Computer Science Department

B.Sc. Computer Science
Computer Science Department

ARMv8 Snapshot Fuzzing

Candidate

Anna Groza

Student ID 700061570

Supervisor

Achim D. Brucker

University of Exeter

Academic Year
2022/2023

Abstract

Fuzz Testing is a widely recognized method for identifying defects in software. However, the majority of available fuzzing tools necessitate access to the source code of the program being tested. This presents a challenge when dealing with closed-source software. Extensive research has been conducted on binary-only fuzzing, with some of the most effective solutions targeting x86 CPUs. As part of my dissertation, I developed a high-performance black box fuzzer compatible with ARMv8 targets.

	Yes	No
I certify that all material in this dissertation which is not my own work has been identified.	☑	☐
I give the permission to the Department of Computer Science of the University of Exeter to include this manuscript in the institutional repository, exclusively for academic purposes.	☑	☐

Contents

List of Figures

List of Tables

List of Algorithms

List of Code Snippets

List of Acronyms

AFL American Fuzzy Lop

ASAN Address Sanitizer

VM Virtual Machine

PML Intel's Page Modification Logging

PT Intel Processor Trace

FDT Flattened Device Tree

1 Background context

1.1 Introduction

The widespread adoption of computing technology in recent years has brought about numerous benefits, but also poses challenges in terms of reliability and security. The criticality of these systems, many of which underpin essential infrastructure and hold sensitive information, makes the issue of security particularly pressing. Despite significant advances in the field over the last quarter-century, vulnerabilities in computer systems remain a concern.

One significant contributor to this problem is the use of non-memory-safe programming languages, such as C and C++, for the development of widely-used software. Errors in code written in these languages can result in memory corruption bugs, which in turn can lead to serious security vulnerabilities, including the potential for sensitive data leakage and arbitrary code execution[30][24][34].

To this day, memory corruption is still responsible for 70% of the vulnerabilities found in Microsoft products, despite decades of engineering effort invested into finding these bugs and making them harder to exploit[33].

The distribution of vulnerable code to users can have dire consequences. There is a substantial market for these vulnerabilities[37], which are often bundled with surveillance software and sold to governments that abuse them. These vulnerabilities have been linked to the surveillance of journalists[18] and human rights activists[15], spying on political opponents[13], and the persecution of Uighurs in China[1].

It is thus important to find and fix these bugs before they can pose a danger to users. Fuzzing is a technique that can be used to discover these bugs, and it has been demonstrated to be effective. The Google OSS-Fuzz project is one example of this, having identified over 40,500 bugs in 650 widely used open-source projects[25].

1.2 An overview of Fuzzing Research

Fuzzing, a software testing technique introduced in 1990, involves feeding a program or function random or mutated inputs and then checking for crashes during input processing[19]. Initially, this technique used random bytes as inputs, but this naive implementation had a low probability of discovering bugs that could only be triggered by a specific sequence of conditions. However, 30 years of research have significantly improved the effectiveness of fuzzing tools.

In order to provide a more concise overview of the relevant research, this review will focus on the studies that had a significant impact on the field.

1.2.1 Coverage Guided Fuzzing

Coverage-guided fuzzing is a software testing technique that uses feedback from the program being tested to guide the exploration of the input space. This technique is used to identify potential vulnerabilities or bugs in the program by providing the fuzzer with information about which inputs are interesting [17].

One well-known implementation of coverage-guided fuzzing is American Fuzzy Lop (AFL), which was developed in 2013 and demonstrated the potential of this approach. AFL uses a modified version of the GCC compiler to instrument the target code, adding instructions that measure which code is executed and sending this information back to the fuzzer [32].

Instrumentation is the process of adding new code to a program without altering its behaviour. In the case of AFL, some code is added before each basic block to track the execution of the program.

A **basic block** is a group of instructions in a program that are executed one after the other, without any branching or jumping to other parts of the code [11].

Code coverage is a useful metric for measuring the performance of a fuzzer, as it indicates the proportion of the program's code that has been executed during testing. By focusing on inputs that maximize code coverage, coverage-guided fuzzers can effectively identify potential vulnerabilities or bugs in the program being tested as "There is a very strong correlation between the coverage achieved and the number of bugs found by a fuzzer" [8].

Algorithm 1 An algorithm to guide fuzzers with code coverage

```
corpus ← {...}
while true do
  newinput ← mutate(corpus) {Mutate a random element of the corpus}
  newcov ← execute(newinput) {Execute with new input and measure code coverage}
  if newcov reaches new code then
    corpus ← corpus+newinput {Save inputs that execute new code}
  end if
end while
```

The fuzzer maintains a corpus of known inputs, which can be either empty or initialized with known valid inputs. For each execution, an element of the corpus is picked and mutated,

and the resulting input is passed to the program. If the input causes new code to be executed, it is considered interesting and added to the corpus for future mutations. This genetic algorithm allows the fuzzer to explore the input space much more quickly than naive implementations [11].

