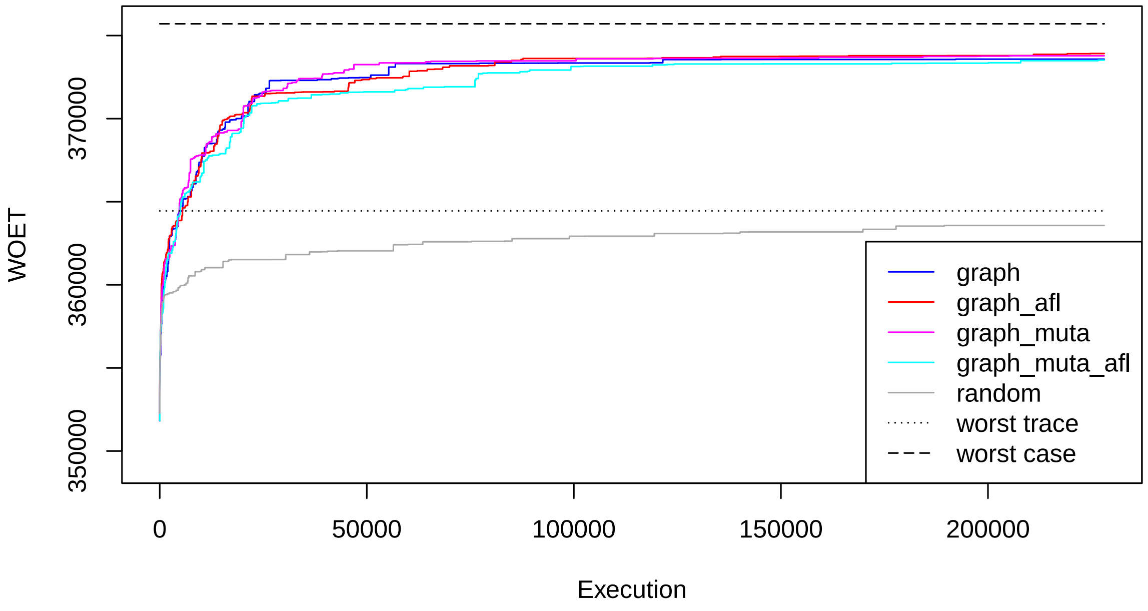


Figure 7.3: Comparison of graph-based coverage fuzzing with and without mutators and an AFL-like feedback. Neither the additional feedback nor the mutators show a significant impact. All of them reach close to the WCET.

configuration	avg. WOET	σ_{WOET}	App $\frac{\text{WOET}}{\text{WCET}}$
afl_mapmax	373662	963	92.65%
afl_queue	373620	1323	92.50%
graph	373581	933	92.36%
graph_afl	373917	1048	93.57%
graph_muta	373782	906	93.09%
graph_muta_afl	373521	1227	92.15%
known_edges	372838	1394	89.69%
state	375331	490	98.65%
state_afl	375359	366	98.75%
random	363574	1052	59.41%

Table 7.1: Average and standard deviation over max execution time during ten runs. The percentage of WCET describes how close each WOET is to the WCET. System setup time is discounted for this calculation, since application code starts after 347870 ticks.

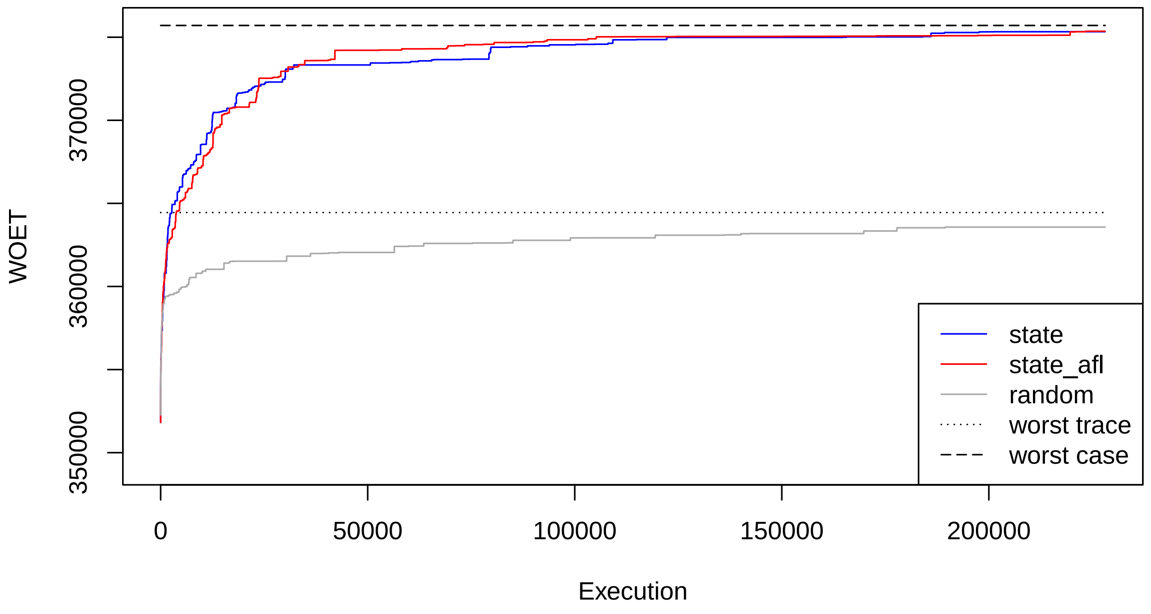


Figure 7.4: Comparison of state-coverage fuzzing with and without additional AFL-like feedback. There is no significant difference visible. Both reach extremely close to the WCET.

