Implementing an emulator for AMD's Platform Security Processor

Alexander Eichner

December 9, 2020

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig und eigenhändig sowie ohne unerlaubte fremde Hilfe und ausschließlich unter Verwendung der aufgeführten Quellen und Hilfsmittel angefertigt habe.

Berlin, December 9, 2020

Alexander Eichner

TU Berlin

Chair for Security in Telecommunications
Institute of Software Engineering and Theoretical Computer Science
Faculty of Electrical Engineering and Computer Science

Master Thesis

Implementing an emulator for AMD's Platform Security Processor

Alexander Eichner

1. Reviewer	Prof. Dr. Jean-Pierre Seifert Chair for Security in Telecommunications TU Berlin
2. Reviewer	Prof. Dr. Jan Nordholz Chair for Secure and Trustworthy Network-Attached System Architectures TU Berlin
Supervisors	Robert Buhren

December 9, 2020

Alexander Eichner
Implementing an emulator for AMD's Platform Security Processor
Master Thesis, December 9, 2020
Reviewers: Prof. Dr. Jean-Pierre Seifert and Prof. Dr. Jan Nordholz
Supervisors: Robert Buhren

TU Berlin
Faculty of Electrical Engineering and Computer Science
Institute of Software Engineering and Theoretical Computer Science
Chair for Security in Telecommunications
Straße des 17. Juni 135
10623 and Berlin

Abstract

Most modern micro processors contain additional undocumented execution cores which are used for system initialization or to aid the main operating system in power management tasks during runtime. Very often these cores serve also as a trusted execution environment for features like TPM or serving as the systems trust anchor. The most prominent example is Intel's ME engine. In the past, security researchers focused on this embedded system revealing multiple security issues in Intel's proprietary firmware. Modern AMD processors feature a similar system called the *Platform Security Processor (PSP)*. The firmware is proprietary and there is no public documentation about the full capabilities of the PSP which makes it an interesting target for researchers to check how secure the system is in reality. The goal of this thesis is to implement an emulator capable of running the proprietary firmware unmodified to give researchers a tool for further analysis.

Zusammenfassung

Die meisten modernen Mikroprozessoren enthalten undokumentierte zusätzliche Ausführungseinheiten, welche für die Systeminitialisierung oder zur Unterstützung des Betriebssystems bei der Energieverwaltung genutzt werden. Außerdem stellen sie häufig Trusted Execution Environments (TEE) für Features wie zum Beispiel Trusted Platform Module (TPM) bereit, oder fungieren als Trust Anchor (Vertrauensursprung bzw. Start einer Vertrauenskette). Das prominenteste Beispiel für ein solches System ist Intels ME engine. Sicherheitsforscher haben sich in der Vergangenheit auf dieses eingebettete System fokussiert und mehrere Sicherheitslücken in Intels proprietärer Firmware aufgedeckt. Moderne Prozessoren von AMD enthalten ein ähnliches eingebettetes System, welches von AMD als Security Platform Processor (PSP) bezeichnet wird. Die Firmware für das System ist ebenfalls proprietär und es gibt keine öffentliche Dokumentation über die vollständigen Fähigkeiten des PSP. Das macht es zu einem interessanten Ziel für Forscher, um die Sicherheitsaspekte der Firmware zu erforschen. Das Ziel dieser Arbeit ist die Implementierung eines Emulators, welcher in der Lage ist, die proprietäre Firmware auszuführen und Forschern damit ein Werkzeug für zukünftige Sicherheitsanalysen an die Hand zu geben.

Acknowledgements

First and foremost I would like to thank my supervisor Robert Buhren for giving me the opportunity to work on such an exciting topic and for supervising me throughout this thesis. Especially his review comments for the written part of this thesis were greatly appreciated. Thank you, Stefanie Lehmann and Christina Quast for proofreading this thesis and giving valueable feedback.

Contents

1 Introduction

Deeply embedded co-processors have been implemented in commodity CPUs for a long time. Intel includes their *Intel Management Engine (ME)* since 2008 into their architecture [@25] while AMD introduced their variant called the *Platform Security Processor (PSP)*, now just called *Security Processor (SP)* since 2013 [@23]. Intel's management engine has been subject to a lot of security research and reverse engineering in the past and new issues are frequently found and fixed by Intel [@12] while AMD's PSP has not been targeted to the same extent so far.

When AMD launched CPUs with their new Zen micro architecture in 2017, they had created a competing micro architecture which was on par with Intel and in many cases even faster than comparable CPUs. They were cheaper as well, resulting in an increased market share of AMD CPUs [@8] making them a viable target for attacks.

1.1 Motivation and Problem Statement

Like Intel, AMD CPUs have multiple embedded micro controllers besides the main x86 cores which are used for bootstrapping and managing the system during runtime. And like Intel the firmware required for these embedded cores is completely proprietary. The most prominent one is the Platform Security Processor (PSP). [@5, p. 156] states that the PSP manages the boot process and initializes various security related mechanisms. Furthermore, [@4] states that the PSP[1] establishes a hardware root-of-trust. This suggests that the PSP is the first general purpose CPU to run when power is applied to the system and executes instructions before the x86 cores are enabled. Despite providing security features like a firmware TPM on the desktop CPUs and AMD's Secure Encrypted Virtualization (SEV) in the server segment it also acts as the systems trust anchor for secure boot by verifying the integrity of the UEFI image before it is executed by the x86 cores[@33]. Due to the high-privileged nature of this component it is a highly valuable target for implanting permanent root kits or attacking the system during runtime by adversaries. In case of SEV an

[1] Named AMD Secure Processor in the source.

untrusted cloud provider could decrypt the memory of a virtual machine when code execution on the PSP is achieved as shown by [11]. At last, AMD plans to make the PSP available as a generic trusted execution environment (TEE) for third parties and already presented patches for the Linux kernel adding support for it [@24] even though there are no applications currently known to the author. All of these are good reasons for security researchers to have a look at the proprietary firmware. Static code analysis of the firmware is possible but the result is only valid for a particular version. If AMD decides to completely restructure the firmware one has to start from the beginning. For example changing layouts of structures between versions require manual work of the researcher to update the analysis to the new version to check for bug fixes or new issues. If the PSP is emulated however one can execute different versions of the firmware observing the behaviour. It also opens up the possibility to easily implement fuzzing for any parsers and communication interfaces as well as asserting quickly that a previously reported bug is indeed fixed. Because the PSP is a system on a chip of its own and a rather complex one as such, emulating all devices is next to impossible because there is no public documentation available. To overcome this limitation, this thesis introduces an emulator for the AMD PSP firmware. The emulator uses a technique similar to in-circuit emulation where the firmware is running inside the emulator but most of the I/O accesses are passed to the real hardware. This requires a small code module running on the PSP (which in turn requires a vulnerability to gain code execution) and a communication channel between the PSP and the emulator. The emulator allows security researchers to analyse multiple firmware generations in a full virtual environment or while it is interacting with a real system. In addition it supports interfacing with an external debugger like GDB to help debugging exploits for found security issues.

1.2 Contributions

In summary, the author's contributions presented in this thesis are:

1. Critical hardware interfaces and software components of the PSP were reverse engineered and publicly documented for the first time, giving an unprecedented insight into the architecture and inner workings of the PSP.
2. An emulator for the PSP firmware with an extensive amount of features was developed, allowing researchers to further analyse the firmware.
3. A communication interface and code module running on the PSP were developed to allow researchers to further explore supported PSPs.
4. Supporting tools and libraries were developed while implementing the emulator which can be incorporated into other projects not related to the PSP.

Together with the author's supervisor Robert Buhren, parts of this research were presented at the BlackHat USA 2020 conference [@10].

1.3 Thesis Structure

This thesis is divided into the following chapters:

Chapter 2

This chapter shortly describes related work done previously related to the PSP and in the area of system emulation and interfacing real hardware.

Chapter 3

This chapter provides an overview about AMD's Zen/Zen+ and to some extent Zen 2 CPU design, showing major components the CPU consists of, as well as connections between those components. It also details the architecture of the Platform Security Processor, key hardware interfaces and firmware components running on it. Understanding the CPU and PSP design is key to understand the architecture of the PSP emulator.

Chapter 4

The architecture and components of the implemented emulator are presented in this chapter in greater detail, describing the implementation of certain features.

Chapter 5

The implementation process and architecture of the proxy mode is explained in this chapter. Proxy mode allows to forward hardware accesses made by the firmware inside the emulator to a physical target system, thus providing a method to analyse the firmware without having to emulate every detail of the hardware.

Chapter 6

This chapter provides an overview about the latest feature implemented in the emulator allowing to execute the x86 UEFI firmware in a VirtualBox VM for analysis and forwarding hardware accesses to a physical target.

Chapter 7

The evaluation of the implemented solution is presented in this chapter.

Chapter 8

A summary of the implemented features and achieved progress throughout this thesis is given in this chapter.

Chapter 9

This last chapter provides an outlook of further research topics in that area and possible further enhancements to the emulator itself.

2 Related Work

Intel's ME has been extensively analysed by researchers in the past and Intel released a lot of security bug fixes till today [@12][@13]. The earliest publicly presented security issues for AMD's PSP known to the author where the ones named "AMDFlaws" by Farkas and Li On in 2018 [@16]. Details about the hardware of the PSP and firmware components itself weren't published so far with the exception of the work done by Robert Buhren and Christian Werling [11]. They analysed the firmware filesystem and published a tool [@28] to inspect and modify it. The research group also assessed the security of the SEV implementation of a Zen based AMD server CPU [18].

Interfacing real hardware with an emulator for dynamic firmware analysis of unknown targets has been done in the past. Avatar2 [21] is such a project, describing itself as a target orchestration framework. Avatar2 provides just the basic framework researchers can use to develop tools tailored to their use case. It interfaces with third party components like OpenOCD[1] to access a real target where the code being analysed would run normally. Avatar2 synchronizes the states between an emulator and the hardware in order to provide hardware access to the firmware running in the emulator. One of the supported emulators is QEMU[2]. However, interfacing with the real hardware through OpenOCD for example requires a proper debug interface, like JTAG for instance, to be exposed on the physical target.

In the case of the PSP almost nothing is known about the hardware peripherals and there is no standard debug interface available. The absence of a standard debug interface requires implementing a specialised target component. Instead of implementing the target component for avatar2, it can be more efficiently integrated into an emulator, to avoid the state synchronization's additional overhead of avatar2. Implementing a standalone emulator not being dependent on any framework grants complete control over the architecture to tailor it to the peculiarities of the PSP.

[1] `http://openocd.org/`
[2] `https://www.qemu.org/`

3 Anatomy of an AMD Zen CPU

This chapter gives an overview of AMD's Zen CPU design and major components. Only the high level architecture of those CPUs was disclosed publicly by AMD along with details about the micro architecture of the x86 cores. Internals about the PSP itself are not documented publicly. Everything presented starting from 3.1 in this chapter was reverse engineered during this thesis through static code analysis.

Starting with Zen, AMD employs what they call a chiplet design for their CPUs. In their design a CPU doesn't consist of a single monolithic silicon die but several smaller ones which are placed on the same package making up the CPU. The package substrate connects all the dies together and provides the contacts to the mainboard. The reason for this is to save cost and increase yield during manufacturing because those dies are smaller compared to previous architectures. Additionally, all CPU models share the same die layout starting with Zen, no matter whether it is a desktop or server CPU model [2, p. 14]. This further reduces cost because the number of lithography masks can be reduced.

Figure 3.1 shows a high level block diagram of a Zen and Zen+ AMD CPU. It consists of one or multiple core complex dies (CCDs) manufactured in a 14nm (12nm for Zen+) process by GlobalFoundries as stated in [@30] and [@31]. The CCD contains up to eight x86 cores split into two Core CompleXes (CCX) each containing up to four cores, a dual channel DDR4 memory controller and I/O peripherals. Each CCD also contains a Platform Security Processor (PSP) and a System Management Unit (SMU) as shown in [@30]. A CPU can have up to four CCDs with a total of 32 cores.

The architecture changed quite a bit with the introduction of Zen2. The memory controllers and I/O peripherals, like PCI-Express links, and USB and SATA controllers were moved to a dedicated I/O die as shown in [1, p. 18] and [@32]. The CCDs only contain the x86 cores as well as L2 and L3 caches, the SMU and PSP. This allows to manufacture the CCDs and I/O dies in different processes as well as having a more uniform memory access latency characteristics across all CCDs.

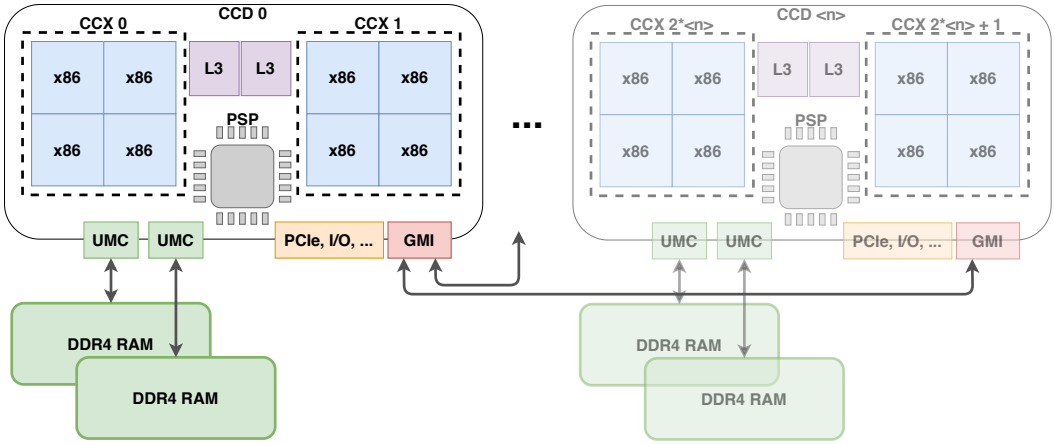

Fig. 3.1.: AMD SoC block diagram of a Zen/Zen+ based CPU

3.1 The Platform Security Processor (PSP)

The following section is giving an overview about the architecture of the PSP hardware and the software running on it. As there is no public documentation available from AMD, everything had to be reverse engineered during this thesis.

Each CCD contains a dedicated Platform Security Processor. It is based on an ARM Cortex-A5 core and is the first known general purpose CPU core inside the AMD SoC executing instructions before the x86 cores start executing. The PSPs are used to bootstrap the system and are for example responsible for initializing the memory controllers and train the communication links to the DDR4 memory modules. The PSP on CCD 0 acts as the master PSP and coordinates the PSPs on the remaining CCDs.

3.1.1 Memory layout of the PSP

Because the Cortex-A5 is a 32-bit ARM architecture the physical address space is 4 GiB in size. AMD divides the address space it into five regions depicted in figure 3.2. The address space of each PSP starts with 256 KiB (320 KiB for Zen2) of local SRAM for code and data. Starting at the 16 MiB boundary there is a 32 MiB sized region to access the System Management Network (SMN) which will be described in the next section. Starting at address `0x3000000` there is a 16 MiB MMIO region which maps the registers of local devices like the IRQ controller, timers and cryptographic accelerator for cryptographic operations into the address space of the PSP. The fourth and by far largest region starting at address `0x04000000` provides access to

the x86 address space, details will be explained in 3.1.4. The last region starting at address `0xffff0000` contains the on-chip bootloader which seems to be contained in an on-chip ROM according to our analysis and is executed first when power is applied to the CPU.

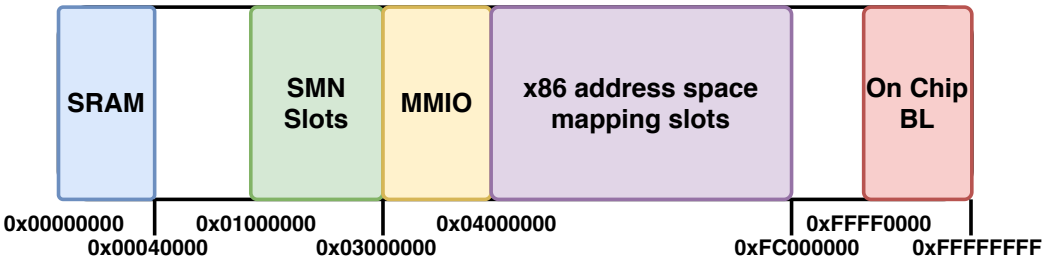

Fig. 3.2.: Regions of the PSP address space for a PSP in a Zen/Zen+ based CPU (not to scale).

3.1.2 Standard MMIO area

The PSP can be viewed as a self contained SoC inside the AMD SoC. As such, it has various peripherals attached which are accessed by the standard MMIO region ranging from $[0x03000000, 0x04000000[$. The following peripherals were identified through static analysis of code inside the PSP firmware accessing the MMIO regions:

- Two timer devices, at least one has the capability to generate interrupts.
- An IRQ controller for managing interrupts and to determine an interrupt source.
- The cryptographic Co-Processor (CCP) for hardware accelerated cryptographic operations.
- SMN mapping control registers controlling access to the system management network explained in more detail in 3.1.3.
- x86 address space mapping control registers controlling access to the full 48bit x86 address space explained in more detail in 3.1.4.
- Fuses to indicate the size of the public key, for example.

3.1.3 System Management Network (SMN)

The System Management Network is a dedicated internal bus with a separate 32-bit address space. Attached to the SMN are various devices like the DDR4 memory controllers or the SMU (System Management Unit). The SPI flash containing the systems firmware is also mapped into the SMN address space at a fixed location by the embedded SPI flash controller.