1.2.2 Address Sanitizer

Address Sanitizer (ASAN) is a feature that is included in most C compilers and is designed to detect memory corruption bugs that might not cause the program to crash. ASAN uses shadow memory to store metadata about which memory regions are accessible. The program instruments every read and write to memory and checks if that address is accessible according to the shadow memory[29].

```
1  char a[10]; // Global variable
2
3  int main() {
4      char b[10]; // Stack variable
5      char* c = (char*)malloc(10); // Heap allocation
6
7      a[12] = 1; // Crash: OOB write to global
8      b[12] = 1; // Crash: OOB write to stack
9      c[12] = 1; // Crash: OOB write to heap
10     a[0]=c[12];// Crash: OOB read on the heap
11
12     free(c); // Free the heap alloc
13
14     c[0]; // Crash: Use after free
15 }
```

Code 1.1: Types of bug caught by Address Sanitizer

To detect out-of-bounds accesses in stack and global variables, ASAN marks the memory around them as non-accessible. These non-accessible areas of memory next to the variable are called *redzones*, and they will cause the program to crash if any read/write operation is performed on them [29].

To detect memory corruption on the heap, ASAN replaces the standard memory allocator with one that also updates the shadow memory metadata. When the *malloc* function is called, a region of memory with the requested size is marked as accessible and surrounded by a *redzone*. When an allocation is freed, all accessible bytes of the allocation are marked as non-accessible. This allows ASAN to identify most types of memory corruption [29].

Instrumenting the binary in this way results in a 2x slowdown[2], but this overhead is considered worthwhile to detect otherwise-undetectable memory corruption bugs.

1.2.3 CmpLog

CmpLog is a technique used to increase the effectiveness of fuzzing by providing the fuzzer with additional information about the comparisons performed by the program at runtime [26].

This instrumentation pass aims to make it easier for the fuzzer to pass hard comparisons such as magic numbers and checksums[10].

CmpLog is implemented by inserting a call to a function before each comparison performed by the program. It also replaces some commonly-used comparison functions in C (e.g. *memcmp*, *strcmp*) with equivalents that also log the compared data [10].

The function called on every compare stores the values of both operands and the address at which the comparison was performed. This data is extremely valuable when used in conjunction with other analysis algorithms [5].

1.2.4 Limitations of current fuzzers

Modern fuzzing techniques are effective at finding bugs in some types of code, but they often struggle to achieve high code coverage for stateful targets that require specific sequences of inputs over time in order to reach bugs [28]. This is a common challenge for many critical attack surfaces, such as kernels, browsers, and network protocols[27].

Most existing instrumentation techniques for fuzzing require the binary to be compiled with a custom compiler, which makes it difficult to find bugs in closed-source software [22].

However, by far the biggest issue with our current approach to fuzzing is discussed in the paper "Fuzzing: On the Exponential Cost of Vulnerability Discovery" [7].

"Our first empirical law suggests that using a non-deterministic fuzzer (a) given the same time budget, each new vulnerability requires exponentially more machines and (b) given the same number of machines, each new vulnerability requires exponentially more time" [7].

This means that fuzzers become exponentially less effective at finding new vulnerabilities as more code is explored [7].

This paper also states that when an application is fuzzed for the first time there is a higher chance of finding security-relevant bugs [7][27]. Therefore creating fuzzing tools that are able to test previously untestable code is more likely to result in discovering bugs. the subsequent section, we will explore a technique that enables testing a broader range of codebases. Conducting further research to improve instrumentation and input mutation techniques is beneficial. However, it is equally crucial to prioritize the development of fuzzers with high performance and compatibility. Historically, the field has not given much attention to these aspects, but this has started to change in recent times with the introduction of snapshot fuzzing.

2

Snapshot fuzzing

2.1 Introduction

Snapshot fuzzing is a relatively new research area in the field of fuzzing that aims to address some of the limitations of existing fuzzing techniques. This approach makes it possible to fuzz almost any code compiled for the same architecture as the fuzzer, regardless of whether the source code is available or whether the binary has been instrumented[28].

Using a snapshot fuzzer typically requires taking a copy of the memory and registers in the target system immediately before the input is processed. The easiest way to save this data is using a debugger, which generates a coredump file containing the full state of the program [16]. We can obtain a coredump for our target by attaching a debugger to the process or the kernel, setting a breakpoint before the input processing, and then saving a coredump at that point [12].

This coredump is loaded into a VM and used as the starting snapshot for the fuzzing process, allowing the fuzzer to mutate the input and continue execution without restarting the target program for each test [28]. After the input has been processed, the VM is reset to the original snapshot state, and another input is processed.

One example of this type of fuzzer is Nyx, which "has been shown to improve test throughput by up to 300x and coverage found by up to 70% compared to traditional fuzzers"[27]. This very significant increase in performance is likely to result in the discovery of more bugs while using fewer computing resources.

Snapshot fuzzers can capture and resume the complete execution of an operating system. In order to concurrently run multiple instances of the fuzzer, it is necessary to execute the captured snapshot within a Virtual Machine (VM). For this purpose, a hypervisor is required to use these fuzzers efficiently.

A **hypervisor** is a software program that enables multiple operating systems to run on a single physical host computer. It provides a layer of abstraction between the physical hardware and the virtual machines running on top of it. Emulated devices are virtual versions of physical

hardware components created by the hypervisor to provide virtual machines with the same functionality and capabilities as their physical counterparts[28].

Snapshot fuzzers are typically implemented by modifying existing hypervisors to include the capability of quickly resetting the state of a virtual machine and collecting coverage data on the code executed by it [23][35].

As a result, snapshot fuzzers have the capability to test any target that can run within a virtual machine, even including other hypervisors [28]. This makes it possible to fuzz complex, multi-layered systems that would be challenging to test with traditional fuzzers. Additionally, snapshot fuzzing can dramatically improve fuzzing performance for targets with high startup costs, such as web browsers. For example, Mozilla uses Nyx to fuzz the Firefox sandbox[20][27].