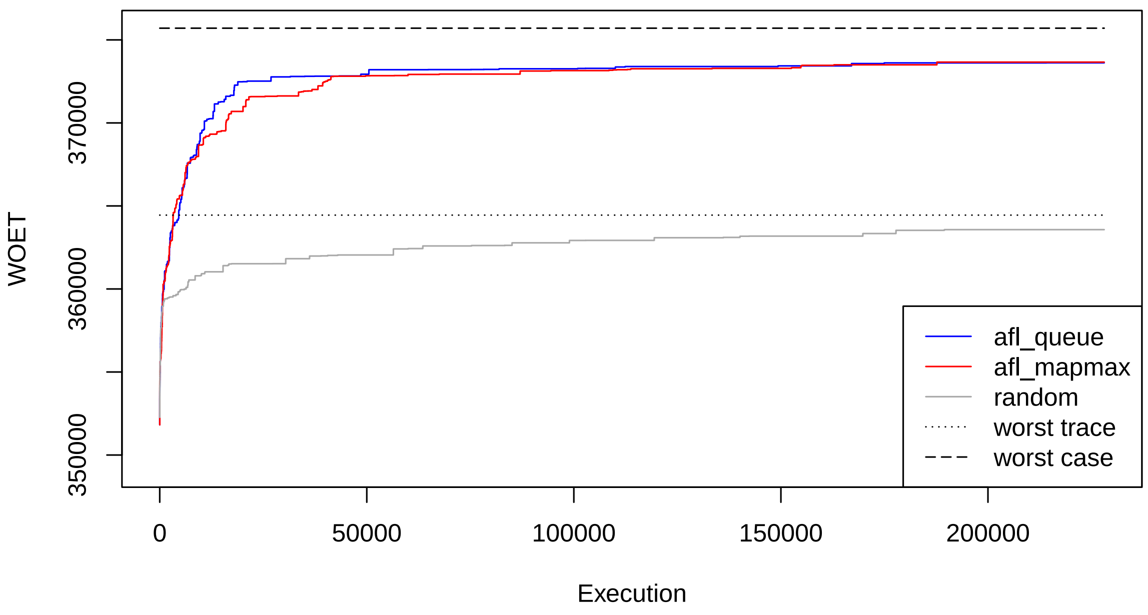


Figure 7.5: Comparison of AFL-like fuzzing with and without prioritizing scheduler. No significant difference between the two is visible. Both reach close to the worst case.

As seen in Figure 7.5, the AFL-based fuzzers did not show a significant difference when using a simple queue corpus or a time maximizing scheduler, which may be explained by the extremely low number of total corpus elements for this fuzzer. This can be seen in Table 7.2. The low count of corpus elements de-emphasizes the importance of a scheduler because there are few to no corpus elements that have been overtaken in the number of every edge.

The low corpus size for the AFL-like feedback is surprising. Tests confirm that inputs which increase loop bounds do increase the edge execution counts accordingly. The feedback is set to reward every increase in the execution count, even of a single edge. One possible explanation for this observation is the difficulty of maximizing the total number of loop executions. The total number of execution for a loop does not only depend on one part of the input but the global control flow, which raises the risk of getting stuck in a local maximum. A concrete example for this might look like this: Assuming the loop in replicate A gets exercised 40 times during a run that terminated without a retry. Another run might exercise it 10 times on the first try and 25 times on a retry. From a global perspective, the second run is more valuable, because it leads to the longer system state path. Meanwhile, the first one might still be considered more interesting by the fuzzer, because it had a greater number of edge executions and potentially also execution time. This could lead a fuzzer to discover inputs that trigger all STG paths only once, but not discover increased edge counts on them because previous local maxima overshadow the discoveries on the new path.

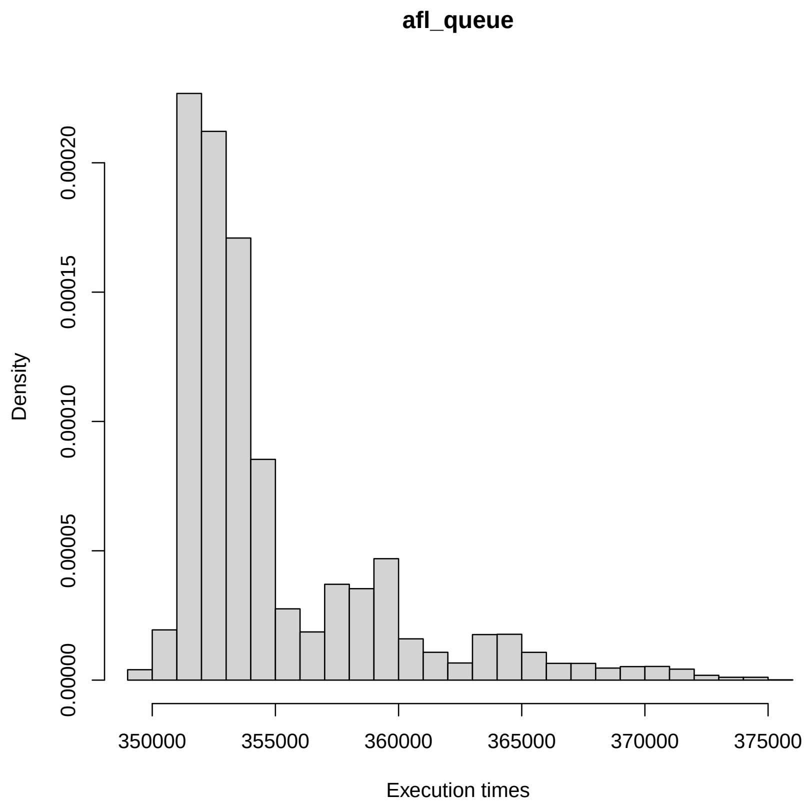
The graph-based method and the mostly AFL-based method are extremely close to each

Configuration	avg. corpus	σ_{corpus}
afl_mapmax	8,1	0,88
state	94,2	18,20
graph	79,1	4,68