The PSP can access the devices attached to the SMN by mapping a portion of the address space into its own 32bit ARM address space providing a window into the address space of the SMN. This is achieved by a set of 16 32-bit wide control registers in MMIO space starting at address `0x03220000`. Each register controls 2 "slots" (AMD's terminology for this is unknown), the lower 16-bit half of control register n controls mapping slot $2 \cdot n$ while the upper half controls mapping slot $2 \cdot n + 1$ for a total of 32 mapping slots each being 1 MiB in size.

In order to create a mapping of SMN address `0xaaa12345` in the first slot starting at address `0x100000000` the SMN address to be mapped needs to be aligned on a 1 MiB boundary and shifted to the left by 20 bits, the result 0xaaa needs to be written to the lower half of the first control register. This will enable the mapping and any access to $[\texttt{0x01000000}, \texttt{0x010fffff}]$ will be mapped to the SMN address $[\texttt{0xaaa00000}, \texttt{0xaaafffff}]$. In order to access `0xaaa12345` the final address after the mapping will be `0x1012345`. The whole mechanism can be seen in figure 3.3 and in the basic framework to run code on the PSP created throughout this thesis[1].

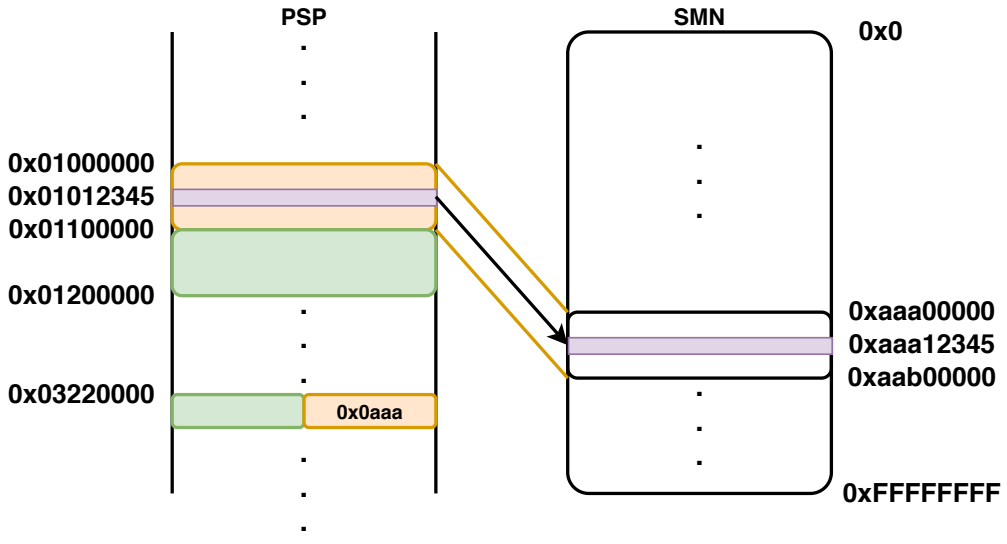

Fig. 3.3.: Process of accessing a location in the SMN address space from the PSP.

[1] `https://github.com/PSPReverse/psp-apps/blob/master/Lib/src/smn-map.c`

3.1.4 x86 address space mapping

The PSP has complete access to the full 48-bit physical x86 address space to access either DRAM or memory mapped devices. The largest part of the PSP address space is taken up by the 62 x86 mapping slots as shown in figure 3.2. Each slot is 64 MiB wide. The MMIO region contains a set of control registers for each slot. The general approach to map a specific region of the 48-bit physical address space into the PSP is the same as with SMN but because the regions are now 64 MiB wide the alignment is different. There is also a distinction between normal DRAM accesses and accesses to MMIO regions in the x86 address space, some slots can only handle MMIO accesses while others can only handle DRAM accesses. Trying to access a physical address backed by normal DRAM with a slot configured for MMIO or the other way around causes a data abort exception inside the PSP. Each slot is controlled by a set of six registers, the purpose of most of them being unknown. They are mostly set to fixed values inside the PSP off-chip bootloader. As with SMN the whole mechanism can be seen again in the PSP framework[2].

3.2 The bootstrapping process

The following section gives a broad overview of the bootstrapping process and the code modules running on the PSP. Figure 3.4 shows the major components involved in this process. It is not publicly documented and was reverse engineered during this thesis.

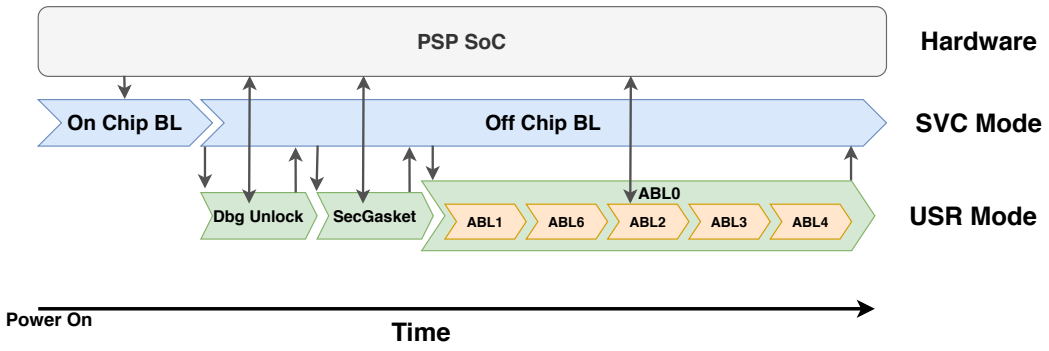

Fig. 3.4.: PSP boot process part 1 from power on until the memory controllers are initialized and all ABL stages returned to the off-chip bootloader

[2] `https://github.com/PSPReverse/psp-apps/blob/master/Lib/src/x86-map.c`

3.2.1 On-Chip bootloader

Our analysis showed that after power on or a reset the x86 cores are halted. Then the PSP starts executing the on-chip bootloader at the high vector location starting at `0xffff0000` which is a standard ARM feature explained in the ARM Architecture Reference Manual at [@9]. The goal of the on-chip bootloader is to bring the system into a minimal initialized state where accessing the SPI flash is possible, searching the SPI flash for the correct off-chip bootloader matching the identified architecture, loading the off-chip BL into SRAM, verifying the signature to ensure that only code authorized and signed by AMD will be executed and finally calling into the off-chip bootloader. The on-chip bootloader stores some information about the system in the last 4 KiB of SRAM (for Zen/Zen+ it starts at `0x3f000`) for use by the off-chip bootloader. Among other things, the block of information contains parts of the filesystem read from the flash device which are validated already. Likewise, the on-chip bootloader already verified and stored the used AMD public key which is used for signature verification of the individual firmware components in this memory region. The layout of the region as known up to now is given in listing 3.1.

3.2.2 Off-Chip bootloader

In the next phase the loaded off-chip bootloader takes control over the PSP and continues initializing auxiliary subsystems of the CPU in order to continue with the main bootstrapping phase. The off-chip bootloader sets up page tables containing mostly identity mapped regions (virtual address equals physical address) and enables the MMU. The virtual address space is divided into a supervisor mode code region ranging from `0x0` up to `0x14fff`. Starting at `0x15000` the user mode region begins where the off-chip bootloader loads and executes various code modules in a specific order as shown in 3.4. The user mode region ranges until `0x3efff` in case of Zen and Zen+ and `0x4efff` for a Zen2 system. However, the last two pages are used for the user mode stack. All code modules loaded and executed by the off-chip bootloader are running in user mode. The off-chip bootloader provides a syscall interface to the user mode code. The functionality ranges from syscalls for loading entries from the SPI flash, mapping and unmapping SMN and x86 addresses into the PSP address space to various syscalls providing access to the cryptographic co-processor for certain operations like SHA and AES operations and zlib decompression.

The first code module the off-chip bootloader executes is called `DebugUnlock`. The purpose of this code module is unknown but the name implies that it might be possible to enable a special debug mode for the PSP firmware which helps with

debugging. It could be used by mainboard vendors to enable debugging output aiding in firmware and mainboard development. The next module being executed is called `SecurityGasket` and as with the previous module the purpose is unknown. Both don't seem to have any noticeable impact on the bootstrapping phase.

Most of the system initialization is done in the ABL (AGESA Boot Loader) stages which are executed afterwards. The module consists of a main module called `ABL0` which is executed first and loads the different stages from the SPI flash and calls directly into them. The order in which these stages are executed depends on the ACPI sleep state the system is coming from. The different stages are named `ABL1` to `ABL6` and are mostly responsible for initializing the memory subsystem and generating information about the installed memory modules for the UEFI firmware. Control is returned to the off-chip bootloader when the ABL stages complete successfully. The off-chip bootloader then continues to load and verify the UEFI firmware into the now available DRAM and releases the x86 cores from reset by setting a bit in a register located at a specific SMN address. To our knowledge the bootstrapping process is the same between all CPU segments up to this point no matter whether the model is a Ryzen, Threadripper or EPYC CPU.

On a Ryzen CPU the off-chip bootloader loads a new operating system called SecureOS into the secure DRAM region along with some code modules and a loader. It then continues to disable the MMU and jump to the loader in secure DRAM, which is a special memory region in DRAM set aside by the PSP and not being accessible by the x86 cores. The purpose of the loader is to copy the SecureOS into the PSPs local SRAM, thus overwriting the off-chip bootloader completely and calling into it. According to our analysis SecureOS is based on the Kinibi TEE and is still mostly a black box as of now. Our analysis suggests that it is at least used for the firmware TPM option available on the consumer CPU models if a hardware TPM is not present on the mainboard.

The off-chip bootloader's last stage is completely different for EPYC CPUs. On EPYC, SecureOS is completely absent and the off-chip bootloader goes into an idle loop waiting for interrupts from external devices and services them as they occur. EPYC offers the Secure Encrypted Virtualization feature which involves the PSP for key management and programming the AES keys into the memory controllers. A dedicated code module exists for SEV which is loaded and executed by the off-chip bootloader when the host operating system running on the x86 cores issues a request to the PSP through a mailbox register interface. This module implements the SEV API related functionality as defined in [@7].

The difference between the boot process between Ryzen and EPYC is depicted in figure 3.5. Threadripper wasn't looked at due to lack of the required hardware.

This chapter presented the results of our static firmware analysis. These findings were leveraged to implement the initial version of the PSP emulator which is presented in more detail in the next chapter.

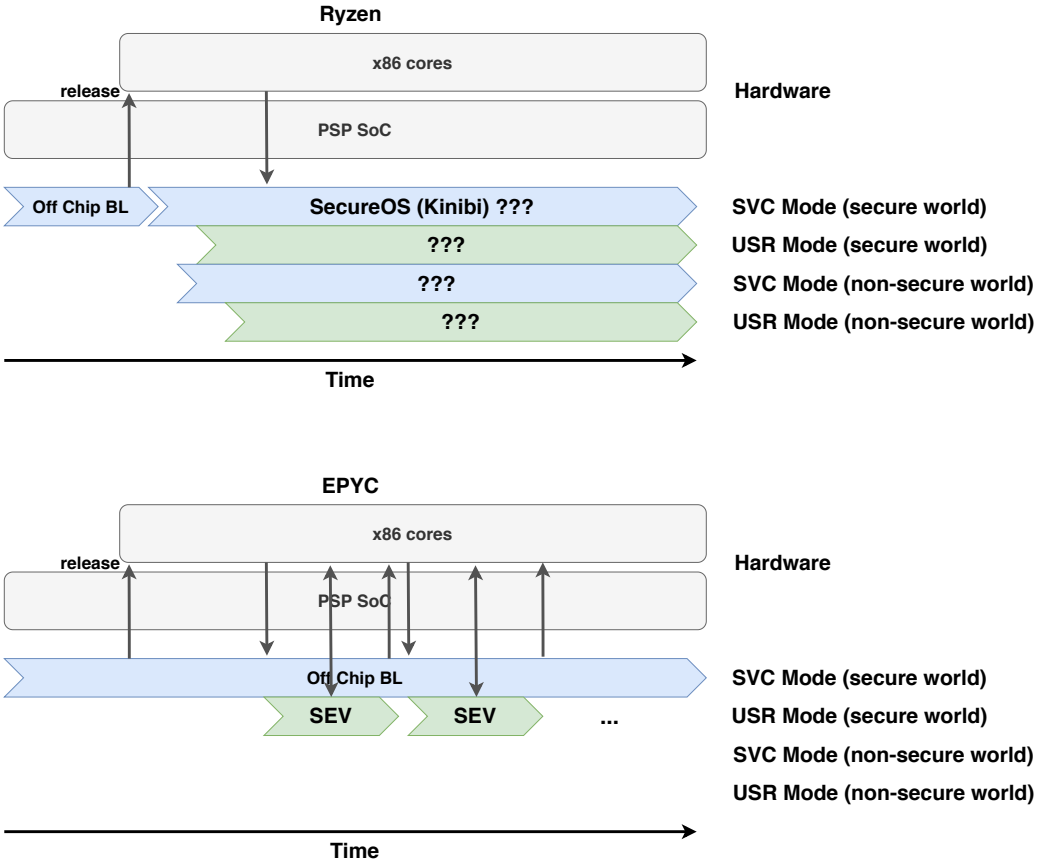

Fig. 3.5.: PSP boot process difference between Ryzen and EPYC CPUs showing the absence of SecureOS on the latter.

```
/**
 * The boot ROM service page structure as known,
 * residing at 0x3f000. Set up by the on-chip bootloader.
 */
typedef union PSPROMSVCPG
{
    /** Byte view. */
    uint8_t                     ab[_4K];
    /** Structured view. */
    struct
    {
        /** 0x00-0x0f: The flash directory header. */
        PSPFFSDIRHDR            FfsDirHdr;
        /** 0x10-0x40f: The flash directory entries. */
        PSPFFSDIRENTRY          aFfsDirEntries[64];
        /** 0x410-0x64f: AMD public key. */
        uint8_t                 abAmdPubKey[576];
        /** 0x650-0xa13: Unknown. */
        uint8_t                 abUnknown0[964];
        /** 0xa14: Boot mode (everything < 2 means signature checks disabled, 2 is 'secure'). */
        uint32_t                u32BootMode;
        /** 0xa18 - 0xa1d: Unknown. */
        uint8_t                 abUnknown1[6];
        /** 0xa1e: Number of physical x86 cores per CCX */
        uint8_t                 cCoresPerCcx;
        /** 0xa1f: Number of CCXs on die. */
        uint8_t                 cCcxs;
        /** 0xa20: Number of physical x86 cores on the CCD. */
        uint8_t                 cCoresPerCcd;
        /** 0xa21: Unknown. */
        uint8_t                 bUnknown2;
        /** 0xa22 - 0xa23: logical Cores per Complex */
        uint8_t                 logCoresPerComplex[2];
        /** 0xa24 - 0xa43: coreinfo structs - only seen access to the first 8 so far */
        PSPCOREINFO             aCoreInfo[16];
        /** 0xa44 - 0xa4f: unknown */
        uint8_t                 abUnknown3[12];
        /** 0xa50: Physical die ID of the PSP. */
        uint8_t                 idPhysDie;
        /** 0xa51: Socket ID. */
        uint8_t                 idSocket;
        /** 0xa52: Package type. */
        uint8_t                 u8PkgType;
        /** 0xa53: System socket count. */
        uint8_t                 cSysSockets;
        /** 0xa54: Unknown. */
        uint8_t                 bUnk4;
        /** 0xa55: Number of dies per socket. */
        uint8_t                 cDiesPerSocket;
        /** 0xa56 - 0xfff: unknown */
        uint8_t                 abUnknown5[1450];
    } Fields;
} PSPROMSVCPG;
```

Listing 3.1: Layout of the boot ROM service page as left behind by the on-chip bootloader for a Zen/Zen+ based system

4 PSPEmu - Architecture of the emulator

The goal of the PSP emulator is to ease dynamic analysis of the proprietary PSP firmware for researchers and help with the creation of new exploits for found security issues. It should implement at least the following set of features:

- Support execution of multiple firmware versions.
- Extensive logging of SMN, MMIO and x86 accesses, syscalls, etc.
- A debugger interface stub to allow using GDB for stepping through the code, breakpoints, watchpoints, etc. with the ability for source level debugging for code targeted for the PSP.
- Emulation of the most critical devices like the CCP.
- Creation of coverage traces to aid static firmware analysis.
- Allowing certain parts of the firmware to run in a fully virtual environment.
- A mode to pass through hardware accesses to a real PSP for non emulated devices allowing initialisation of a real system.

The resulting emulator should be easy to use, consisting of only a single binary. The configuration should be done completely through arguments given to the emulator to try out different options without having to change the implementation and re-compile the code. The basic usage of the implemented emulator is shown in listing 4.1. Further examples are available in the `psp-docs`[1] repository on GitHub.

```
$ ./PSPEmu \
    --emulation-mode sys \          # Sets the emulation mode to start at the off-chip BL stage
    --cpu-profile ryzen7-1800x \    # Selects a Ryzen 7 1800X for emulation (selects PSP profile)
    --flash-rom ./flash.ROM \       # The flash image to use for the emulated flash device
    --timer-real-time \             # Emulated timers tick in host real time
    --trace-log ./log \             # Destination for the log
    --intercept-svc-6 \             # Intercept and log svc 6 debug log syscalls
    --trace-svcs                    # Log all syscalls being made from usermode
```

Listing 4.1: Example invocation of the PSP emulator.