2.1.1 Current implementations

To better understand how snapshot fuzzers work at a low level, We read the code and used several open-source snapshot fuzzers, including Nyx[23], What The Fuzz(WTF)[35] and Tartiflette[31].

One of the key challenges of snapshot fuzzing is the need to quickly restore the virtual machine's state at the end of each fuzzing iteration. To do this, three components of the VM must be reset: the registers, the memory, and the emulated devices (e.g., hard disks, keyboards, and screens)[28].

Although the three aforementioned fuzzers have similar implementations, comprehending their differences can illustrate the tradeoffs that must be considered when implementing a snapshot fuzzer.

Resetting the registers is typically accomplished by writing all their values to memory at the start of a fuzz cycle and then reading them back after the cycle is complete.

Fast Memory Resets are implemented by using Intel's Page Modification Logging (PML) extension [6]. This CPU feature pushes every page of memory that has been modified onto a stack.

During the fuzzing iteration, the snapshot fuzzer enables PML and allows the target to run. After the execution is completed, the fuzzer needs to iterate over all the modified pages on the stack and restore their original content by copying it from the initial snapshot. This allows us to restore only a few pages after each execution instead of the entire memory [28].

Where these fuzzers start to diverge is in how they gather coverage on their targets. There are many different ways to gather coverage information, and each snapshot fuzzer may implement a different technique depending on its specific requirements. Some of the most commonly used techniques include:

- **Intel PT Code Coverage**: Intel Processor Trace (PT) is a hardware feature capable of collecting information about the code being executed by the CPU. This collected data is then stored in memory as highly compressed packets.

  While the performance overhead of Intel PT during execution is relatively low, decoding the compressed packets requires a significant amount of CPU resources. The impact of using Intel PT for tracing varies depending on the specific implementation of the packet

decoding process. In most cases, however, the use of Intel PT results in a performance slowdown of between 20% to 50%[9][22].

- **Breakpoints Code Coverage**: WTF and Tartiflette gather code coverage by implementing a lightweight debugger in the hypervisor. Before the target is executed, a disassembler is used to identify all the basic blocks in the target. This list is then used to replace the first instruction of every basic block with an int3 instruction (a software breakpoint on x86)[14].

  Every time a breakpoint instruction is executed, the debugger in the hypervisor receives an interrupt. When the hypervisor receives the interrupt, it marks the basic block as executed, replaces the breakpoint with the original instruction, and continues execution. This technique has a high-performance penalty the first time a basic block is executed due to the overhead of context switches, but it has no penalty for subsequent executions[21].

  This method of gathering code coverage[21] is slower and produces less information on the target compared to other techniques, but it has the advantage of working on non-Intel CPUs and being easy to implement.

Another area to consider when implementing a snapshot fuzzer is how to handle interactions between the tested program and the operating system or hardware. In general, the best approach will depend on the specific requirements of the target being tested and the trade-offs between simplicity, performance, and test coverage.

2.2 ARM Snapshot Fuzzers

Since all currently available snapshot fuzzers only support x86 targets, implementing support for ARM targets would enable testing of previously untestable systems. This is particularly useful given the widespread use of ARM CPUs in mobile and embedded computing. The ability to test binary-only targets compiled for ARM would greatly facilitate the discovery of some critical vulnerabilities[36].

A snapshot fuzzer for ARM CPUs must implement the same features as an x86 snapshot fuzzer (memory, device, and register resets) without using Intel-specific CPU features.

2.2.1 ARM CPU Fragmentation

In contrast to Intel and AMD, ARM does not engage in the actual manufacturing of CPUs; rather, it licenses its designs to other manufacturers, who can customize and manufacture them. While this is a central reason why ARM has gained such widespread popularity, it has also resulted in significant feature fragmentation within their CPUs. The decision of CPU designers to incorporate specific ARM features is optional, and not including them generally results in lower production costs. Consequently, while ARM may possess features that are equivalent to those previously described, these are frequently not implemented in hardware.

Furthermore, each CPU manufacturer must specify the included features in a device tree, which can sometimes be inaccurate. CPU manufacturers also frequently modify kernels to support platform-specific features, but this practice often results in the use of outdated kernels.

Even in cases where CPU features are implemented, they are often slower and more difficult to use compared to their x86 counterparts. For instance, CoreSight experiences a 50% performance slowdown when executing on ARM CPUs, compared to a 25% slowdown on Intel CPUs when parsing PT traces[4][9].

2.2.2 Design decisions

Following the evaluation of various development platforms, we opted to deploy the fuzzer on an M1 Mac Mini that runs Asahi Linux. This decision was informed by the fact that Asahi Linux employs a cutting-edge kernel and supports all features implemented by the CPU. Additionally, it provides the potential to execute Apple-specific code in the future, which can leverage certain CPU features unique to Apple and not available on other devices. Due to the previously discussed issues with CPU Fragmentation, We've decided to avoid using any optional ARM CPU features to maximise the number of platforms my fuzzer can run on.

2.2.3 Success Criteria

The goal of this project is to make it possible to test previously un-fuzzable targets. Success in this project will be evaluated by the ability to test new targets, or outperform existing fuzzers, such as AFL++, in terms of the number of test cases that can be executed in a similar timeframe. In either case, this project is a useful resource for developing more advanced ARM snapshot fuzzers in the future, and due to my design choices, it opens the possibility of running Android or XNU on it in the future.