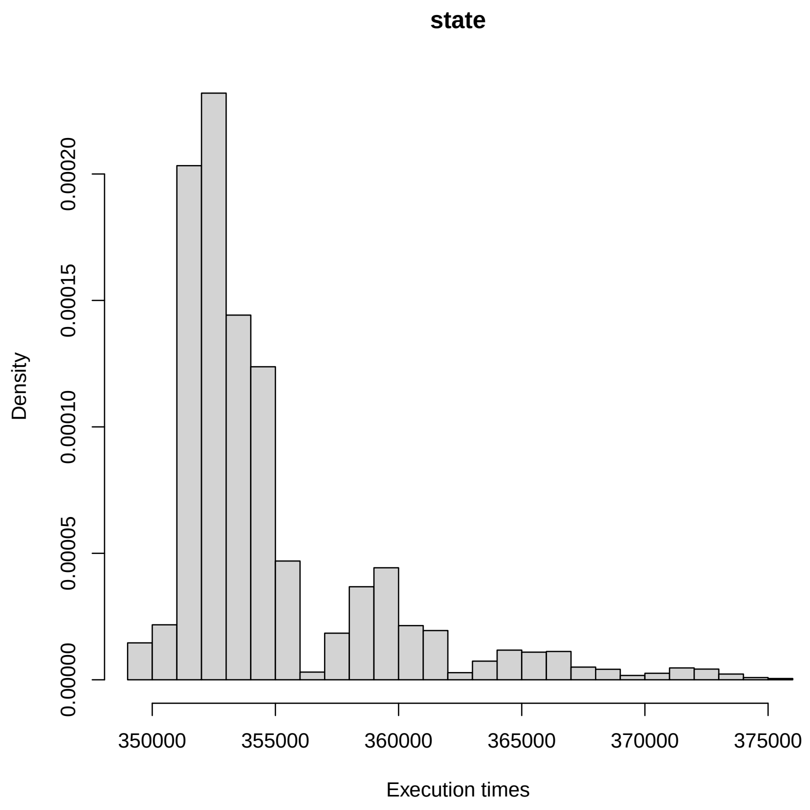
Table 7.2: Average corpus size and its standard deviation for the three main configurations

other. This is surprising since the graph-based method is an extension of the state-based method. As seen in Table 7.2, the state-based method even adds more elements to its corpus overall. One possible explanation for this phenomenon is, that the increased number of inputs that the graph scheduler regards as relevant at the same time slows down the fuzzing process. This is caused by the large number of states caused by every branch in the graph and could be mitigated in future works by merging states from different branches.

The previous figures focussed on the WOET values over time, but it is also interesting to observe the overall distribution of observed execution times because it shows how well the fuzzer prioritizes long executing inputs. Figures 7.6a and 7.6b show histograms of the execution times for the `afl_queue` and `state` configurations. High frequencies of long-running executions are desired, but neither of them shows this. While the overall distribution is very similar, Figure 7.6b shows four distinct clusters of execution times, separated by valleys. Those clusters are presumably caused by the system-state transition graph and the fact, that the state fuzzer targets each path separately. As described in Section 7.2, the application can retry the calculations multiple times if certain conditions are met. The more pronounced clusters in Figure 7.6b may be explained by the fuzzer's scheduler, which continues to fuzz all paths, even after longer ones have been discovered.
Overall the configuration using unique end-to-end paths in the system transition graph performed the best. It discovered progressively worse paths quickly and reached the overall maximum the fastest. This shows that this basic degree of system-state awareness can guide a fuzzer to the worst paths relatively quickly, while previous coverage-based fuzzers are also able to reach close to the worst cases. The graph-based mutators have shown no significant advantage over random mutators in a graph-based fuzzer, which itself also did not significantly outperform the AFL-based feedback and scheduling.



(a) Distribution of execution times in the `afl_queue` fuzzer. The large majority of executions had execution times below 355000 ticks.



(b) Distribution of execution times in the `state` fuzzer. The large majority of executions had timed below 355000 ticks. Multiple bumps in the distribution are visible, presumably corresponding to the major retry steps in the application.

