Technische Universität Berlin

Fakultät IV - Elektrotechnik und Informatik
Institut für Softwaretechnik und Theoretische Informatik
Security in Telecommunications (SecT)

Master thesis

Fuzzing the AMD SP's ROM bootloader with LibAFL using QEMU full-system emulation

Patrick Gersch
381425

1. Reviewer Prof. Dr. Jean-Pierre Seifert
Security in Telecommunications
Technische Universität Berlin

2. Reviewer Prof. Dr. Thomas Magedanz
Architekturen der Vermittlungsknoten
Technische Universität Berlin

Supervisors Christian Werling
Hans Niklas Jacob
Vincent Quentin Ulitzsch

February 4, 2023

Abstract

Advanced Micro Devices (AMD's) processor security features like Secure Encrypted Virtualization (SEV), Hardware-validated Boot (HVB), and firmware-based Trusted Platform Module (fTPM) all depend on the integrity of the responsible firmware. Modern AMD systems (Ryzen and Epyc) implement these functionalities in a Trusted Execution Environment (TEE) running on a dedicated security processor, the AMD Secure Processor (ASP). It powers up before any x86 core and executes the ROM bootloader. Its primary task is to load and verify the off-chip bootloader from flash before permanently handing off the execution to it.

This thesis presents the first fuzzer for the ASP's proprietary ROM bootloader firmware using full-system emulation for Ryzen Zen1, Zen+, Zen2, Zen3 and an updated Zen1 version for Tesla products (ZenTesla). Fuzzing is a technique to test the functionality of a target automatically. The modular ASP fuzzer presented in this thesis re-discovered a known buffer overflow yielding arbitrary code execution on Zen1 and Zen+ ASP's. No bug was found in the ROM bootloader for Zen2, Zen3 or ZenTesla, even though the fuzzer input-depending code was covered entirely.

Zusammenfassung

Die Sicherheitsfunktionen der Prozessoren von Advanced Micro Devices (AMD) wie Secure Encrypted Virtualization (SEV), Hardware-validated Boot (HVB) und das Firmware-basierte Trusted Platform Module (fTPM) hängen alle von der Integrität der zuständigen Firmware ab. Moderne AMD-Systeme (Ryzen und Epyc) implementieren diese Funktionalitäten in einer Trusted Execution Environment (TEE), die auf einem speziellen Sicherheitsprozessor, dem AMD Secure Processor (ASP), liegt. Er schaltet sich vor jedem x86-Kern ein und führt den ROM-Bootloader aus. Seine primäre Aufgabe ist es, den Off-Chip-Bootloader aus dem Flash zu laden und zu verifizieren, bevor die Ausführung dauerhaft an ihn übergeben wird.

Diese Arbeit präsentiert den ersten Fuzzer für die ASP ROM-Bootloader-Firmware für Ryzen Zen1, Zen+, Zen2, Zen3 und eine aktualisierte Zen1 Version für Tesla Produkte (ZenTesla). Dabei ist Fuzzing eine Technik zum automatischen Testen der Funktionalität eines Ziels. Der in dieser Arbeit vorgestellte modulare ASP Fuzzer entdeckte erneut einen bekannten Buffer Overflow, der die Ausführung von beliebigem Code auf Zen1 und Zen+ ASP's ermöglicht. Kein Fehler wurde im ROM-Bootloader für Zen2, Zen3 oder ZenTesla gefunden, obwohl der gesamte vom Fuzzer input-abhängiger Code abgedeckt wurde.

Acknowledgement

First of all, I want to thank my supervisors for offering this thesis to me and supporting me along the way. It was a pleasure to work on this project at SecT.

Without the insights into the AMD Secure Processor from Niklas and Christian, the fuzzer would have never gotten to the point where it is now. Vincent, thank you for introducing me to fuzzing and guiding me through the early stages of this project.

Special thanks go to Dominik Maier for helping me with fundamental fuzzing decisions and questions. Andrea Fioraldi provided great help with his knowledge and expertise for everything concerning LibAFL and QEMU. I am sorry for all the question you two had to stand from me due to my lack of experience in this field. I hope I could give at least a bit back by contributing to the LibAFL project.

Honorable mentions also go to WorksButNotTested for the Ghidra lightkeeper plugin and the fast support he provided for it. Finally, I thank all of the "Awesome Fuzzing" Discord community for the warm welcome and their support throughout my fuzzing journey.

Contents

1 Introduction

The AMD Secure Processor (ASP) is a System-on-a-Chip (SoC) present on all Advanced Micro Devices (AMD) chips which are built ontop of the Zen architecture. It contains and executes the first code (on-chip bootloader) after the chip's boot-up before any x86 core starts. This ASP on-chip bootloader loads and verifies the off-chip bootloader, which is co-located with the x86 firmware and usually stored on a dedicated non-volatile flash storage. Therefore, it is the Root of Trust (RoT) for all later boot stages and also for security features like the Trusted Execution Environments (TEE) or Hardware-validated Boot. The security of this first boot stage is thus of utmost importance as the integrity of the entire chip and any code executed on AMD chips depend on it. Furthermore, the proprietary on-chip bootloader code is in Read-only memory (ROM) on the SoC, thus also called ROM bootloader and can not be patched after manufacturing.

This thesis presents ASPFuzz, the first open-source fuzzer for the ASP in Ryzen Zen1, Zen+, Zen2, Zen3 and an updated Zen1 version for Tesla products (ZenTesla). The fuzzer was built on top of QEMU full-system mode. Therefore it does not require any AMD hardware to run and scales solely by the number of servers used. LibAFL was chosen was the underlying fuzzing library, which comes with native multi-core support. As a result, the fuzzer scales almost linearly with the number of CPU cores. Furthermore, multiple custom snapshotting implementations for the ASP are built into ASPFuzz. Next to the lightweight hooks for LibAFL, this results in a fuzzing throughput of over 5000 test-cases per second. The fuzzer is also configurable over YAML Ain't Markup Language (YAML) files, making it easy to run fuzzing campaigns without changing any code. Knowledge of the on-chip bootloader for Ryzen Zen1, Zen+, Zen2, Zen3 and ZenTesla was used to configure the fuzzer. This could also be done for the AMD Epyc series, other Zen generations or even extending it to the off-chip bootloader. The fuzzer re-discoverd a known buffer overflow in the on-chip bootloader for Zen1 and Zen+ within seconds, showcasing the performance of ASPFuzz. No bug in the on-chip bootloader for Zen2, Zen3 or ZenTesla was found by the fuzzer, even though all fuzzer input-depending code was covered.

Motivation. The on-chip bootloader is proprietary ROM firmware code by AMD running on the proprietary ASP coprocessor. Prior research was able to gain code execution on the ASP using fault injection [Buh+17]. The authors could therefore dump the ROM memory and partly reverse it to build an emulator for the ASP [Buh+21; Eic20]. This leaves an opportunity to check the security of the on-chip bootloader code using a software security technique like fuzzing, either building trust or casting doubts on the security of AMD's proprietary ROM firmware.

The on-chip bootloader parses the Firmware File System (FFS) [Wer19], which is stored in flash memory, to load the off-chip bootloader. An attacker with physical hardware access or other capabilities to overwrite the flash memory is well inside AMD threat model for technologies relying on the ASP [KPW16]. Therefore, the fuzzer can only write arbitrary inputs to the flash memory region to qualify for this threat model. This thesis only focuses on the ASP in Ryzen series chips, leaving the Ryzen Threadripper or Epyc series chips as future work.

Contributions. To summarize the contributions of this thesis:

- The first full-system emulation, coverage guided fuzzer for the ASP was built.
- ASPFuzz is publicly available on Github[1].
- The fuzzer scales almost linearly with the used cores and servers and is heavily configurable.
- Fuzzer configurations for the proprietary on-chip bootloader for Ryzen Zen1, Zen+, Zen2, Zen3 and ZenTesla were found by extending previous binary analysis.
- 100% code coverage was reached for all flash memory-dependent code.
- ASPFuzz found a known buffer overflow in Zen1 and Zen+ within seconds while not finding any other security-critical bugs.
- On the other hand, no bugs in the on-chip bootloader for Zen2, Zen3 and ZenTesla were found by the fuzzer.

[1] `https://github.com/TeumessianFox/ASPFuzz`

Organisation of this thesis. In chapter 2 the related work is presented. Furthermore, the ASP with its on-chip bootloader, the basics of emulation and the fundamentals of fuzzing are discussed in-depth. Next, chapter 3 highlights how QEMU for the ASP with LibAFL integration was built. The LibAFL-based ASPFuzz fuzzer is presented in chapter 4. Especially the custom snapshotting and configuration options are covered to a great extent. Chapter 5 uses knowledge of the on-chip bootloaders to configure the fuzzing campaigns for it. Finally, the results and performance of ASPFuzz are shown in chapter 6 together with a conclusion in chapter 7. Future work is briefly suggested in chapter 8.

2 Background

This section first presents the related work to put this thesis in perspective to existing research. The later sections introduce basic concepts and general background knowledge to understand the main chapters of this thesis. References for all major topics will be provided along the way.

2.1 Related Work

The related work section mainly focuses on existing research in the area of full-system emulated fuzzing. ASP and fuzzing specific research will be discussed in section 2.2 and section 2.4 respectively.

Full-system fuzzing is mainly used to analyze Operating Systems (OSs), specifically kernel interfaces, syscalls or drivers. The open-source fuzzer Syzkaller [Goo19] was developed by Google to fuzz the Linux kernel, but was extended to other OSs as well. Agamotto [Son+20] is also a kernel device driver fuzzer. Agamotto showcases the lack of performance in kernel fuzzing due to slow snapshotting of the emulator state. Agamotto proposes lightweight checkpoints as a snapshotting strategy to increase the throughput of the fuzzer compared to kernel fuzzers like Syzkaller. A disadvantage is that the fuzzer only works for QEMU with Kernel-based Virtual Machines (KVMs).

Internet of Things (IoT) firmware security is another research area that uses full-system fuzzers. IoTFuzzer [Che+18] directly operates on the real hardware without requiring access to the firmware code. FIRMADYNE [Che+16], on the other hand, emulates embedded devices in a full-system manner yielding higher throughput and more coverage information but requiring an emulator and the firmware code for every target. Similarly, FIRMWIRE [Her+22] is a cellular protocol fuzzer to analyze binary firmware for baseband processors in full-system emulation. FIRM-AFL [Zhe+19] fuzzes IoT firmware using augmented emulation. Augmented emulation only runs firmware code in QEMU full-system mode if an Emulated Memory Management Unit (softMMU) is required for specific memory translations. Otherwise, the code is executed in the faster QEMU user-mode.

Multiple authors [Lum22; Bog19; Har+22] present full-system fuzzer for the Open Portable Trusted Execution Environment (OP-TEE) target. OP-TEE is an open-source implementation of TEE for certain ARM-based chips. They require OP-TEE to run as a Guest OS inside a QEMU Virtual Machine (VM). TEE is one of the technologies which can use the ASP as a RoT. AMD Secure Encrypted Virtualization (AMD SEV) [KPW16] and Intel Trust Domain Extensions (Intel TDX) [Int20] are both solutions for confidential computing in the cloud environment. The aim is to protect VMs from malicious Hypervisors. Hetzelt et al. [Het+21] propose the fuzzing framework VIA to detect missing input validations in the device drivers of VMs.

TheHuzz [Kan+22] takes a different approach and fuzzes the handware design in Hardware Description Language (HDL). This approach is advantageous if the HDL is publicly available. At the time of writing, no HDL design for the ASP was released.

No matter which target, all full-system fuzzer suffer from undetected memory corruptions (silent memory corruption). While user-code emulations use the underlying OS to detect faulty states, embedded devices often lack these features due to their resource constraints. As a result, memory corruption which would typically result in crashes, might not be noticed as embedded devices mostly come without an Memory Management Unit (MMU) and avoid extra memory security like sanitizer because of their low computational power. Therefore, critical bugs may stay unnoticed during fuzzing as no crash or hang is triggered during the emulation. [Mue+18; Sal+22]

2.2 AMD Secure Processor

The ASP is an Advanced RISC Machines (ARM) Cortex-A5 processor embedded into every AMD SoC that uses the Zen architecture alongside the x86 cores since the Zen CPU architecture was first released. The Ryzen, Ryzen Threadripper and Epyc series are based on the Zen architecture and all include the ASP. Multiple Zen generations have been released by AMD, currently including Zen (referred to as Zen1), Zen+, Zen2, Zen3, Zen3+ and Zen4. The architecture of AMD chips used by Tesla [Tes23] is denoted as ZenTesla. Most public knowledge about the ASP either come from AMDs whitepapers [KPW16; Mal21], from one of these masterthesises [Wer19; Eic20; Jac22; Küh22] or the following papers [Buh+17; BWS19; Buh+21; Buh22]. While the ASP stayed unchanged from Zen1 to Zen+, there were some minor changes in the memory layout and hardware values for Zen2, Zen3 and ZenTesla.

The following subsections 2.2.1 and 2.2.2 describe the memory layout and boot procedure of the ASP. The last subsection describes how the proprietary chip and its firmware were extracted and analyzed.

2.2.1 Memory Layout

The ARM Cortex-A5 implements a 32-bit ARMv7-A Instruction Set Architecture (ISA) with 16 32-bit registers. Figure 2.1 shows the 32-bit memory layout of the ASP. Further details concerning the memory layout can be found in [Eic20; BJE22] and will be mentioned as needed throughout this thesis.

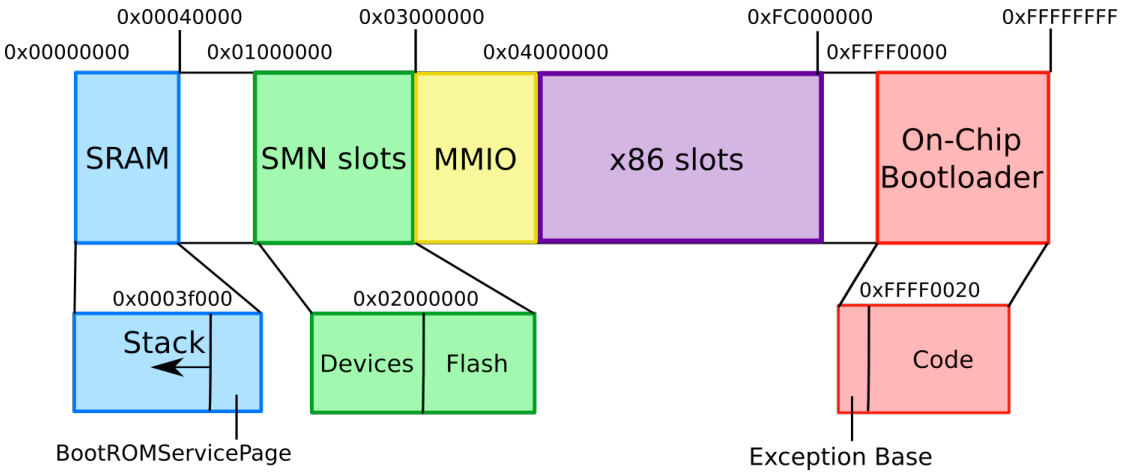

Fig. 2.1: ASP memory layout of the Zen1 & Zen+ generation

SRAM. Zen1, Zen+ and ZenTesla come with 256kiB of Static Random-Access Memory (SRAM) and Zen2 and Zen3 with 320kiB of SRAM. The SRAM is mapped to `[0x0,0x0003ffff]` and `[0x0,0x0004ffff]` respectively. For simplicity, all addresses from this point on will be only for the Zen1, Zen+ and ZenTesla architecture. The Zen2 and Zen3 addresses can be computed by offsetting with the Zen2 & Zen3 SRAM space. The last 4KiB `[0x0003f000,0x0003ffff]` of SRAM are used for system information (referred to as BootROMServicePage), while `0x0003f000` is the initial stack pointer with the stack growing downwards.

MMIO. The Memory-Mapped I/O (MMIO) space includes at least an Interrupt Request (IRQ) controller, two timers, the System Management Network (SMN) slot controller,

the x86 slot controller, fuses and a cryptographic accelerator called Cryptographic Co-Processor (CCP). The CCP can do different cryptographic operations and provides a passthrough operation. The passthrough operation copies data from one memory area to another. Therefore it can be seen as a hardware-accelerated *memcpy* function.

SMN. The SMN has a separate 32-bit memory space which can be used to map various devices like the DDR4 memory controller into the address space of the ASP. This can be achieved by configuring the 32 1MiB SMN slots in the ASP memory space `[0x01000000, 0x02ffffff]` using the SMN controller registers in the MMIO space. Each slot can be mapped to an address in the internal 32-bit memory space of the SMN. The Unified Extensible Firmware Interface (UEFI) image is stored in 16MiB of Serial Peripheral Interface (SPI) flash memory connected over an SPI bus to the internal SMN memory space. Parts of the UEFI image are statically mapped to `[0x02000000, 0x02ffffff]` by the on-chip bootloader.

X86. The x86 memory slots are similar to the SMN slots but map x86 address spaces into the ASP memory space. These slots only get important after the on-chip bootloader is finished.

On-chip bootloader. Lastly, the on-chip bootloader code laying in ROM is mapped to `[0xffff0000, 0xffffffff]`. For that reason, it is also referred to as ROM bootloader. The high vector (`0xffff0000`) is set for the exception and interrupt vector [ARM16]. Therefore, the Cortex-A5 starts at the reset exception base `0xffff0000` on system start-up.

2.2.2 Boot Structure

The x86 cores are halted during the system power-on while the ASP initializes the system. The on-chip bootloader is the first code being executed on the ASP. The corresponding on-chip bootloader code is at the high vector location (`[0xffff0000,0xffffffff]`) of the ARM chip in ROM (see section 2.2.1). Even though the on-chip bootloader is proprietary firmware code by AMD, the binary code was already partly reverse-engineered by [Eic20; BWS19] after the code was extracted from the hardware.

1. First, the on-chip bootloader configures some MMIO devices including the external SPI flash memory where the UEFI image is written to.
2. Next, it parses the FFS, which is part of the UEFI image, to find the correct off-chip bootloader. The FFS was intensely studied by Werling [Wer19; BWS19] who also provides PSPTool [Wer22] to analyze the FFS. A Firmware Entry Table (FET) is the entry point of the FFS. The FET can contain one or more directories. Each directory can contain combo directories, secondary directories and/or entries. Two types of entries are known: the Public Key Entry and the Header Entry.
3. The AMD Root Signing Key (ARK) is a Public Key Entry which is loaded from the SPI flash memory into SRAM. A SHA256 hash of the key is compared to a stored hash value in ROM.
4. Finally, the off-chip bootloader (Header Entry) is loaded to SRAM starting at `0x00000100`. The signature of the loaded off-chip bootloader entry is verified using the ARK before handing the execution over to it.