3

Implementing an ARM Snapshot fuzzer

To implement a snapshot fuzzer, it is essential to first develop a virtual machine monitor. The fuzzer was constructed on top of KVM to ensure maximum compatibility, enabling it to operate on any Linux-based ARM device that supports virtualization.

3.1 Tardis

In the following section, I will expand upon the design decisions and operational aspects of my fuzzer, aptly named Tardis. The name is an allusion to the iconic phone box in the TV series Doctor Who. It was chosen because the tool, akin to the Tardis, facilitates time travel for virtual machines.

3.1.1 Rust-VMM

The fuzzer which I have developed is built upon the Rust-VMM framework. Rust-VMM comprises a collection of Rust libraries that facilitate the creation of Virtual Machine Monitors. Initially created by Amazon for AWS Lambda, it has now secured extensive support from prominent industry players like Google and Intel.

This decision was made based on several factors, including Rust-VMM's established support for fast virtual machine (VM) migrations - a process of moving a running VM from one physical host to another, a well-defined API for serializing device states, and thoroughly tested support for Arm virtual machines. Notably, Google utilizes Rust-VMM to virtualize Android, and thus employing these libraries would simplify future Android support.

3.2 Booting and Snapshotting Linux

3.2.1 Booting Linux

To begin fuzzing within the virtual machine, it is essential to first boot an operating system within it and establish a mechanism for virtual machine snapshots. To achieve this, we followed the Linux guide on booting the operating system on ARM CPUs, and the process is depicted in the figure below.

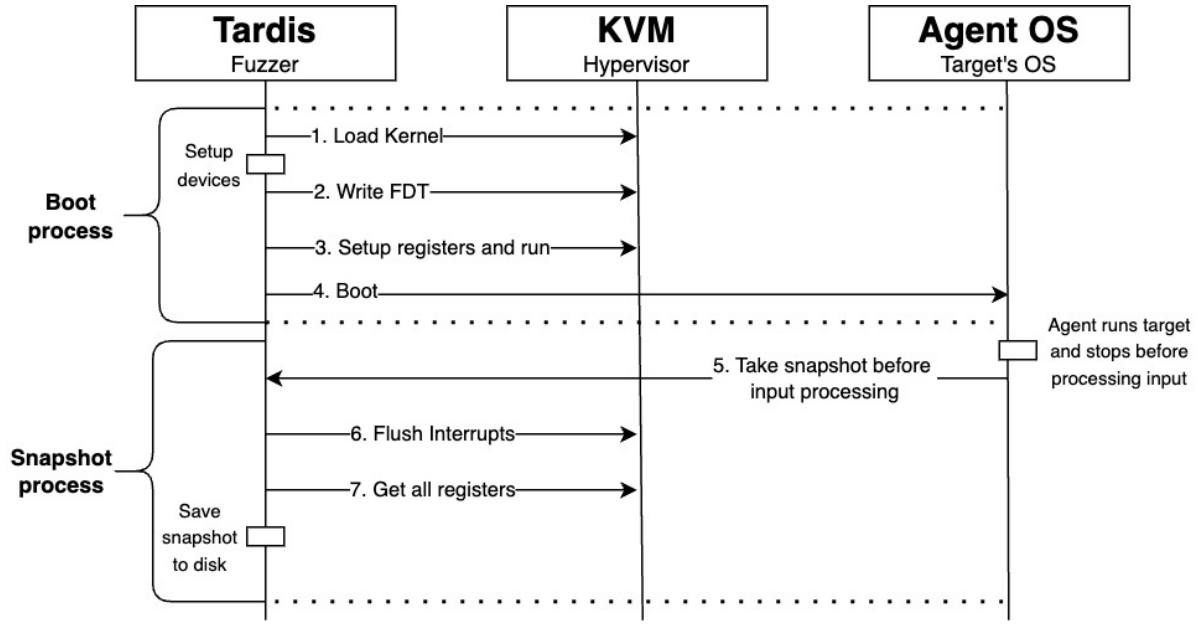

Figure 3.1: Overview of the boot and snapshotting process in Tardis and its various components

The process commences with the creation of a new virtual machine in KVM, followed by the allocation of memory for said virtual machine, and the transfer of the Linux kernel into it. In order to enable the booting of Linux with console output, we set up a few devices, such as an interrupt controller, a serial console, and a clock. Although additional devices could provide more functionality, such as networking or a separate disk, we kept the initial devices to a minimum to reduce complexity.

Regrettably, due to the absence of a disk device, we add files within the virtual machine by recompiling the Linux image to include the files in its initial ramdisk, which is utilized during boot. This method increases the complexity of adding new targets to the fuzzer. Nonetheless, the inclusion of a disk device to the virtual machine would readily resolve this issue.

After creating the devices, we generate a description of the virtual machine's hardware configuration, known as a Flattened Device Tree (FDT) on Linux, which is written to memory. Next, we configure some registers, and upon execution of the virtual machine, Linux will boot.

3.2.2 Tardis-cli

As our targets operate within a Linux virtual machine, we developed a command-line interface tool that acts as a mediator between the virtual machine and the fuzzer, and can be utilized within the virtual machine. One of the tool's functions is the launch of another process, which serves as our target, with certain security features disabled, such as ASLR. Address Space Layout Randomization (ASLR) is a security feature that randomizes the memory layout of a process to make it harder for attackers to exploit vulnerabilities. By disabling ASLR, the process's memory layout becomes predictable, and thus allowing us to place breakpoints in the target more easily.