7.5 Fuzzing with interrupts

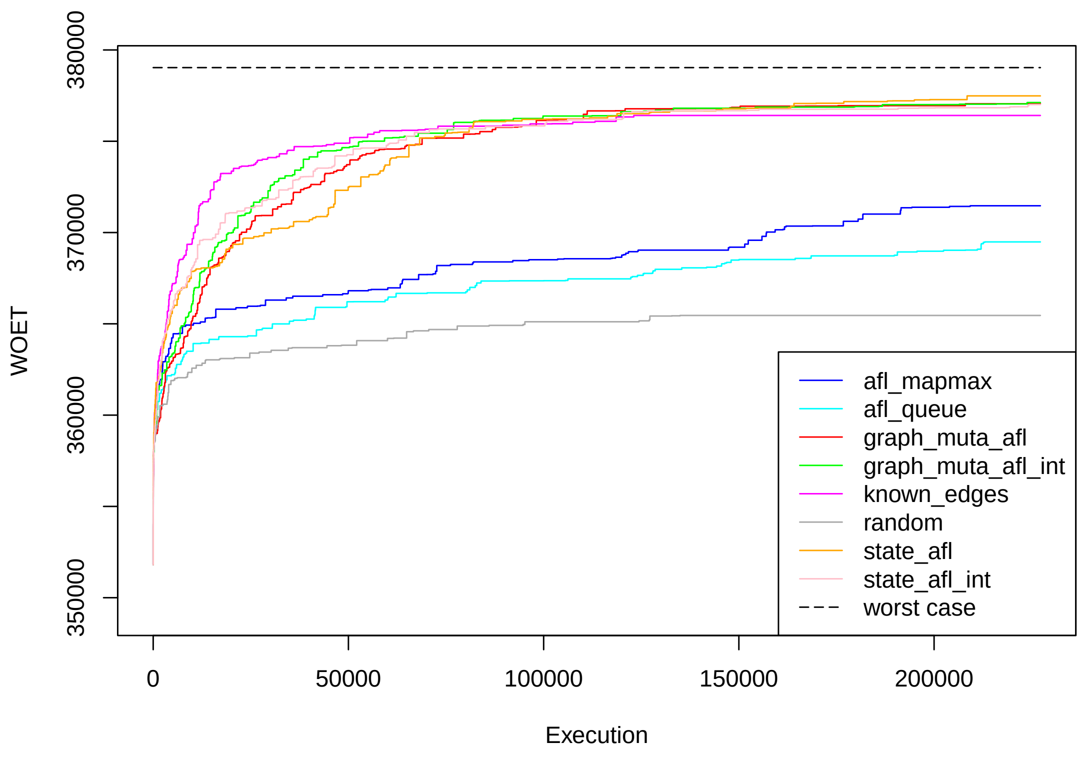


Figure 7.7: Comparison of WOET over time for multiple configurations with active interrupts. The int suffix represents configurations with a mutator affecting interrupts. State- and graph-based fuzzers perform close to each other, while the AFL-based configurations perform much worse.

The previous section focussed only on the input directly consumed by the target application while disabling interrupts. As can be seen in figure 7.7, when enabling interrupts and firing them based on the input, the performance of the AFL-like fuzzer suffers from worse performance than with disabled interrupts. All other variants meanwhile perform very similarly to each other and again reach close to the new worst-case execution time. The on average best performing fuzzer was `state_afl`, which reached an average maximum of 377485 ticks during one hour of fuzzing. The manually constructed WCET was 379033 ticks, while the application code of the system only starts execution at 347870 ticks. All execution before that point consists of the initialization of the RTOS and the application tasks. This means the fuzzer found a case that causes the application to reach 95.0% of its WCET on average, while random fuzzing on average reaches about 56.4% of the worst-case during one hour of fuzzing. `afl_mapmax` reached an average maximum of 371465 ticks, which covers around 75.7% of the worst-case execution time. As Table 7.3 shows, the size of the corpus is massively increased for all fuzzers, which emphasizes the importance of the

Configuration	avg. corpus	σ_{corpus}
afl_mapmax	340.3	2.31
state	1365.3	70.65
graph	2392.3	111.46

Table 7.3: Average corpus size and its standard deviation for the three main configurations with an interrupt included in the input

corpus scheduler to prioritize the most promising inputs. The even worse performance of the `afl_queue` configuration confirms that the corpus scheduler for CFG edges does indeed increase performance, but was insignificant in the case without interrupts. The likely explanation for the performance degradation of the CFG edge-based fuzzers is the increased number of unique edges resulting from the interrupts. Inputs that interrupt the target in a new block produce new edges and thus create new interesting inputs. This observation is consistent with prior work on kernel fuzzing [33]. The same effect is not as pronounced for the graph and state-based fuzzers, because the implementation of the interrupt handler only activates a task that can diverge the state path in a limited number of states.

7.6 Scalability

Testing-based methods are generally not able to cover the full input space, which makes the fuzzing process open-ended and the question of scalability difficult to answer. The main limit to scalability of the fuzzers described in this thesis is the number of corpus elements, as storing many relatively uninteresting test cases slows the fuzzer down and consumes memory. While culling can be used to reduce it, the scheduling algorithms already focus on a subset of the corpus. The size of this preferred subset differs between fuzzing methods and still presents a scaling limit under culling. Tables 7.2 and 7.3 show only the gross corpus size but are still useful to reason about the scalability of the approaches. The edge-based fuzzer is not scalable when used in a scenario with enabled interrupts, due to the large number of edges generated by the interrupt handling. The corpus sizes of the other fuzzers increased less dramatically. Interrupts could increase the preferred subset in an edge-based fuzzer to about $O(WCET) + O(edges_{CFG})$ inputs, as the interrupt can cause a new edge to be created every few instructions. This effect can be seen in the growth of the average corpus size by a factor of around 42 for `afl_mapmax`, while both other fuzzers increased their corpus sizes by much smaller factors (30 for `graph` and 14 for `state`): The graph-based fuzzer as presented in this thesis limits its preferred set by $O(paths_{STG} * states_{STG})$, because all branches contain duplicate states. The system-path based fuzzer focuses on only $O(paths_{STG})$ in its corpus. Another possible coverage-based fuzzing approach would be to merge diverging branches of the STG and use the edges of

the resulting graph for coverage, which would scale even better with a preferred around $O(edges_{STG})$. Since this was not part of this thesis the best remaining method is based on the state path coverage, which scales with $O(paths_{STG})$.
System-aware static timing analyzers - while not directly comparable - also see the complexity of the analysis increase with the number of STG paths [1]. One of the key advantages of using a measurement-based analysis is that the number of paths to consider shrinks, as only feasible paths are encountered. This is similar to the scaling of system-aware static timing analysis methods, except they still need to consider some infeasible paths through the STG, while fuzzing discovers only feasible paths.
Overall the scaling potential of fuzzing for timing analysis is promising. Future work in system-aware fuzzing has the potential for improving scalability by merging STG branches, developing better prioritization heuristics and deploying well-known fuzzing strategies like culling of the corpus.