Once the off-chip bootloader takes over, there are multiple stages until the ASP is fully initialized and the x86 cores are running. The following stages are described in [BWS19; Buh+21]. It includes features like Platform Secure Boot, runtime TEE, Firmware Trusted Platform Module (fTPM) and AMD SEV, which are provided by the ASP in the later stages. Therefore, the on-chip bootloader is the RoT for all these security features, especially as it loads and verifies the ARK and off-chip bootloader.

2.2.3 ROM Extraction

Even though the on-chip bootloader is in the ASPs ROM and was not published by AMD, the binary code was extracted for Zen1, Zen+, Zen2, Zen3 and ZenTesla.

[BWS19] found a buffer overflow in the parsing of the Header Entry header in the on-chip bootloader for Zen1 and Zen+. Only the Header Entries body is verified using the ARK, leaving complete control over the header. An attacker with physical access or root/admin permission can overwrite the UEFI image and thus freely change the header. The header includes a size field, which indicates the length of the body. Due to an insufficient bound check, the attacker can gain arbitrary code execution by setting the most significant bit of the size field. This results in a buffer overflow which allows the attacker to overwrite the stack. The Link Register (LR) saved on the stack can thus be overwritten, resulting in

arbitrary code execution on the ASP. Code execution then allowed dumping the ROM code.

For Zen2, Zen3 and ZenTesla Voltage Fault Injection [Buh+21] and Electromagnetic Fault Injection [Küh+22] were used to skip the ARK verification. Skipping the ARK verification allows attacker-chosen public keys for the verification of Header Entries. Therefore, any attacker-chosen Header Entry can be loaded and executed instead of a valid off-chip bootloader, which allows dumping the ROM again.

Dumping the ROM code for each generation allowed static analysis of the on-chip bootloader in tools like Ghidra [Ghi19]. This ultimately resulted in the first emulator for the ASP using that knowledge (see section 2.3.2).

2.3 Emulation

Emulators allow the execution of binaries compiled for one host CPU architecture to run on another guest CPU architecture. Thereby, emulators are grouped into user-mode emulators and full-system emulators.

User-mode emulation. In user-mode emulation, instructions are only translated from one CPU ISA to another. Any system calls or peripheral accesses are passed to the underlying host OS exactly as regular user-land applications do. Memory translation works on the host OSs virtual addresses. As a result, only user-land applications can be emulated. A typical example is software compiled for non-x86 machines, which need to run on x86 desktop machines for testing purposes.

Full-system emulation. Compared to user-mode emulation, which operates on the host OSs virtual addresses, the full-system emulation uses a softMMU. This allows emulation of arbitrary memory layouts and access permissions, including ROM, SRAM, fuses, MMIO and custom peripheral devices mapped into the address space. Therefore, allowing the emulation of OSs or bare-metal devices. A common use-case is emulating an ARM chip on an x86 desktop machine.

Section 2.3.2 presents the existing emulators for the ASP. To understand the terminologies and concepts used in chapter 3 and chapter 4, section 2.3.1 and section 2.3.3 explain the fundamentals of dynamic instruction translation and QEMU.

2.3.1 Basic Blocks & Translation

Most emulators are based on Just In Time (JIT) compilation. JIT means that the compilation for the target architecture happens at run time. An emulator needs JIT as it runs code compiled for one CPU on another CPU architecture. Instead of doing one full sweep, emulators translate one instruction or basic block at a time. This is advantageous as the code flow is only known during execution. Without knowledge about the code flow, it is difficult to predict which sections contain code that needs to be translated and what program data is. Also, emulators might not have access to all the code at the emulation start.

Even though emulators can translate one assembly instruction at a time, most of them translate it in blocks. This is mainly for performance reasons. Assembly instructions are typically grouped into basic blocks. The transition from one block to another is called an edge. An edge is defined by a start and an end block, just like in a graph context for nodes. Basic blocks play a significant role in compiler design and, thus, in emulators' dynamic translation. A basic block is defined as a code sequence with no branches except for the exit. Transferring that to assembly code means that it is a sequence of assembly instructions with only one entry point and one exit point. One entry point refers to the prerequisite that only the very first instruction can be the target of a branch/jump/call instruction. The exit point is the last assembly instruction in the sequence and has to be a branch/jump/call instruction or before the beginning of the following basic block entry point. This definition ensures that if the first instruction is executed, the following instruction of the basic block will also be executed as there is no branch/jump/call instruction changing the execution flow in between.

An example can be seen in fig. 2.2. The figure shows five basic blocks of the on-chip bootloader code. Each block consists of one or more ARM instructions. Besides one block, all of them end on a branch instruction (*beq, bne, b, bl*). The one block without a branch instruction ends because the next instruction (*bl 0xffff45f8*) is the entry point for another block. Note that emulators relax the basic block definition as they cannot predict if a basic block has to end earlier because the next instruction is the entry for another

basic block. Therefore, basic blocks for emulators always end on a branch/jump/call instruction, unlike shown in fig. 2.2.

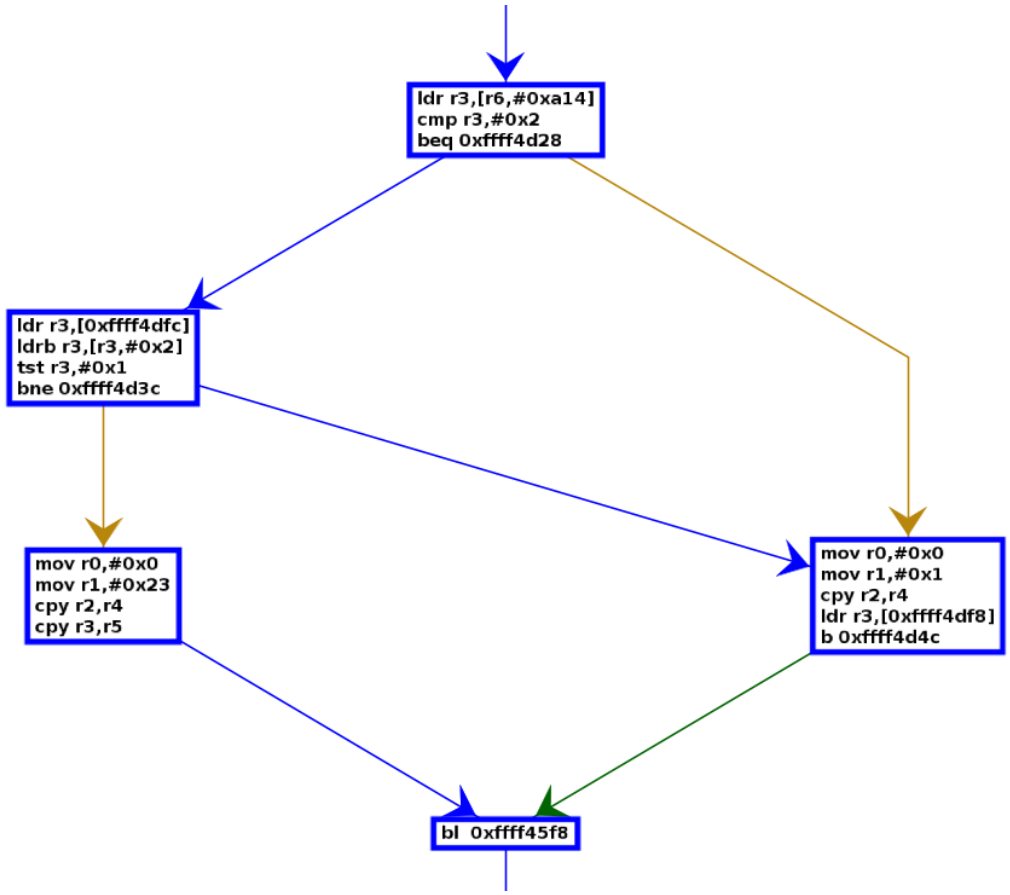

Fig. 2.2: Example basic blocks from the ASP on-chip bootloader represented by Ghidra [Ghi19]

2.3.2 ASP Emulators

[Eic20] presented PSPEmu [Eic21], the first emulator for the ASP. The Unicorn-based [Uni22] emulator can run the on-chip bootloader and most of the off-chip bootloader, but it breaks during the SecureOS execution. It supports the emulation of Ryzen Zen1, Zen+, Zen2 and Zen3 ASPs. Hardware accesses are forwarded to the real hardware for components that the emulator does not implement. Hence, this is not a full-system emulation but a partial emulation. Partial emulation forwards "peripheral interactions to the physical device" [Mue+18, p.8]. The author also points that out and mentions that Unicorn was not a good choice for the emulation of the ASP [Eic20, p.18].

To overcome the limitation of Unicorn, the ASP for QEMU [Har22] was built based on PSPEmu. Native QEMU, therefore, replaced Unicorn in this project. Besides supporting full-system mode, QEMU is also considered reasonably faster. At the time of writing, this emulator can only run the on-chip bootloader for Ryzen Zen1 and Zen+. Later versions might be able to run more Zen generations or even the whole off-chip bootloader.

2.3.3 QEMU

While many emulator choices exist, most only support specific architectures or applications. Frida [Fri22] only support user-mode emulation, WINE [Win22] only windows binaries, Nyx [SA22] only KVM-based emulation and Unicorn [Uni22] only user-mode or partial emulation. Only PANDA [PAN22] and QEMU [QEM22] allow full-system emulation for ARM32 targets which are required to emulate the ASP.

It would be possible to use PSPEmu for the fuzzer, but performance is one of the essential properties needed for fuzzing (see chapter 4). A brief comparison showed that PSPEmu is at least 10x slower than QEMU, which is why it was not chosen due to its slow emulation speed. On the other hand, PANDA would require porting PSPEmu to PANDA again. For this reason, the existing QEMU-based ASP emulator was picked as the baseline for this thesis because no new hardware implementation of the ASP is needed and it provides a reasonable emulation speed.

QEMU does JIT using the Tiny Code Generator (TCG). The TCG is responsible for lifting executed basic blocks into an Intermediate Representation (IR). Afterward, the IR is translated into the host CPUs architecture. Each of these Translated Blocks (TBs) is then cached to avoid re-translating every block on every single execution. Especially for loops and functions that are executed many times, this approach results in a significant speed-up.

The physical memory space of a CPU can be emulated using QEMU's softMMU. QEMU allows splitting the memory space into different memory regions with read/write/execute permissions. This can be used to implement SRAM or ROM regions. Peripheral devices can also be implemented and mapped into the guest CPUs physical address space. While this enables full-system emulation, it also slows down the emulation as every memory access has to be mapped to the correct memory region or peripheral device and the device's functionality has to be simulated. Therefore full-system emulations are significantly slower than user-mode emulations [Zhe+19].

2.4 Fuzzing

Fuzzing or fuzz-testing was introduced in 1990 by Miller et al. [MFS90]. It describes the automated testing of programs/devices/target using at least some actual random data from the fuzzer. It is similar to testing using test-cases, but instead of choosing inputs based on expert knowledge, the fuzzer generates the inputs. While test-cases come with predefined outputs, a fuzzer does not check the functional correctness but instead searches for abnormalities like crashes, hangs or undefined behavior. A fuzzer is, therefore, a tool to generate inputs and observes the outputs repetitively. This technique is used to find bugs and vulnerabilities in applications, most often software. Each entire run of the fuzzer for a target is called a fuzzing campaign. While hardware fuzzers [Kan+22], web API fuzzer [Li+13] and many more exist, this thesis will focus on software fuzzing. Therefore, the target of fuzzing campaigns is assumed to be a program under test running on a CPU. In the case of this thesis, it is the ASP's on-chip bootloader firmware running on a bare-metal ARM CPU.

Section 2.4.1 shows the general structure of a fuzzer and explains the different steps of a fuzzing campaign. In section 2.4.2 fuzzers are differentiated based on their input generation strategy and in section 2.4.3 on their target access. Section 2.4.4 explains the concept of instrumentation to gather feedback from fuzzing runs. Lastly, one specific fuzzer, the LibAFL fuzzer is presented in section 2.5.

2.4.1 Structure

Figure 2.3 shows the general structure of a fuzzer. The following 7 steps describe the stages of a fuzzing campaign.

1. A fuzzer can be seeded with initial inputs. This can be known, valid inputs or knowledge from prior knowledge. These inputs are stored in the corpus.
2. The fuzzer then generates new test-cases/inputs from the stored inputs in the corpus using a mutator or generator following an input generation strategy (see section 2.4.2).
3. First the target needs to be initialized. Depending on the target this might involve resetting the target to a specific state using a snapshot, reverting changes from any previous run or simply restarting the target. Afterward, a test-case is fed to

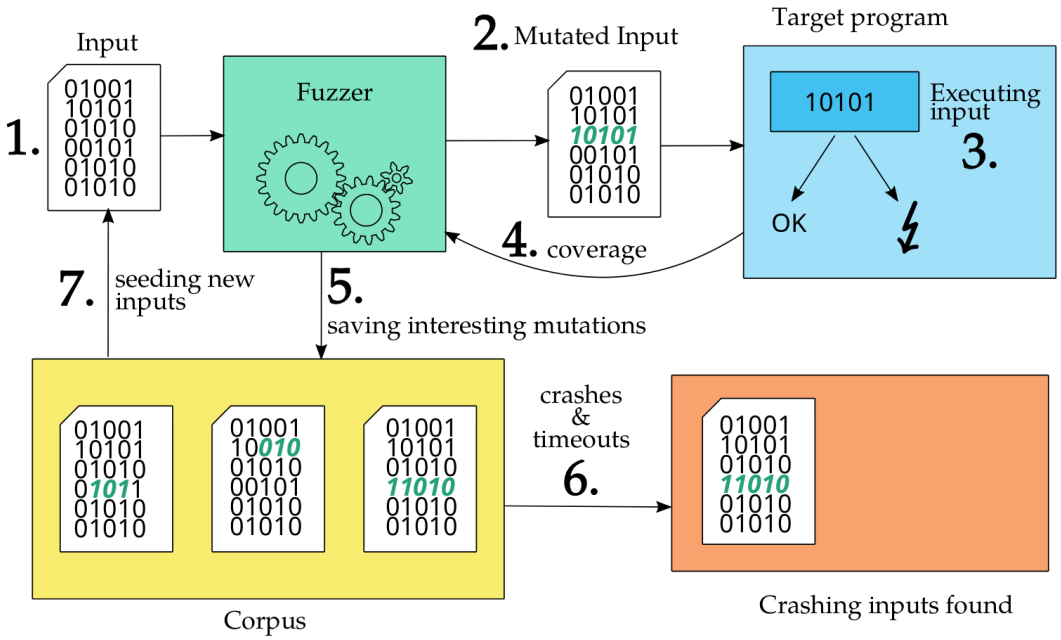

Fig. 2.3: General structure of a fuzzer [Lum22, p.5]

the target and the target then runs. The outputs might be evaluated to decide the outcome of the execution. These steps are referred to as the fuzzing harness.

4. Besides the outcome and any crashes or hangs, the code coverage also might be fed back to the fuzzer for coverage-guided fuzzers (section 2.4.2).
5. Interesting inputs are saved to the corpus.
6. The crashing inputs hold all inputs that are a solution. A solution can be anything but is always defined by an objective. Objectives can be any predefined conditions, states or outcomes. The most common ones are hangs or crashes.
7. The corpus is then used to pick new inputs for the next fuzzing iteration. A scheduler decides which inputs from the corpus are used for the next iteration.

2.4.2 Input Generation Strategy

While a fuzzer could generate random inputs, this is not a common approach as it would not be better than brute-forcing all possible inputs. Fuzzers are classed based on how

they generate inputs into mutation-based fuzzers, generation-based fuzzers and coverage-guided fuzzers. These classes are not mutually exclusive. Many fuzzers have features from multiple of these classes.

Mutation-based. Fuzzers generating new inputs by mutating an old input using specific rules are called mutation-based fuzzer. An example would be that the mutator picks a random byte in the input and flips all bits.

Generation-based. Generation-based fuzzers on the other hand generate new inputs based on a grammar. A grammar is a way to describe structures or formal languages like YAML. Therefore if only valid YAML files are expected by a program under test, then a generation-based fuzzer can be advantageous as it can consistently generate valid inputs, unlike a mutation-based fuzzer.

Coverage-guided. A fuzzer that uses code coverage feedback is called coverage-guided. To decide which inputs are interesting/novel, it checks if the test-case run reached new code paths. These inputs are then kept in a corpus and the whole corpus is used for the next round of mutations. Bypassing roadblocks like comparisons with large constants is the main reason for using coverage feedback in fuzzers [Fio+20, p.3].

2.4.3 Blackbox, Graybox and Whitebox Fuzzer

Blackbox fuzzing. In blackbox fuzzing, targets are treated as a blackbox. No instrumentation (see section 2.4.4) or feedback is being used for the generation of new inputs. The only information the fuzzer gets is if the target crashes or hangs. Looking at fig. 2.3, that means that steps 4,5 and 7 don't exist for blackbox fuzzing. This type of fuzzing is only necessary if the program/device under test cannot be instrumented (see section 2.4.4). Therefore, no feedback can be gathered from the target besides crashes or hangs. A target could be a remote application or a hardware device without access to the internals. IoTFuzzer [Che+18] is one example of a blackbox fuzzer. The ASPs CCP would also be a potential candidate in this category.

Whitebox fuzzing. Whitebox fuzzing is the opposite of blackbox fuzzing. It involves having full access to the target and deep knowledge of the underlying system. For software that would mean source code access. This allows analyzing the target after the execution using the collected coverage. Constraints can be formulated from roadblocks and solved to generate new inputs which bypass conditions in the executions. The approach is similar to symbolic execution [Kin76], just that it is not an exhaustive search but an iterative one. SAGE [GLM08] is one of the earliest tools for whitebox fuzzing. This technique is typically applicable to open-source projects or company internal, as all information about the target is available. AMD could apply this approach to their on-chip and off-chip bootloader code as they have the source code and complete knowledge about the ASP.

Graybox fuzzing. A graybox fuzzer does not require full access to the target, unlike whitebox fuzzer. For software, this means that only the binary image and not the source code might be available. Compared to blackbox fuzzing, it requires at least some sort of instrumentation, most predominately in the form of code coverage. This approach represents the case shown in fig. 2.3. Graybox fuzzer can often outperform whitebox fuzzers, because of their high speed due to the lightweight instrumentation [Yun+18]. AFL++ [Fio+20] and LibAFL [Fio+22] are prominent fuzzers supporting graybox fuzzing. The fuzzer built in this thesis is a graybox fuzzer as code coverage feedback from the emulation of the ASP is used.