[1] `https://github.com/PSPReverse/psp-docs/tree/master/Logs/ryzen7-1800x`

At the core the PSP emulator requires an ARM instruction emulator capable of emulating the ARMv7 ARM and Thumb2 instruction set. The ARM instruction emulator has to support full system emulation maintaining the complete Cortex-A5 CPU state including MMU and Co-processor states as well as support for passing MMIO accesses to the device emulations. Because the emulator should be platform independent, virtualisation solutions like KVM which are restricted to an ARM host were discarded quickly. Using QEMU[2] with its ARM full system emulation might have been possible, but the complexity and scarce documentation of the QEMU internals makes it more difficult. Being forced into the QEMU architecture could have also complicated emulation of the SMN and x86 address space or the hardware pass through mode which was not conceivable upfront due to the lack of documentation.

A single library was desired for the ARM emulation which allows building the emulator around that, thus having complete control over the rest of the emulators architecture. Unicorn[@27] was chosen as the base for the PSP emulator, a lightweight CPU emulator library based on QEMU internally. It has a simple to use but powerful API enabling the emulator to implement features for tracing code execution of the firmware easily. During development however it turned out that Unicorn is not capable to be used as a full system emulator because those introspection features interfere with for example the MMU, which is heavily used by the SecureOS later on in the PSP boot process. TrustZone is not properly handled as well so Unicorn had to be modified in order to support those features properly[3]. A list of changes done to Unicorn in order to be usable for the emulator is given in appendix C.

An overview of the emulators architecture can be seen in figure 4.1. Not all interactions between components are shown for simplicity reasons. The tracing component `psp-trace` for example is used by almost everything else for logging to the trace log. The emulator is split into self contained components which interact with distinct defined interfaces enabling easy extension of the emulator with new device emulations and features for tracing without touching the implementation of the interfaced components itself. It also enables easy replacement of implementations if the need arises in the future. For example, it is possible to replace Unicorn with another ARM system emulation implementation by just rewriting the implementation of the `psp-core` component. A brief overview of most of the individual components is given next.

[2] `https://www.qemu.org/`
[3] `https://github.com/PSPReverse/unicorn/tree/mmio`

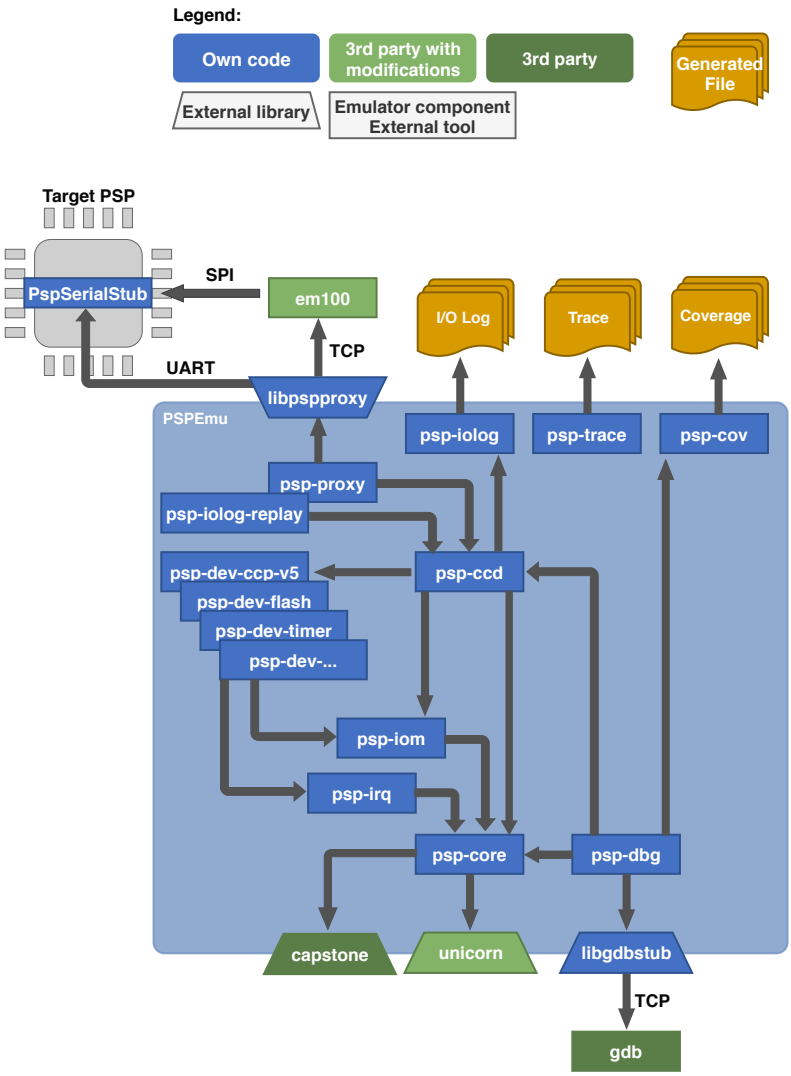

Fig. 4.1.: Architectural diagram of the PSP Emulator

4.1 psp-core - The ARM emulation core

`psp-core` is the first of the two main components of the emulator and implements an ARMv7 based CPU core by wrapping the Unicorn API and providing an abstract API. The API provides functionality to register memory regions, handlers for MMIO ranges and means to register handlers which are called when a certain code range is executed. Additionally it provides helpers to read or write either physical or virtual memory based on the current CPU mode, transparently handling virtual to physical address translation.

4.2 psp-iom - I/O Manager

`psp-iom` is the second main component and handles everything related to MMIO. It implements the SMN and x86 regions and handles the control registers in the MMIO region to enable the emulated PSP firmware to access devices behind SMN and x86 addresses. It also implements interfaces to call a registered handler for a certain device access being made by the firmware, no matter at which SMN or x86 slot the device is currently mapped. This is used for tracing I/O access being made by the firmware and to enable I/O breakpoints for the debugging component.

4.3 psp-irq - IRQ manager

`psp-irq` emulates the register interface of the interrupt controller present in real PSPs and allows individual devices to assert and clear interrupts to the firmware. The implementation is incomplete because AMD didn't use a publicly available interrupt controller design available for ARM but created something new specific to the PSP. This requires reverse engineering the register interface. The interrupt controller is not used by the off-chip bootloader during the early system initialisation so the implementation was started at the end of this thesis.

4.4 psp-dev-... - Device emulations

The individual device implementations like the flash device, CCP and x86 UART each have their dedicated source file. They all have a registration record which allows instantiation of multiple devices of each kind and allows adding new emulations quickly. All devices make use of the I/O manager API to register handlers for their respective regions.

4.4.1 CCP - Cryptographic Co-Processor device emulation

Of all the emulated devices the CCP emulation together with the flash device emulation are the most important and sophisticated ones. It implements all requests the firmware requires like copying data between memory locations, AES encryption and decryption, RSA encryption, SHA hashing, zlib decompression required for compressed code modules as well as certain ECC operations.

4.4.2 psp-dev-flash - SPI flash controller emulation

The SPI flash controller emulation provides the emulated firmware access to the flash image used with the emulator. It allows creating a special trace log which is compatible with PSPTools `psptrace`[4] command to view SPI flash accesses made by the firmware and correlate them with the raw flash image.

4.5 psp-ccd - Core Complex Die emulation

`psp-ccd` wraps all components related to a single PSP and presents what is a single CCD in a real system with a dedicated PSP. Wrapping all components into `psp-ccd` allows implementing support for multiple PSPs in the emulator in a later stage.

4.6 psp-dbg/libgdbstub - Debugging interface

`psp-dbg` implements a stub to access the emulator from a debugger like GDB. It uses the `libgdbstub`[5] library which implements a parser for GDBs remote serial protocol and was specifically created for `PSPEmu` but can also be incorporated into other projects. The `psp-dbg` component wraps the library and implements the necessary callbacks to create, modify and delete breakpoints, step through the executed firmware, and provide means to read and write registers. It also implements special requests specific to `PSPEmu`, for example to configure I/O breakpoints. These special commands can be accessed from within GDB's own `monitor` command. Figure 4.2 shows an example debugging session inside GDB.

[4] `https://github.com/PSPReverse/PSPTool/blob/master/bin/psptrace`
[5] `https://github.com/AlexanderEichner/libgdbstub`

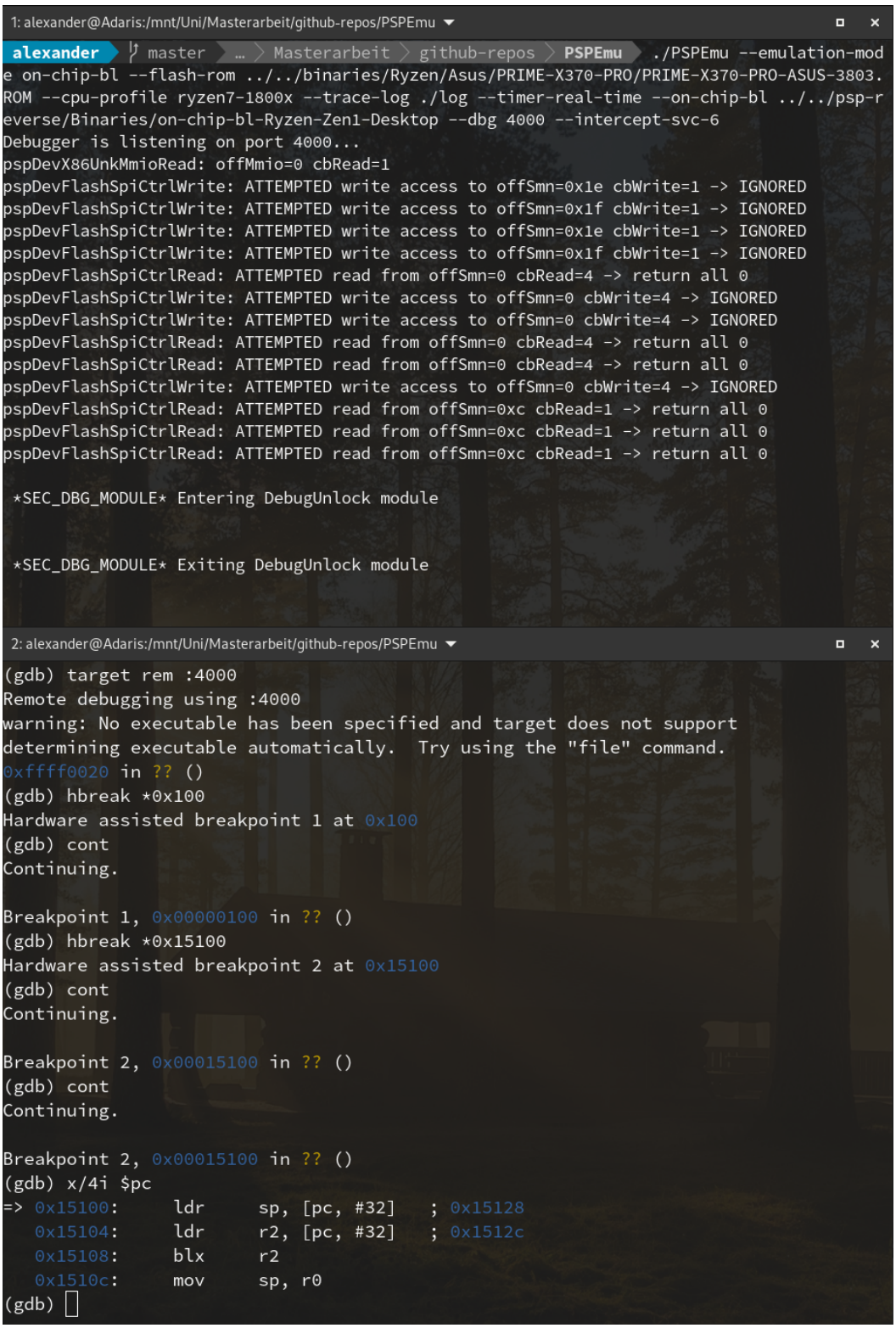

Fig. 4.2.: Screenshot of PSPEmu running with GDB attached and breakpoints configured

4.7 psp-trace - Trace log writer

`psp-trace` provides an interface to write a human readable trace log of a firmware run inside the emulator. It is accessed from all other components. Listing 4.2 shows the start of a trace from a run of the firmware with proxy mode enabled. Among other things, the log allows a user to quickly identify which mode the CPU is in when a certain event like a MMIO or SMN access happens. The log is split into several columns, each having a different purpose:

1. A monotonically increasing event identifier.

2. Event severity ranging from debug events to errors.

3. The event source. The example shown in listing 4.2 shows that first four events were generated from the proxy component and the last one from a MMIO access.

4. Information about the emulated ARM core state when the event occurred.

    a) Program counter value

    b) Link register value

    c) CPU mode (Supervisor, user, etc.)

    d) Indicator for secure or non-secure execution environments

    e) Indicator whether the MMU is enabled or not

    f) Interrupt masked indicator

    g) Fast interrupt masked indicator

    h) Value of TTBR0, the register holding the physical page table root address

5. Event dependent data, the example shows four debug strings sent from the PSP stub and the last event is a device register read which was forwarded to the real PSP with the given address, size and returned value.

```
1  [...]
```

Listing 4.2: Excerpt from a trace log showing 4 debug strings sent from the PSP stub and one register read from the firmware which is forwarded to the real PSP.

4.8 psp-cov - Coverage trace log writer

`psp-cov` enables `PSPEmu` to write coverage traces compatible with the DrCov format described at [@15] in order to allow researchers to follow the flow of instructions with tools like Ghidra or IDA. Coverage traces can be either created for a complete run or can be created during runtime for certain code locations from GDB. Figure 4.3 shows how Ghidra highlights executed basic blocks from a loaded coverage trace.

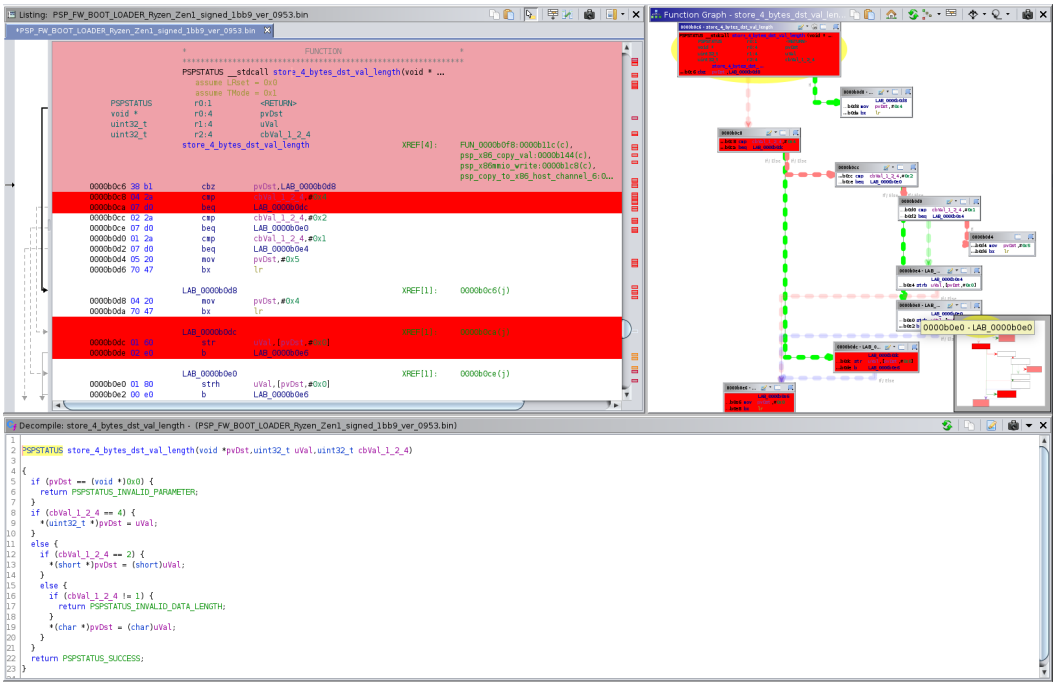

Fig. 4.3.: Screenshot of Ghidra with a loaded coverage trace marking the executed basic blocks in red for a single invocation of the displayed function

4.9 psp-flash - Flash filesystem parser

`psp-flash` allows the emulator to load the off-chip bootloader or individual components like public keys or single user space modules from a given flash image by implementing a simple parser of the flash filesystem, similar to PSPTool. This is a convenience function for researchers allowing usage of unmodified UEFI firmware images from mainboard vendors and avoiding extra work extracting the different components from the flash image using PSPTool and loading them manually into `PSPEmu`.

4.10 psp-proxy - Real hardware access proxying

This component is used to forward hardware accesses performed by the firmware, which are not emulated right now, to a real PSP running a code stub. This allows inspecting firmware interaction with real hardware and provides access to the protected AES keys inside a real CCP. The implementation and details of `psp-proxy` will be discussed in greater detail in the following chapter.

4.11 psp-iolog/psp-iolog-replay - I/O log writer and replay

One of the unique features of `PSPEmu` is to create a binary I/O log of a particular run of the firmware with real hardware and proxy mode enabled for example, and to replay it later when access to the hardware is not possible. It is presented in more depth in section 5.8.