7.7 Searching Witness Inputs

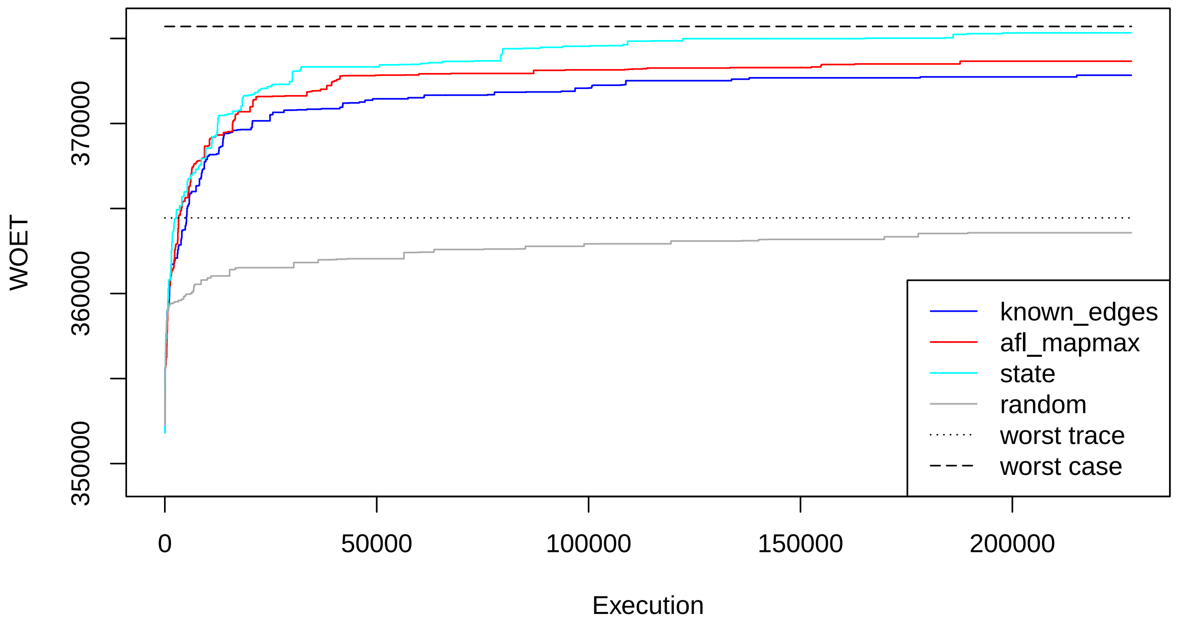


Figure 7.8: Comparison of fuzzing for known edges with other methods without interrupts

The last fuzzing approach presented in this thesis is focused on guiding the fuzzer using a feedback function that rewards decreasing differences between the observed CFG edges and ones produced using a hypothetical compatible static analysis. While there are a few static timing analysis methods available that analyze the whole system, they can not be used for comparison, as their output is not directly comparable to the edges measured using the emulator. Thus, the target edges for fuzzing are taken from the execution of the true worst-case input instead. The result was that this fuzzer variant did not find the worst-case input during the one-hour test. As seen in Figure 7.8 on average its generated inputs do not outperform the state-based fuzzer or the AFL-based one during fuzzing without interrupts. The corpus grew approximately as large as the AFL-based fuzzers during this test. This is not surprising, because most of the time both feedbacks reward increasing the number of times an edge gets executed unless the MSE decreases overall or the edge was not included in the worst case. During fuzzing with interrupts the corpus of the edge targeting fuzzer does increase very little, as the edges caused by the interrupt are generally not in the target set. This causes it to outperform the AFL-based one, as seen in Figure 7.7.

7.8 Resume

This section presented a small real-time system hosting a simple application that processes sensor input and is protected by triple modular redundancy. Multiple fuzzer configurations based on the concepts of this thesis were evaluated against this example, aiming to trigger multiple retries of the redundancy, while maximizing the time each replicates spends on the processing. The fuzzer using the coverage of system-state paths was the most effective one. Withing one hour of fuzzing it increased the observed execution times of the application in the system to over 98% of the worst-case in a scenario without interrupts and 95% with interrupts, compared to random testing which reached around 59% and 56% respectively. The fuzzer based on STG-node coverage was less effective and its specialized mutators did not show improvements over the base mutators either.
With these results the main questions of this thesis can be answered as follows: First off, repurposing existing coverage-guided fuzzing techniques is possible and can lead to high execution time test cases. In the tested cases, it produced much higher execution times on average than what random testing would produce within the same time. Secondly, fuzzing guided by STG paths proved to be even more effective than the CFG-edge-based one. When interrupts are activated the situation is similar, but the fuzzer based on CFG-edges performs much worse. For the last question about using the edge-frequencies provided by another analysis, guiding the fuzzer using known worst-case edge numbers was unable to find the worst-case witness and overall did not outperform the state-path-guided fuzzer.

Chapter 8

Discussion and Conclusion

This chapter summarizes the work and discusses its limitations and directions for future work.

8.1 Discussion

This thesis presents a system-state-aware fuzzing approach to generate inputs that cause running executions of a real-time system. The prototype which was evaluated in Chapter 7 used a simplified hardware model and made assumptions about interrupt handling, as well as implementation-specific assumptions about the system state. This Section discusses these assumptions and provides starting points for future research.
The main causes for increased execution time in this thesis were the overall control flow enabled by the input and preemption caused by interrupts. Since the execution model was simplified, it did not consider hardware effects, such as caches and pipelines. Hardware effects can be included in the future by using a cycle-accurate emulator of a target machine. This would likely sacrifice emulation performance and potentially require re-writing code for basic block analysis, which was implicitly performed in QEMU by the separation in translation blocks. Modeling of caches would also open new opportunities for graph-based mutators. A mutator could focus on inserting an interrupt at a point where it will cause the current task to be preempted due to lower priority. This would be a useful heuristic to increase execution times, as preemption of the current task causes cache eviction of its working set, which needs to get re-loaded at a later time to complete the preempted task. As discussed in 3, the point of preemption (or migration) in code is important because the active working-set size changes over the lifetime of a task. This could also be incorporated into a more advanced heuristic that uses knowledge of the evolution of a task's working-set size. It may move an interrupt that causes a preemption in the execution of a low-priority task at the point where its active working set is at its maximum, to cause the largest preemption-overhead.

The graph-based fuzzer did not outperform the simpler state trace-based fuzzer. This might be caused by duplication of states between branches, which introduces more nodes for test cases to cover that are ultimately on the wrong branches. It is possible to merge the graph and fuzz over edges instead, which should increase the scalability and bring it close to CFG-based coverage-guided fuzzing.

In the graph settings, the mutators also did not show advantages, which might be down to multiple reasons. On one hand, some mutators are focussed on exploiting relative advantages. Without state merging relative advantages within a single trace are limited. On the other hand, a possible explanation is that the MOPT scheduler might decrease the probability of these mutators being used due to low initial success.