2.4.4 Instrumentation

Instrumentation describes the ability to provide feedback from the program under test back to the fuzzer. Instrumentation of the target can be done statically or dynamically, statically by sanitizer, for example, provided by LLVM and GCC or dynamically by emulators like QEMU or Unicorn.

Static instrumentation. Sanitizers are compile-time instrumentations to detect bugs and collect code coverage. Besides gathering code coverage for graybox and whitebox fuzzing, sanitizers are primarily used to dynamically detect bugs that would otherwise not necessarily result in a crash. Sanitizers insert extra code at compile-time to detect these bugs and throw crash reports at runtime. A fuzzer without these sanitizers might miss

bugs because they don't lead to crashes. A typical example would be a buffer overflow by only a few bytes. Popular sanitizers are AddressSanitizer (ASan) for memory safety issues such as use-after-free, null pointer dereferencing and buffer overflow/underflow, UndefinedBehaviourSanitizer (UBSan) to detect signed integer overflows, misaligned pointers and more, MemorySanitizer (MSan) for uninitialized memory, LeakSanitizer (LSan) and ThreadSanitizer (TSan). Especially ASan [Ser+12] is widely adopted as buffer overflows are one of the most critical kinds of bugs.

Sanitizers can also be used if only the compiled target code is available as binary code. This technique is called binary rewriting. It requires decompiling the binary back into an IR and re-compiling it with sanitizers. Schulte et al. [SFB22] give an overview of the existing binary rewriting tools and their capabilities. Many different binary rewriting tools exist for the x86 architecture, like RetroWrite [Din+20] and Pin [Int05]. RetroWrite [Bar21] also supports aarch64. μSBS [SHC20] is a binary rewriting tool specifically for ARM architecture and likely the only one which potentially could work for the ASP on-chip bootloader.

Dynamic instrumentation. If the source code is unavailable or changing the binary image is not an option, then static instrumentation is the only option. While there are some existing implementations of sanitizer-like features for emulators, they are sparse. This is mainly because many metadata, like buffer sizes, are lost at compile time which would be required to detect bugs. Also, each emulator has its own instrumentation capabilities, as each target needs a specific solution for dynamic instrumentation. Dynamic instrumentation builds on top of the emulator's translation and inserts additional instructions into the translation to detect bugs. Code coverage is one feature that can be easily retrieved using the emulator's translation. In QEMU this needs to happen in the TCG (see section 2.3.3). Vanilla QEMU does not come with dynamic instrumentation capabilities. Besides that, QEMUAddressSanitizer (QASan) [FDQ20] implements ASan for QEMU user-mode emulations. The author of this thesis is not aware of any ASan-like feature for QEMU full-system emulation.

2.5 LibAFL

There exist many different state-of-the-art fuzzer like AFL++ [Fio+20], libFuzzer [Pro18] or honggfuzz [Swi22]. While they are all widely used, most researchers and security

analysts have their own version/fork of them to integrate their target or change parts of the functionality. As a result, many different flavors of each fuzzer exist because combining different forks is cumbersome. LibAFL [Fio+22] was recently published and tries to fill this gap by providing a framework to build modular and reusable fuzzers in Rust. The aim is that LibAFL only provides the general structure of the fuzzer while each component can be individually picked or built. A component like a mutator is part of every fuzzer. LibAFL provides a *trait* for a generic mutator but leaves it to the user to pick one of the provided mutators or implement a new one. Instead of customizing the fuzzer using command-line flags, LibAFL is a library where each component has to be configured in Rust code. Therefore, if researchers implement a promising new version of a component, it can be easily added to the LibAFL project without breaking anything. Users can choose if they want to use it or not.

LibAFL consists of many components, as seen in fig. 2.4. The most important ones will be introduced shortly, leaving the details for the later section where they are needed.

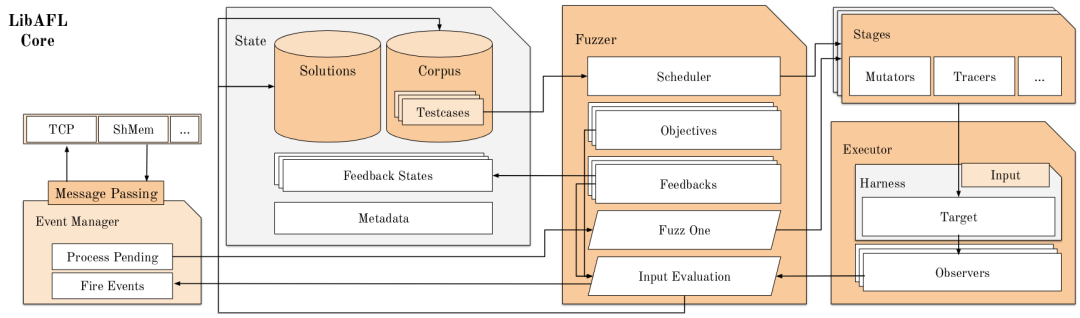

Fig. 2.4: Simplified structure of the LibAFL components [Fio+22, p.5]

Input. This component is the representation of the input. In the simplest case, this is a byte-array.

Corpus. The corpus is the way the inputs and their metadata are stored. They are either stored in memory for performance reasons or on disk if memory is a bottleneck. One corpus exists for the interesting test-cases and one for the solutions of the fuzzer. Both of these can be individually defined using the feedback component.

Scheduler. Whenever the fuzzer needs a new test-case for the stage component, the scheduler picks one entry from the corpus. This can be a simple random selection, a First-In First-Out (FIFO) or any more advanced decision-making algorithm.

Stage. The stage defines the actions which are performed on a single test-case. The fuzzer gets this test-case from the scheduler and runs every stage on this input. Stages can, for example, be one or more mutations.

Observer. This component holds information about a single execution of the target. A common example is a coverage map.

Executor. The executor runs the target with the input given from the fuzzer. The executor uses the harness to place the inputs where the target expects them.

Feedback. The feedback component classifies the outcome of a test-case execution. The observer information can be used to decide whether an input is considered interesting or not. Multiple different feedbacks can also be combined. Feedback is referred to as objective if it is used for the corpus of the solutions.

Mutator. Mutators derive new inputs from existing inputs.

Generator. The generator component can generate new inputs from scratch. This can be useful for initial inputs or a generation-based fuzzer using a grammar.

3 Emulation

To run the on-chip bootloader code for fuzzing there are three options.

1. The straightforward approach would be to run it on the actual hardware. This would mean that the input has to be provided over SPI to an AMD Zen chip using a flash programmer. This way, only limited feedback could be gathered as Joint Test Action Group (JTAG) debugging is not enabled for the ASP on Zen chips. Furthermore, the speed of the fuzzer would be limited by the number of AMD chips connected to the machine running the fuzzer. Especially the price for scaling would be very high as each ARM chip needs at least a motherboard, power supply and a flash programmer. Every new Zen generation would also require new chips and potentially new motherboards.

2. The second option would be to use a different ARMv7-A ISA compatible chip with KVM support. ARM instructions could be executed natively while memory layout-specific operations like MMIO or SMN read/writes have to be emulated. Feedback could be gathered using the ARM CoreSight [ARM22] feature. This approach would have the advantage of gathering feedback, but the scaling issue persists. Even though the price would be significantly decreased, orchestrating all devices and connecting them via cable or over the network would be a non-neglectable effort.

3. The last option is to emulate the entire ASP on a different CPU. It includes translating the ARM instructions into the host CPU ISA, emulating all memory accesses and simulating all MMIO devices. Therefore it requires the existence of a full-system emulator for the ASP, which did not exist for a long time (see section 2.3.2). Especially for proprietary hardware, this is barely ever the case and a unique situation for the ASP. It comes with the advantage that feedback can be gathered from the emulator (see section 2.3.3) while scaling with the computational power of the machines running the fuzzer. Increasing the fuzzing throughput only requires more on-sight servers or renting cloud servers which is cheaper and less tedious than the first two options. One disadvantage is that the emulation might not be

the same as executing the on-chip bootloader on real hardware. Therefore, false positives and false negatives are possible, meaning that a bug in the emulation might exist that does not exist in hardware and vice versa. Even with access to the HDL design, this could never be ruled out.

The third option was chosen for this thesis as it provides the most feedback while keeping the cost and setup effort low. Therefore, section 2.3 already presented the general concept of emulation. Specifically, the existing emulators for the ASP were introduced in section 2.3.2. Section 3.1 describes the changes made to the existing QEMU-based emulator for the ASP. In section 3.2 the QEMU patches to integrate LibAFL are explained. The last subsection (section 3.3) shows how the two patched QEMU versions were merged and some additional patches which were applied afterward. The resulting QEMU fork is named qemu-libafl-asp and is publicly available over GitHub[1].

3.1 QEMU ASP

QEMU ASP [Har22] is the QEMU-based emulator supporting the ASP for Ryzen Zen1 and Zen+. Section 2.3.3 describes how QEMU works in general and section 2.2 the behaviour of the ASP that was implemented in QEMU ASP. After building QEMU, the ASP emulator can be started as shown in listing 3.1. The environment variables ZEN_GENERATION, ASP_UEFI_IMAGE and ASP_ROM_BL have to be provided. The ZEN_GENERATION sets the Zen generation, which should be emulated. ASP_ROM_BL is the path to the ROM code binary for the specified Zen generation. Acquiring the bootloader involves one of the attacks presented in section 2.2.3. ASP_UEFI_IMAGE is the path to the UEFI image to use. These can be gathered from the motherboard provider's website.

```
./build/arm-softmmu/qemu-system-arm
    --machine \
    amd-psp-$ZEN_GENERATION \
    --nographic \
    -device \
    loader,file=$ASP_ROM_BL,addr=0xffff0000,force-raw=on \
    -global \
    driver=amd_psp.smnflash,property=flash_img,value=$ASP_UEFI_IMAGE \
    -bios \
    $ASP_UEFI_IMAGE \
```

Listing 3.1: Bash command to start the ASP emulator

[1] https://github.com/TeumessianFox/qemu-libafl-asp

To improve the emulator and add extra functionality, the following changes were made to the QEMU ASP emulator by Haprecht.

SCTLR. To start the ASP at the high vector address `0xffff0000` (see section 2.2.1), the Program Counter (PC) was set to `0xffff0000` at the start of the emulation. While this works to start the CPU at the correct address, it does not set the exception and interrupt vector to the high vector address. As a result, whenever an interrupt or exception happened, the emulator used the address `0x0` as the exception base, not `0xffff0000`. Therefore the exception could not be handled correctly as the desired exception vector table is located in ROM at `0xffff0000`. Especially for fuzzing, where triggering exceptions is desired, this resulted in undesired behavior. To fix this, the SCTLR [ARM16, p.88] has to be set to the correct default value (`0x00c52078`). The SCTLR configures features like the exception vector base address, instruction caching, data caching or memory access alignments faults (fixed in commit 62e7d93).

CCP. The original implementation of the CCP tried to resemble the hardware behavior as closely as possible. In real hardware, the CCP needs a small amount of time to handle an operation. Even though the precise timing is unknown, the CCP was implemented to take 100ns for each operation. During fuzzing, this resulted in errors because many threads were running on the machine. Therefore threads might not be scheduled all the time. In the worst case, a CCP operation is started and the thread stops executing for more than 100ns. The same happens when debugging the ASP using gdb if the execution is stopped during a CCP operation using a breakpoint. As a result, the CCP operation finishes much faster than expected. If the operation writes to memory like the passthrough operation, then the memory can be overwritten, resulting in abnormal behavior by the ASP. To fix this issue, the CCP was changed to execute the operation only once the code checks the CCP status register to see if the last operation has already been finished (commit 9549754). This change improved the emulation speed and stability.

Zen2 and Zen3. Thanks to the work by Pascal Haprecht, QEMU ASP already included most functionality needed for Ryzen Zen2 and Zen3. The emulator successfully ran the on-chip bootloader for Zen2 and Zen3 until the flash memory was mapped to the CPU's physical memory using the SMN. Using static and dynamic binary analysis, the missing components were found. A missing fuse value could be taken straight from PSPEmu. Furthermore, Zen2 and Zen3 use SHA384 and RSA4096 instead of SHA256 and RSA2048.

Both were implemented in QEMU, resulting in functional on-chip bootloaders for Zen2 and Zen3 in QEMU ASP (commit 9549754 and 7e63490).

ZenTesla. The Zen architecture in AMD chips used by Tesla is referred to as ZenTesla. ZenTesla is similar to Zen1 and Zen+ but does not have the known buffer overflow present in the on-chip bootloader for Zen1 and Zen+ chips. Multiple MMIO addresses slightly differ from Zen1 and Zen+ (commit c57a0dc). Additionally, the ZenTesla chips do not use AMDs ARK. Instead, the SHA256 for Teslas ARK is believed to be written to a fuse and used for the key verification. The relevant fuse values were not implemented for the QEMU emulator. Therefore the ARK verification has to be skipped to run the complete on-chip bootloader in QEMU.

3.2 LibAFL Integration

Running QEMU with LibAFL requires some changes to QEMU. These changes are necessary to allow feedback and hooks. The qemu-libafl-bridge project [AFL22b] provides precisely these changes.

While working with qemu-libafl-bridge a mistake in the command line option parser during the build process was discovered. An early wildcard character in the configure script prevented the enabling/disabling of certain QEMU features. The pull request #12 to qemu-libafl-bridge addressed the issue and was merged in the process of writing this thesis.

3.2.1 Hooks

Hooks are callbacks to the fuzzer whenever a specific constraint during the execution of the target program is met. Running target code like the on-chip bootloader in a full-system emulator allows hooking any event. Hooks have to be provided by the fuzzer, but the emulator needs to support them. Therefore hooks rely on the capabilities of the emulator. The most common ones are read, write, compare, block and edge hooks. Read hooks trigger on memory read operations like *ldr* for ARM. Write hooks likewise trigger on memory writes like *str*. Triggering on compare instructions is used to overcome

roadblocks like input comparisons with 32-bit constants. Block hooks and edge hooks are used for coverage feedback or to build a call stack.

No matter which kind of hook, they always require two callback functions in qemu-libafl-bridge, a generation and an execution callback. The generation callback is only triggered once when the basic block is translated. As described in section 2.3.3, basic blocks are only translated once in QEMU and cached to avoid re-translating the same block. An example would be the translation of a basic block, including a *ldr* instruction. Every generation callback for read hooks would be triggered. The callback gets the memory address where the *ldr* instruction is at. Based on the location, each generation hook can decide if the corresponding execution hooks should be triggered whenever this instruction is hit. This is useful if only specific load instructions should be hooked. The execution hook is then inserted into the translated basic block and triggered every single time this basic block runs. Execution hooks then get, for example, the address from which the load instruction reads data. This can be used to gather data from the emulation based on any criteria defined by execution callback functions. A typical example is a write hook to a memory area that should not be written to.

3.2.2 Feedback

Lightweight coverage feedback is the most essential feature for graybox fuzzing. QEMU will be linked with LibAFL (see chapter 4). Therefore LibAFL can access all global variables and functions provided by QEMU. Inserting some code into QEMU allows for intercepting the execution if certain conditions are met with callbacks. This approach is called a hook. Hooks are explained in section 3.2.1. For example, whenever a transition from one basic block to another one happens, an execution callback will be triggered, which runs some LibAFL code. This LibAFL code can thus save the edge coverage in a coverage map which is then accessible by the fuzzer. The same could be done to acquire block coverage instead of edge coverage. A coverage map can be as simple as an array with one entry for each edge. Each entry can either represent how often each edge was executed or only if the edge was ever executed. As the number of edges is unknown at the emulation start, the array has to scale dynamically.

3.3 Custom QEMU Build

Figure 3.1 shows how the different QEMU forks were built on top of vanilla QEMU. The qemu-libafl-asp project is a merged version of QEMU ASP and qemu-libafl-bridge with some additional patches. QEMU ASP and qemu-libafl-bridge are both forks of QEMU version 7, but different minor versions. Both projects irregularly rebase to the latest vanilla QEMU version. Therefore the qemu-libafl-asp project is never up-to-date with the latest vanilla QEMU version as it would require both projects to be up-to-date. In general, QEMU regularly changes function interfaces making it hard to merge forks with different minor versions. The additional changes made to the qemu-libafl-asp emulator for fuzzing are described below.

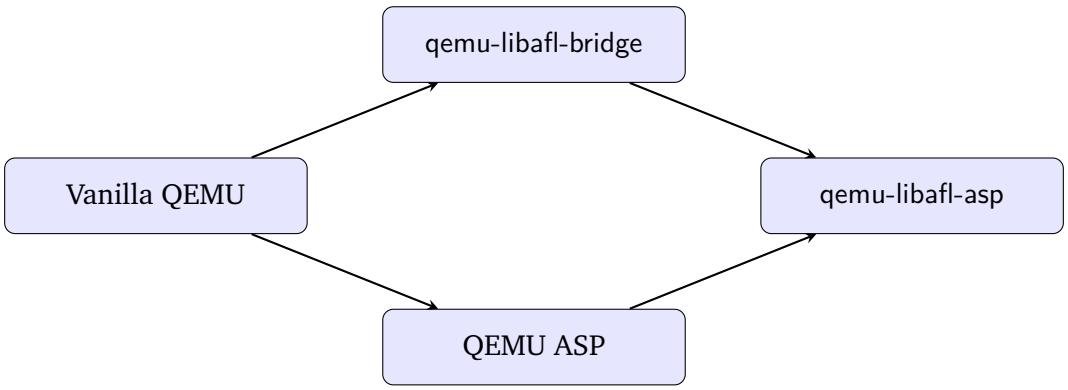

Fig. 3.1: Figure showing the dependency of each used QEMU fork to the vanilla QEMU project

Timer/SMN. The two timers are part of the MMIO memory region. To preserve the state of the timers in between fuzzing test-cases, the fuzzer needs the ability to read and write the internal state of the timers. Therefore the internal timer values for the count and control were made accessible for both timers through global variables. The same approach was made for the SMN controller. This allows the fuzzer to reset any changes to the SMN slots during the fuzzing execution back to the original state. As the QEMU code is compiled into the fuzzer, these globals are then accessible in the LibAFL fuzzer. Further details about the linking can be found in chapter 4.

Flash memory. QEMU had to be extended to provide the fuzzer with a fast and reliable method to insert fuzzing inputs into the flash memory. Even though QEMU already exposes a function to write to the physical memory of the CPU, this only help when the

flash memory is mapped to the SMN slots. For Zen2 and Zen3 the flash memory is never completely mapped into the SMN slots. Therefore adding the functionality to directly write to the flash memory space in the SMN internal memory space allows the fuzzer to insert inputs at any moment during the emulation.