4.12 psp-x86-ice - x86 UEFI firmware hardware access proxying

One of the last features implemented in `PSPEmu` and not complete yet is the ability to execute the x86 UEFI firmware inside a specially patched VirtualBox VM[6] and forward hardware accesses made by the x86 UEFI firmware using the PSP proxy to real hardware, allowing analyses of the interaction between the firmware running

[6] `https://github.com/PSPReverse/vbox-ice`

on the x86 cores and the PSP. This requires a dedicated stub to run on the x86 core of a real AMD CPU[7]. This component will be presented in more detail in chapter 6.

[7] `https://github.com/PSPReverse/x86-stub`

5 Proxy mode - Communicating with a black box

The initial implementation of the emulator was started from what was known about the PSP from the analysing the proprietary firmware using Ghidra[1], which included knowledge about the ARM core and how the firmware set up the environment. With the initial version of the emulator starting to execute the firmware, the next step was to implement emulation for devices accesses by the firmware. This was done by executing the firmware until it hung or unicorn returned with an error because an unmapped memory region was accessed. The executed code was analysed and the functionality was then implemented to a point where the firmware would continue to run. This approach resulted in quick progress up to the point where the firmware would run the first user code modules from the ABL stage, as shown in the boot process figure 3.4. This step required a working CCP emulation. Because there is no public documentation about the register interface, the open source CCP driver in the Linux kernel was used as a reference [@3] to implement the initial emulation.

However, after this point was reached another approach had to be chosen as reverse engineering and implementing every accessed register without initially knowing the purpose and behaviour is a lengthy reverse engineering process. For example, side effects due to register accesses and how they would influence other devices and registers are very difficult to predict due to the complexity of the PSP firmware. The idea was to pass through device accesses to a real PSP and return the result back to the firmware through the emulator. The emulator would act as proxy between the firmware and the real PSP, hence the name proxy mode for this type of operation. This mode however has two preconditions to be usable:

1. Code execution on the PSP in order to communicate with the emulator.
2. A stable and fast communication channel for exchanging data between the emulator and real PSP.

The first requirement was already achieved by the previous work of Robert Buhren and Christian Werling [11]. It was possible to run code at the highest privilege

[1] `https://ghidra-sre.org`

level of the PSP and therefore have complete control over it. This was achieved by exploiting a buffer overflow inside the on-chip bootloader. The second requirement turned out to be a much bigger challenge as the PSP and its surrounding peripherals where mostly a black box at this time. An interface was required which would be available immediately after the PSP started while the rest of the system was not fully initialised still.

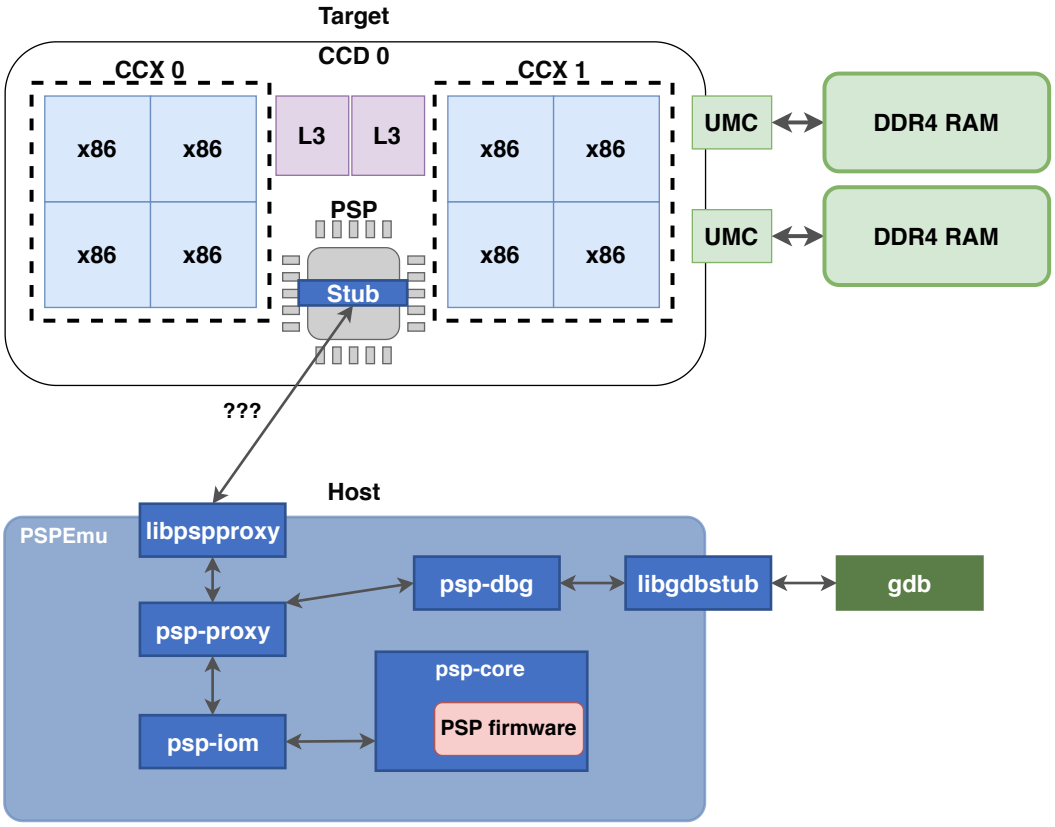

Fig. 5.1.: The general architecture of the proxy mode.

Figure 5.1 shows the overall architecture of the proxy mode functionality. Whenever the firmware tries to access an SMN, MMIO or x86 address which is not handled by an emulated device the I/O manager `psp-iom` passes the request to the newly introduced `psp-proxy` component. `psp-proxy` handles the communication to the real PSP by using `libpspproxy` which implements the lowlevel communication backend and data exchange with the stub running on the real PSP. The stub will execute the I/O access and return the result to `psp-proxy`. The result is then passed back to the emulated firmware and instruction emulation is resumed. Other components are able to access the proxy interface as well. An example is `psp-dbg` to allow custom commands, allowing researchers to directly access the real PSP address space from within GDB.

5.1 Communication requirements

As stated previously, one of the requirements for proxy mode is a stable and fast communication channel between the stub running on the PSP and the emulator. The channel consists of two parts, each having its own requirements:

1. A physical interface to connect the target system containing the real PSP and the host system running the emulator, allowing basic data exchange which should be reliable for optimal performance.

2. A protocol to forward requests from the emulator to the stub and return responses as well as notifications in the opposite direction when something on the PSP happens which was not initiated by previous request, like interrupts for instance.

For the physical interface there are several options present on modern mainboards, each having its own advantages and disadvantages as shown in table 5.1. The SPI flash interface was chosen initially because it was the only accessible physical interface at the time when the proxy mode implementation was started.

Interface	**Easily accessible from outside**	**Hardware required for access**	**Speed**	**Early access possible**
Serial port	++	Serial cable	–	Yes[2]
SPI flash	+	Flash emulator	+	Yes
USB3 debug port[3]	++	USB3 debug cable	++	No[4]
Network interface	++	Network cable	++	No[4]

Tab. 5.1.: Overview of the possible communication channels available

The protocol on top of the physical interface has to fulfil the following requirements:

- Independence from the underlying physical link to allow switching to a faster and/or more reliable interface when it becomes available.

- Detection of dropped data and corruptions in order to prevent running into invalid emulation states when an I/O access returned corrupted data previously.

The design of the devised protocol is explained in 5.3.2.

[2] The initialization sequence was not known initially, see section 5.5

[3] `https://www.usb.org/sites/default/files/documents/usb_debug_class_rev_1_0_final_0.pdf`

[4] Requires DRAM to be initialized, which disqualifies it for early bootstrapping. Also requires rather complex drivers in the stub.

5.2 Hardware Setup

The following hardware was chosen as a starting point to implement proxy mode:

- AMD Ryzen 7 1800X, because unlike the smallest EPYC it is a single CCD design, meaning only one PSP has to be emulated.
- Asus Prime X370-Pro as the motherboard because it exposes the SPI flash interface as a standard pin header with 2.5mm pitch. This makes it easy to attach wires to the SPI bus instead of soldering something onto the embedded SPI flash chip and allows reverting the setup quickly to the original state if required.
- 2 x 4 GiB of DDR4-RAM in a dual channel configuration.

The V_{CC} pin on the SPI flash soldered onto the motherboard was desoldered to not interfere with the flash emulator. A wire was soldered onto the pin which can be connected with the correct pin on the header to get it working again quickly. The result can be seen in figure 5.2.

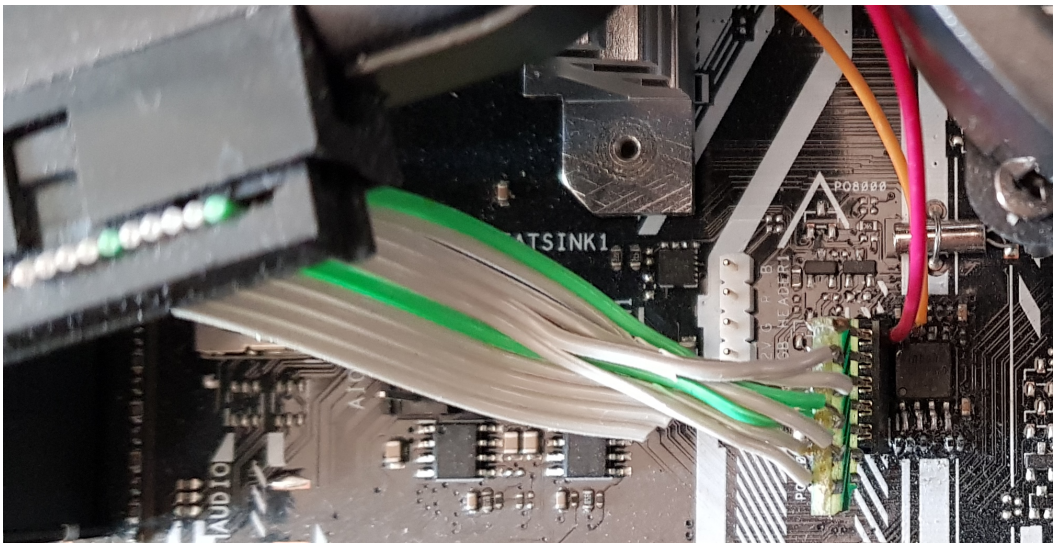

Fig. 5.2.: The SPI header connected with the flash emulator (left) and the embedded SPI flash modification (right).

5.3 Accessing the SPI Flash - First attempt

From the work done by Christian Werling and Robert Buhren in [11] the only known device MMIO range being accessible from outside of the CPU, was the SPI flash storing the firmware. The SPI flash controller inside the AMD CPU maps the content of the SPI flash chip into the SMN address space starting at a constant address. The start address depends on the version of the Zen architecture. When the firmware accesses a particular offset inside the flash, either by reading or writing the start SMN address plus offset, the SPI flash controller will translate that to the necessary commands which are sent to the flash chip and the response is parsed and returned to the firmware. The SPI flash controller caches read operations for performance reasons.

In order to communicate with code running on the PSP over the SPI flash, a protocol was devised to only require read and write accesses to the SPI flash. This requires monitoring SPI transactions related to the flash and updating the content of the flash while the system is running. This was achieved by using a commercially available Flash Emulator, specifically the DediProg EM100Pro-G2[5]. This flash emulator allows tracing all SPI commands and sending the trace over USB to the host it is attached to. Furthermore, it is possible to change the content of the flash image stored in the flash emulator's internal RAM on the fly with some caveats which will be discussed in the next section when the protocol is described in detail. The availability of the open source tool `em100`[6] and the possibility to control the emulator from a Linux host has driven the final decision to use this flash emulator because the controlling tool needs to be modified in order to implement the devised protocol.

5.3.1 The SPI message channel protocol

One of the biggest obstacles when using the flash emulator is the fact that updating the flash content on the fly requires disabling the SPI emulation. This results in the PSP reading all bits set when accessing the flash because the SPI side of the flash emulator goes into the high impedance state and the internal pull-up resistors tie the line to V_{CC}. This would result in data corruption when the PSP tries to read from the flash when the host side modifies the content at the same time. Avoiding that requires a locking mechanism when the code running on the PSP wants to access the flash to ensure that the flash content is accessible throughout the read operation.

[5] `https://www.dediprog.com/product/EM100Pro-G2`
[6] `https://review.coreboot.org/cgit/em100.git`

The mechanism is depicted in figure 5.3 and implemented by defining a fixed location in the flash at offset `0xaa0000`. This location is written with a fixed value, in the following referred to as a magic value, by the PSP whenever it wants to lock the flash. The modified `em100` tool maintains the current locking state and starts in the UNLOCKED state (**1**), allowing the tool to update the flash content at any time, for example when new data to be read by the stub on the PSP arrives. When the stub wants to access the flash to check whether there is new data available for reading, it writes the value SPI_FLASH_LOCK_LOCK_REQ_MAGIC to the fixed location (**2**) and then starts to poll the location which will still return SPI_FLASH_LOCK_UNLOCKED_MAGIC (**3**). The write operation to the flash by the PSP stub generates a trace record inside the flash emulator and the trace record is parsed by `em100` (**4**). However, the write operation will not update the flash content inside the flash emulator itself. Upon encountering the SPI_FLASH_LOCK_LOCK_REQ_MAGIC value from the write process, `em100` sets the locking state to LOCKED (**5**) and disables the flash emulation (**6**). This will cause the stub to read `0xFFFFFFFF` when polling the fixed location (**7**). `em100` now updates the fixed location with the SPI_FLASH_LOCK_LOCKED_MAGIC value (**8**) and enables the flash emulation (**9**). The next read operation done by the PSP stub will now return SPI_FLASH_LOCK_LOCKED_MAGIC (**10**) causing the PSP stub to exit the polling loop and complete the locking. The stub now can safely access the flash content. Unlocking the flash by the PSP stub is done in the same fashion but with different magic values.

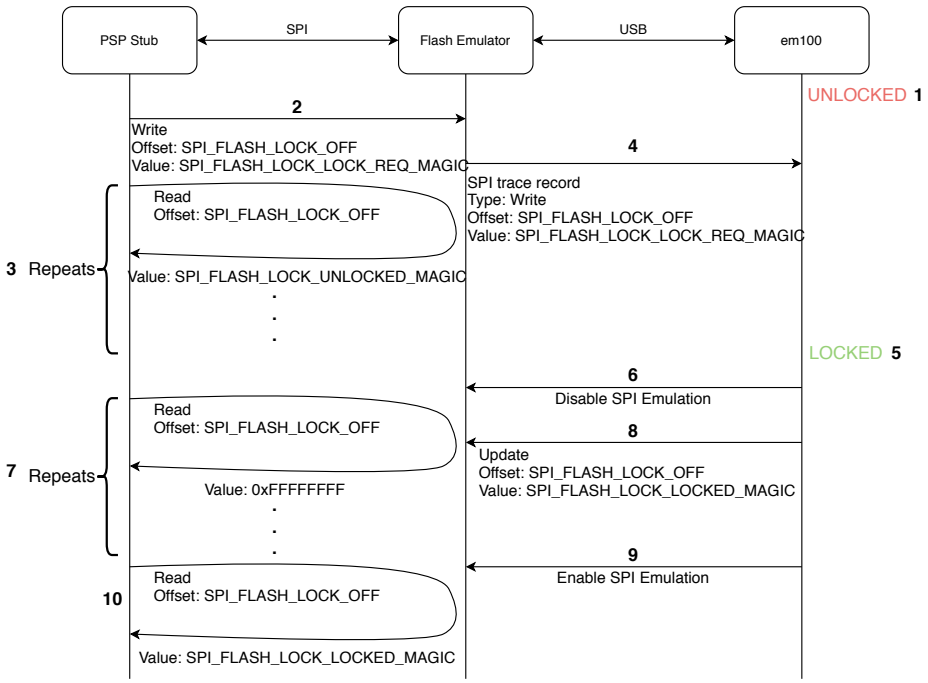

Fig. 5.3.: The SPI flash locking protocol

With the locking in place a reliable way for exchanging data between the PSP and the host exists. The rest of the SPI message channel protocol includes fixed locations in the flash for reading data by the PSP, a register indicating how much bytes are available for reading from the data location and locations for status registers to indicate whether the protocol is actually available. The implementation of the protocol was split into a dedicated backend[7] on the side of the PSP to make adding new transport channels easier when they become available, while still being able to revert to older variants. On the host side `em100`[8] gained a network mode. `libpspproxy` communicates over a TCP connection with `em100` implementing the low level SPI message channel protocol.

5.3.2 The PSP serial protocol

After basic data exchange was established, a protocol was needed to convey the actual requests to the PSP to read from memory, MMIO, SMN, etc., fulfilling the requirements mentioned in 5.1. The devised protocol[9] is tailored towards the PSP but can be easily modified for other environments. The protocol defines a Protocol Data Unit (PDU) which is of varying length. It consists of a fixed-sized header followed by a payload, with length and content depending on the operation. The PDU is completed with a fixed-sized footer. The header starts with a distinct marker value denoting the direction of the transmission. It is followed by the size of the PDU excluding the header and footer size and a counter which is monotonically incremented for every PDU sent in each direction. After that comes an ID denoting the type of PDU. Three types are defined:

- Requests — Sent from the host to the PSP requesting some operation to be executed on the PSP.
- Responses — Sent from the PSP to the host in response to a previously received request, there is only one response for every request.
- Notifications — Sent from the PSP to the host when something happens on the PSP which is not a response from a previous request like, for example an interrupt happening on the PSP.