The STG is already used for a simplified taint analysis, which relies on the assumption that the behavior of each state block is mostly determined by the last part of the input which was read. This assumption can be eliminated in future work if real taint analysis using symbolic execution is used.

Symbolic execution can also be used for concolic fuzzing. Concolic fuzzing would allow deriving new state paths by solving SMT formulas about the branch predicates. In aggregate, such a system could be classified as a hybrid timing analysis, but instead of the established pipeline of static timing analysis, supplemented by timing information from testing this would do the reverse. Like any approach which attempts to cover all CFG paths, it would still run into similar scalability problems as full static timing analysis.

Another aspect left out is the portability between different RTOSes. The current prototype is focused on FreeRTOS. The changes made inside it to accommodate the emulated environment, namely the way inputs are consumed from memory that is written by the fuzzer, should be very portable to other library OSes, as long as their compiled kernels contain ELF-symbols to find the input fields for the fuzzer. More complex systems like Linux with user-space and protected mode separation would require a different mechanism to insert input data, using virtual devices emulated using QEMU for example. The changes necessary to the emulator instrumentation are more complex, as the OS-specific data structures are not portable. They require custom code to extract the state per target OS. Generating struct bindings from the layout information contained in debugging information adhering to the DWARF standard should also be possible. This solution would be more portable than the prototype implementation, which requires the generation of struct definitions and manual post-processing of them to fit the host memory layout.

As mentioned in Chapter 7, CFG edges are created when interrupt handlers are triggered. Other kernel-focussed fuzzers have encountered similar problems [33], their workaround could be implemented similarly to how the detection of application against kernel code works. Checking if edges lead to the interrupt handler could exclude those edges.

8.2 Summary

This thesis proposed three techniques to leverage coverage-guided fuzzing to find inputs with long execution times for real-time systems. The first one is based on the common control flow edge coverage popularized by AFL. The second one traces the system state of the target system, which consists of many system facts, such as the active and ready tasks, their priorities, notification states and the number of held mutexes. The fuzzer uses coverage over these traces and execution times to determine if an input is interesting and to select the next input for mutation. The third fuzzer conceptualized goes one step further and uses the state traces to build a state transition graph. It then uses coverage of nodes (states) in the graph and additionally uses information from the states during mutation. At last one non-coverage-based fuzzer was proposed in this thesis, which uses pre-determined target control flow edges from a (theoretical) IPET analysis to search a witness input for said analysis.
The concepts were implemented using the modular fuzzer library LibAFL and its fork of QEMU. To host a real-time system the instrumentation in QEMU was changed from user-mode to system-mode with the state reset between runs being handled using QEMU's snapshot functions. Additionally, support for injecting interrupts based on precise time inputs was implemented.
The evaluation was performed against a small FreeRTOS system running multiple tasks which implement a simple workflow. In this case, triple modular redundant processing of some data being part of the fuzzing input. The results from testing showed that in absence of interrupt handling all three base fuzzers performed well in the sense that they quickly produced inputs that reached close to the worst-case execution time while performing significantly better than testing of random inputs. With enabled interrupt handling the performance of the edge-based fuzzer decreased dramatically. The most likely explanation is the increased number of edges produced by the interrupt handler, which is consistent with observations made by other researchers focussing on fuzzing kernels [33]. The overall best-performing fuzzer was the one based on state traces, instead of the state transition graph. Edges resulting from executing the known worst-case input were used for evaluating the fuzzer targeting known edges. The fuzzer was unable to find the worst-case input and did not even find worse inputs than the coverage-based fuzzers.
The main questions of this thesis were whether coverage-guided fuzzing can be effective in searching long executing inputs for a real-time system and whether this can be improved using system state-based techniques. The results show that both of them can be answered positively. However, an efficient search for witness inputs using edge frequencies from a theoretical IPET analysis could not be demonstrated.

8.3 Future Work

Since the results of the fuzzers turned out positive, an obvious next step is to move the execution mode closer to real hardware. This could be achieved by exchanging QEMU for a cycle-accurate emulator of the target system or even real hardware. This would make the results more comparable to existing MBTA tools.
Another direction would be to improve the system state analysis. The coverage of STG nodes did not produce the anticipated improvement over system state traces, despite the finer granularity it offers. Merging graph nodes from different branches and using coverage of STG edges would shrink the graph and increase the similarity with CFG coverage for potentially better guidance and scalability.
The wider field of fuzzing research has many techniques to offer which might be useful for timing analysis. Concolic fuzzing for example is a white-box technique that leverages symbolic execution and an SMT solver to construct inputs that exercise new control flow paths. This might also be applicable on a system-wide level to exercise new STG paths. Other grey-box techniques such as deeper taint analysis might also be promising, as they are less complex than white-box techniques, yet effective.

List of Figures

Bibliography

[1] C. Dietrich, P. Wägemann, P. Ulbrich, and D. Lohmann, "Syswcet: Whole-system response-time analysis for fixed-priority real-time systems (outstanding paper)", in *Proceedings of the 23nd IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, IEEE, 2017, pp. 37–48.

[2] R. Wilhelm, J. Engblom, A. Ermedahl, *et al.*, "The worst-case execution-time problem—overview of methods and survey of tools", *ACM Transactions on Embedded Computing Systems*, vol. 7, no. 3, 36:1–36:53, May 8, 2008, ISSN: 1539-9087. DOI: `10.1145/1347375.1347389`.

[3] F. Mueller and J. Wegener, "A Comparison of Static Analysis and Evolutionary Testing for the Verification of Timing Constraints", *Real-Time Systems*, vol. 21, no. 3, pp. 241–268, 2001, ISSN: 0922-6443. DOI: `10.1023/A:1011132221066`.

[4] U. Khan and I. Bate, "WCET analysis of modern processors using multi-criteria optimisation", in *Proceedings of the 1st International Symposium on Search Based Software Engineering*, IEEE, 2009, pp. 103–112.