This feature is employed during the snapshotting phase to insert a breakpoint in the target code just before any input is processed. Upon reaching the specified instruction, the target code halts, and the fuzzer gains control, enabling it to capture a snapshot of the virtual machine's state right before any input is processed in the target.

3.2.3 Snapshotting

Upon returning control to Tardis, a snapshot of the virtual machine's state is captured. Initially, the values of all registers are copied and saved, following which the device states are obtained and serialized utilizing the existing Rust-VMM code. However, Rust-VMM lacks serialization support for the interrupt controller (GICv3) - a vital component in a virtual machine that is responsible for managing and distributing interrupt requests from devices to the CPU. The GICv3 depends on the CPU's existing interrupt controller instead of being fully emulated.

To preserve the state of the interrupt controller, we must first flush all pending operations to memory to prevent the loss of state during the process. Subsequently, we save the values of specific system registers related to GICv3, which results in functional snapshots. Finally, all the compiled data is serialized and saved in a file alongside the full contents of the VM's memory.

The saved file can be utilized to replicate the virtual machine in a state that is identical to the captured snapshot at a later point in time. As we utilize Rust-VMM's device serialization code, extending support to other compatible devices (such as disks and networking) in the future would be a straightforward process.

3.3 Fuzzing

Once we have obtained a snapshot of our target, we can initiate the fuzzing process. This process is relatively complex, and a simplified depiction of it is illustrated in the accompanying figure. In this section, I will elaborate on its functionality and how we achieve the required features for fast snapshot resets without depending on CPU-specific features.

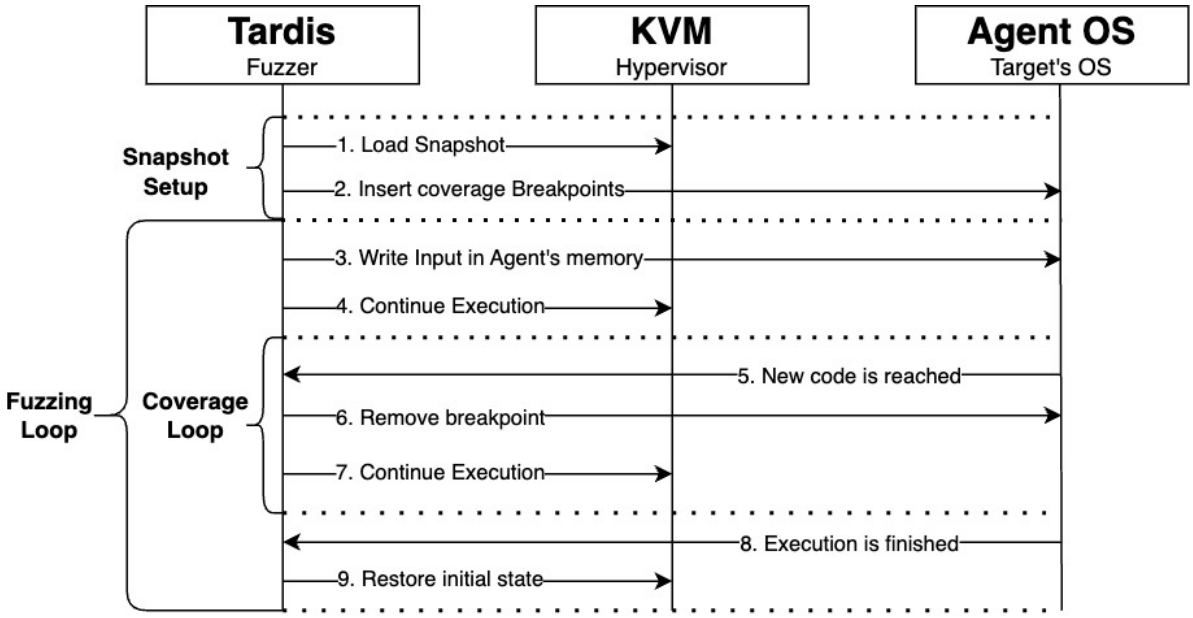

Figure 3.2: Overview of the fuzzing steps in Tardis and its various components

3.3.1 Setup

This is the initial phase of the fuzzing process, which commences with the fuzzer creating a virtual machine within KVM. Then, we allocate the necessary memory for the VM and duplicate the contents of the snapshot's memory into it. Once that's complete, we create the same devices that were utilized in the original VM, but we initialize them with the serialized state obtained from the snapshot. Subsequently, we restore the state of the interrupt controller by setting the vGIC3 system registers to the values we saved during the previous execution. This enables us to resume the snapshot execution in precisely the same state as when we captured it. However, to fuzz the target, we must also obtain code coverage and quickly restore the modified memory between executions.

3.3.2 Coverage

In Tardis, code coverage is obtained by utilizing software breakpoints. To achieve this, a disassembler is utilized to determine the addresses of all basic blocks, and software breakpoints are then placed at the beginning of each basic block. KVM is then configured to redirect all software breakpoints to the fuzzer, allowing us to measure code coverage.

However, the primary issue faced during the implementation process was dealing with the different address spaces in our VM. Specifically, we needed to modify the memory of the target process, but we do not know where the target's memory is located in physical memory.