[7] `https://github.com/PSPReverse/psp-apps/blob/master/PspSerialStub/pdu-transp-spi-flash.c`
[8] `https://github.com/PSPReverse/em100/blob/network-mode-v1/net.c`
[9] `https://github.com/PSPReverse/psp-includes/blob/master/psp-stub/psp-serial-stub.h`

The footer completes the PDU with an additive checksum for the payload and a closing marker value. The markers in the header and footer make it easier to observe the data flow during debugging of the protocol and involved code because they visualize PDU boundaries. An example of exchanged PDUs is shown in figure 5.4. PDU processing on the host is done by libpsspproxy[10] and on the PSP by the stub code[11].

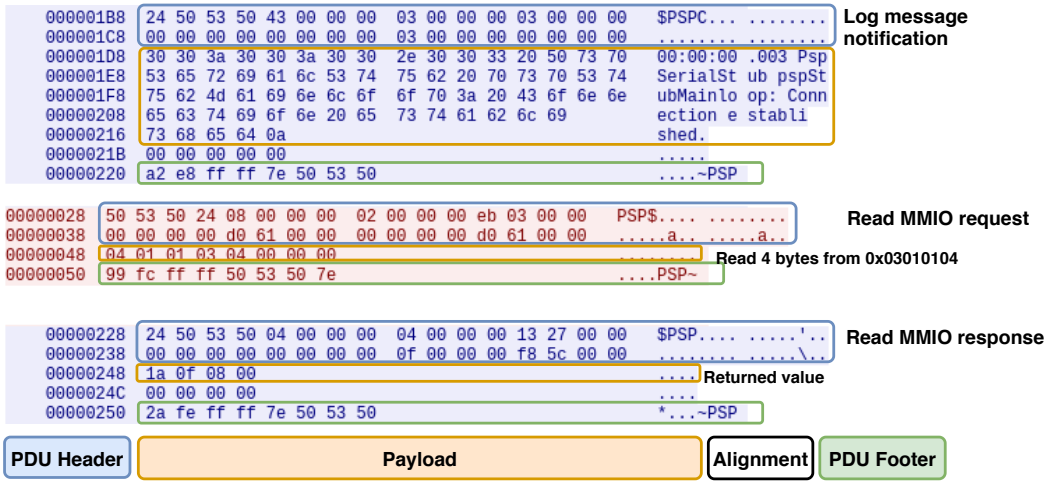

Fig. 5.4.: PDU exchange between the PSP stub and the emulator. The PDUs for the last two log lines in listing 4.2 are shown.

5.3.3 Preliminary evaluation

With the previously explained components a stable communication was established, which allowed forwarding I/O accesses from the emulator to the PSP. However, it turned out to be extremely slow, making it unusable for emulating a complete run of the PSP firmware. With the SPI message channel protocol in place it was possible to do only two to three register accesses per second. Running the on-chip bootloader inside the emulator took approximately 20 minutes and it is not doing many I/O accesses compared to the off-chip bootloader. The low performance is due to the high overhead imposed by the frequent locking and unlocking of the flash channel by the stub. It continuously polls the location which indicates how many bytes there are to be read. Each iteration requires a lock and unlock operation. The flash emulator creates new trace records for the SPI accesses which the `em100` tool has to parse and act upon. Disabling and enabling the SPI flash emulation in the flash emulator is also not instantaneous, increasing the overhead even further. While

[10] `https://github.com/PSPReverse/libpsspproxy/blob/master/psp-stub-pdu.c`
[11] `https://github.com/PSPReverse/psp-apps/blob/master/PspSerialStub/main.c`

not being feasible for the emulator itself, the SPI message channel turned out to be useful for further exploration of the PSP peripherals by using the interactive python shell and `libpspproxy`. This avoids the need to modify and recompile code for the PSP and parse the output on the SPI bus with a logic analyser or the flash emulator, which is even more time consuming.

5.4 Enabling the DediProg Hyper Terminal mode

After the first attempt to transport data over the SPI bus turned out to be too slow for the emulator, attention was turned towards making it possible to send arbitrary commands over the SPI bus. This enables the possibility to use the flash emulator's so called Hyper Terminal mode [@14, p. 24], allowing to exchange data between the stub and the host controlling the flash emulator efficiently. The Hyper Terminal mode defines additional SPI commands which can read and write internal FIFOs in the flash emulator, bypassing the SPI flash emulation completely. However, this requires low-level access to the SPI bus, which was not available at this point, to send arbitrary commands. The initial analysis of the PSP firmware with Ghidra revealed a set of functions which seemed to implement the required functionality in order to read information about the attached flash chip's size. This was done by accessing a special set of registers. Experiments using those registers did not result in any reliable communication on the SPI bus though. However, AMD's Processor Programming Reference manuals [@6], which are intended for OS and UEFI firmware developers, explain how to access and program various embedded hardware on AMD's SoCs. The PPRs describe a low-level SPI controller interface, enabling the x86 UEFI firmware to send arbitrary commands to the attached SPI flash, which is exactly what is required in order to use the flash emulator's Hyper Terminal mode.

With the described register interface in the PPR and the address of the registers in SMN space gathered from the firmware it was finally possible to send arbitrary commands on the SPI bus, enabling implementation of a transport backend for the PSP stub[12] using the DediProg hyper terminal interface. With a few other fixes it was now possible to run the complete off-chip bootloader including all ABL stages in about 5 minutes when using this communication channel because the overhead for exchanging data decreased greatly from the previous method. It is not required to disable and re-enable the SPI flash emulation inside the flash emulator when

[12] `https://github.com/PSPReverse/psp-apps/blob/master/PspSerialStub/pdu-transp-spi-em100.c`

sending data to the stub anymore and the stub does not have to use the lengthy lock and unlock protocol when trying to access the SPI flash and receive data.

Figure 5.5 shows the result of the work. While the system is not fully booting yet, the display output shows that the UEFI firmware started execution on the x86 cores and got to a point where it also initialized the attached graphics adapter to show an error message. This proves that the PSP successfully executed all ABL stages as shown in the boot process figure in 3.4 and also got passed the point to release the x86 cores as shown in figure 3.5. The error message is due to an incorrect emulation of the CCP detailed in 5.6, resulting in the PSP firmware entering some sort of recovery mode which the UEFI firmware detects. This causes the UEFI to show an error message and not continuing further.

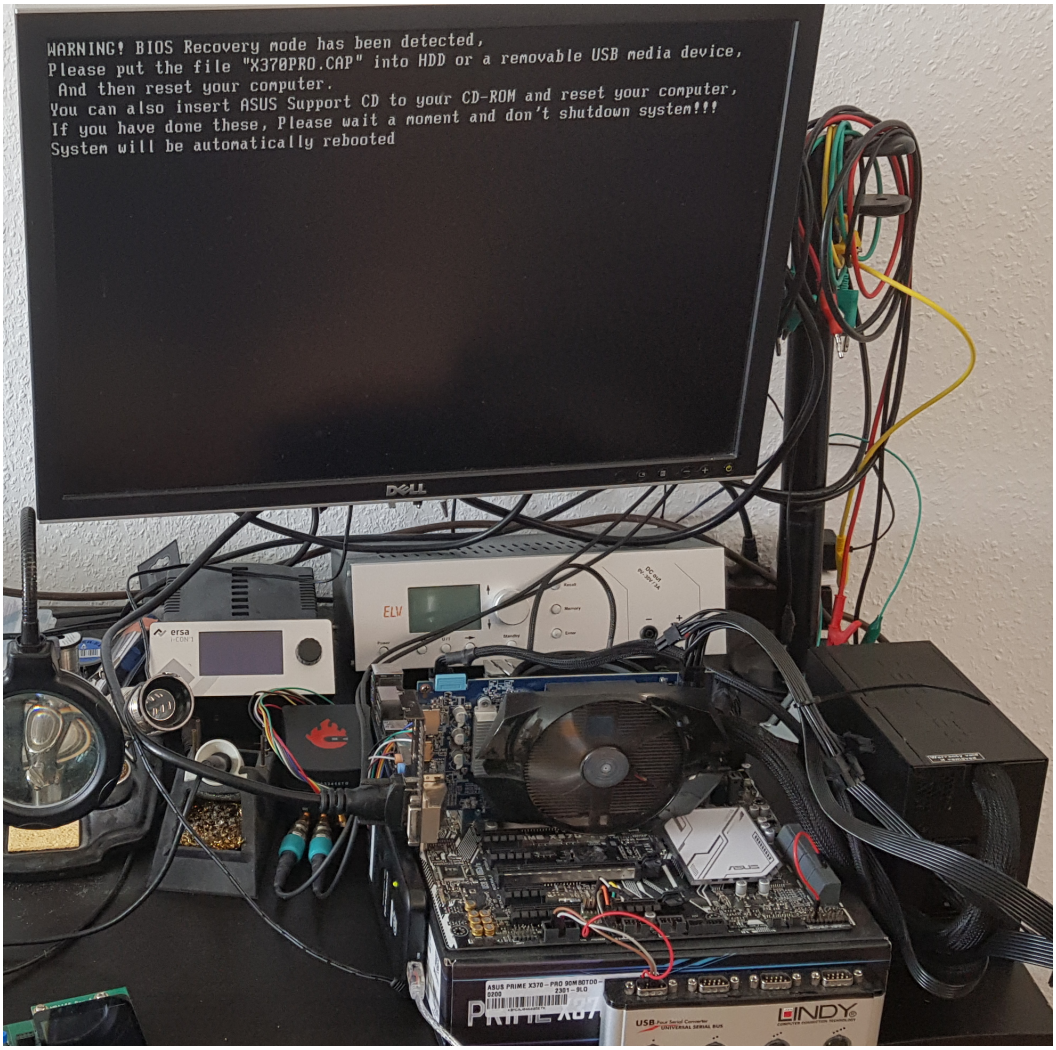

Fig. 5.5.: Run of the emulator initializing the real target system and releasing the x86 cores to get the UEFI firmware started to execute for the first time during this thesis.

5.5 Enabling the legacy UART

With the SPI communication channel working stable and fast, attention was switched to getting the serial port working in order to have an easily accessible and low cost solution available as shown in section 5.1.
The AMD SoC has integrated UARTs but these are not exposed on most of the consumer mainboards. The exposed serial port on the used mainboard is controlled by a so called SuperIO chip, which among other things handles legacy serial ports. Those SuperIO chips are attached via the Low Pin Count (LPC) bus to the SoC. The LPC bus was devised by Intel in the late 90s to replace the ISA bus, requiring to route far fewer signals on a PCB. The specification is freely available [@19]. SuperIO chips are microcontrollers on their own, requiring a special configuration sequence to enable and configure exposed devices. Most of these chips like the `ITE8665E` on the used mainboard don't have public documentation which complicates finding the correct configuration sequence. However, the UEFI firmware initializes the SuperIO chip, so figuring out the correct sequence is just a matter of sniffing the transactions on the LPC bus with a logic analyser, decoding them, looking for a configuration sequence, and replicating the transactions in the PSP stub. The LPC bus operates at 33.3 MHz which requires a fast logic analyser capable of capturing at least six signals with the required bandwidth. The Saleae Logic Pro 16 was chosen for this. Wiring had to be short to reduce possible interferences while the capture is running. The used mainboard exposed a direct connection to the LPC bus through the TPM header which saved the author from soldering wires onto the SuperIO chip itself. The resulting setup is shown in figure 5.6. The flash emulator is not necessary for capturing the LPC transactions but made switching between the official UEFI firmware image and the PSP stub much easier and faster.

Figure 5.7 shows a single transaction on the LPC bus captured with the logic analyser. To avoid having to decode those signals manually a tool called `lpc-dec`[13] was created. It takes a capture and produces a human readable output of all captured transactions on the LPC bus. Listing 5.1 shows what was later identified as the configuration sequence which enables the legacy UART.

[13] `https://github.com/AlexanderEichner/lpc-dec`

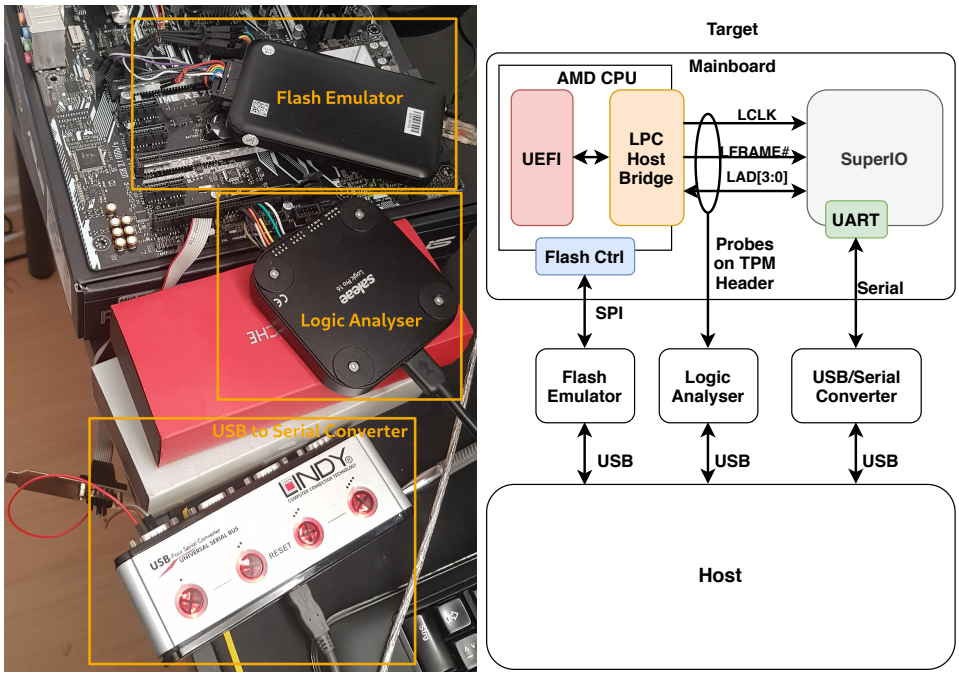

Fig. 5.6.: Setup for capturing the LPC transactions with the logic analyser being attached to the TPM header on the mainboard and the serial port being connected to a serial to USB converter.

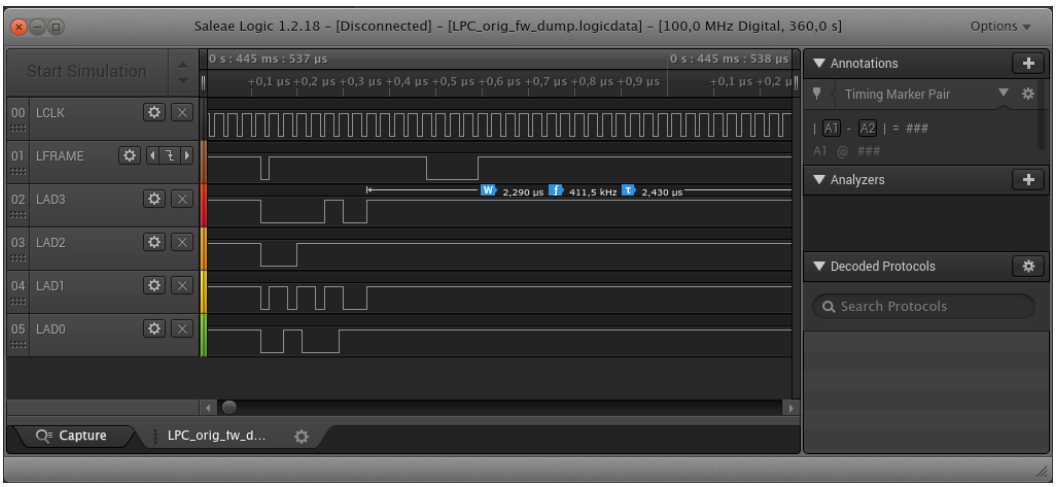

Fig. 5.7.: Single LPC transaction captured with a logic analyser

```
$ ./lpc-dec --input ../../LPC_UEFI.bin --verbose
[...]
155655809: I/O Write 0x002e: 0x87 WAIT_LFRAME_ASSERTED -> START -> ADDR -> DATA -> TAR -> SYNC -> TAR
155656010: I/O Write 0x002e: 0x01 WAIT_LFRAME_ASSERTED -> START -> ADDR -> DATA -> TAR -> SYNC -> TAR
155656196: I/O Write 0x002e: 0x55 WAIT_LFRAME_ASSERTED -> START -> ADDR -> DATA -> TAR -> SYNC -> TAR
155656712: I/O Write 0x002e: 0x55 WAIT_LFRAME_ASSERTED -> START -> ADDR -> DATA -> TAR -> SYNC -> TAR
155656901: I/O Write 0x002e: 0x23 WAIT_LFRAME_ASSERTED -> START -> ADDR -> DATA -> TAR -> SYNC -> TAR
155657093: I/O Read  0x002f: 0x40 WAIT_LFRAME_ASSERTED -> START -> ADDR -> TAR -> SYNC -> DATA -> TAR
155657297: I/O Write 0x002f: 0x40 WAIT_LFRAME_ASSERTED -> START -> ADDR -> DATA -> TAR -> SYNC -> TAR
155657486: I/O Write 0x002e: 0x07 WAIT_LFRAME_ASSERTED -> START -> ADDR -> DATA -> TAR -> SYNC -> TAR
155657678: I/O Write 0x002f: 0x01 WAIT_LFRAME_ASSERTED -> START -> ADDR -> DATA -> TAR -> SYNC -> TAR
155657876: I/O Write 0x002e: 0x61 WAIT_LFRAME_ASSERTED -> START -> ADDR -> DATA -> TAR -> SYNC -> TAR
155658068: I/O Write 0x002f: 0xf8 WAIT_LFRAME_ASSERTED -> START -> ADDR -> DATA -> TAR -> SYNC -> TAR
155658266: I/O Write 0x002e: 0x60 WAIT_LFRAME_ASSERTED -> START -> ADDR -> DATA -> TAR -> SYNC -> TAR
155662397: I/O Write 0x002f: 0x03 WAIT_LFRAME_ASSERTED -> START -> ADDR -> DATA -> TAR -> SYNC -> TAR
155662595: I/O Write 0x002e: 0x30 WAIT_LFRAME_ASSERTED -> START -> ADDR -> DATA -> TAR -> SYNC -> TAR
155662781: I/O Write 0x002f: 0x01 WAIT_LFRAME_ASSERTED -> START -> ADDR -> DATA -> TAR -> SYNC -> TAR
155662973: I/O Write 0x002e: 0x02 WAIT_LFRAME_ASSERTED -> START -> ADDR -> DATA -> TAR -> SYNC -> TAR
155663171: I/O Write 0x002f: 0x02 WAIT_LFRAME_ASSERTED -> START -> ADDR -> DATA -> TAR -> SYNC -> TAR
155663360: I/O Read  0x03fb: 0x00 WAIT_LFRAME_ASSERTED -> START -> ADDR -> TAR -> SYNC -> DATA -> TAR
[...]
```

Listing 5.1: Excerpt of the output of `lpc-dec` for a capture during the UEFI phase when initializing the SuperIO chips legacy UART, and showing the first read from the UARTs FIFO control register.