[5] M. Zalewski, *American fuzzy lop*, version 2.57b, Google, Jun. 30, 2020. [Online]. Available: `https://github.com/google/AFL` (visited on 05/25/2022).

[6] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++ : Combining Incremental Steps of Fuzzing Research", in *Proceedings of the 14th USENIX Workshop on Offensive Technologies (WOOT '20)*, USENIX Association, Aug. 2020. [Online]. Available: `https://www.usenix.org/conference/woot20/presentation/fioraldi`.

[7] J. Schneider, "Why You Can't Analyze RTOSs without Considering Applications and Vice", Nov. 24, 2002.

[8] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: State of the Art", *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 1199–1218, Sep. 2018, ISSN: 1558-1721. DOI: `10.1109/TR.2018.2834476`.

[9] F. Bellard and multiple others, *QEMU*. [Online]. Available: `https://www.qemu.org/` (visited on 05/25/2022).

[10] A. Fioraldi and D. Maier. “The LibAFL book”. (May 12, 2022), [Online]. Available: `https://aflplus.plus/libafl-book/libafl.html` (visited on 05/12/2022).

[11] M. T. Yourst, “PTLsim: A cycle accurate full system x86-64 microarchitectural simulator”, in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems & Software*, IEEE, 2007, pp. 23–34.

[12] K. C. Wang, “Embedded Real-Time Operating Systems”, in *Embedded and Real-Time Operating Systems*, K. Wang, Ed., Cham: Springer International Publishing, 2017, pp. 401–475, ISBN: 978-3-319-51517-5. DOI: `10.1007/978-3-319-51517-5_10`.

[13] J. W. S. Liu, *Real-time Systems*. Englewood Cliffs, NJ, USA: Prentice Hall PTR, 2000, ISBN: 0-13-099651-3.

[14] S. K. Baruah, A. K. Mok, and L. E. Rosier, “Preemptively scheduling hard-real-time sporadic tasks on one processor”, pp. 182–190, Dec. 1990. DOI: `10.1109/REAL.1990.128746`.

[15] F. Reghenzani, G. Massari, and W. Fornaciari, “The Real-Time Linux Kernel”, *ACM Computing Surveys (CSUR)*, Feb. 21, 2019. DOI: `10.1145/3297714`. (visited on 07/02/2022).

[16] Amazon Web Servcies. “FreeRTOS - Market leading RTOS for embedded systems.”, FreeRTOS. (May 11, 2022), [Online]. Available: `https://www.freertos.org/index.html` (visited on 05/11/2022).

[17] F. E. Allen, “Control flow analysis”, *SIGPLAN Not.*, vol. 5, no. 7, pp. 1–19, Jul. 1970, ISSN: 0362-1340. DOI: `10.1145/390013.808479`. [Online]. Available: `https://doi.org/10.1145/390013.808479`.

[18] C. Dietrich, M. Hoffmann, and D. Lohmann, “Cross-Kernel Control-Flow–Graph Analysis for Event-Driven Real-Time Systems”, *ACM SIGPLAN Notices*, Jun. 4, 2015. DOI: `10.1145/2808704.2754963`.

[19] P. Puschner, “Zeitanalyse von Echtzeitprogrammen”, Ph.D. dissertation, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 1993.

[20] S. Edgar and A. Burns, “Statistical analysis of wcet for scheduling”, in *Proceedings of the 11nd IEEE Real-Time Systems Symposium (RTSS '01)*, Dec. 2001, pp. 215–224. DOI: `10.1109/REAL.2001.990614`.

[21] C. Ferdinand and R. Heckmann, “aiT: Worst-Case Execution Time Prediction by Static Program Analysis”, in *Building the Information Society*, ser. IFIP International Federation for Information Processing, R. Jacquart, Ed., vol. 156, Boston, MA: Springer US, 2004, pp. 377–383, ISBN: 978-1-4020-8157-6. DOI: `10.1007/978-1-4020-8157-6_29`.

[22] Y.-T. S. Li and S. Malik, "Performance Analysis of Embedded Software Using Implicit Path Enumeration", in *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, &amp; Tools for Real-Time Systems*, (La Jolla, California, USA), ser. LCTES '95, New York, NY, USA: Association for Computing Machinery, 1995, pp. 88–98, ISBN: 978-1-4503-7308-1. DOI: 10.1145/216636.216666.

[23] D. Kästner, M. Pister, S. Wegener, and C. Ferdinand, "TimeWeaver: A Tool for Hybrid Worst-Case Execution Time Analysis", in *Proceedings of the 19th International Workshop on Worst-Case Execution Time Analysis (WCET 2019)*, S. Altmeyer, Ed., ser. OpenAccess Series in Informatics (OASIcs), vol. 72, Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2019, 1:1–1:11, ISBN: 978-3-95977-118-4. DOI: 10.4230/OASIcs.WCET.2019.1.

[24] S. Schuster, P. Wägemann, P. Ulbrich, and W. Schröder-Preikschat, "Proving real-time capability of generic operating systems by system-aware timing analysis", in *Proceedings of the 25th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '19)*, IEEE, 2019, pp. 318–330.

[25] S. Schuster, P. Wägemann, P. Ulbrich, and W. Schröder-Preikschat, "Annotate once – analyze anywhere: Context-aware WCET analysis by user-defined abstractions", in *Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, New York, NY, USA: Association for Computing Machinery, Jun. 22, 2021, pp. 54–66, ISBN: 978-1-4503-8472-8. DOI: 10.1145/3461648.3463847.

[26] J. M. Calandrino, H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson, "LITMUSˆRT : A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers", in *Proceedings of the 27th IEEE International Real-Time Systems Symposium (RTSS '06)*, Dec. 2006, pp. 111–126. DOI: 10.1109/RTSS.2006.27.