4 ASPFuzz: The ASP Fuzzer

ASPFuzz[1] is the fuzzer for the ASP built in this thesis. Even though this thesis only focuses on the on-chip bootloader, the ASPFuzz fuzzer is also capable of fuzzing the off-chip bootloader once the underlying QEMU version supports the execution of the off-chip bootloader. It is based on the LibAFL [Fio+22] fuzzing framework. LibAFL was already briefly introduced in section 2.5. One major advantage is that LibAFL supports QEMU (qemu-libafl-bridge) natively while being easily extensible. Therefore qemu-libafl-asp, which is also a fork of qemu-libafl-bridge (see section 3.3), can be used as the ASP emulator for fuzzing with LibAFL.

LibAFL is a relatively recent project under active development and constantly changing. Even though all the state-of-the-art features for a fuzzer are already present, the project improves the functionalities, features and extendability with every major version. To preserve the functionality of ASPFuzz, the LibAFL version was fixed to version 0.8.2 commit 0515eeb from Nov 20, 2022 (see section 4.2.1 for the reasoning). Later versions might break the fuzzer as function interfaces and trait definitions often change, especially for the QEMU mode.

First, section 4.1 introduces how QEMU is integrated into LibAFL. Section 4.2 shows the patches made to LibAFL to improve its functionality. In section 4.3, multiple custom snapshotting implementations are presented. Objectives can be defined as specified in section 4.4. ASPFuzz has to be configured using the YAML parser from section 4.5. The fuzzer output structure is shown in section 4.6.

4.1 QEMU Integration

LibAFL is written in Rust. Rust was developed as a secure system programming language similar to C but providing memory safety guarantees at compile-time. Memory safety is reached by Rust's borrow checker, which statically analyses the lifetime of variables,

[1] https://github.com/TeumessianFox/ASPFuzz

theoretically not producing any memory-unsafe code at runtime. Sometimes Rust cannot infer if a variable or operation is memory-safe using static analysis. Therefore Rust provides the *unsafe* keyword. It allows inherently unsafe operations in well-defined bounds but leaves it to the programmer to ensure memory safety. It can be seen as a second, memory-unsafe programming language within Rust. These “unsafe superpowers” [Rus22] include

- dereferencing raw pointers
- calling unsafe functions
- accessing or modifying mutable static variables
- implementing unsafe traits
- accessing fields in unions

Especially calling unsafe functions is essential to integrate QEMU into LibAFL. QEMU is written in C and can be linked into Rust. This allows calling functions from QEMU in Rust using the Foreign Function Interface (FFI) [Rus22] and vice versa. FFI’s *extern* keyword allows importing functions from a foreign programming language or making Rust functions accessible to foreign programming languages. The *unsafe* keyword can then be used to execute the foreign functions. Listing 4.1 shows an example usage of the FFI in LibAFL for QEMU. The C function "libafl_add_edge_hook" is exposed to Rust using *extern* (line 1-7) and called using *unsafe* (line 14). This function allows the insertion of generation and execution callbacks for edge hooks as described in section 3.2.1 to gather edge coverage feedback.

```
1  extern "C" {
2      fn libafl_add_edge_hook(
3          gen: Option<extern "C" fn(GuestAddr, GuestAddr, u64) -> u64>,
4          exec: Option<extern "C" fn(u64, u64)>,
5          data: u64,
6      );
7  }
8  pub fn add_edge_hooks(
9      &self,
10     gen: Option<extern "C" fn(GuestAddr, GuestAddr, u64) -> u64>,
11     exec: Option<extern "C" fn(u64, u64)>,
12     data: u64,
13 ) {
14     unsafe { libafl_add_edge_hook(gen, exec, data) }
```

```
15 }
```

Listing 4.1: Example for the Rust FFI taken from LibAFL source code [Fio+22]

4.2 LibAFL Patches

This section presents the patches made to LibAFL through the development of ASPFuzz. Each section describes why these changes were necessary for this work and their usage in ASPFuzz. Therefore these sections are core contributions of this thesis. Additionally, three minor changes were also made, which have almost no impact on the fuzzer as they are only for user convenience (pull requests #699, #752 and #970). All of them were suggested to the LibAFL community as a pull request on the public GitHub repository and eventually merged into LibAFL.

4.2.1 Full-System ARM Support

To get familiar with LibAFL and QEMU a simple dummy was built based on the existing "qemu_launcher" example. The existing "qemu_launcher" could fuzz a modified libpng parser using QEMU user-mode emulation. The libpng parser was therefore compiled for x86. The newly created "qemu_arm_launcher" compiles the libpng parser for ARM32 and modifies the LibAFL fuzzer to function with ARM assembly instead of x86 assembly (pull request #708). This test showed that LibAFL is capable of fuzzing ARM32 code, which is required for the ASP on-chip bootloader as it is ARM32 code. In the testing process, it was noticed that the fuzzer could not recover from crashes resulting in interrupts/exceptions. The issue was that LibAFL could only access the 16 general-purpose registers to reset the state between test-cases and not the Current Processor Status Register (CPSR). The CPSR holds the status of the execution including exception masks, condition flags, the mode field and the thumb execution mode. If not properly reset, the processor will keep trigger exceptions right after the start of each new test-case (fixed with pull request #800). The last thing that was left is switching QEMU from user-mode to full-system emulation. There was no existing example for full-system emulations and LibAFL was also crashing if QEMU was set to full-system emulation. First, the cargo build script has been adapted to support a systemmode flag and build full-system QEMU so that no matter which libraries are present on a machine, it compiles and links correctly with

LibAFL. Furthermore, this systemmode flag is passed to the underlying Rust code as a Rust feature to allow full-system emulation of specific LibAFL code. Address translation was fixed to use physical addresses instead of virtual ones. Also, the *cpu_reset* function from QEMU was exposed to LibAFL to enable resetting the processor from within LibAFL. Lastly, ARM addresses were corrected to properly handle ARM interworking addresses (pull request #737). With these changes, it was possible to run the ASP using `qemu-libafl-asp` in LibAFL.

Starting from Nov 21, 2022, there has been a lot of activity in the LibAFL community involving full-system QEMU. An example fuzzer was created and the whole interface and support for QEMU in LibAFL started to be reworked. Every new patch brought new features, incompatibilities with old versions, and sometimes new problems. To keep the fuzzer functional, the LibAFL version for ASPFuzz was fixed to a version before the changes started. Once the rework of full-system QEMU in LibAFL has finished, it will be helpful to upgrade LibAFL again to use the new features.

4.2.2 Multi-Core Fuzzing

LibAFL comes with the capability to do fuzzing on multiple instances/cores. For that purpose, LibAFL implemented the Low Level Message Passing (LLMP) protocol to exchange information like test-cases, metadata and statistics between the fuzzing instances. One instance is therefore chosen as the "broker" which forwards messages from any "client" instance to all other ones and filters them for messages to display to the user. This also works for instances connected over TCP [AFL22a].

For Unix-like systems, LibAFL spawns the instances using *fork*. While this is the fastest for most user-mode applications, it does not work for full-system QEMU because of its multi-threaded nature. Therefore LibAFL also supports spawning cores with *execve* on Unix. Instead of forking the process, the broker spawns clients by starting the same program with the same environment variables as the broker in a new process. This new process is then informed about its client status over an environment variable. Unfortunately, only Unix systems without *fork* support were able to use LibAFL with *execve*. Pull request #806 and pull request #814 address this issue. As a result, LibAFL with full-system QEMU and, therefore also ASPFuzz can fuzz on as many cores as the machine running the fuzzer provides. Finally, test-cases can be forwarded to every client using LLMP, while the broker shows statistics. The multi-core performance and statistics from ASPFuzz are shown in chapter 6.

4.2.3 DrCov

DrCov is a commonly used coverage file format. Plugins like Lighthouse [Gaa22] for IDA Pro and Binary Ninja or Lightkeeper [Wor22] for Ghidra can highlight code coverage using DrCov file. The DrCov file system is not officially documented but is yet widely used.

The DrCov file header is decoded in ASCII and, therefore human-readable. A DrCov file header starts with some metadata like the DrCov version. After that, there is the module table. Each module has an id, a start, an end address and a file path to the actual modules binary. This can be seen in listing 4.2. In the case of the ASP there is only the on-chip bootloader as a module. Lastly, there is the basic block table. While the number of basic blocks is still part of the ASCII-encoded header, the actual basic blocks are encoded in binary and can be seen as the DrCov file body. Each basic block has a start and an end address as well as an id to identify which module it is in [Ayr18].

```
1 DRCOV VERSION: 2
2 DRCOV FLAVOR: libafl
3 Module Table: version 2, count 1
4 Columns: id, base, end, entry, checksum, timestamp, path
5 000, 0x0, 0xffff9000, 0x0, 0x0, 0x0, on-chip-ryzen-zen.bl
6 BB Table: 719 bbs
```

Listing 4.2: DrCov file header from an ASPFuzz run

Even though LibAFL supports DrCov for the Frida emulator and can write DrCov files, there was no support for QEMU. To support DrCov each basic block has to be hooked using block hooks. Execution callbacks can be used to count the number of block hits but slow down the emulator as the execution hook is triggered on each basic block execution. On the other hand, generation callbacks are enough to decide if a basic block was ever executed, as they only get called once per basic block. Compared to execution callbacks, generation callbacks almost do not cause any slowdown of the emulator. Both versions were implemented for LibAFL. QEMU execution and generation callbacks only provide the start address of a basic block. This is because QEMU inserts the callbacks for hooks before the basic block translation. Therefore the basic block end address is not known but is needed to output the DrCov file. As a solution, the assembly code can be gathered from QEMU and translated one instruction at a time until a branch/jump/call instruction is reached. Capstone [Cap22] is the binary disassembler used to disassemble bytes into the corresponding ARM instructions. Now that the start and end address of each executed basic block (and potentially the execution count) are known, the DrCov

file writer, which was already built into LibAFL, could be used to output the DrCov file (pull request #878).

Loading a DrCov file from an ASPFuzz run into Ghidra's lightkeeper plugin highlights the executed basic blocks and shows the code coverage per function. An example can be seen in fig. 4.1. A visual representation of the code coverage is helpful to investigate which code sections were not reached by the fuzzer, because of roadblocks. These roadblocks can then be analyzed and the fuzzer can be adapted to overcome them. This process was used throughout the thesis to improve the fuzzer.

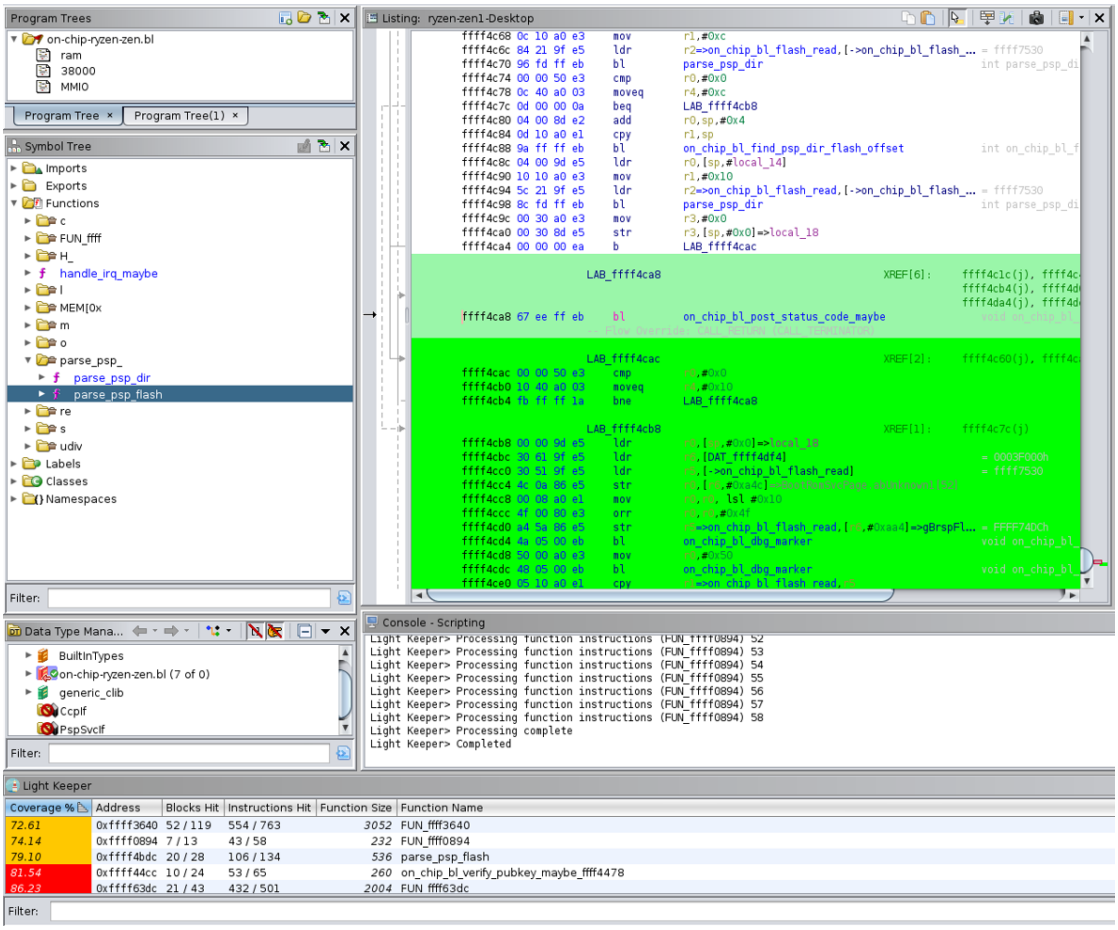

Fig. 4.1: Code coverage shown in Ghidra's lightkeeper plugin using a DrCov file

4.3 Custom Snapshotting

Snapshotting is the way for the fuzzer to reset the emulator's state in between the execution of test-cases. Vanilla QEMU comes with a snapshotting feature. QEMU's snapshotting feature was built to save an emulator's complete state to continue execution another time and not for fuzzing, where snapshots are loaded and saved multiple times a minute. Therefore it is no surprise that QEMU's snapshotting feature is not usable for fuzzing. Prior research has shown that QEMU snapshots can take up to $0.5s$ to load a snapshot [Ngu+22] depending on the machine, resulting in low fuzzing throughput. Tests with similar machines as the ASP resulted in at least $15 - 20ms$ for loading a snapshot. To drastically increase the fuzzing speed, multiple custom snapshot implementations were built. They vary by the amount of state information they save and load. An overview and the average speed of each snapshotting implementation can be found in table 4.1.

	SuperLazy	Lazy	RustSnapshot	HardReset
16 Registers	x	x	x	x
CPSR	x	x	x	x
BootROMServicePage		x	x	x
SRAM			x	x
Timer			x	x
SMN Controller			x	x
CPU				x
Zen1 Speed	$1.31\mu s$	$2.69\mu s$	$82.98\mu s$	$13711.58\mu s$
Zen+ Speed	$1.27\mu s$	$2.61\mu s$	$83.63\mu s$	$13768.22\mu s$
Zen2 Speed	$1.25\mu s$	$2.56\mu s$	$103.80\mu s$	$11217.86\mu s$
Zen3 Speed	$1.29\mu s$	$2.69\mu s$	$103.80\mu s$	$11403.29\mu s$

Tab. 4.1: Comparision of the different snapshot levels. The average snapshot load speed was calculated over 10000 load executions on an Intel Xeon E7-4870 at 2.40GHz.

***SuperLazy*.** The *SuperLazy* snapshot only saves the 16 registers (R0-R15) and the CPSR register. If the target program does not include any memory operation or changes the internals of MMIO devices, it might already be enough only to load and save these registers. Interrupts/exceptions are already appropriately handled in this snapshotting

version as the CPSR is saved. This is very important for full-system fuzzing, as exceptions are likely to occur if a buffer overflow is found. If memory operations only involve stack operations, resetting the Stack Pointer (SP) register is often sufficient.

Lazy. Besides the stack, there can also be a heap or global variables in SRAM. The on-chip bootloader does not have a heap. Therefore only global variables like the BootROMServicePage have to be saved. The BootROMServicePage is the largest known area that stores information during the on-chip bootloader. *Lazy* snapshots thus save the same as the *SuperLazy* snapshot and additionally the SRAM from `[0x3ED00,0x40000]` (`[0x4ED00,0x50000]` for Zen2 and Zen3). The BootROMServicePage is part of the saved SRAM region. While the *SuperLazy* snapshot could be used for a generic ARM Cortex device, this snapshot is adapted to the on-chip bootloader using prior knowledge of the binary.

RustSnapshot. The *RustSnapshot* is similar to the QEMU snapshot. Instead of saving the internal structs for the CPU and all devices, it copies only the necessary CPU and device information to reset their state. This includes all 16 registers, the CPSR, the SRAM region (`[0x0,0x40000]` for Zen1 and Zen+ and `[0x0,0x50000]` for Zen2 and Zen3) the internal state of the ASP timers and the SMN controller. Note that the differences in load speed in table 4.1 for the Zen generations come from the size of the SRAM. While QEMU snapshot is built to snapshot a machine in QEMU, this snapshot variant is implemented strictly for the ASP machine. This has the advantage that knowledge about the devices can be used to decide whether a state needs to be saved. While the timers have an internal counter value that needs to be saved, the fuses MMIO device, for example, does not change during the execution. Here the QEMU snapshot would load/save the fuse internal struct in its completeness, which is not required and only results in a slower snapshotting. Also, much more metadata than the counter value would be saved/loaded for the timer. The SMN controller state has to be saved, as the slot mapping can change during the execution. A thorough analysis has shown that the state of the CCP does not need to be saved if the harness is not set up to start in the middle of a CCP operation. Therefore the CCP state was not saved because the harnesses described in chapter 5, follow this rule.

HardReset. The *HardReset* snapshot resets the CPU, timer devices, SMN devices and the SRAM to the initial state at machine start. It can be seen as a restart of the entire

emulator, just that it is faster. The state at the machine start might differ from that at the beginning of the test-case. That is because a harness can decide that only parts of the program depend on the fuzzer input. Therefore initial setup code before that does not change the outcome of a fuzzing run and thus only needs to run once. This is achieved by running the program once from the start until a chosen point in the program, then taking the snapshot and always starting test-cases from this point. For the case of the on-chip bootloader, this is done because the first code executed is independent of the fuzzer input, because it only configures MMIO devices. After resetting the machine, the *HardReset* snapshot runs the on-chip bootloader until the harness entry. The *HardReset* is faster for Zen2 and Zen3 as the initial code is less than for Zen1 and Zen+.