What the LPC capture did not show however, was that the LPC to host bridge embedded in the AMD SoC required some initialisation as well in order to pass I/O port accesses onto the LPC bus. The Processor Programming Reference manual contained some information about it and the remaining information required was taken from the `PSPEmu` trace log because the off-chip bootloader initialises the LPC to host bridge as well. The resulting initialisation sequence as implemented in the PSP stub is shown in listing 5.2. With the serial port finally working, a serial port backend was added to `libpspproxy`[14] to make that communication channel accessible to `PSPEmu` as well. Testing whether the firmware was able to bootstrap the system using the serial port inside the emulator turned out to be successful, although it took around 45 minutes compared to the 5 minutes to get to the same point when using the SPI flash emulator. This is caused by the lower bandwidth of the serial port compared to the SPI bus.

[14] `https://github.com/PSPReverse/libpspproxy/blob/master/psp-proxy-provider-serial.c`

```
static void pspStubX86MmioWriteU32(PPSPSTUBSTATE pThis, X86PADDR PhysX86Addr, uint32_t u32Val)
{
    volatile uint32_t *pu32 = NULL;
    int rc = pspStubX86PhysMap(pThis, PhysX86Addr, true /*fMmio*/, (void **)&pu32);
    if (STS_SUCCESS(rc))
    {
        *pu32 = u32Val;
        pspStubX86PhysUnmapByPtr(pThis, (void *)pu32);
    }
}

static void pspStubX86MmioWriteU8(PPSPSTUBSTATE pThis, X86PADDR PhysX86Addr, uint8_t bVal)
{
    volatile uint8_t *pb = NULL;
    int rc = pspStubX86PhysMap(pThis, PhysX86Addr, true /*fMmio*/, (void **)&pb);
    if (STS_SUCCESS(rc))
    {
        *pb = bVal;
        pspStubX86PhysUnmapByPtr(pThis, (void *)pb);
    }
}

static void pspStubSerialSuperIoInit(PPSPSTUBSTATE pThis)
{
    pspStubX86MmioWriteU32(pThis, 0xfffe000a3048, 0x0020ff00);
    pspStubX86MmioWriteU32(pThis, 0xfffe000a30d0, 0x08fdff86);
    pspStubX86MmioWriteU8(pThis, 0xfed81e77, 0x27);
    pspStubX86MmioWriteU32(pThis, 0xfec20040, 0x0);
    pspStubX86MmioWriteU32(pThis, 0xfffe000a3044, 0xc0);
    pspStubX86MmioWriteU32(pThis, 0xfffe000a3048, 0x20ff07);
    pspStubX86MmioWriteU32(pThis, 0xfffe000a3064, 0x1640);
    pspStubX86MmioWriteU32(pThis, 0xfffe000a3000, 0xffffff00);
    pspStubX86MmioWriteU32(pThis, 0xfffe000a30a0, 0xfec10002);
    pspStubX86MmioWriteU32(pThis, 0xfed80300,     0xe3020b11);
    pspStubX86MmioWriteU8(pThis, 0xfffdfc000072, 0x6);
    pspStubX86MmioWriteU8(pThis, 0xfffdfc000072, 0x7);
    pspStubSmnWrU32(pThis, 0x2dc58d0, 0x0c7c17cf);
    pspStubX86MmioWriteU32(pThis, 0xfffe000a3044, 0xc0);
    pspStubX86MmioWriteU32(pThis, 0xfffe000a3048, 0x20ff07);
    pspStubX86MmioWriteU32(pThis, 0xfffe000a3064, 0x1640);
    pspStubX86MmioWriteU8(pThis, 0xfffdfc00002e, 0x87);
    pspStubX86MmioWriteU8(pThis, 0xfffdfc00002e, 0x01);
    pspStubX86MmioWriteU8(pThis, 0xfffdfc00002e, 0x55);
    pspStubX86MmioWriteU8(pThis, 0xfffdfc00002e, 0x55);
    pspStubX86MmioWriteU8(pThis, 0xfffdfc00002e, 0x07);
    pspStubX86MmioWriteU8(pThis, 0xfffdfc00002f, 0x07);
    pspStubX86MmioWriteU8(pThis, 0xfffdfc00002e, 0x24);
    pspStubX86MmioWriteU8(pThis, 0xfffdfc00002f, 0x00);
    pspStubX86MmioWriteU8(pThis, 0xfffdfc00002f, 0x10);
    pspStubX86MmioWriteU8(pThis, 0xfffdfc00002e, 0x02);
    pspStubX86MmioWriteU8(pThis, 0xfffdfc00002f, 0x02);
    pspStubX86MmioWriteU8(pThis, 0xfffdfc00002e, 0x87);
    pspStubX86MmioWriteU8(pThis, 0xfffdfc00002e, 0x01);
    pspStubX86MmioWriteU8(pThis, 0xfffdfc00002e, 0x55);
    pspStubX86MmioWriteU8(pThis, 0xfffdfc00002e, 0x55);
    pspStubX86MmioWriteU8(pThis, 0xfffdfc00002e, 0x23);
    pspStubX86MmioWriteU8(pThis, 0xfffdfc00002f, 0x40);
    pspStubX86MmioWriteU8(pThis, 0xfffdfc00002f, 0x40);
    pspStubX86MmioWriteU8(pThis, 0xfffdfc00002e, 0x07);
    pspStubX86MmioWriteU8(pThis, 0xfffdfc00002f, 0x01);
    pspStubX86MmioWriteU8(pThis, 0xfffdfc00002e, 0x61);
    pspStubX86MmioWriteU8(pThis, 0xfffdfc00002f, 0xf8);
    pspStubX86MmioWriteU8(pThis, 0xfffdfc00002e, 0x60);
    pspStubX86MmioWriteU8(pThis, 0xfffdfc00002f, 0x03);
    pspStubX86MmioWriteU8(pThis, 0xfffdfc00002e, 0x30);
    pspStubX86MmioWriteU8(pThis, 0xfffdfc00002f, 0x01);
    pspStubX86MmioWriteU8(pThis, 0xfffdfc00002e, 0x02);
    pspStubX86MmioWriteU8(pThis, 0xfffdfc00002f, 0x02);
}
```

Listing 5.2: LPC host bridge and SuperIO initialisation sequence as implemented in the PSP stub.

5.6 CCP pass-through

The last issue remaining was the off-chip bootloader going into recovery mode and not continuing to load and execute the SecureOS. Investigation with the emulator and GDB showed that the off-chip bootloader tries to decrypt an entry dubbed WRAPPED_IKEK by `PSPTool` as shown in listing 5.3.

The entry contains an encryption key for other components and is itself encrypted. Decrypting the key is only possible when using a real CCP as it contains the required key in special local memory which can't be read to protect against key extraction. The request is shown in listing 5.4. To use that key for decryption the request has to use the special identifier of the memory buffer to indicate that this particular key should be used. Overcoming that issue required extending proxy mode in order to be able to pass through certain CCP requests to the real CCP by using the stub. Just passing through the MMIO accesses to the CCP registers in the target is not possible, as the CCP has direct access to the local SRAM of the PSP. Passing through any operation to the real CCP requires copying input buffers to the real PSP before the operation, rewriting the request descriptors and copying output buffers back, which would increase the overhead and require a special CCP emulation. Most operations like zlib decompression or memory copy operations don't need to be executed on the real PSP anyway and can be more efficiently done inside the emulator with the CCP device emulation. The stub itself didn't require any modifications as the primitives to access the local PSP's SRAM and MMIO range were already available. The implementation can be found in the proxy code[15], in particular in the functions `pspEmuProxyCcpAesDo()` and `pspEmuProxyCcpReqExec()`. The CCP emulation was modified to use the pass-through function offered by proxy mode if access to the special CCP key buffer was detected and proxy mode was configured as shown in listing 5.5. If proxy mode is not available an error is printed to the trace log to make detection easier.

```
$ psptool PRIME-X370-PRO-ASUS-3803.ROM
[...]
+---+-------+----------+---------+--------------------------------+----------+-------------+[...]
|   | Entry |  Address |    Size |                           Type | Magic/ID |     Version |[...]
+---+-------+----------+---------+--------------------------------+----------+-------------+[...]
|   |     0 |  0xd1400 |   0x240 |             AMD_PUBLIC_KEY-0x0 |     1BB9 |             |[...]
|   |     1 | 0x361400 |  0xd280 |         PSP_FW_BOOT_LOADER-0x1 |     $PS1 |    0.9.0.53 |[...]
[...]
|   |     6 | 0x114300 |    0x10 |              WRAPPED_IKEK-0x21 |          |             |[...]
[...]
```

Listing 5.3: `PSPTool` showing the wrapped IKEK entry in the UEFI firmware image.

[15] `https://github.com/PSPReverse/PSPEmu/blob/master/psp-proxy.c`

```
CCP Request 0x0000d280:
    u32Dw0:             0x00000019 (Engine: AES, AES Type: AES128, Mode: ECB, Encrypt: 0, Size: 0)
    cbSrc:              16
    u32AddrSrcLow:      0x0003fa00
    u16AddrSrcHigh:     0x00000000
    u16SrcMemType:      0x00000002 (MemType: 2, LsbCtxId: 0, Fixed: 0)
    u32AddrDstLow:      0x0003f890
    u16AddrDstHigh:     0x00000000
    u16DstMemType:      0x00000002 (MemType: 2, Fixed: 0)"
    u32AddrKeyLow:      0x00000080
    u16AddrKeyHigh:     0x00000000
    u16KeyMemType:      0x00000001
```

Listing 5.4: The passed through AES decryption request for unwrapping the IKEK.

```
[...]
    /* If the request uses a protected LSB and CCP passthrough is available we use the real CCP. */
    if (   CCP_V5_MEM_TYPE_GET(pReq->u16KeyMemType) == CCP_V5_MEM_TYPE_SB
        && CCP_ADDR_CREATE_FROM_HI_LO(pReq->u16AddrKeyHigh, pReq->u32AddrKeyLow) < 0xa0)
    {
        if (pThis->pDev->pCfg->pCcpProxyIf)
            return pspDevCcpReqAesPassthrough(pThis, pReq, uMode == CCP_V5_ENGINE_AES_MODE_CBC ? true :
                false /*fUseIv*/);
        else /* No key in the protected LSB means that the output is useless, leave an error. */
            PSPEmuTraceEvtAddString(NULL, PSPTRACEEVTSEVERITY_FATAL_ERROR, PSPTRACEEVTORIGIN_CCP,
                                    "CCP: Request accesses protected LSB for which there is no key set,
                                        decrypted output is useless and the emulation will fail\n");
    }
[...]
```

Listing 5.5: Changes to the CCP emulation to pass through certain AES requests accessing a protected key buffer.

With this, the off-chip bootloader successfully decrypted the IKEK and continued loading and executing the SecureOS.

5.7 SecureOS emulation

The general design of SecureOS is already explained in 3.2.2. However, most of the details are unknown at this point as it is a completely different from the off-chip bootloader running on the PSP requiring further analysis which is out of scope for this thesis. Nevertheless, an attempt was made to get SecureOS working inside the emulator. SecureOS requires a working implementation of the ARM TrustZone extension, which the modified variant of QEMU in unicorn is not able to do correctly. Some work was invested into the `psp-core` component to work around that. The resulting changes make it possible for SecureOS to switch between the secure and non-secure world and execute code, as well as exchanging information as shown in listing 5.6. At last it was attempted to get interrupts working inside the emulator by implementing the interrupt controller register interface and hooking devices like the timers and the CCP up to it. However, the UEFI firmware still wouldn't continue

further and the target system just showed a black screen now. Understanding SecureOS and getting it to work in the emulator is a topic for future research.

```
1 [...]
2 00739134          WARNING              MMIO 0x01f0437c[0x01f02be5][  SVC, S, M,NI,NF,0x00009400] DEV WRITE
      <PROXY>                                    0x301020c 4 0x00000000
3 00739135             INFO               SMC 0x01f08630[0x01f0860c][  SVC, S, M,NI,NF,0x00009400] SMC ENTRY
      0 0x00000001 00000000 0x01f097ec 00000000 LR=00000000
4 00739136             INFO               SMC 0x00035264[0x00034e81][  SVC,NS,NM,NI, F,0x0003c000] SMC ENTRY
      0 0x81000008 0x0000000b 00000000 00000000 LR=00000000
5 00739137             INFO               SMC 0x01f001fc[0x00035264][  IRQ, S, M,NI,NF,0x00009400] SMC ENTRY
      0 0x81000008 0x01f09908 0x00000007 00000000 LR=00000000
6 00739138             INFO               SMC 0x00035264[0x00034e2f][  SVC,NS,NM,NI, F,0x0003c000] SMC ENTRY
      0 0x81000002 0x0000fffd 0xfb000000 0x00118000 LR=00000000
7 [...]
```

Listing 5.6: Excerpt from a trace log showing a transition between secure and non-secure world in the SecureOS.

5.8 I/O log creation and replay

During analysis of the proprietary firmware it might be required to recreate a previous run of the firmware without having access to the hardware. Also, researchers might not have access to AMD hardware at all but still want to analyse the PSP firmware. For those use cases the emulator offers creation of a special log file containing all I/O transactions done in proxy mode. The log can be used with the emulator to pass the correct value for a read access for which there is no device emulation available without requiring proxy mode. This allows the firmware to progress to the same point as if proxy mode would have been used. Replaying an I/O log is also much faster than using proxy mode. The only requirement for this is using the exact same firmware version for replay as for creation of the I/O log at the moment. This is due to the replay functionality just looking for the next matching I/O access from the current point in the log and returning the value. Any non-matching accesses are skipped. Write operations for example are completely ignored in the current implementation. A different firmware version might change the order of I/O accesses resulting in undefined behaviour because of that.

The functionality is accompanied by a small helper utility called `psp-iolog-tool`[16] which can either just dump the content of a given I/O log in a human readable form as shown in listing 5.7 or read the entire log and create a map of all accessed peripherals divided into SMN, x86 and standard MMIO regions. Listing 5.8 shows an example line for a captured I/O log when using the register map mode. The first column gives the address of the register/memory location, in this example for an

[16] `https://github.com/PSPReverse/PSPEmu/blob/master/psp-iolog-tool.c`

SMN address. After that comes the size of the region in the second column which is 4 bytes. Then come some statistics about the number of read and write accesses and some information about the last read and write access, like the program counter value at which the access happened. In the example only one write occurred from PC `0x151a0` and it was I/O access number 8204 in the log.

This mode is useful to get a quick overview over all accessed addresses for each address space and how frequently individual addresses are accessed allowing comparison between different firmware versions.

```
$ ./psp-iolog-tool --iolog-input ../../ryzen_1800X.iolog --mode dump
CCD       PC       Address space       Direction       Address    Size     Value
00     0x000051e6     PSP/MMIO            READ         0x3010104    4     0x1a060900
00     0x000028ac     PSP/MMIO            WRITE        0x3006000    4     0x00000001
[...]
```

Listing 5.7: Example output of `psp-iolog-tool` from a given I/O log dumping the content in human readable form.

```
$ ./psp-iolog-tool --iolog-input ../../ryzen_1800X.iolog --mode reg-map
SMN:
0x0001c064  4  READS: 0  WRITES: 1  LWPC: 0x000151a0  LWIOID: 000000008204  LRPC: 0x00000000  LRIOID: 000000000000
[...]
```

Listing 5.8: Example output of `psp-iolog-tool` from a given I/O log when using the register map mode

6 UEFI firmware emulation

After the emulator was able to successfully execute all ABL stages in proxy mode and the off-chip bootloader releases the x86 cores from reset the x86 UEFI starts executing. The UEFI firmware starts bootstrapping the x86 side of the SoC and in that process communicates with the off-chip bootloader using a mailbox register interface. The off-chip bootloader provides requests to set the SMM region [@29] for the x86 UEFI for example. When the target system was bootstrapped using the PSP emulator in proxy mode the UEFI firmware would hang at a certain point resulting in the system not being able to continue to boot. Trying to find the cause for the hang by checking what the SecureOS, started by the off-chip bootloader, was doing turned out to be unsuccessful. A different approach was required and the intuitive solution was to do the same with the UEFI firmware as for the PSP firmware components, namely to execute the UEFI firmware on a different system by using emulation and to forward hardware accesses. The resulting architecture of the solution can be seen in figure 6.1.