To address this problem, we wrote code that translates Virtual Addresses to Physical Addresses, as outlined in the ARM reference manual. This process is the same as what the CPU uses to translate these addresses. Currently, my implementation does not support translating kernel addresses.

3.3.3 Dirty Memory

In order to track dirty pages on ARM CPUs, where there is no equivalent feature to PML, we make use of KVM's *KVM_MEM_LOG_DIRTY_PAGES* flag. This flag is set during initial memory mapping within the virtual machine, which enables KVM to track any modified memory on our behalf. This feature works by examining every mapped page within the virtual machine and verifying if the Access Flag bit is set[3]. The results are then converted into a bitmap that provides information about which pages are dirty. However, using this API can be inefficient for VMs with a large amount of RAM, as parsing the resulting bitmap can be time-consuming. To address this issue, we optimized the code that parses the bitmap by making use of SIMD instructions outputted by the compiler, resulting in much better performance.

3.3.4 Final implementation

To turn this project into a fuzzer, we first load the target's snapshot and insert breakpoints at the start of each basic block. Then, using the virtual address translation code, we write our test case directly in the target's memory. As the execution of the snapshot proceeds, each time a breakpoint is hit, the basic block is marked as executed and the breakpoint is replaced with the original instruction. This continues until we reach the end of our test case. Afterwards, the fuzzer restores the registers, device states and replaces the dirtied memory with its original content, returning the snapshot to its initial state. At this point, we can write a different test case to memory and execute the snapshot again.

4

Results and Evaluation

4.1 Evaluating Tardis

In this section, we will assess the effectiveness of Tardis in meeting the success criteria outlined earlier. Initially, we will conduct fuzzing on a challenging target, which cannot be adequately fuzzed with conventional fuzzers. Subsequently, we will compare the performance of Tardis against that of other conventional fuzzers and deliberate on the circumstances that warrant the use of such a fuzzer.

4.2 Fuzzing bash

To showcase the capacity of Tardis to fuzz non-conventional targets, I conducted a test using Bash, which is typically a challenging target to fuzz due to the potential creation of files or execution of other processes. As a result, the cleanup process following each execution is time-consuming and resource-intensive for traditional fuzzers. However, with Tardis, this is not a concern as the complete state of the virtual machine utilized for fuzzing Bash is restored after each execution.

Despite the significant advantages of using Tardis for this purpose, the preparation required to run the target in the fuzzer is a relatively intricate process. Therefore, in the following subsections, we will provide an illustrative example of how to utilize this tool from a user's perspective.

4.2.1 Picking a target function

We initiated the process by examining the target's code and identifying a function to be targeted. In the context of Bash, we selected the function $with\_input\_from\_string$, which is executed when Bash executes a command passed to it with the -c command line flag. Through

this method, we can directly write our inputs into Bash's memory instead of providing them via the standard input (stdin) mechanism.

Subsequently, we employed a disassembler to locate the address of this function within the target binary, after which we proceeded to the next stage.

4.2.2 Preparing the snapshot

Once we have the address of our target function, we start Tardis in 'run' mode, which allows us to create snapshots. We then use *tardis_cli* to create our snapshot right before the target function is called:

```
# Start a Linux VM with 128mb of ram, save the snapshot in state-bash
tardis-fuzz run -k pe-linux -s state-bash -m 128
...
[    0.141925] Run /init as init process
Hello, world, from the Tardis reference VM!

# Snapshot Bash when executing the with_input_from_string function
/$ tardis-cli -b 0x403A74 -- bash -c "echo hi" > /dev/null
[DEBUG tardis_fuzz::vmm::setup] Saving state
```

Code 4.1: Creating a snapshot with tardis-cli

After preparing our snapshot, we also need to generate a coverage map containing all the basic blocks for our target process. This can be done with Tardis's gen-cov command:

```
# Find all basic blocks in the bash binary and write them to bash_cov.json
tardis-fuzz gen-cov -b bash -o bash_cov.json
```

Code 4.2: Creating a coverage file with Tardis

Now we have a full snapshot of our target alongside all the necessary information to fuzz it. In the next stage we will modify Tardis in order for it to write inputs correctly into our target and allow us to fuzz it.

4.2.3 Preparing the fuzzer

Once we have a snapshot that will be used for fuzzing, we need to modify Tardis so that it can correctly write inputs inside the memory of Bash. In the case of *with_input_from_string* our input is in a pointer to a string as its first argument, which will be in register $x0$ due to ARM calling conventions. We then modify the harness function in src/fuzzer/fuzzer.rs to write the inputs at the address contained in $x0$. This function will be called before every execution and it has the responsibility of writing the passed input inside the target.

```
let mut harness = |vm: &mut VM, input: &[u8]| {
    // Get the pointer to the input in the guest's addr space
    let input_addr = vm.get_reg(aarch64::X0);
    // Write the input to memory
    vm.write_slice(input_addr, input);
```

```
};
```

Code 4.3: Writing inputs inside a snapshot

Once this is done, we can finally fuzz our target. To do this, we run Tardis in fuzz mode:

```
# Fuzz the bash snapshot on 8 cores
tardis-fuzz fuzz -s state-bash -c bash-cov.json -v 8 --corpus-path corpus
```

Code 4.4: Creating a coverage file with Tardis

This will successfully start fuzzing Bash. In order to compare performance, we also attempted to fuzz the same target using a traditional fuzzer, in this case AFL++. However, this approach led to the fuzzer encountering timeouts and running out of memory within a few minutes.