4.4 Objective Definition

The objective is a feedback for the solutions. The feedback defines which conditions must be met for a test-case so that it is added to the corpus or solutions. Solutions are also a corpus in LibAFL only that they hold the outcomes of a fuzzing campaign compared to the actual corpus, which holds the interesting inputs (see section 2.5) Therefore test-cases are added to the solutions depending on the objectives defined by the fuzzer. Objectives are defined to detect as many bugs as possible.

ASPFuzz uses multiple different feedbacks, which are connected by boolean operators, to construct the final objective. First, it uses exceptions as an objective. An exception (*ExceptionFeedback*) is defined as a jump to any entry in the exception vector which is at `[0xffff0000,0xffff001c]`. Crashes are the second objective and the predominant one. A crash (*CrashFeedback*) can be triggered on any PC value or by write and execute permission on memory areas. The YAML configuration file allows setting arbitrary values for the PC and memory areas that are not allowed to be written to or executed. Section 4.5 will describe these configuration options in-depth. Lastly, the coverage feedback (*MaxMapFeedback*), which is also used for the corpus feedback, is used to avoid multiple solutions with the same crash or exceptions. Therefore, only if the test-case results in a crash or exception and has a new novel coverage map is added to the corpus of the solutions. This approach using the coverage map as an objective already results in solution deduplication. Instead of keeping every test-case in the corpus of the solutions that triggers a crash or exception, only those with a novel coverage map are kept. Even though the same bug can still be multiple times in the solutions, maximum once for every

possible code path leading to this bug. A representation as a boolean expression can be seen in eq. (4.1).

$$(ExceptionFeedback \,\&\, CrashFeedback) \,||\, MaxMapFeedback \tag{4.1}$$

4.5 YAML Configuration

YAML is a human-readable language mostly used for configuration files as it can conveniently serialize data structures. YAML files consist of key-value pairs. The keys are referred to as tags. The YAML parser is the most essential component of the ASPFuzz fuzzer. It allows configuring the fuzzer using a YAML file at runtime. Therefore adding new functionality, new Zen generations or even fuzzing the off-chip bootloader can be as simple as writing a new YAML file. This section describes all the configuration options available over a YAML file with Zen1 as an example. Chapter 5 will present how the configuration values for Zen1 were found.

QEMU. Listing 4.3 shows the configuration options for the QEMU tag. In this section of the YAML file, the Zen generation and the path to the on-chip bootloader binary have to be provided.

```
qemu:
    # Zen generation to emulate
    zen:                "Zen1"
    # On-chip bootloader to use
    on_chip_bl_path:    "bins/on-chip-bl-Ryzen-Zen1-Desktop"
```

Listing 4.3: YAML options to configure QEMU in ASPFuzz

Flash. The flash memory is the only viable input for the fuzzer as it is the only attacker-controlled input for the on-chip bootloader. This tag gives information about where the flash is in the SMN memory space and where it is mapped to in the SMN controller. The size of the flash memory has to be set as well. Also, a base flash image has to be provided. This image must be valid for the Zen generation chosen in the QEMU tag. An example configuration for Zen1 can be seen in listing 4.4.

```
flash:
    # Start of flash mmap in SMN memory space
    start_smn:      0x0a000000
    # Size of flash memory
    size:           0x01000000
    # Start of flash mmap area in cpu physical memory
    start_cpu:      0x02000000
    # Base image in flash memory
    base:           "bins/PRIME-X370-PRO-ASUS-3803.ROM"
```

Listing 4.4: YAML options to configure the flash memory in ASPFuzz

Input. This tag of the YAML file configures how the fuzzer handles the inputs to the target. First, one or more UEFI images can be given to the fuzzer as initial inputs. If no inputs are provided, the fuzzer starts from an empty corpus. Next, the memory areas in the flash image where the fuzzer should write the inputs must be specified. As the whole flash memory space could be chosen for fuzzing inputs, this could mean an input space of up to 16 MiB. Therefore this tag allows multiple smaller subsections of the whole flash memory for the fuzzing inputs. How this can be used without losing the opportunity to find bugs can be found in chapter 5. Lastly, some fixed values can be written to memory. Listing 4.5 shows the input YAML configuration for Zen1.

```
input:
    # Initial inputs for the fuzzer
    initial:
        -   "bins/PRIME-X370-PRO-ASUS-3803.ROM"
    # Input bytes in-order to flash memory
    mem:
        # FET
        - addr:     0x00020000
          size:     0x40
        # Combo Dir
        - addr:     0x000c0000
          size:     0x300
        # Dir
        - addr:     0x000d1000
          size:     0x300
        # Entry header
        - addr:     0x00361400
          size:     0x100
    # Set fixed values at certain addresses
    fixed:
```

```
        # Combo Dir addr
        - addr:     0x00020014
          val:      0xff0c0000
        # Dir addr
        - addr:     0x000c0028
          val:      0xff0d1000
        # Public key addr
        - addr:     0x000c0018
          val:      0xff0d1400
        - addr:     0x000d1018
          val:      0xff0d1400
        # Entry header addr
        - addr:     0x000c0038
          val:      0xff361400
        - addr:     0x000d1028
          val:      0xff361400
```

Listing 4.5: YAML options to configure fuzzer inputs in ASPFuzz

Harness. The harness tag sets the start address of the fuzzing campaign. The fuzzer will execute the on-chip bootloader once until the start address is reached. Every consecutive test-case run of the fuzzer will start from this start address, not executing any code before that again. This is achieved by snapshotting using the snapshot implementation from section 4.3. When the emulator hits the start address the first time, a snapshot is saved and then always loaded after every test-case. Secondly, the harness tag defines every sinks. A sink is any address that marks the end of a fuzzing test-case run. If no *CrashFeedback* or *ExceptionFeedback* was hit until this point in the execution, the test-case run is finished without being considered an objective. Listing 4.6 shows the start address and the two sinks for Zen1.

```
harness:
    # parse_psp_flash() after on_chip_bl_init_SPI_maybe()
    start:          0xffff4bfc
    sinks:
        # on_chip_bl_post_status_code_maybe() in loop wfi
        - 0xffff064c
        # call_off_chip
        - 0xffff48e4
```

Listing 4.6: YAML options to configure the harness in ASPFuzz

Tunnels. The tunnels tag was built to overcome roadblocks in the on-chip bootloader. Roadblocks are comparisons with large integers like constants or checksums. The fuzzer will eventually find the correct value at random for constant integers. On the other hand, checksums are dynamically calculated and likely change for every fuzzing input. The tunnel tag provides a method to overcome both options to reach higher code coverage with fewer fuzzing inputs. Once the PC register reaches a specific address, any register can be set to an arbitrary value or another register. Listing 4.7 shows how the register R0 can be set to the value of register R3.

```
tunnels:
    cmps:
        # on_chip_bl_fletscher32() #1
        - addr:     0xffff4344
          r0:       "R3"
        # on_chip_bl_fletscher32() #2
        - addr:     0xffff4400
          r0:       "R3"
```

Listing 4.7: YAML options to configure the tunnels in ASPFuzz

Crashes. The *CrashFeedback* is one of two mechanisms to detect objectives during fuzzing (see section 4.4). Crashes are the most common objective to find bugs in software. In general, any abnormal behavior can be a bug. Therefore this tag allows defining crashes in many different ways. Hitting a breakpoint/address can be defined as a crash if a specific code should never be reached. Furthermore, write and execute permission on memory areas can be used to detect crashes. If the execution continues at an address where no code is laying, this is often caused by a buffer overflow overwriting the LR on the stack. Setting all memory to not be executable besides the on-chip bootloader code area allows the detection of these kinds of crashes. Precisely this was done for the YAML configuration file for Zen1 (see listing 4.8). If certain memory regions should not be written to, this can also be defined as a crash using the write permission on the memory area. Memory writes can be hooked using write hooks or checked whenever the CCP passthrough function is used. While hooking writes directly works for any write to the memory space, it also comes at a very high cost as every *ldr* instruction will trigger a callback. The CCP passthrough function is used to copy chunks of memory from the SMN to the SRAM. As the flash memory is mapped into the SMN space and the on-chip bootloader mainly copies chunks from the SMN, hooking only the CCP passthrough will cover most fuzzer-controlled memory writes. Therefore hooking the passthrough function

results in almost the same accuracy as hooking each write instruction but comes at an order of magnitude lower performance penalty.

```
crashes:
    # Breakpoints as crashes
    breakpoints:
        - null
    mmap:
        # Defining the none executable address space
        no_exec:
            - begin:  0x0
              end:    0xffff0000
        # Flash read function for no_write_flash_fn
        flash_read_fn:  0xffff7530
        # Only hooks the on_chip_bl_flash_read()
        # don't hook call at no_hook
        # (recommended)
        no_write_flash_fn:
            # SRAM region (BootROMServicePage) which can be used
            # as a hash for the public key
            - begin:    0x3f8a0
              end:      0x3f8c0
              no_hook:  null
            # SRAM region (BootROMServicePage) storing the public key
            - begin:    0x3f410
              end:      0x3f650
              no_hook:
                    -   0xffff4460
        # Write hooks on every ldr operation
        # (NOT recommended, very slow)
        no_write_hooks:
            - begin:  null
              end:    null
              no_ldr: null
```

Listing 4.8: YAML options to define crashes in ASPFuzz

Snapshot. The state in between each test-case has to be reset using a snapshot. While section 4.3 presented the available snapshot levels, the snapshot tag specifies which snapshot to use in which situation. Three different situations are possible. After an objective/crash, a rather comprehensive snapshot like *RustSnapshot* or *HardReset* should be used. If no objective happened, it is often enough to use a simple snapshot like *SuperLazy* or *Lazy*. Periodically it makes sense to use a snapshot similar to the crash

snapshot to avoid silent memory corruptions. The period length and all three situations can be defined in the snapshot tag, as seen in listing 4.9.

```
# Snapshotting behavior:
# - Use enum for "default", "on_crash", "periodically":
#   ["SuperLazy", "Lazy", "RustSnapshot", "HardReset"]
# - "period":
#   number of testcases before running state_rest "periodically"
snapshot:
    default:        "SuperLazy"
    on_crash:       "HardReset"
    periodically:   "RustSnapshot"
    period:         100000
```

Listing 4.9: YAML options to configure the snapshotting in ASPFuzz

4.6 Output

Each run of the ASPFuzz fuzzer produces an output directory that includes all files created during the campaign. Figure 4.2 shows this output directory structure for an example run of the fuzzer. The used YAML file is copied to the output directory. All initially created inputs from the input tag (see section 4.5) and solutions with their metadata are also stored. Some logs are kept for debugging purposes, especially the DrCov file from section 4.2.3.

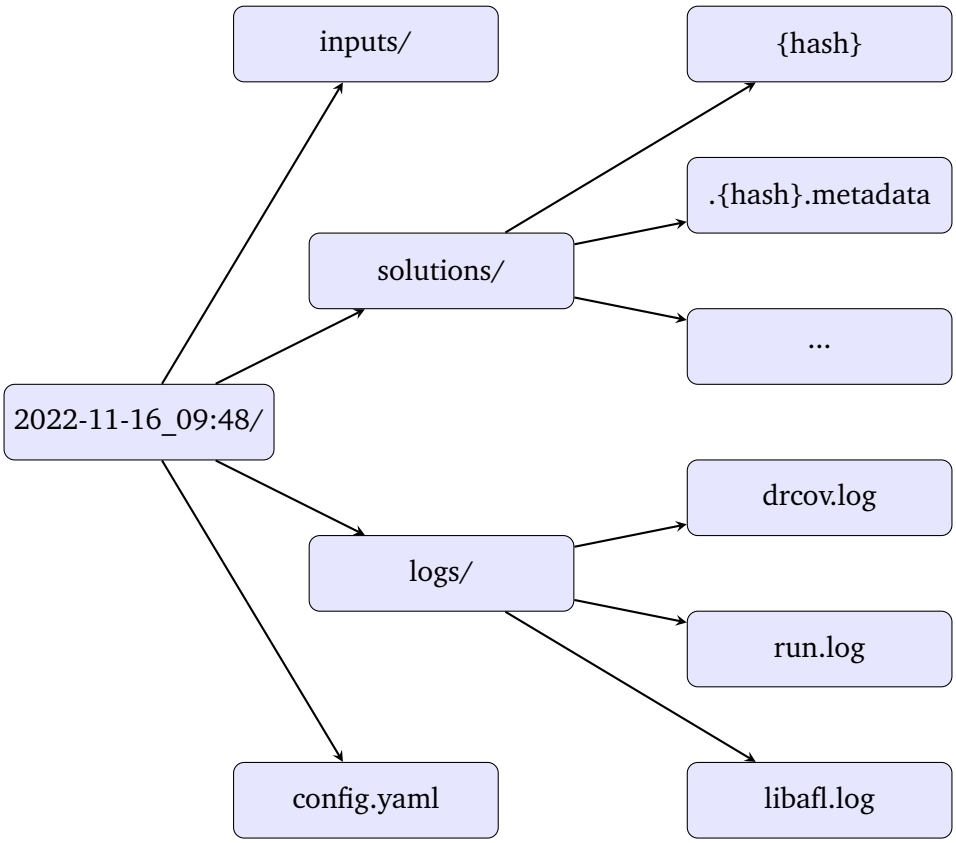

Fig. 4.2: Output structure from ASPFuzz. {hash} is the hash of the solutions input bytes

5 Harness

In chapter 4, the ASPFuzz fuzzer and the configuration options available over the YAML file were described. Even though the fuzzer can run qemu-libafl-asp in full-system QEMU mode and emulate the ASP for Ryzen Zen1, Zen+, Zen2, Zen3 and ZenTesla, it is missing the harness specification. The harness handles how the generated fuzzer inputs are inserted into the target, where to start, when to stop and how outputs from the target are evaluated. All these functionalities are already built into ASPFuzz with the YAML parser using different tags. This section discusses how the on-chip bootloader was analyzed to gather YAML configuration parameters for the different Zen generations. Once the underlying QEMU version supports the off-chip bootloader, the same process can be done to find the configuration parameters for the off-chip bootloader.

The following subsections describe the analysis knowledge that went into Ryzen Zen1. All addresses mentioned are specific to the Zen1 on-chip bootloader and might vary slightly for other generations. The other Zen generations were analyzed following the same steps. Differences between Zen1 and other Zen generations are mentioned in section 5.4. Section 5.1 shows how the start and end addresses of the on-chip bootloader were found and which operations are done by the ASP in between start and end. To limit the amount of input the fuzzer has to provide, section 5.2 describes the simplifications made to the input space. Finally, section 5.3 shows which behavior of the on-chip bootloader is defined as a crash.

5.1 Flash Parsing

Ghidra [Ghi19] was used to analyze the on-chip bootloader code. This work builds on top of prior work by other researchers [BWS19; Buh+21]. The ASP starts at `0xffff0000`. This is the exception vector base address. The first entry is the reset address (`0xffff0020`). Therefore this address can be considered the start of the *main* function. It first initializes the CPU and some MMIO devices. Eventually, it jumps to the function at `0xffff4bdc` which will be called *parse_asp_flash()* from here on. A simplified version of the code

inside the *parse_asp_flash()* function can be seen in listing 5.1. Some functions are split into two functions to increase code readability.

```
1  void parse_asp_flash()
2  {
3      // Configuring SPI MMIO device
4      configure_SPI_flash();
5      // Mapping the flash memory into SMN slots
6      map_flash_into_SMN();
7      // Finding the FET in flash memory
8      if (find_FET()) {
9          // Parsing the FFS directory structure
10         parse_asp_directory();
11         // Loading public key entry
12         load_public_key();
13         // Verifying public key using a hash
14         if (verify_public_key_hash()) {
15             // Loading the off-chip bootloader entry
16             if (load_entry(0x1)) {
17                 // Verifying the off-chip entry
18                 if (verify_entry()) {
19                     // Starting the off-chip bootloader
20                     call_bootloader();
21                 }
22             } else {
23                 // Loading the recovery bootloader entry
24                 load_entry(0x3);
25                 // Verifying the recovery bootloader entry
26                 if (verify_entry()) {
27                     // Starting the recovery bootloader
28                     call_bootloader();
29                 }
30             }
31         }
32     }
33     // Posting a status code and loop forever
34     post_status_code_and_halt();
35 }
```

Listing 5.1: Simplified codeflow of the parse_asp_flash function.

This function is ideal for fuzzing as it starts by mapping the flash memory and ends by calling the off-chip bootloader. Therefore the fuzzing harness start tag can be set right after the flash memory is mapped into the SMN. As the on-chip bootloader hangs if *post_status_code_and_halt()* is reached, this can be set as a sink (line 34). The same goes

for the *call_bootloader()* function, as any valid off-chip/recovery bootloader is also not interesting for finding bugs (line 20 & 28). We decided not to fuzz the public key. Therefore it is almost impossible for the fuzzer to find an entry that can bypass the signature verification. How to still detect verification bypasses is explained in section 5.3.

The start address and all sinks must be hooked. Furthermore, all checksum checks in the on-chip bootloader binary must be found. These can then be skipped by manipulating the registers involved in the comparison operation. The *cmp* instruction in ARM32 assembly is responsible for comparison for equality of two registers. Thus the tunnel tag can be used to set the value of one register to the value of the other before the *cmp* instruction is executed. In the case of the on-chip bootloader, there are only fletscher32 checksums for the FFS directories. Therefore the corresponding *cmp* instructions in the *parse_asp_directory()* function were hooked using the tunnnel tag (line 10). This is preferable over skipping the checksum calculation and comparison in its completeness as it allows finding buffer overflows in the checksum calculation.

5.2 Input Structure

There is no upper limit for the input size in fuzzing, but inputs greater than 2kiB should be avoided as the input space grows exponentially. The UEFI image and thus the flash memory is 16MiB large for most chips. The UEFI image for the ASUS Prime X370 Pro 3803 [ASU23b] was used as a base image for the flash and input tag. Therefore the fuzzing input space is up to 16MiB. As this is not viable for a fuzzer, this section will describe how the input space can be significantly decreased without lowering the capabilities of finding bugs. The core idea is that the FFS is a tree-like structure that is only traversed down to one leaf once during the on-chip bootloader. As a result, it is sufficient to fuzz only one child for each parent node in the tree. All the other children share the same behavior as the fuzzed child. A graphical representation of the FFS can be seen in fig. 5.1. This basic idea is further described in the following paragraphs.