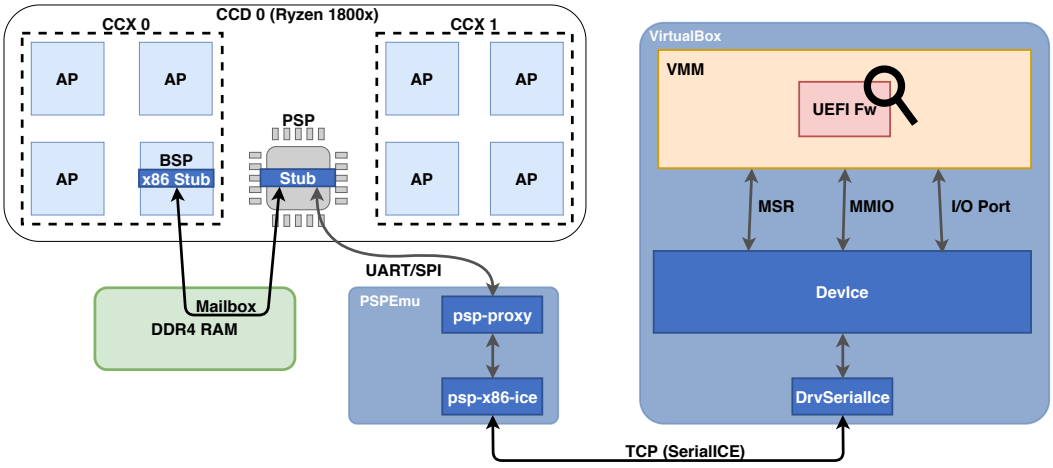

Fig. 6.1.: Architectural block diagram of the components involved in the x86 UEFI firmware emulation

Instead of creating an x86 system emulator with unicorn, which would have been a tremendous effort, VirtualBox was chosen as a widely used x86 virtualisation solution. VirtualBox was modified to include a special device emulation[1] to intercept

[1] https://github.com/PSPReverse/vbox-ice

accesses to I/O ports, MMIO addresses and MSR ranges, forwarding them to the PSP emulator using the SerialICE protocol. The PSP emulator gained a new component called `psp-x86-ice` which receives requests from VirtualBox and forwards them using the proxy mode component to the target. VirtualBox and the PSP emulator communicate over a standard TCP connection to convey I/O access requests and responses. SerialICE[2] was chosen as the protocol. It was created by the CoreBoot project to allow prototyping and debugging x86 firmware inside a specially patched version of QEMU, forwarding hardware accesses to real hardware just like the PSP emulator is doing in proxy mode. Using SerialICE instead of designing a different protocol allows others to use different tools instead of VirtualBox when working on x86 firmware or use VirtualBox with something other than the PSP emulator in the future.

In the first iteration of the implementation the stub running on the PSP would do the I/O port and MMIO accesses. The drawback was that MSR accesses couldn't be forwarded that way because the PSP has no access to the x86 CPU MSRs. And as it turned out, certain I/O port and MMIO ranges are not accessible by the PSP either but result in a data abort exception. To overcome these issues, a second stub[3] was implemented destined to run on the x86 bootstrap processor (BSP) where the UEFI firmware would run usually. The x86 stub is tailored towards AMD CPUs and leverages the fact that DRAM is fully initialised when it starts executing. This made the stub easy to implement because it only has to set up a global descriptor table (GDT) and to switch to unpaged 32-bit protected mode in order to access all MMIO regions. After initial setup the stub will then continuously poll a fixed location in DDR4 DRAM for a certain magic indicating that a new request has arrived from the PSP stub. Upon reading the magic the stub parses the request, executes it and returns the result. This allows forwarding accesses to both the PSP and x86 BSP over a single connection. Loading the x86 stub is done by the PSP emulator when the off-chip bootloader inside the emulator tries to release the x86 cores from reset. VirtualBox itself contains a powerful debugger[4] and logging infrastructure so that the firmware can be analysed. VirtualBox recently gained a GDB stub implementation[5] based on `libgdbstub` which will be part of the next major release.

[2] `https://www.serialice.com/Main_Page`
[3] `https://github.com/PSPReverse/x86-stub`
[4] `https://www.virtualbox.org/browser/vbox/trunk/src/VBox/Debugger/DBGCEmulateCodeView.cpp`
[5] `https://www.virtualbox.org/browser/vbox/trunk/src/VBox/Debugger/DBGCGdbRemoteStub.cpp`

6.1 Results

For testing the implemented solution, the same setup as described in section 5.2 was used. First, the PSP emulator was started to run in proxy mode doing the early system initialisation. When the point was reached where the off-chip bootloader would release the x86 cores it would copy over the stub instead and release the real x86 cores afterwards. The emulator would then halt in the GDB debugger at the point where control is handed over to the SecureOS. Then, the VirtualBox VM was started, which was configured to use the UEFI firmware image. The UEFI firmware was able to do very early x86 initialisation inside the VM, for instance switching to protected mode. The PSP emulator trace logs also showed that the UEFI can read/write I/O registers and access MSRs on the real target. However, it would then hang in a loop polling a single I/O port without continuing further. The cause for this hang is completely unknown so far. The following incomplete list gives an overview of possible causes:

- The UEFI firmware waits for an interrupt which doesn't occur inside the virtualized environment because interrupt forwarding from the x86 BSP to VirtualBox is not implemented.
- Some mismatch in the CPU profile presented in the virtual environment to the firmware from the real CPU causes the firmware to execute differently. The CPU profile in VirtualBox presented to the guest does not have to match the host CPU in order to allow executing the firmware on a host system with a completely different CPU, like an Intel one for instance.
- Different timings in the virtual environment could causes the firmware to execute differently. A bug in the firmware might get triggered due to changed timings.

Unfortunately the problem couldn't be investigated further due to time constraints. Debugging should be possible in a future attempt with the basic infrastructure in place and working.

7 Evaluation

To evaluate the validity of the approach, three of the main goals presented in chapter 4 were picked to show the capabilities of the implemented emulator.

7.1 Executing firmware in a fully virtual environment

The first goal to be evaluated is running the PSP firmware inside a fully virtual environment. Listing 7.1 shows the output produced. While the firmware fails with an error in the first ABL user mode stage it shows that the off-chip bootloader is executing and can load code modules from flash and execute them in user mode. The error happens because a lot of the device emulations are still missing which can be worked around with either proxy mode presented in the previous chapter or by using an I/O log of a previous run. Listing 7.2 shows the beginning of the output of the firmware from such a run. ABL stage 1 completes successfully and ABL stage 6 is getting executed. This proves the usefulness of the fully virtual environment.

```
1  $./PSPEmu --emulation-mode sys --flash-rom ../../binaries/Ryzen/Asus/PRIME-X370-PRO/PRIME-X370-PRO-ASUS
       -3803.ROM --cpu-profile ryzen7-1800x --trace-log ./log --timer-real-time --intercept-svc-6 --trace-
       svcs
2  [...]
3   *SEC_DBG_MODULE* Entering DebugUnlock module
4   *SEC_DBG_MODULE* Exiting DebugUnlock module
5   *SEC_DBG_MODULE* Entering DebugUnlock module
6   *SEC_DBG_MODULE* Exiting DebugUnlock module
7  SMU: Executing request 0x1 with argument 0x3
8  SMU: Executing request 0x4 with argument 0x1010000
9  ABL0 - Main ABL Execution
10   AGESA MEM - Initial APCB Support
11   AGESA MEM - Initial APCB Support
12 Calling ABL 1 BL
13 ABL1 Loaded
14 Slave: ABL, Init Slave states
15 Cannot find the APCB BoardID Data: APCB_PSP_TYPE_BOARD_ID_GETTING_METHOD
16   AGESA MEM - Initial APCB Support
17 SPD Data Sync
18 ABL 1 - AGESA FATAL/ERROR detected
19 Returned from ABL 1 BL
20 All ABLs Complete (pass control back to PSP BL
```

Listing 7.1: The output of the PSP firmware when executed in a fully virtual environment with the emulator.

```
./PSPEmu --emulation-mode sys --cpu-profile ryzen7-1800x --flash-rom ../../binaries/Ryzen/Asus/PRIME-X370-
    PRO/PRIME-X370-PRO-ASUS-3803.ROM --trace-log run.tracelog --timer-real-time --emulate-devices ccp-v5:
    flash:timer2 --iom-log-all-accesses --trace-svcs --intercept-svc-6 --io-log-replay ./proxy-mode.iolog
PSP I/O log: Opening ./proxy-mode.iolog
PSP I/O log: Opened ./proxy-mode.iolog
[...]
 *SEC_DBG_MODULE* Entering DebugUnlock module
 *SEC_DBG_MODULE* Exiting DebugUnlock module
 *SEC_DBG_MODULE* Entering DebugUnlock module
 *SEC_DBG_MODULE* Exiting DebugUnlock module
Entering security gasket programming application
ABL0 - Main ABL Execution
  AGESA MEM - Initial APCB Support
  AGESA MEM - Initial APCB Support
Calling ABL 1 BL
ABL1 Loaded
Slave: ABL, Init Slave states
Cannot find the APCB BoardID Data: APCB_PSP_TYPE_BOARD_ID_GETTING_METHOD
  AGESA MEM - Initial APCB Support
SPD Data Sync
Debug Sync
Debug Sync Disabled - Die count <= 1
Returned from ABL 1 BL
Calling ABL 6 BL
Starting AGESA 6 BL
Slave: ABL, Init Slave states
Restore
Locate
[...]
```

Listing 7.2: Beginning of the output of the PSP firmware when executed in a fully virtual environment with the emulator and replaying a given I/O log.

7.2 Proxy mode and multiple firmware versions

The second main goal of the emulator is to make analysis of different firmware versions easier for further research. After the emulator was working with a single UEFI firmware image in proxy mode, other images were obtained from the vendors website and tested as well. The setup for testing is the same as described in 5.2 and the generated trace log was observed to determine how far those firmware images were able to get. The tested images are given in 7.1 along with the versions of the included off-chip bootloader and ABL stages as reported by PSPTool.

Filename	Off-Chip BL version	ABL version
PRIME-X370-PRO-ASUS-0502.ROM	0.5.0.30	17.1.12.0
PRIME-X370-PRO-ASUS-0810.ROM	0.5.0.35	17.5.15.1
PRIME-X370-PRO-ASUS-3401.ROM	0.5.0.35	17.9.20.30
PRIME-X370-PRO-ASUS-3803.ROM	0.5.0.38	17.12.11.30

Tab. 7.1.: Tested UEFI firmware images with the contained off-chip bootloader and ABL stage versions as reported by PSPTool.

All tested images were able to reach the final stage were the off-chip bootloader would load the secure OS and jump to it. This means all tested firmware versions were able to fully run the ABL stages successfully and bootstrap the system, initializing the DDR4-DRAM and releasing the x86 cores.

This shows that the emulator is capable to emulate different versions of the firmware stack in proxy mode for Zen CPUs if the CPU contains only a single PSP which excludes Threadripper and EPYC CPUs for now. Zen2 couldn't be tested because, although code execution on the PSP of those CPUs is possible, it requires substantial changes to the stub code which was not possible to implement as part of this thesis due to time constraints. Zen+ wasn't tested but as it is only a minor architectural change it should work with only minor modifications as well.

As stated in section 5.5 there are two communication channels available for proxy mode. While SPI is fast and allows initializing the system in about 5 minutes it requires an expensive flash emulator. The serial port on the other side is easily accessible on the mainboard and doesn't require expensive development tools but just the ability to flash the embedded flash chip with the custom flash image. However, the serial port is much slower, requiring about 45 minutes to reach the same point compared to SPI. Furthermore, the initialisation sequence of the SuperIO chip depends on the exact model present on the mainboard and will need modifications if another model is being used.

7.3 Debugging code targeted for the PSP with the emulator

The last evaluated goal is the ability of the emulator to provide an easy to use environment for researchers to debug code targeted for the PSP before it gets executed on real hardware. As an example, exploitation of a bug in the on-chip bootloader is shown which was presented by Robert Buhren and the author at 2020 BlackHat USA [@10]. Modifying the on flash filesystem in a certain way can cause the on-chip bootloader to issue a request to the CCP with a large size overflowing the buffer and overwriting the on-chip bootloader's stack. This can be used to overwrite the return address of the currently executed function making it possible to inject arbitrary code into the PSP. Figure 7.1 shows the use of the emulator with GDB attached and listing 7.3 shows the CCP request in the trace log causing the buffer overflow triggering the exploit. The flash image used with the emulator is used unmodified to inject the PSP stub into a real target for proxy mode. Parts of

the created PSP stub and the shown exploit were developed using the debugging capabilities of the emulator, proving that it is a capable tool for researchers when developing code targeted for the PSP.

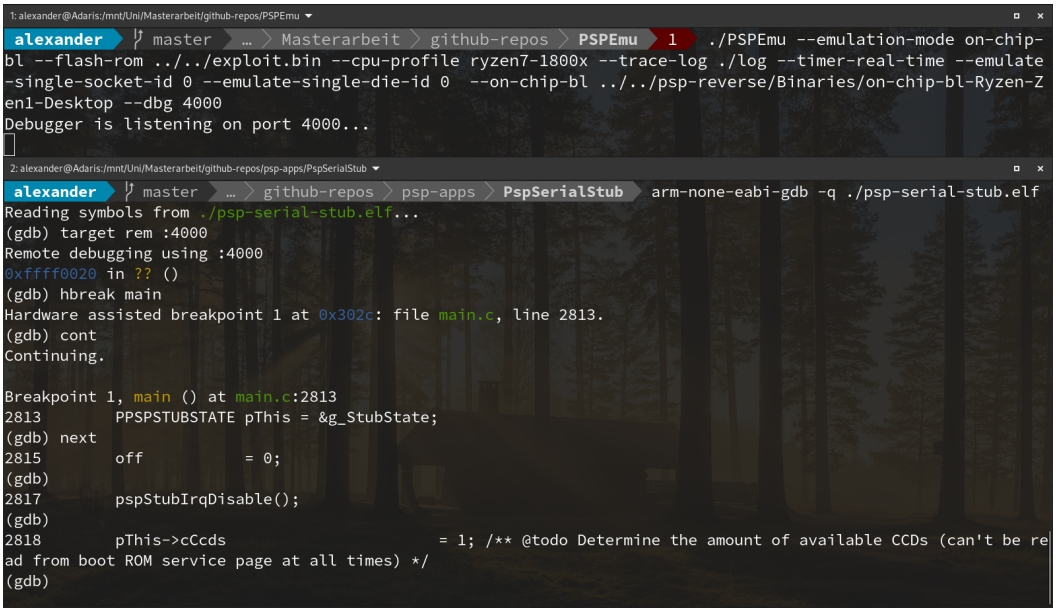

Fig. 7.1.: Source level debugging of code written for the PSP inside the emulator.

```
1  STRING "CCP Request 0x0003f900:"
2  STRING "    u32Dw0:             0x00500011 (Engine: PASSTHROUGH, ByteSwap: NOOP, Bitwise: NOOP, Reflect:
       0)"
3  STRING "    cbSrc:              2147537024"
4  STRING "    u32AddrSrcLow:      0x02361500"
5  STRING "    u16AddrSrcHigh:     0x00000000"
6  STRING "    u16SrcMemType:      0x00000006 (MemType: 2, LsbCtxId: 1, Fixed: 0)"
7  STRING "    u32AddrDstLow:      0x00000100"
8  STRING "    u16AddrDstHigh:     0x00000000"
9  STRING "    u16DstMemType:      0x00000002 (MemType: 2, Fixed: 0)"
10 STRING "    u32AddrKeyLow:      0x00000000"
11 STRING "    u16AddrKeyHigh:     0x00000000"
12 STRING "    u16KeyMemType:      0x00000000"
```

Listing 7.3: The CCP request overflowing the on-chip bootloader's stack, `cbSrc` containing a large value.

The debugging component and modular architecture of the emulator makes it easy to extend it with additional functionality. For example, others used Unicorn in conjunction with AFL[1] for fuzzing [20]. Similar functionality can be added to the PSP emulator as well, in order to allow fuzzing certain sub components of the proprietary PSP firmware by emulating externally accessible devices like the flash or data stored in the SPD EEPROM of DRAM modules. The fuzzer can create corrupted data to test the parsers contained in the PSP firmware.