4.2.4 Performance analysis

Snapshot fuzzers tend to outperform traditional fuzzers in terms of performance. The performance of snapshot resets was measured without the added overhead of measuring coverage. To achieve this, a benchmark was implemented that dirties a single page and then resets the virtual machine (VM). When this benchmark was run on all the cores of the Apple M1 APL1102 CPU, a significant difference in performance was observed depending on which core the CPU was running. Four of the cores consistently achieved approximately 18,000 resets per second, while the other four achieved around 11,000 resets per second. This is due to the two types of cores present in M1 CPUs: 4 performance cores and 4 efficiency cores. In total, when a single page of memory is dirty, we can achieve around 110,000-120,000 resets per second. However, this benchmark is not representative of the real usage of the fuzzer and only provides an upper bound on performance.

To test performance on a more realistic target, such as Bash, the LibAFL integration was used. In experiments, a full Bash VM can be reset between 5,000 and 7,000 per second per core, depending on the core the fuzzer is running on. Unfortunately, due to a bug in LibAFL, the fuzzer is only able to run on half the CPU cores on the CPU, resulting in a speed of between 20,000 and 28,000 executions per second. It is important to note that the performance of our fuzzer improves over time due to the high initial overhead of measuring code coverage. However, this overhead becomes smaller over time as the most common paths in the program are explored, and the breakpoints in them are removed. In a separate benchmark that doesn't use LibAFL but still gathers coverage data, we achieved roughly 42,000 executions per second while fuzzing Bash. This gives us a better idea of the fuzzer's performance without the bug that limits our speed.

However, this information does not provide us with a clear understanding of how Tardis compares to other fuzzers. To make a more precise comparison between the different execution modes of LibAFL and Tardis, we selected a target (LibPNG) that could be used in all cases and conducted fuzzing on all targets. The results are presented in Table 4.1.

LibAFL LibPNG (In-process executor)	LibAFL LibPNG (In-process fork executor)	LibAFL LibPNG (Forkserver executor)	Tardis LibPNG (Snapshot executor)
980k exec/s	38k exec/s	28k exec/s	46k exec/s

Table 4.1: Comparing LibAFL performance with Tardis

It is evident that Tardis performs better than LibAFL in all execution modes, except for the In-Process Executor. The reason for the high throughput of this execution mode is that the fuzzer and the target code run within the same binary. However, this execution mode is only applicable to certain targets with source code available and where previous executions do not impact the output of future ones.

5 Conclusion

According to our previous benchmarks, Tardis is an excellent choice for fuzzing black box ARMv8 targets. It outperforms all other available LibAFL execution modes for black box targets and provides the previously discussed benefits of using a snapshot fuzzer. However, the more complicated setup and lower speeds make it impractical for codebases where in-process fuzzing is possible.

5.1 Bottlenecks and future improvements

Despite achieving good performance metrics, we have conducted a profiling analysis on our fuzzer to identify potential areas for further optimization. Our analysis revealed that approximately 40% of the CPU time outside of the virtual machine is dedicated to altering registers via KVM. While KVM enables multiple register changes with a single call on x86, it only permits one register change at a time on ARMv8. To enhance performance, we can either modify KVM to support multiple register changes at once on ARMv8, or selectively restore only the necessary registers.

5.1.1 Reusing AFL instrumentation

Since we have full access to the guest's memory we can reuse AFL's compile-time instrumentation, which have been the subject of a large amount of research over the years. This would enable us to obtain more effective instrumentation(such as CmpLog, ASAN, etc) for targets where we have access to the source code. Additionally, this method is substantially more efficient in comparison to breakpoint coverage and should result in better performance.

References

[1] *A message about iOS security*. URL: https://www.apple.com/newsroom/2019/09/a-message-about-ios-security/.

[2] *Address Sanitizer Performance Numbers*. URL: https://github.com/google/sanitizers/wiki/AddressSanitizerPerformanceNumbers.

[3] ARM. *AArch64 memory model - Access Flag*. URL: https://developer.arm.com/documentation/102376/0100/Access-Flag.

[4] *ARMored CoreSight: Towards Efficient Binary-only Fuzzing*. URL: https://ricercasecurity.blogspot.com/2021/11/armored-coresight-towards-efficient.html.

[5] Cornelius Aschermann et al. "REDQUEEN: Fuzzing with Input-to-State Correspondence". In: *Symposium on Network and Distributed System Security (NDSS)*. 2019.

[6] Stella Bitchebe et al. *Intel Page Modification Logging, a hardware virtualization feature: study and improvement for virtual machine working set estimation*. 2020. DOI: 10.48550/ARXIV.2001.09991. URL: https://arxiv.org/abs/2001.09991.

[7] Marcel Böhme and Brandon Falk. "Fuzzing: On the Exponential Cost of Vulnerability Discovery". In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2020. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 713–724. ISBN: 9781450370431. DOI: 10.1145/3368089.3409729. URL: https://doi.org/10.1145/3368089.3409729.

[8] Marcel Böhme, László Szekeres, and Jonathan Metzman. "On the Reliability of Coverage-Based Fuzzer Benchmarking". In: *Proceedings of the 44th International Conference on Software Engineering*. ICSE '22. Pittsburgh, Pennsylvania: Association for Computing Machinery, 2022, pp. 1621–1633. ISBN: 9781450392211. DOI: 10.1145/3510003.3510230. URL: https://doi.org/10.1145/3510003.3510230.