The flash memory is mapped to `[0x0a000000,0x0b000000]` in the SMN memory space. The SMN controller for Zen1 statically maps this memory region to the CPUs physical memory at `[0x02000000,0x03000000]`. Throughout this section, all addresses are in UEFI image representation (`[0x0,0x01000000]`). Note that this simplification was done to avoid confusion. Therefore, depending on the context, these addresses need to be translated to the corresponding memory space for the actual implementation. To find the

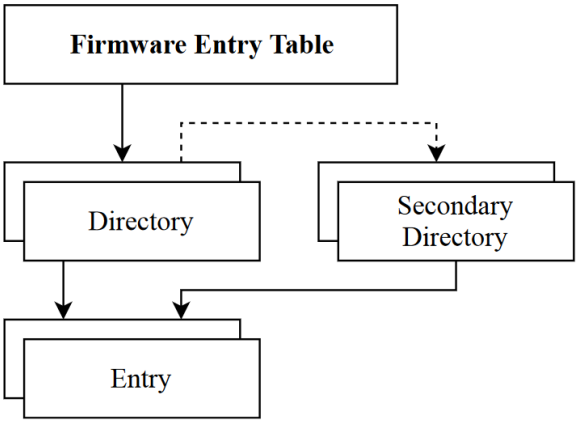

Fig. 5.1: General structure of the FFS [Wer19, p.10]

parts of the flash memory that the fuzzer has to write to the *parse_asp_flash()* code can be used (see listing 5.1). Table 5.1 shows a complete list of the flash memory regions that are used for Zen1. These can be inserted into the YAML input tag, as seen in listing 4.5. The following paragraphs show how each of these regions was found.

FFS component	Start address	End address	Size
FET	0x00020000	0x00020040	0x40
Combo directory	0x000c0000	0x000c0300	0x300
Directory	0x000d1000	0x000d1300	0x300
Entry header	0x00361400	0x00361500	0x100
Total			0x740

Tab. 5.1: Input flash memory spaces for the Zen1 fuzzer

In line 8 (listing 5.1) the on-chip bootloader tries to find the FET. Even though this could be anywhere in the flash memory, the on-chip bootloader has six hard-coded addresses where to search for it. In Zen1 these are `0x00fa0000`, `0x00f20000`, `0x00e20000`, `0x00c20000`, `0x00820000` and `0x00020000`. If any of them start with the magic bytes `0x55aa55aa` they are a valid FET and will be used for *parse_asp_directory()*. As we are in control of all these addresses and it does not matter which one is used, we only give the fuzzer input control over the latter one. The FET has seven 32-bit addresses pointing to directories after the four magic bytes. Therefore the first fuzzer input is written to

[0x00020000,0x00020040]. Though the FET has seven pointers to directories, it always uses 0x00020014 as the directory to parse. If the fuzzer can write arbitrary bytes to this address, most often, it would be an address not in the flash memory. As we want to parse a fuzzer/attacker-controlled input, we write a fixed value to it. Therefore, the directory to parse is in flash memory at 0x000c0000.

The *parse_asp_directory()* function at line 10 parses the FFS directory structure. Each directory starts with four fields of each four bytes, a magic value denoting the type of directory, a checksum, the number of entries and an unknown value [Wer19]. The on-chip bootloader differentiates between two types. In the first case, it is not a combo directory. 16 bytes will be loaded for each entry and the directory checksum will be checked. The second case, where it is a combo directory, also loads 16 bytes for each entry and checks its checksum. After that, it iterates over each entry and picks one based on four of the 16 bytes by comparing it to a value stored in a fuse in the MMIO address space. This entry is then used as a primary directory and parsed again. First, loading 16 bytes per entry and then doing the checksum check, exactly as in the first case. Therefore two new flash memory regions have to be added to the fuzzer. The first one starts at 0x000c0000 and can be any length. To decrease the input space and because the directory parser checks if the number of entries is greater than 64, there is little value in adding more than 65 entries. As a result, [0x000c0000,0x000c0300] is sufficient for fuzzing. Again if this is a combo directory, it will search for the primary directory. Therefore, we have to fix the address where the parser looks for the primary directory. Fixing this address is important because we want the primary directory to lay in the fuzzer-controlled flash memory space and not at a random address. The address was chosen to be 0x000d1000 and thus, the fuzzer can also write inputs to [0x000d1000,0x000d1300].

The 0x240 byte public key entry is loaded in line 12. As the on-chip bootloader calculates and compares a 32-byte SHA hash over the whole public key entry in line 14, the public key cannot be altered by the fuzzer. The chance of finding a hash collision is almost zero.

Lastly, the on-chip bootloader loads the off-chip bootloader entry, verifies it using the public key and then starts the off-chip bootloader. If that fails, it will attempt the same with the recovery bootloader, which is simply a different entry in the directory. The *load_bootloader(int type)* searches for an entry with a specific type value in the directory. For the off-chip bootloader, it is 0x1 and for the recovery bootloader it is 0x3. If a matching entry was found, the directory also includes a pointer address to the entry. Therefore this address for the off-chip bootloader was fixed to 0x00361400 to get an

attacker-controlled entry. The entry header is always 256 bytes, so the fuzzer can control the header at `[0x00361400,0x00361500]`. The entry body is verified using the public key. Therefore the body can only be manipulated if the public key can be changed beforehand. This is why the fuzzer cannot overwrite the entry body, because it would need to guess the signature correctly, which is almost impossible. In theory, the recovery bootloader could also be fuzzed, but it would not help to find new bugs as it uses the same functions as the off-chip. The only difference would be the memory location, which does not matter as the fuzzer can already be chosen arbitrarily in the flash memory region.

5.3 Crash Definition

In general, fuzzing aims to find bugs to mitigate or exploit them. In user-land applications, bugs are detected mainly by running the application with a fuzzer input and checking if it crashes. A crash happens because the underlying OS detects behavior that is not permitted by the OS. This is mainly triggered by the MMU as each application only gets read/write/execute permission on some memory pages. If the application accesses a memory address on a page for which it does not have access rights, the OS stops it and throws an error. Other crashes can also come from improper use of syscalls, memory allocators and more. Less flashy bugs can be found by triggering crashes with some form of sanitization. These methods rely on the OS to detect the crashes and find bugs.

There is no underlying OS for the ASP's on-chip bootloader as it is bare-metal firmware running in machine mode. The on-chip bootloader code can access any memory address and controls the exception vector. Any triggered exception jumps to the corresponding entry in the on-chip bootloader-controlled exception vector. Even if the on-chip bootloader accesses an unmapped memory region like `[0x00040000,0x01000000]`, it will only result in a restart of the on-chip bootloader. Whenever memory corruptions occur, they often are silent memory corrupts as they do not lead to an observable difference in the outcome. This is a known issue for bare-metal firmware fuzzing. Muench et al. [Son+19] presents this problem and analyzes why it is much harder to find bugs in embedded systems. This section focuses on which bugs can still be detected in the on-chip bootloader.

Spatial memory safety error These are any accesses to memory outside the bounds of its intended range. Buffer under-/overflows, heap under-/overflows and global under-/overflows are the most common ones. They can be caught by the OS if they exceed

a page or even by a single byte with ASan for instrumented binaries. In the case of the on-chip bootloader, no heap exists and the binary is not instrumented. Therefore under-/overflows for buffers and globals can only be caught by their effects. If the under-/overflow results in access to an unmapped memory area, this is detected by the *ExceptionFeedback* as a reset or data abort happens. An example would be an overflow of the global BootROMServicePage struct, which lies at the end of the SRAM memory region. Using the crash tag to mark memory regions as none-executable also allows finding under-/overflows resulting in a corruption of the stack. Bugs causing this issue would be any underflow of the global BootROMServicePage struct or the stack itself. The stack stores the saved LR to build the call stack. If any of them is overwritten, the execution will eventually continue at a random memory location. Therefore any memory area that is not the on-chip bootloader ROM region is set as none-executable (`[0x0,0xffff0000]` see listing 4.8). Jumps to data within the on-chip bootloader can also be detected as it is often not a valid ARM assembly instruction. The *ExceptionFeedback* catches these by triggering undefined instruction exceptions. Lastly, overflow within the global BootROMServicePage struct can result in corruption with unknown side effects. Two of them were analyzed to be particularly interesting. The first one is any writes to the ARK stored in the BootROMServicePage (`[0x3f410,0x3f650]`) after the hash was verified. Secondly, any writes to `[0x3f8a0,0x3fc0]`. The on-chip bootloader can use this memory area to compare the hash for verifying the ARK instead of the hashes stored in ROM. Any ARK could be verified if an under-/overflow bug allows overwriting this memory region. Both of them would lead to an authentication bypass. Under-/overflows by only a few bytes cannot be detected with this method, as an OS can only detect them if they exceed a page. Only ASan can find under-/overflows by single bytes.

Temporal memory safety error The on-chip bootloader has no heap thus, no use-after-free bugs can be present. Besides, that use-after-scope and use-after-return are other temporal memory safety errors. Unfortunately, ASPFuzz is unable to find them, as it would require building a call stack and checking for each stack access if the access is within the scope of the current function in the call stack. It would require similar features as QASan just for full-system QEMU, which is still an unsolved issue (see section 2.4.4).

Uninitialized variables error Using uninitialized variables can have various side effects but primarily results in information leakage. ASPFuzz has no capabilities to detect them besides if they change the execution flow to memory not within the on-chip bootloader

ROM. Information leakage would be an issue, but as there are already existing attacks on all Zen generations to gain arbitrary code execution, no new knowledge could be gained. Implementing an uninitialized variable detection is possible using read hooks. Every memory read operation would need to be checked, resulting in a not neglectable overhead in the execution.

Pointer type error Pointer type errors can occur when casting pointer from one data type to another. As no source code is available for the on-chip bootloader, these kinds of bugs cannot be detected.

Variadic function error Variadic functions are functions with a variable amount of input parameters. The most prominent bug of this type is format string errors. The on-chip bootloader has no string output. Therefore, it is not unexpected that no string operations like *printf* were found in the on-chip bootloader binary. In general, no variadic function was found in the on-chip bootloader. Because of that, no checks for argument length mismatch were implemented in ASPFuzz.

Integer errors Integer errors can either be integer overflows or arithmetic errors like divide-by-zero. The effect of such errors can reach from undefined behavior to out-of-bounds array access. While the latter can be detected the same as any other overflow, its effect cannot recognize undefined behavior. To find bugs associated with integer arithmetic, every arithmetic instruction would need to be hooked and the result needs to be checked for validity. ASPFuzz does not provide these capabilities.

Functional errors To check for functional bugs, it would require an oracle that can be used to compare the behavior of the on-chip bootloader to the ground truth. As no oracle exists, the only functional verification the fuzzer can do is that no verification bypasses exist because the on-chip bootloader is the RoT for the entire system. Therefore, it is assumed that a functional on-chip bootloader should not include any authentication bypass. The necessary checks were already explained in the spatial memory safety paragraph.

5.4 Differences in Zen Generations

Zen1 and Zen+ have the same memory layout. Therefore, most addresses stay the same. The input spaces for Zen1 can also be used for Zen+ (see table 5.1). Also, the functionalities of the on-chip bootloaders are seemingly identical. Zen+ has slightly more code than Zen1. The main difference is that some functions have different memory addresses. This is most likely caused by some minor changes in the code, resulting in a different placement by the compiler.

Zen2 and Zen3 have different fuse values and MMIO memory mappings. Especially the flash memory is mapped to `0x44000000` instead of `0x0a000000` in the SMN internal memory layout. Also, the on-chip bootloader does not statically map the whole flash memory to the SMN slots but only maps the area of the flash memory needed for a particular operation. Furthermore, the ASUS Prime B450M-A 1201 [ASU23a] UEFI image was used as a base image for fuzzing. The input space was adapted according to the new base image, as seen in table 5.2.

FFS component	Start address	End address	Size
FET	0x00020000	0x00020040	0x40
Combo directory	0x000c0000	0x000c0300	0x300
Directory	0x00299000	0x00299300	0x300
Entry header	0x006a8400	0x006a8500	0x100
Total			0x740

Tab. 5.2: Input flash memory spaces for the Zen2 and Zen3 fuzzer

ZenTesla is mostly the same as Zen1 and Zen+. Only a couple MMIO addresses slightly vary. As the emulator for ZenTesla does not provide the correct fuse values to successfully emulate the ARK verification, the fuzzer skips the comparison instructions for the verification. The corresponding input space can be seen in table 5.3.

FFS component	Start address	End address	Size
FET	0x00020000	0x00020040	0x40
Combo directory	0x00030000	0x00030300	0x300
Directory	0x000d1000	0x000d1300	0x300
Entry header	0x00030600	0x00030700	0x100
Total			0x740

Tab. 5.3: Input flash memory spaces for the ZenTesla fuzzer

6 Result

This section presents the results of ASPFuzz. All experiments were conducted on a server with four Intel Xeon E7-4870 and 512GiB RAM. The server has 40 cores (80 threads) at 2.40 GHz. The data used for the results and all the plots are publicly available at Zenodo[1] for openness and reproducibility. In section 6.1 the performance of the basic fuzzer and different features are presented. The scaling of ASPFuzz to more than one core is shown in section 6.2. Finally, section 6.3 analyzes the objectives found during the fuzzing campaign and associates them with bugs in the on-chip bootloader.

6.1 Performance

First, the ASPFuzz fuzzer was compared for Zen1, Zen+, Zen2, Zen3 and ZenTesla. Figure 6.1 shows the executions per seconds while fig. 6.2 the edge coverage and the number of objectives. The fuzzing campaigns for the Zen generations were executed on a single core for 24 hours each. As the inputs are randomly mutated, the fuzzer is non-deterministic. Therefore, ten trials for each generation were run and the average, minimum and maximum were plotted.

The executions per second decrease over the runtime for all Zen generations. This is due to the increasing number of edges, resulting in a longer emulation time. Furthermore, for Zen1 and Zen+ the number of solutions increases over the fuzzing campaigns. After an objective/crash, the *HardReset* snapshot is used to guarantee that the emulator's state is reset correctly before the next test-case. As the *HardReset* takes significantly longer to load (see section 4.3), the runtime will decrease the more often an objective is hit. The edge coverage of Zen+ is a little bit higher than Zen1, which is expected as the on-chip bootloader for Zen+ has slightly more code. Zen2, Zen3 and ZenTesla have significantly less edges, not because the code size for parsing the flash content is smaller. Instead, the crashes in the on-chip bootloader for Zen1 and Zen+ result in additional edges because extra basic blocks are executed as a result of the crashes (more details

[1] ASPFuzz data on Zenodo

on the crash behaviour in section 6.3). After eight hours of the fuzzing campaign, the edge coverage does not increase any further. Using the DrCov file and the lightkeeper plugin, the edge coverage was visualized in Ghidra. A manual analysis of the visual code coverage did not find any basic blocks with branches depending on flash memory that was not found by the fuzzer. Code that was not covered mostly depends on fuse values or other MMIO device values that are not controlled by the fuzzer.

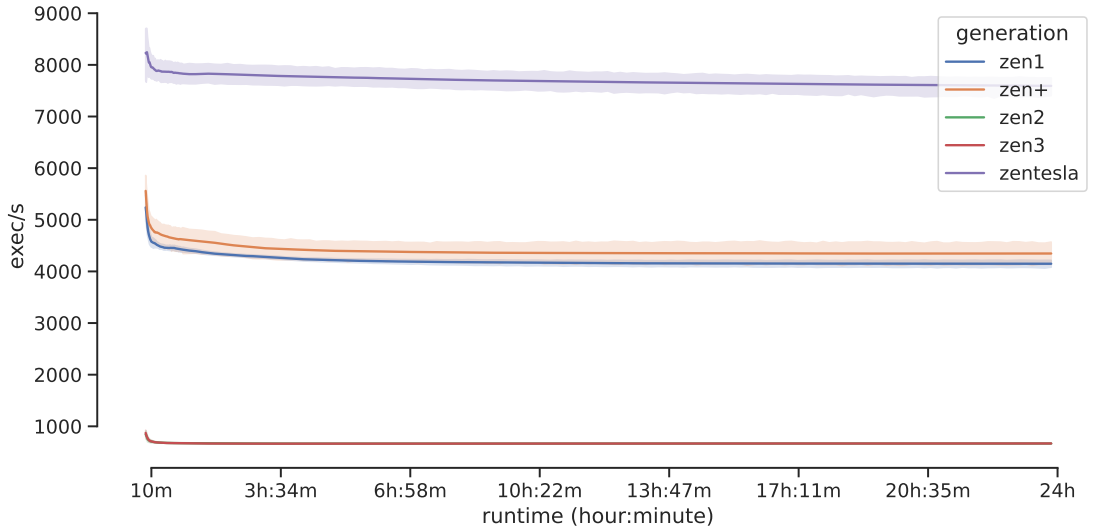

Fig. 6.1: Execution per second for each Zen generation over ten trials for 24 hours each

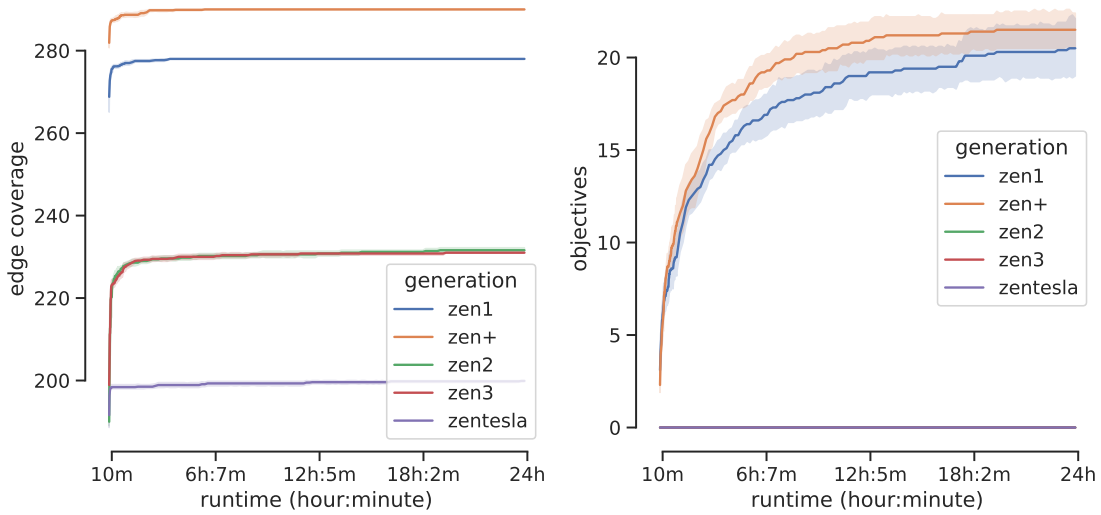

Fig. 6.2: Edge coverage and objectives for each Zen generation over ten trials for 24 hours each

The four custom snapshotting implementations were already presented in section 4.3 and their average load times. The fuzzer can be configured to use a different snapshotting level for three different situations, the default one after any test-case, the one after an objective/crash happened and a periodic one. This means 64 different combinations of snapshotting can be configured over the YAML file. Figure 6.3 highlights the executions per second for 5 chosen snapshot configurations. Each snapshotting configuration is given by a tuple of the form (default, crash, periodical). The campaigns were run ten times for each configuration and data was collected for one hour. Using the *Superlazy* snapshot for all three situations reaches the highest throughput with $\approx 14k$ exec/s. It comes with the penalty that memory corruptions from previous test-cases affect the following test-cases. On the other hand, using only *HardReset* snapshots has the worst performance with ≈ 100 exec/s. As the CPU is restarted after each test-case, previous test-cases do not affect other test-cases. For reference, the snapshotting built into vanilla QEMU would reach under 50 exec/s. As a trade-off between throughput and stability, all other ASPFuzz campaigns in the result section use (*SuperLazy*, *HardReset*, *RustSnapshot*).