[27] P. Denning, "Working sets past and present", *IEEE Transactions on Software Engineering*, vol. SE-6, no. 1, pp. 64–84, 1980. DOI: 10.1109/TSE.1980.230464.

[28] J. A. Brown, L. Porter, and D. M. Tullsen, "Fast thread migration via cache working set prediction", in *IEEE 17th International Symposium on High Performance Computer Architecture*, 2011, pp. 193–204. DOI: 10.1109/HPCA.2011.5749728.

[29] W. Lunniss, S. Altmeyer, C. Maiza, and R. I. Davis, "Integrating cache related preemption delay analysis into edf scheduling", in *Proceedings of the 19th Real-Time and Embedded Technology and Applications Symposium (RTAS '13)*, 2013, pp. 75–84. DOI: 10.1109/RTAS.2013.6531081.

[30] P. Raffeck, P. Ulbrich, and W. Schröder-Preikschat, "Work-in-progress: Migration hints in real-time operating systems", in *Proceedings of the 40th IEEE International Real-Time Systems Symposium (RTSS '19)*, IEEE, 2019, pp. 528–531.

[31] T. Klaus, P. Ulbrich, P. Raffeck, *et al.*, “Boosting Job-Level Migration by Static Analysis”, in *Proceedings of the 15th Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT ’19)*, D. Lohmann and A. Lackorzynski, Eds., Jul. 2019, pp. 17–22.

[32] P. Rafeck, W. Schröder-Preikschat, and P. Ulbrich, “Revisiting Migration Overheads in Real-Time Systems: One Look at Not-So-Uniform Platforms”, in *Proceedings of the 16th Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT ’22)*, D. Lohmann and R. Mancuso, Eds., Jul. 2022, pp. 41–48.

[33] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, “kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels”, in *Proceedings of the 26th USENIX Security Symposium (USENIX Security ’17)*, Vancouver, BC: USENIX Association, Aug. 2017, pp. 167–182, ISBN: 978-1-931971-40-9. [Online]. Available: `https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/schumilo`.

[34] S. Schumilo, C. Aschermann, A. Abbasi, S. Wörner, and T. Holz, “Nyx: Greybox hypervisor fuzzing using fast snapshots and affine types”, in *Proceedings of the 30th USENIX Security Symposium (USENIX Security ’21)*, 2021. [Online]. Available: `https://www.usenix.org/conference/usenixsecurity21/presentation/schumilo`.

[35] C. Lyu, S. Ji, C. Zhang, *et al.*, “MOPT: Optimized Mutation Scheduling for Fuzzers”, presented at the 28th USENIX Security Symposium (USENIX Security 19), 2019, pp. 1949–1966, ISBN: 978-1-939133-06-9.

[36] D. Maler, A. Floraldi, D. Zhang, *et al.*, *LibAFL, the fuzzer library.* Advanced Fuzzing League ++, May 11, 2022. [Online]. Available: `https://github.com/AFLplusplus/LibAFL` (visited on 05/12/2022).

Eidesstattliche Versicherung

(Affidavit)

Berger, Alwin	192971
Name, Vorname (surname, first name)	Matrikelnummer (student ID number)

☐ Bachelorarbeit
(Bachelor's thesis)

[✓ Masterarbeit]
(Master's thesis)

Titel
(Title)

Emulator-based Fuzzing of Operating-system State-transition Graphs

Ich versichere hiermit an Eides statt, dass ich die vorliegende Abschlussarbeit mit dem oben genannten Titel selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

I declare in lieu of oath that I have completed the present thesis with the above-mentioned title independently and without any unauthorized assistance. I have not used any other sources or aids than the ones listed and have documented quotations and paraphrases as such. The thesis in its current or similar version has not been submitted to an auditing institution before.

Dortmund, 11.07.22	A. B. Berger
Ort, Datum (place, date)	Unterschrift (signature)

Belehrung:
Wer vorsätzlich gegen eine die Täuschung über Prüfungsleistungen betreffende Regelung einer Hochschulprüfungsordnung verstößt, handelt ordnungswidrig. Die Ordnungswidrigkeit kann mit einer Geldbuße von bis zu 50.000,00 € geahndet werden. Zuständige Verwaltungsbehörde für die Verfolgung und Ahndung von Ordnungswidrigkeiten ist der Kanzler/die Kanzlerin der Technischen Universität Dortmund. Im Falle eines mehrfachen oder sonstigen schwerwiegenden Täuschungsversuches kann der Prüfling zudem exmatrikuliert werden. (§ 63 Abs. 5 Hochschulgesetz - HG -).

Die Abgabe einer falschen Versicherung an Eides statt wird mit Freiheitsstrafe bis zu 3 Jahren oder mit Geldstrafe bestraft.

Die Technische Universität Dortmund wird ggf. elektronische Vergleichswerkzeuge (wie z.B. die Software „turnitin") zur Überprüfung von Ordnungswidrigkeiten in Prüfungsverfahren nutzen.

Die oben stehende Belehrung habe ich zur Kenntnis genommen:

Official notification:
Any person who intentionally breaches any regulation of university examination regulations relating to deception in examination performance is acting improperly. This offense can be punished with a fine of up to EUR 50,000.00. The competent administrative authority for the pursuit and prosecution of offenses of this type is the Chancellor of TU Dortmund University. In the case of multiple or other serious attempts at deception, the examinee can also be unenrolled, Section 63 (5) North Rhine-Westphalia Higher Education Act (*Hochschulgesetz, HG*).

The submission of a false affidavit will be punished with a prison sentence of up to three years or a fine.

As may be necessary, TU Dortmund University will make use of electronic plagiarism-prevention tools (e.g. the "turnitin" service) in order to monitor violations during the examination procedures.

I have taken note of the above official notification:*

Dortmund, 11.07.22	A. B. Berger
Ort, Datum (place, date)	Unterschrift (signature)

***Please be aware that solely the German version of the affidavit ("Eidesstattliche Versicherung") for the Bachelor's/ Master's thesis is the official and legally binding version.**