[9] Yaohui Chen et al. "PTrix: Efficient Hardware-Assisted Fuzzing for COTS Binary". In: *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*. Asia CCS '19. Auckland, New Zealand: Association for Computing Machinery, 2019, pp. 633–645. ISBN: 9781450367523. DOI: 10.1145/3321705.3329828. URL: https://doi.org/10.1145/3321705.3329828.

[10] *CompareCoverage*. URL: https://github.com/googleprojectzero/CompareCoverage.

[11] Andrea Fioralldi et al. "Registered report: Dissecting american fuzzy lop - A fuzzbench evaluation". In: *FUZZING 2022, 1st International Fuzzing Workshop, 24 April 2022, San Diego, CA, USA / Co-located with NDSS 2022*. San Diego, 2022.

[12] *Fuzzing Modern UDP Game Protocols With Snapshot-based Fuzzers*. URL: https://blog.ret2.io/2021/07/21/wtf-snapshot-fuzzing/.

[13] *Greece wiretap and spyware claims circle around PM Mitsotakis*. URL: https://www.bbc.com/news/world-europe-62822366.

[14] *Interrupt 3—Breakpoint Exception (BP)*. URL: https://xem.github.io/minix86/manual/intel-x86-and-64-manual-vol3/o_fe12b1e2a880e0ce-209.html.

[15] *Jordanian Human Rights Defenders and Journalists Hacked with Pegasus Spyware*. URL: https://citizenlab.ca/2022/04/peace-through-pegasus-jordanian-human-rights-defenders-and-journalists-hacked-with-pegasus-spyware/.

[16] *Linux Programmer's Manual - core dump file*. URL: https://man7.org/linux/man-pages/man5/core.5.html.

[17] Valentin J. M. Manès et al. "Fuzzing: Art, Science, and Engineering". In: *CoRR* abs/1812.00140 (2018). arXiv: 1812.00140. URL: http://arxiv.org/abs/1812.00140.

[18] *Mexican Journalists Investigating Cartels Targeted with NSO Spyware Following Assassination of Colleague*. URL: https://citizenlab.ca/2018/11/mexican-journalists-investigating-cartels-targeted-nso-spyware-following-assassination-colleague/.

[19] Barton P. Miller, Lars Fredriksen, and Bryan So. "An Empirical Study of the Reliability of UNIX Utilities". In: 33.12 (Dec. 1990), pp. 32–44. ISSN: 0001-0782. DOI: 10.1145/96267.96279. URL: https://doi.org/10.1145/96267.96279.

[20] Mozilla. *Effectively Fuzzing the IPC Layer in Firefox*. URL: https://blog.mozilla.org/attack-and-defense/2021/01/27/effectively-fuzzing-the-ipc-layer-in-firefox/.

[21] Stefan Nagy and Matthew Hicks. "Full-Speed Fuzzing: Reducing Fuzzing Overhead through Coverage-Guided Tracing". In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019, pp. 787–802. DOI: 10.1109/SP.2019.00069.

[22] Stefan Nagy et al. "Breaking Through Binaries: Compiler-quality Instrumentation for Better Binary-only Fuzzing". In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 1683–1700. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/nagy.

[23] *Nyx Github*. URL: https://github.com/nyx-fuzz/Nyx.

[24] *Once upon a free()*. URL: http://phrack.org/issues/57/9.html.

[25] *OSS Fuzz*. URL: https://github.com/google/oss-fuzz.

[26] *SanitizerCoverage*. URL: https://clang.llvm.org/docs/SanitizerCoverage.html.

[27] Sergej Schumilo et al. "Nyx-Net: Network Fuzzing with Incremental Snapshots". In: *CoRR* abs/2111.03013 (2021). arXiv: 2111.03013. URL: https://arxiv.org/abs/2111.03013.

[28] Sergej Schumilo et al. "Nyx: Greybox Hypervisor Fuzzing using Fast Snapshots and Affine Types". In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 2597–2614. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/schumilo.

[29] Konstantin Serebryany et al. "AddressSanitizer: A Fast Address Sanity Checker". In: *USENIX ATC 2012*. 2012. URL: https://www.usenix.org/conference/usenixfederatedconferenceswe addresssanitizer-fast-address-sanity-checker.

[30] *Smashing the stack for fun and profit*. URL: http://phrack.org/issues/49/14.html.

[31] *Tartiflette Github*. URL: https://github.com/MattGorko/Tartiflette.

[32] *Technical "whitepaper" for afl-fuzz*. URL: https://lcamtuf.coredump.cx/afl/technical_details.txt.

[33] *Trends, challenge, and shifts in software vulnerability mitigation*. URL: https://github.com/microsoft/MSRC-Security-Research/tree/master/presentations/2019_02_BlueHatIL.

[34] *Vudo malloc tricks*. URL: http://phrack.org/issues/57/8.html.

[35] *What The Fuzz Github*. URL: https://github.com/0vercl0k/wtf.

[36] Google Project Zero. *MMS Exploit Part 1: Introduction to the Samsung Qmage Codec and Remote Attack Surface*. URL: https://googleprojectzero.blogspot.com/2020/07/mms-exploit-part-1-introduction-to-qmage.html.

[37] *Zerodium Exploit Acquisition Program*. URL: https://zerodium.com/program.html.

Acknowledgments

acknowledgements if any otherwise leave empty, it does not count towards the number of pages.