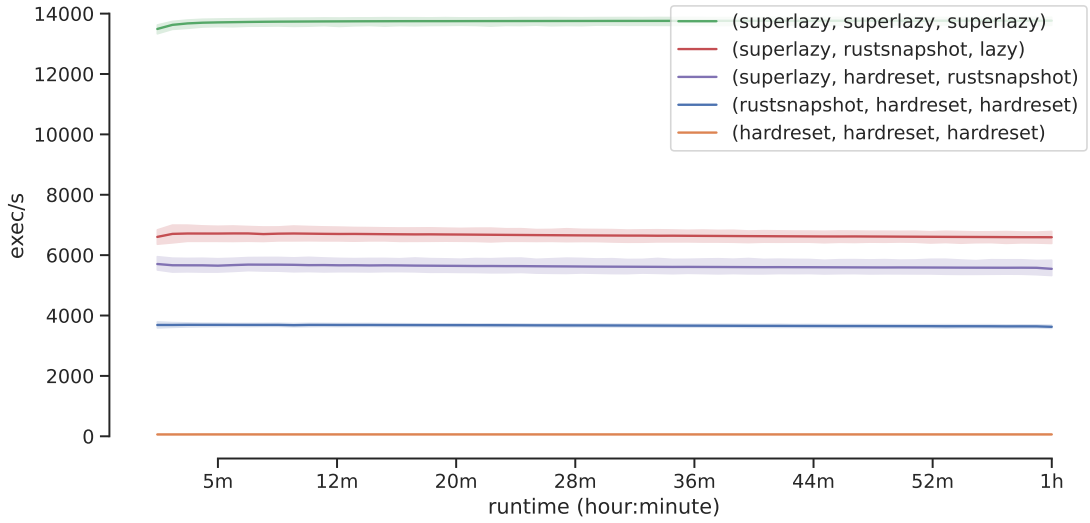

Fig. 6.3: Execution per second for Zen1 with different snapshotting strategies over ten trials for one hour each. The tuple represents the three different snapshotting situations from section 4.3 as (default, crash, periodically).

Lastly, an empty initial corpus was compared to a single valid UEFI image in the corpus. The average over ten trials for 24 hours shows that providing a valid flash image to the fuzzer significantly increases the performance. Not only the edge coverage is much higher, but also more objectives are found quicker. Only the performance of the fuzzing campaign with an empty corpus is better. This is because the edge coverage and, therefore the

emulation speed of each test-case is much lower and fewer objectives also lead to fewer *HardReset* snapshots.

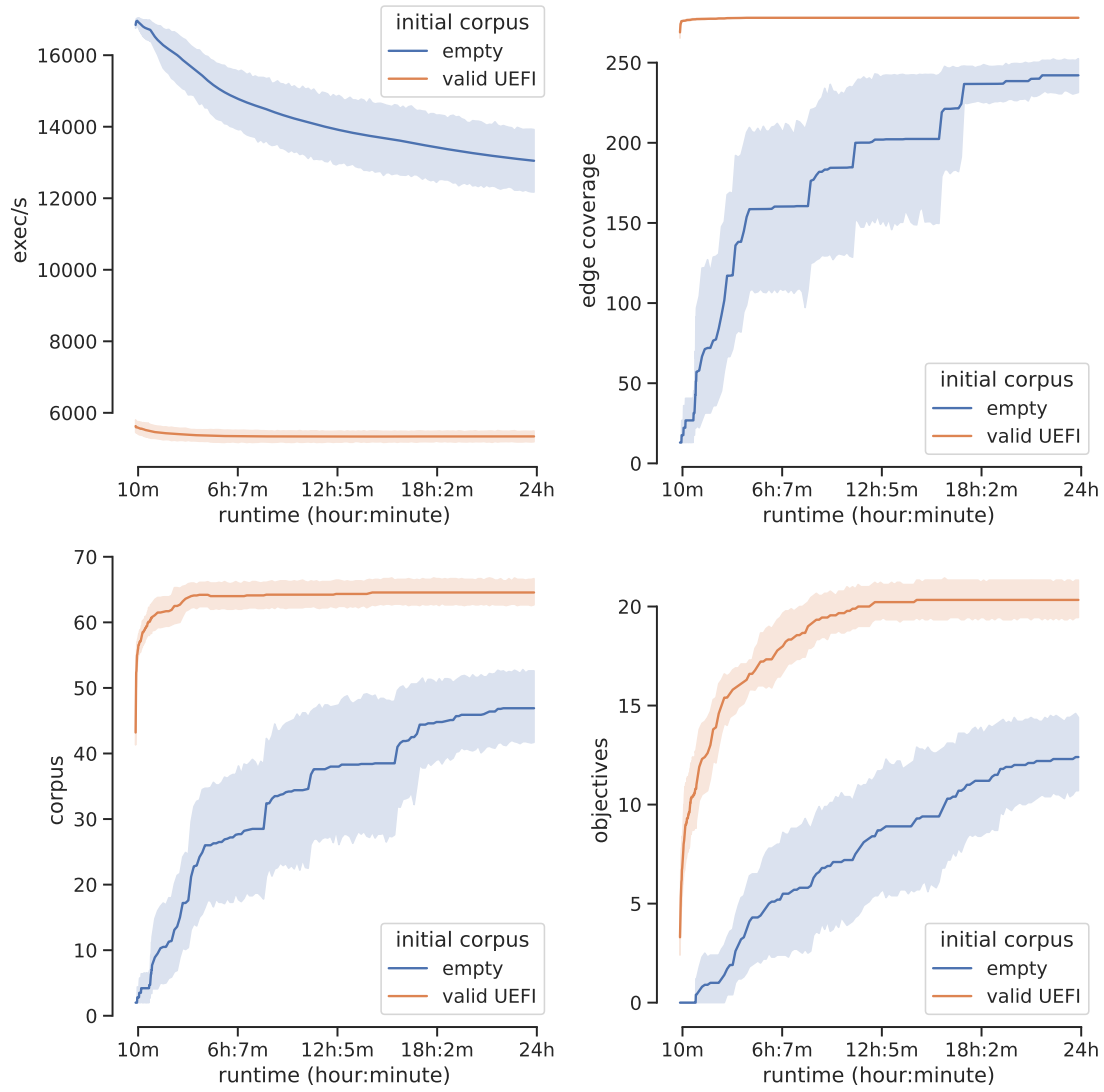

Fig. 6.4: Execution per second for Zen1 with a valid UEFI image in the initial corpus or no image on one core over ten trials for 24 hours each

6.2 Scalability

LibAFL also can run the fuzzer on multiple cores/instances (see section 4.2.2). This capability was fully utilized to drastically increase the throughput of the fuzzer on the 40 cores (80 threads) machine. Figure 6.5 shows executions per second for different amounts of fuzzing cores over one hour. Compared to the other experiments in the result section, this one was only conducted once instead of 10 times. The reason is that it would take days to run the experiment multiple times. Also, the minimum and maximum executions per second would not deviate that much as it is a similar setup to fig. 6.3.

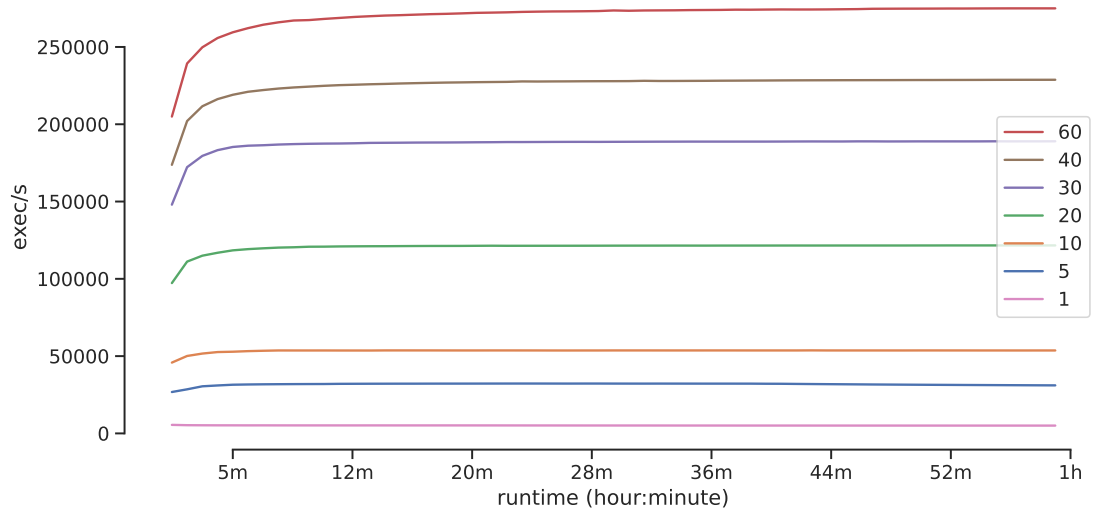

Fig. 6.5: Executions per second for Zen1 with different amounts of fuzzing cores over one trial for one hour each

From fig. 6.5, it is safe to say that the performance of the fuzzer increases with the number of cores. Table 6.1 shows the same data but takes the one core performance as a baseline. Every number of cores is compared to the single-core performance and the following formula calculates the speed-up.

$$\text{speed-up} = \frac{\#\text{Cores } exec/s}{\text{1-Core } exec/s}$$

The result shows that the scaling of the fuzzer is almost linear, with the number of cores up to 40 cores. After 40 cores, the speed-up decreases. The machine running the fuzzer has 40 physical cores, which explains this barrier. Also, full-system QEMU is multithreaded. Therefore every new fuzzing core comes with multiple threads. Even though the 80

threads of the machine allow more than 40 fuzzing cores to run simultaneously while increasing the throughput, it does not scale linearly anymore. That said, running the fuzzer with nearly as many threads as the machine allows, is still the best choice to reach the best performance regarding executions per second.

#Cores	Max. exec/s	Per core	Speed-up
1	5520	5520	1.00x
5	32257	6451	5.84x
10	53716	5371	9.73x
20	121614	6080	22.03x
30	188985	6299	34.24x
40	228773	5719	41.44x
60	274974	4582	49.81x

Tab. 6.1: Multi-core performance comparison for ASPFuzz. The speed-up is computed based on the single-core performance.

6.3 Findings

The fuzzer ran each Zen generation on 60 cores for four days. For Zen1 and Zen+, the fuzzer found over 30 unique objectives. Objectives similar for each fuzzing instance were hereby removed. For Zen2, Zen3 and ZenTesla no objectives were found. Each objective was analyzed using the emulator and static analysis of the resulting UEFI image.

Some objectives are caused by a write operation to the memory space in the range `[0x7fffff00,0x80037f00]`. All of these write operations happen during the *load_entry* function (listing 5.1 line 16 & 24) using the CCP passthrough (`0xffff47a4` for Zen1 and `0xffff4750` for Zen+) or a C implementation of *memcpy* (`0xffff47f4` for Zen1 and `0xffff47a0` for Zen+). Both copy operations write data from the flash memory to the x86 slot memory region as long as the start address of the copy operation is in the range `[0x7fffff00,0x80037f00]`. After reversing the corresponding section of the binary, it was found that this source address comes from the Header Entries header in the flash memory and can therefore be chosen by the fuzzer. The on-chip bootloader does two checks on the source address. Listing 6.1 shows how a decompiled version of both checks

could look like, where the source address is denoted as *src_addr*. The first check most likely tries to ensure that the source address lies within the SRAM memory. Instead of checking the whole source address, the most significant bit is removed before the check. For the copy operations, it is not removed. Therefore allowing source address not only in the range `[0x100,0x38000]`, but also `[0x80000000,0x80038000]`. This leads to the source addresses in the x86 slot memory mentioned before. The slight difference in the address range comes from a subtraction by `0x100` before the copy operation. As this bug is triggered often during every fuzzing campaign, the corresponding check was removed from the objective in the more recent version of ASPFuzz to avoid the slowdown which comes with it. As the x86 slots are unused during the on-chip bootloader, no exploit could be found for this bug.

```
1 if((src_addr & 0x7fffffff) < 0x38001) {
2     if (src_addr < 0x100) {
3         // return with error
4     }
5     // continue load_entry function
6 }
7 // return with error
```

Listing 6.1: Source address check during the *load_entry* function

Most objectives for both generations were triggered because they executed memory outside the on-chip bootloader memory regions (`[0x0,0xffff0000]`). As the code flow should never reach these addresses, the fuzzer must have overwritten the PC register. The PC can either be directly overwritten by a data-dependent jump instruction or if the LR saved on the stack was overwritten. Using the metadata, the last valid PC value was found to be `0xffff715c` for Zen1 (`0xffff7108` for Zen+). The ARM assembly instructions at both addresses load the LR saved on the stack to the PC. Therefore the fuzzer must have overwritten the SRAM region containing the stack, which then changed the code flow. If this address loaded to the PC can be arbitrarily chosen, a malicious UEFI image would be able to gain arbitrary machine mode code execution on the ASP. Using the emulator, the LR on the stack was monitored and the moment of corruption was found. It was overwritten by the CCP when copying the Header Entry body from the flash memory to SRAM. The size of the copy operation comes from a 4-byte value in the Header Entries header. As the Header Entry header is part of the UEFI image in flash memory and there is no signature check on it, the fuzzer was able to manipulate the size field and thus overwrite the complete SRAM memory, including the stack. Even though the on-chip bootloader checks the size field, due to a mistake in the conversion between

unsigned and signed values, any size larger than `0x80000000` bypasses this check. The same buffer overflow in the copy operation of the CCP was already found by Buhren et al. [BWS19] and used to acquire the on-chip bootloader from ROM for Zen1 and Zen+ (see section 2.2.3). Finding this bug takes the fuzzer only a couple of seconds, as seen in fig. 6.2.

7 Conclusion

This thesis presented ASPFuzz, the first fuzzer for the ASP in the Ryzen Zen1, Zen+, Zen2, Zen3 and ZenTesla architecture. ASPFuzz uses the uprising LibAFL fuzzing framework together with QEMU to fuzz AMD's on-chip bootloader. The ASP machine in QEMU was extended to emulate the on-chip bootloader for Zen2, Zen3 and ZenTesla and also added extra fuzzing support. LibAFL was enhanced for ARM32 full-system emulations.

ASPFuzz is configurable using YAML files. This includes the definition of objectives, memory regions the input is written to and custom snapshotting mechanisms. The YAML file configuration makes the fuzzer highly reusable and limits the need to change any code. Running a fuzzing campaign with a new configuration or target can be as simple as changing a single file. This might include running Ryzen Threadripper or Epyc ASPs, more recent Ryzen Zen generations or even fuzzing the off-chip bootloader.

Full-system emulation fuzzers are known to be slow as they need to emulate the MMU. Due to the custom snapshot implementations and the lightweight hooks, the throughput of the ASPFuzz is as high as some application fuzzers. At the same time, it also scales with the computational power of the fuzzing campaign's machine. Therefore the speed of the fuzzer can be increased by simply acquiring more servers.

For the on-chip bootloader, the fuzzer can manipulate the content of the flash memory, just like an attacker with hardware access or other possibilities to overwrite the UEFI image. This lies well in AMD's threat model, as the ASP is the system's RoT. As a result, the fuzzer found the known buffer overflow in Zen1 and Zen+ within seconds. Running the fuzzer for Zen1 and Zen+ for multiple days found only one new bug. However, this bug seems to be unexploitable. As the on-chip bootloader code is minimal and the fuzzer reached a high code coverage, no further critical bugs likely exist. No objectives were found by the fuzzer for Zen2, Zen3 and ZenTesla. Because the fuzzer can find the known bug, but no further critical bugs, the proprietary on-chip bootloader by AMD for the new Zen generations seems secure from a software perspective.

In conclusion, the initial goal of building a fuzzer for the ASP's on-chip bootloader was achieved while even providing a configurable, open-source fuzzer for the whole ASP.

8 Future Work

ASPFuzz can still be improved in many different ways. Some ideas are presented below.

ASP applications The fuzzing can be extended to the off-chip bootloader and any later functionalities of the ASP. This includes functionalities the ASP provides to the x86 cores after boot-up. Again the machines for the corresponding Zen generations in QEMU need to be improved to emulate everything that should be fuzzed. Afterward, new YAML files can be written to fuzz these functionalities.

More targets. Newer Ryzen Zen generations or Ryzen Threadripper and Epyc chips could be added to ASPFuzz. Therefore QEMU needs to be extended to support these machines and then YAML files can be created to fuzz the on-chip bootloader from these generations.

Generation-based fuzzing. Even though ASPFuzz can already configure the memory regions the inputs are written to and also set fixed values, it might be better for the performance to build a generation-based fuzzer. The grammar for the generation-based fuzzer is given by the FFS structure which was already partially reversed by other researchers [Wer19]. LibAFL comes with Gramatron, which can be used to perform this grammar-aware fuzzing.

Full-system emulation sanitizer. To avoid silent memory corruptions (see section 2.1), it would be highly beneficial to build QASan also for full-system emulation. Unfortunately, no implementation is publically available at the time of writing this thesis. As soon as it exists, adding this sanitizer would be greatly beneficial as any buffer/global under-/overflow by any number of bytes could be caught. This would improve the spatial/temporal memory safety error detection and could replace all existing methods of ASPFuzz.

Binary rewriting. While no implementation of QASan for full-system QEMU exists, it might be possible to use tools like μSBS to insert sanitizer through binary rewriting (see section 2.4.4). Only a few binary rewriting tools are capable of rewriting ARM32 assembly. Furthermore, bare-metal firmware, like the on-chip bootloader, is even more cumbersome to rewrite. Some of the on-chip bootloader code might be timing- or ordering-critical and the binary rewriter is also unaware of the physical memory layout. For these reasons, this approach would likely involve multiple patches to the existing binary rewriting tools.

More crash definitions. Currently, all cores of ASPFuzz run the same YAML file with the same hooks and objectives. To improve bug detection, some cores could run additional checks with a higher performance penalty. This is a widespread technique in fuzzing, where every core only uses one sanitizer instead of all viable sanitizers. Thus each fuzzing core gets a specific bug type to search for. An example for ASPFuzz would be to hook all read operations to check for uninitialized variables. Another one would be to hook all integer operations to check for integer overflows or undefined behavior.

Bibliography

[AFL22a] AFLplusplus. *Message Passing - The LibAFL Fuzzing Library*. Dec. 2022. URL: https://aflplus.plus/libafl-book/message_passing/message_passing.html (visited on Dec. 15, 2022) (cit. on p. 30).

[AFL22b] AFLplusplus. *QEMU LibAFL Bridge*. Dec. 2022. URL: https://github.com/AFLplusplus/qemu-libafl-bridge (visited on Dec. 13, 2022) (cit. on p. 23).

[ARM22] ARM. *CoreSight Architecture*. 2022. URL: https://developer.arm.com/Architectures/CoreSight%20Architecture (visited on Dec. 12, 2022) (cit. on p. 20).

[ARM16] ARM. *Cortex-A5 Technical Reference Manual*. en. 2016. URL: https://developer.arm.com/documentation/ddi0433/a (visited on Dec. 5, 2022) (cit. on pp. 7, 22).

[ASU23a] ASUS. *PRIME B450M-A Mainboard*. de-de. 2023. URL: https://www.asus.com/de/motherboards-components/motherboards/prime/prime-b450m-a/ (visited on Jan. 11, 2023) (cit. on p. 51).

[ASU23b] ASUS. *PRIME X370-PRO Mainboard*. de-de. 2023. URL: https://www.asus.com/de/motherboards-components/motherboards/prime/prime-x370-pro/ (visited on Jan. 11, 2023) (cit. on p. 45).

[Ayr18] Ayrx. *DrCov File Format*. en. Oct. 2018. URL: https://www.ayrx.me (visited on Dec. 16, 2022) (cit. on p. 31).

[Bar21] Luca Di Bartolomeo. “ArmWrestling: efficient binary rewriting for ARM”. In: 2021 (cit. on p. 17).

[Bog19] Martijn Bogaard. *Fuzzing OP-TEE with AFL*. 2019. URL: https://static.linaro.org/connect/san19/presentations/san19-225.pdf (visited on Dec. 2, 2022) (cit. on p. 5).

[Buh22] Robert Buhren. “Resource control attacks against encrypted virtual machines”. en. In: (2022). URL: https://depositonce.tu-berlin.de/handle/11303/16488 (visited on Apr. 1, 2022) (cit. on p. 5).

[Buh+17] Robert Buhren, Shay Gueron, Jan Nordholz, Jean-Pierre Seifert, and Julian Vetter. “Fault Attacks on Encrypted General Purpose Compute Platforms”. In: *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*. CODASPY ’17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 197–204. URL: https://doi.org/10.1145/3029806.3029836 (visited on Apr. 1, 2022) (cit. on pp. 2, 5).

[BJE22] Robert Buhren, Hans Niklas Jacob, and Alexander Eichner. *psp-docs*. 2022. URL: https://github.com/PSPReverse/psp-docs (visited on Dec. 6, 2022) (cit. on p. 6).

[Buh+21] Robert Buhren, Hans-Niklas Jacob, Thilo Krachenfels, and Jean-Pierre Seifert. "One Glitch to Rule Them All: Fault Injection Attacks Against AMD's Secure Encrypted Virtualization". In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. CCS '21. New York, NY, USA: Association for Computing Machinery, Nov. 2021, pp. 2875–2889. URL: https://doi.org/10.1145/3460120.3484779 (visited on Apr. 1, 2022) (cit. on pp. 2, 5, 8, 9, 43).

[BWS19] Robert Buhren, Christian Werling, and Jean-Pierre Seifert. "Insecure Until Proven Updated: Analyzing AMD SEV's Remote Attestation". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. CCS '19. New York, NY, USA: Association for Computing Machinery, Nov. 2019, pp. 1087–1099. URL: https://doi.org/10.1145/3319535.3354216 (visited on Apr. 1, 2022) (cit. on pp. 5, 7, 8, 43, 60).

[Cap22] Capstone. *The Ultimate Disassembly Framework*. 2022. URL: https://www.capstone-engine.org/ (visited on Dec. 16, 2022) (cit. on p. 31).

[Che+16] Daming D. Chen, Manuel Egele, Maverick Woo, and David Brumley. "Towards Automated Dynamic Analysis for Linux-based Embedded Firmware". en. In: *Proceedings 2016 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2016. URL: https://www.ndss-symposium.org/wp-content/uploads/2017/09/towards-automated-dynamic-analysis-linux-based-embedded-firmware.pdf (visited on Dec. 2, 2022) (cit. on p. 4).

[Che+18] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, et al. "IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing". en. In: *Proceedings 2018 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2018. URL: https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_01A-1_Chen_paper.pdf (visited on Dec. 2, 2022) (cit. on pp. 4, 15).

[Din+20] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. "RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization". In: *2020 IEEE Symposium on Security and Privacy (SP)*. ISSN: 2375-1207. May 2020, pp. 1497–1511 (cit. on p. 17).

[Eic20] Alexander Eichner. "Implementing an emulator for AMD's Platform Security Processor". en. In: (2020), p. 82 (cit. on pp. 2, 5–7, 11).

[Eic21] Alexander Eichner. *PSPEmu - Emulator for AMDs (Platform) Secure Processor*. 2021. URL: https://github.com/PSPReverse/PSPEmu (visited on Dec. 6, 2022) (cit. on p. 11).

[FDQ20] Andrea Fioraldi, Daniele Cono D'Elia, and Leonardo Querzoni. "Fuzzing Binaries for Memory Safety Errors with QASan". In: *2020 IEEE Secure Development (SecDev)*. Sept. 2020, pp. 23–30 (cit. on p. 17).

[Fio+20] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. “AFL++: Combining Incremental Steps of Fuzzing Research”. en. In: (2020), p. 12 (cit. on pp. 15–17).

[Fio+22] Andrea Fioraldi, Dominik Maier, Dongjia Zhang, and Davide Balzarotti. “LibAFL: A Framework to Build Modular and Reusable Fuzzers”. In: *Proceedings of the 29th ACM conference on Computer and communications security (CCS)*. CCS '22. event-place: Los Angeles, U.S.A. ACM, Nov. 2022 (cit. on pp. 16, 18, 27, 29).

[Fri22] Frida. *Frida*. en-US. 2022. URL: https://frida.re/ (visited on Dec. 6, 2022) (cit. on p. 12).

[Gaa22] Markus Gaasedelen. *Lighthouse - A Coverage Explorer for Reverse Engineers*. Dec. 2022. URL: https://github.com/gaasedelen/lighthouse (visited on Dec. 16, 2022) (cit. on p. 31).

[Ghi19] Ghidra. *Ghidra SRE*. 2019. URL: https://ghidra-sre.org/ (visited on Dec. 6, 2022) (cit. on pp. 9, 11, 43).

[GLM08] Patrice Godefroid, Michael Levin, and David Molnar. “Automated Whitebox Fuzz Testing”. In: Jan. 2008 (cit. on p. 16).

[Goo19] Google. *syzkaller - kernel fuzzer*. 2019. URL: https://github.com/google/syzkaller (visited on Dec. 2, 2022) (cit. on p. 4).

[Har22] Pascal Harprecht. *QEMU ASP*. Apr. 2022. URL: https://github.com/pascalharp/qemu (visited on Dec. 6, 2022) (cit. on pp. 12, 21).

[Har+22] Lee Harrison, Hayawardh Vijayakumar, Rohan Padhye, Koushik Sen, and Michael Grace. “PARTEMU: Enabling Dynamic Analysis of Real-World TrustZone Software Using Emulation”. en. In: (2022), p. 19 (cit. on p. 5).

[Her+22] Grant Hernandez, Marius Muench, Dominik Maier, et al. “FirmWire: Transparent Dynamic Analysis for Cellular Baseband Firmware”. In: Jan. 2022 (cit. on p. 4).

[Het+21] Felicitas Hetzelt, Martin Radev, Robert Buhren, Mathias Morbitzer, and Jean-Pierre Seifert. “VIA: Analyzing Device Interfaces of Protected Virtual Machines”. In: *Annual Computer Security Applications Conference*. ACSAC. New York, NY, USA: Association for Computing Machinery, 2021, pp. 273–284. URL: https://doi.org/10.1145/3485832.3488011 (visited on Apr. 1, 2022) (cit. on p. 5).

[Int20] Intel. *Intel® Trust Domain Extensions*. en. 2020. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html (visited on Dec. 2, 2022) (cit. on p. 5).

[Int05] Intel. *Pin - A Dynamic Binary Instrumentation Tool*. en. 2005. URL: https://www.intel.com/content/www/us/en/developer/articles/tool/pin-a-dynamic-binary-instrumentation-tool.html (visited on Dec. 9, 2022) (cit. on p. 17).

[Jac22] Hans Niklas Jacob. “Voltage Fault Injection Attacks on AMD’s Secure Processor”. en. In: (2022), p. 111 (cit. on p. 5).

[Kan+22] Rahul Kande, Addison Crump, Garrett Persyn, et al. “TheHuzz: Instruction Fuzzing of Processors Using Golden-Reference Models for Finding Software-Exploitable Vulnerabilities”. en. In: 2022, pp. 3219–3236. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/kande (visited on Nov. 29, 2022) (cit. on pp. 5, 13).

[KPW16] David Kaplan, Jeremy Powell, and Tom Woller. *AMD Memory Encryption Whitepaper*. 2016 (cit. on pp. 2, 5).

[Kin76] James C. King. “Symbolic execution and program testing”. In: *Communications of the ACM* 19.7 (July 1976), pp. 385–394. URL: https://doi.org/10.1145/360248.360252 (visited on Dec. 11, 2022) (cit. on p. 16).

[Küh22] Niclas Kühnapfel. *Evaluating the Electromagnetic Fault Injection Resistance of Modern CPUs: A Case Study on AMD’s Secure Processor*. 2022 (cit. on p. 5).

[Küh+22] Niclas Kühnapfel, Robert Buhren, Hans Niklas Jacob, et al. *EM-Fault It Yourself: Building a Replicable EMFI Setup for Desktop and Server Hardware*. arXiv:2209.09835 [cs]. Sept. 2022. URL: http://arxiv.org/abs/2209.09835 (visited on Dec. 5, 2022) (cit. on p. 9).

[Li+13] Li Li, Qiu Dong, Dan Liu, and Leilei Zhu. “The Application of Fuzzing in Web Software Security Vulnerabilities Test”. In: *2013 International Conference on Information Technology and Applications*. Nov. 2013, pp. 130–133 (cit. on p. 13).

[Lum22] Moritz Lummerzheim. “Fuzzing An Open-Source Trusted Execution Environment Using Rust”. de. In: (2022), p. 74 (cit. on pp. 5, 14).

[Mal21] Akash Malhotra. “AMD RYZEN™ PRO 5000 SERIES MOBILE PROCESSORS MAKING DEFENSES COUNT: DESIGNING FOR SUBSTANTIAL DEPTH”. en. In: (2021), p. 6 (cit. on p. 5).

[MFS90] Barton P. Miller, Lars Fredriksen, and Bryan So. “An empirical study of the reliability of UNIX utilities”. In: *Communications of the ACM* 33.12 (Dec. 1990), pp. 32–44. URL: https://doi.org/10.1145/96267.96279 (visited on Dec. 7, 2022) (cit. on p. 13).

[Mue+18] Marius Muench, Jan Stijohann, Frank Kargl, Aurelien Francillon, and Davide Balzarotti. “What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices”. en. In: *Proceedings 2018 Network and Distributed System Security Symposium*. San Diego, CA: Internet Society, 2018. URL: https://www.ndss-symposium.org/wp-content/uploads/2018/02/ndss2018_01A-4_Muench_paper.pdf (visited on June 16, 2022) (cit. on pp. 5, 11).

[Ngu+22] Trung Nguyen, Antonio Binachi, Kyungtae Kim, and Dave Jing Tian. *TruEMU: an extensible, open-source, whole-system iOS emulator*. Aug. 2022. URL: https://i.blackhat.com/USA-22/Thursday/US-22-Nguyen-TruEmu.pdf (cit. on p. 33).

[PAN22] PANDA. *PANDA*. 2022. URL: https://panda.re/ (visited on Dec. 6, 2022) (cit. on p. 12).

[Pro18] LLVM Project. *libFuzzer – a library for coverage-guided fuzz testing*. 2018. URL: https://llvm.org/docs/LibFuzzer.html (visited on Dec. 8, 2022) (cit. on p. 17).

[QEM22] QEMU. *QEMU*. Dec. 2022. URL: https://github.com/qemu/qemu (visited on Dec. 6, 2022) (cit. on p. 12).

[Rus22] Rust. *Unsafe Rust - The Rust Programming Language*. 2022. URL: https://doc.rust-lang.org/book/ch19-01-unsafe-rust.html#unsafe-superpowers (visited on Dec. 15, 2022) (cit. on p. 28).

[Sal+22] Majid Salehi, Luca Degani, Marco Roveri, Daniel Hughes, and Bruno Crispo. "Discovery and Identification of Memory Corruption Vulnerabilities on Bare-metal Embedded Devices". In: *IEEE Transactions on Dependable and Secure Computing* (2022). Conference Name: IEEE Transactions on Dependable and Secure Computing, pp. 1–1 (cit. on p. 5).

[SHC20] Majid Salehi, Danny Hughes, and Bruno Crispo. *uSBS: Static Binary Sanitization of Bare-metal Embedded Devices for Fault Observability*. en. 2020. URL: https://www.usenix.org/conference/raid2020/presentation/salehi (visited on June 2, 2022) (cit. on p. 17).

[SFB22] Eric Schulte, Vlad Folts, and Michael Brown. *Binary Lifter Evaluation*. en. Mar. 2022. URL: http://arxiv.org/abs/2203.13231 (visited on June 2, 2022) (cit. on p. 17).

[SA22] Sergej Schumilo and Cornelius Aschermann. *QEMU-NYX*. Nov. 2022. URL: https://github.com/nyx-fuzz/QEMU-Nyx (visited on Dec. 6, 2022) (cit. on p. 12).

[Ser+12] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. "AddressSanitizer: A Fast Address Sanity Checker". en. In: 2012, pp. 309–318. URL: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany (visited on Nov. 23, 2022) (cit. on p. 17).

[Son+20] Dokyung Song, Felicitas Hetzelt, Jonghwan Kim, et al. "Agamotto: Accelerating Kernel Driver Fuzzing with Lightweight Virtual Machine Checkpoints". en. In: 2020, pp. 2541–2557. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/song (visited on Nov. 23, 2022) (cit. on p. 4).

[Son+19] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, et al. "SoK: Sanitizing for Security". In: *2019 IEEE Symposium on Security and Privacy (SP)*. ISSN: 2375-1207. May 2019, pp. 1275–1295 (cit. on p. 48).

[Swi22] Robert Swiecki. *Honggfuzz*. Dec. 2022. URL: https://github.com/google/honggfuzz (visited on Dec. 8, 2022) (cit. on p. 17).

[Tes23] Tesla. *Tesla: Electric Cars, Solar & Clean Energy*. en. 2023. URL: https://www.tesla.com/ (visited on Jan. 18, 2023) (cit. on p. 5).

[Uni22] Unicorn. *Unicorn – The Ultimate CPU emulator*. 2022. URL: https://www.unicorn-engine.org/ (visited on Dec. 6, 2022) (cit. on pp. 11, 12).

[Wer22] Christian Werling. *PSPTool*. 2022. URL: https://github.com/PSPReverse/PSPTool (visited on Dec. 6, 2022) (cit. on p. 8).

[Wer19] Christian Werling. *Security Analysis of the AMD Secure Processor*. en. 2019 (cit. on pp. 2, 5, 8, 46, 47, 62).

[Win22] Wine. *WineHQ*. en. 2022. URL: https://www.winehq.org/ (visited on Dec. 6, 2022) (cit. on p. 12).

[Wor22] WorksButNotTested. *lightkeeper plugin for Ghidra*. Nov. 2022. URL: https://github.com/WorksButNotTested/lightkeeper (visited on Dec. 16, 2022) (cit. on p. 31).

[Yun+18] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. "QSYM: A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing". en. In: (2018), p. 18 (cit. on p. 16).

[Zhe+19] Yaowen Zheng, Ali Davanian, Heng Yin, et al. "FIRM-AFL: High-Throughput Grey-box Fuzzing of IoT Firmware via Augmented Process Emulation". en. In: (2019), p. 16 (cit. on pp. 4, 12).

Acronyms

AMD Advanced Micro Devices

ASP AMD Secure Processor

SoC System-on-a-Chip

ROM Read-only memory

TCG Tiny Code Generator

JIT Just In Time

YAML YAML Ain't Markup Language

AMD SEV AMD Secure Encrypted Virtualization

Intel TDX Intel Trust Domain Extensions

TEE Trusted Execution Environments

fTPM Firmware Trusted Platform Module

RoT Root of Trust

OS Operating System

UEFI Unified Extensible Firmware Interface

LLMP Low Level Message Passing

IR Intermediate Representation

OP-TEE Open Portable Trusted Execution Environment

ARM Advanced RISC Machines

KVM Kernel-based Virtual Machine

VM Virtual Machine

HDL Hardware Description Language

IoT Internet of Things

softMMU Emulated Memory Management Unit

MMU Memory Management Unit

ISA Instruction Set Architecture

SRAM Static Random-Access Memory

SPI Serial Peripheral Interface

MMIO Memory-Mapped I/O

FFS Firmware File System

CCP Cryptographic Co-Processor

IRQ Interrupt Request

SMN System Management Network

ARK AMD Root Signing Key

FET Firmware Entry Table

PC Program Counter

LR Link Register

SP Stack Pointer

CPSR Current Processor Status Register

TB Translated Block

QASan QEMUAddressSanitizer

ASan AddressSanitizer

UBSan UndefinedBehaviourSanitizer

MSan MemorySanitizer

LSan LeakSanitizer

TSan ThreadSanitizer

FIFO First-In First-Out

JTAG Joint Test Action Group

FFI Foreign Function Interface

List of Figures

List of Tables

List of Listings