[1] `https://lcamtuf.coredump.cx/afl/`

8 Conclusion

The PSP emulator designed and implemented in this thesis provides a powerful and feature rich platform for further analysis of AMD's proprietary PSP firmware components. In summary, the following features were implemented to reach the defined functionality as described in chapter 4:

- A full system ARMv7 emulation core based on a modified version of unicorn.
- Support for Zen and Zen+ PSP variants, and to some extent Zen 2.
- Basic emulations for devices like flash and timers, enabling the capability to analyse multiple firmware versions in a completely virtual environment.
- An advanced emulation of the Cryptographic Co-Processor (CCP).
- GDB interface for debugging and analysing the firmware components offering full source code level debugging for own code targeted for the real PSP.
- Tracing infrastructure to log key events during firmware execution like I/O accesses.
- Generating code coverage traces and exporting them in a DrCov compatible format, allowing to overlay the execution path in static analysis tools like Ghidra.
- Proxy mode, allowing to bootstrap a real target system in a bearable amount of time without requiring to understand and emulate every aspect of the hardware.

Furthermore, the following features not part of the initially defined feature set were implemented during this thesis:

- Record and playback I/O logs for a particular firmware enabling others without hardware access to analyse the firmware by just providing the I/O log.
- A basic framework to execute own code on a real PSP.
- The basic framework to run the x86 UEFI firmware inside a VirtualBox VM and pass-through hardware accesses to the x86 side of the AMD SoC.

`libgdbstub`, which was created as a standalone library for the GDB debug support, and other implemented components can also be incorporated into projects unrelated to the PSP emulator. The LPC bus decoder, created while working on the emulator, might also prove valuable for other research projects. The code is publicly available on GitHub in the repositories mentioned in appendix B.

9 Future Work

While the basic PSP emulator is complete and working, there is still a lot to be done, with the current state of the emulator providing a solid foundation. The author hopes that the emulator will be as easy and pleasant to use and extensible by others, as it was a pleasure creating it in the first place. The following sections give a non conclusive overview of possible areas for improvement.

9.1 Extend functionality of the PSP emulator

The most important feature lacking right now is the ability to fully emulate the SecureOS in conjunction with proxy mode. This prevents fully bootstrapping a real target system from the emulator. Getting SecureOS working requires understanding the internals and fixing any incompatibilities with the emulator like interrupt handling for instance. Another desired functionality missing currently is integration with a fuzzing engine in order to fuzz subsystems like the syscall interface of the privileged off-chip bootloader, parsers for the flash filesystem, or the APCB included on the flash by the mainboard vendor as described in [@22].

9.2 Implement support for other communication channels

Proxy mode supports only two communication channels currently. The first one uses the SPI bus the flash chip on the mainboard is attached to. While this channel is fast it requires an expensive flash emulator and a mainboard with a pin header exposing the SPI bus. Otherwise soldering is required when used with a mainboard without such a header. The second channel uses the standard serial port which is easily accessible if the mainboard exposes a serial port. However, it is very slow. A third option could be implemented where a FPGA is hooked up directly to the LPC bus responding to accesses to otherwise unused I/O ports in order to implement the communication with the stub over those I/O ports. On the host side the FPGA

would present itself as a USB serial device to make use of it without requiring any changes to the emulator or `libpspproxy`. Communication using the LPC bus directly instead of using the serial port should be faster because the LPC bus operates at 33.3 MHz compared to the 115.2 kHz maximum frequency of a standard serial port. The most basic FPGA development kits start at around 40€[1], which is much cheaper than the used flash emulator[2]. More powerful FPGA development kits start at around 100€[3]. The FPGA can be connected directly to the LPC bus through the TPM header available on many mainboards, making it easy to revert the setup. The only modification required for the mainboard is overwriting the embedded flash with an image containing the exploit and PSP stub developed in this thesis. This is required for the serial port access method as well when the flash emulator is absent.

9.3 Support other AMD CPUs

So far only the Ryzen7 1800X was tested because it is a Zen CPU containing only a single CCD. In order to support other models such as EPYC and Threadripper in the emulator the ability to emulate multiple CCDs needs to be implemented, as well as the communication between all CCDs. The Zen+ micro architecture should only require minimal fixes but needs testing with proxy mode. Zen2 support is incomplete and the recently released Zen3 micro architecture was not looked into this thesis as well.

Furthermore, the focus was solely on the consumer x86 CPU based PSP variants. However, the PSP can also be found on recent AMD GPUs and is also present on the semi custom SoC designs of the newly released gaming consoles Playstation PS5 [@26] and XBox Series X [@17] which both rely on AMD's Zen2 architecture. It is not known whether and to what extend the security measures implemented in those consoles rely on the PSP. With AMD finally providing extremely competitive CPU designs, gaining a significant market share is only a matter of time. And with more security features being dependent on the PSP, like Secure Encrypted Virtual Machine (SEV) for example, being able to asses the security of new proprietary firmware versions by independent researchers more easily will be invaluable. Data centre providers and cloud customers don't have to rely on AMD's promises only.

[1] `https://www.digikey.de/product-detail/en/lattice-semiconductor-corporation/ICE40HX1K-STICK-EVN/220-2656-ND/4289604`
[2] `https://thelabeshop.com/products/dediprog-spi-flash-emulator`
[3] `https://store.digilentinc.com/arty-s7-spartan-7-fpga-board-for-hobbyists-and-makers/`

Bibliography

[1] Inc. Advanced Micro Devices. “Delivering the Future of High-Performance Computing with System, Software and Silicon Co-Optimization”. Hot Chips: A Symposium on High Performance Chips. 2019 (cit. on p. 7).

[2] Inc. Advanced Micro Devices. “Zen2”. Hot Chips: A Symposium on High Performance Chips. 2019 (cit. on p. 7).

[11] Robert Buhren, Christian Werling, and Jean-Pierre Seifert. “Insecure Until Proven Updated: Analyzing AMD SEV’s Remote Attestation”. In: (Aug. 2019) (cit. on pp. 2, 5, 27, 31).

[18] Felicitas Hetzelt and Robert Buhren. “Security Analysis of Encrypted Virtual Machines.” In: *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (2017) (cit. on p. 5).

[20] Dominik Maier, Benedikt Radtke, and Bastian Harren. “Unicorefuzz: On the Viability of Emulation for Kernelspace Fuzzing”. In: *13th USENIX Workshop on Offensive Technologies (WOOT 19)*. Santa Clara, CA: USENIX Association, Aug. 2019 (cit. on p. 52).

[21] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. “Avatar^2: A Multi-target Orchestration Platform”. In: *Workshop on Binary Analysis Research, San Diego, California* (2018) (cit. on p. 5).

Webpages

[@3] Advanced Micro Devices, Inc. *AMD Cryptographic Coprocessor (CCP) driver*. URL: `https://elixir.bootlin.com/linux/latest/source/drivers/crypto/ccp/ccp-dev.h` (visited on Nov. 5, 2020) (cit. on p. 27).

[@4] Advanced Micro Devices, Inc. *AMD PRO Security*. URL: `https://www.amd.com/en/technologies/pro-security` (visited on Nov. 26, 2020) (cit. on p. 1).

[@5] Advanced Micro Devices, Inc. *BIOS and Kernel Developer’s Guide (BKDG) for AMD Family 16h Models 30h-3Fh Processors*. URL: `https://www.amd.com/system/files/TechDocs/52740_16h_Models_30h-3Fh_BKDG.pdf` (visited on Nov. 26, 2020) (cit. on p. 1).

[@6] Advanced Micro Devices, Inc. *Developer Guides, Manuals & ISA Documents*. URL: `https://developer.amd.com/resources/developer-guides-manuals/` (visited on Nov. 12, 2020) (cit. on p. 35).

[@7] Advanced Micro Devices, Inc. *Secure Encrypted Virtualization API Version 0.22*. URL: https://www.amd.com/system/files/TechDocs/55766_SEV-KM_API_Specification.pdf (visited on Nov. 22, 2020) (cit. on p. 13).

[@8] Paul Alcorn. *AMD Reaches Highest CPU Market Share Since 2007, Q3 2020 Report (Updated)*. URL: https://www.tomshardware.com/news/amd-vs-intel-q3-2020-cpu-market-share-report (visited on Dec. 8, 2020) (cit. on p. 1).

[@9] Arm Limited. *ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition - Reset*. URL: https://developer.arm.com/documentation/ddi0406/b/System-Level-Architecture/The-System-Level-Programmers--Model/Exceptions/Reset?lang=en (visited on Nov. 22, 2020) (cit. on p. 12).

[@10] Robert Buhren and Alexander Eichner. *All You Ever Wanted to Know about the AMD Platform Security Processor and were Afraid to Emulate - Inside a Deeply Embedded Security Processor*. URL: https://www.blackhat.com/us-20/briefings/schedule/#all-you-ever-wanted-to-know-about-the-amd-platform-security-processor-and-were-afraid-to-emulate---inside-a-deeply-embedded-security-processor-20106 (visited on Nov. 26, 2020) (cit. on pp. 3, 51).

[@12] Intel Corporation. *2020.2 IPU – Intel® CSME, SPS, TXE, AMT and DAL Advisory*. URL: https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00391.html (visited on Nov. 29, 2020) (cit. on pp. 1, 5).

[@13] Intel Corporation. *Intel Firmware 2018.4 QSR Advisory*. URL: https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00191.html (visited on Nov. 29, 2020) (cit. on p. 5).

[@14] Ltd DediProg Technology Co. *EM100Pro/EM100Pro-G2 Software User Manual*. URL: https://www.dediprog.com/download/save/79.pdf (visited on Dec. 1, 2020) (cit. on p. 35).

[@15] *DrCov File Format*. URL: https://www.ayrx.me/drcov-file-format (visited on Nov. 5, 2020) (cit. on p. 24).

[@16] Uri Farkas, Ido Li On, and CTS Labs. *AMDFlaws A Technical Deep Dive*. URL: https://cts-labs.com/past-publications (visited on Nov. 29, 2020) (cit. on p. 5).

[@17] Andrew Goossen and Jason Ronald. *A Closer Look at How Xbox Series X|S Integrates Full AMD RDNA 2 Architecture*. URL: https://news.xbox.com/en-us/2020/10/28/a-closer-look-at-how-xbox-series-xs-integrates-full-amd-rdna-2-architecture/ (visited on Nov. 22, 2020) (cit. on p. 56).

[@19] Intel Corporation. *Intel® Chipsets Low Pin Count Interface Specification*. URL: https://www.intel.com/content/www/us/en/design/technologies-and-topics/low-pin-count-interface-specification.html (visited on Nov. 5, 2020) (cit. on p. 37).

[@22] The CoreBoot project. *AMD Family 17h in coreboot*. URL: https://doc.coreboot.org/soc/amd/family17h.html (visited on Nov. 26, 2020) (cit. on p. 55).

[@23] The Libreboot Project. *Frequently Asked Questions*. URL: https://libreboot.org/faq.html#amd-platform-security-processor-psp (visited on Dec. 8, 2020) (cit. on p. 1).

[@24] Thomas Rijo-john. *[RFC,0/5] Add TEE interface support to AMD Secure Processor driver*. URL: https://patchwork.kernel.org/project/linux-crypto/cover/cover.1571817675.git.Rijo-john.Thomas@amd.com/ (visited on Nov. 26, 2020) (cit. on p. 2).

[@25] Ned Smith and Intel Coporation. *Storage Protection with Intel® Anti-Theft Technology - Data Protection (Intel® AT-d)*. URL: https://www.intel.com/content/dam/www/public/us/en/documents/research/2008-vol12-iss-4-intel-technology-journal.pdf (visited on Dec. 6, 2020) (cit. on p. 1).

[@26] Ryan Smith. *Sony Teases Next-Gen PlayStation: Custom AMD Chip with Zen 2 CPU & Navi GPU, SSD Too*. URL: https://www.anandtech.com/show/14224/sony-teases-nextgen-playstation-custom-amd-chip-with-zen-2-cpu-navi-gpu-ssd-too (visited on Nov. 22, 2020) (cit. on p. 56).

[@27] *Unicorn Engine*. URL: https://github.com/unicorn-engine/unicorn (visited on Nov. 5, 2020) (cit. on p. 18).

[@28] Christian Werling and Robert Buhren. *PSPTool: Display, extract, and manipulate PSP firmware inside UEFI images*. 2020. URL: https://github.com/PSPReverse/PSPTool (visited on Nov. 5, 2020) (cit. on p. 5).

[@29] Jiewen Yao and Vincent J. Zimmer. *Memory Protection in SMM*. URL: https://edk2-docs.gitbook.io/a-tour-beyond-bios-memory-protection-in-uefi-bios/memory-protection-in-smm (visited on Dec. 1, 2020) (cit. on p. 45).

[@30] *Zen - Microarchitectures - AMD*. URL: https://en.wikichip.org/wiki/amd/microarchitectures/zen (visited on Nov. 5, 2020) (cit. on p. 7).

[@31] *Zen+ - Microarchitectures - AMD*. URL: https://en.wikichip.org/wiki/amd/microarchitectures/zen%2B (visited on Nov. 5, 2020) (cit. on p. 7).

[@32] *Zen 2 - Microarchitectures - AMD*. URL: https://en.wikichip.org/wiki/amd/microarchitectures/zen_2 (visited on Nov. 5, 2020) (cit. on p. 7).

[@33] Vincent Zimmer and Michael Krau. *ESTABLISHING THE ROOT OF TRUST*. URL: https://www.uefi.org/sites/default/files/resources/UEFI%20RoT%20white%20paper_Final%208%208%2016%20%28003%29.pdf (visited on Nov. 26, 2020) (cit. on p. 1).

List of Figures

List of Listings

MicroPython port

A

The emulator relies on components like `libpspproxy`[1] and the `psp-apps`[2] framework to implement proxy mode. These components can be used standalone as well, to explore the environment on the PSP without requiring the emulator. `libpspproxy` for example has python bindings to allow accessing devices from an interactive python shell. Furthermore, the stub running on the PSP allows executing code modules natively on the PSP for timing critical tasks for example. As a showcase, MicroPython[3] was ported to the PSP[4]. Listing A.1 shows the interactive python shell of MicroPython which is running on the PSP. The code module is loaded into the PSP with the `cm-tool`[5] command included in `libpspproxy`. The I/O is transported over the flash emulator in the example but the legacy serial port can be used as well.

```
$sudo ./em100 --stop -c w25q128fw -d ../../exploit.bin -v -p INPUT --start -t -T -n 1237
[...]
Stopped EM100Pro
Configuring SPI flash chip emulation.
Voltage set to 1.8
Chip set to Winbond W25Q128FW.
Hold pin state set to input
Verify: PASS
Started EM100Pro
Starting trace & EM100: Waiting for incoming connection...
EM100: Connected, entering I/O loop
[...]
$ ./cm-tool tcp://localhost:1237 ../micropython/ports/amdpsp/build/firmware.bin
00:00:00.000 PspSerialStub main: Transport channel initialized -> starting mainloop
00:00:00.000 PspSerialStub pspStubMainloop: Entering
00:00:01.441 PspSerialStub Someone connected to us \o/...
00:00:01.442 PspSerialStub pspStubMainloop: Connection established
MicroPython v1.12-388-g6f94599e8-dirty on 2020-04-29; AMD with amd-psp
>>> print("Hello World!")
Hello World!
>>> 1+1
2
>>>
Code module executed successfully and returned 0
```

Listing A.1: Micropython running on the physical PSP

[1] `https://github.com/PSPReverse/libpspproxy`
[2] `https://github.com/PSPReverse/psp-apps`
[3] `https://micropython.org/`
[4] `https://github.com/AlexanderEichner/micropython/tree/master/ports/amdpsp`
[5] `https://github.com/PSPReverse/libpspproxy/blob/master/cm-tool.c`

GitHub repositories

B

The emulator depends on several sub projects, each having its own git repository on GitHub. The following list gives a quick overview of the involved repositories:

- **libgdbstub**
  `Description:` GDB remote serial protocol parser library.
  `Purpose:` Provides GDB support to `PSPEmu`.
  `URL:` `https://github.com/AlexanderEichner/libgdbstub`

- **unicorn**
  `Description:` Unicorn CPU emulator framework.
  `Purpose:` Provides ARM code execution support to `PSPEmu`.
  `URL:` `https://github.com/PSPReverse/unicorn/tree/mmio`
  `Forked from:` `https://github.com/unicorn-engine/unicorn`

- **capstone**
  `Description:` Multi architecture disassembly framework.
  `Purpose:` Provides ARM disassembly support to `PSPEmu`.
  `URL:` `https://github.com/aquynh/capstone`

- **em100**
  `Description:` EM100-Pro command-line utility.
  `Purpose:` Flash emulator control utility for proxy mode.
  `URL:` `https://github.com/PSPReverse/em100/tree/network-mode-v1`
  `Forked from:` `https://review.coreboot.org/cgit/em100.git`

- **psp-includes**
  `Description:` PSP hardware and firmware interface definitions.
  `Purpose:` Central place shared across multiple projects.
  `URL:` `https://github.com/PSPReverse/psp-includes`

- **psp-apps**
  `Description:` Framework to run code on the PSP.
  `Purpose:` Provides the stub required for proxy mode.
  `URL:` `https://github.com/PSPReverse/psp-apps`

- **libpspproxy**
  - `Description:` Independent library to interface with a real PSP running the stub.
  - `Purpose:` Provides the communication interface for proxy mode.
  - `URL:` `https://github.com/PSPReverse/libpspproxy`
- **PSPEmu**
  - `Description:` Emulator for AMD's (Platform) Secure Processor.
  - `Purpose:` The main emulator repository.
  - `URL:` `https://github.com/PSPReverse/PSPEmu`
- **vbox-ice**
  - `Description:` Patches to support custom UEFI firmware emulation with VirtualBox.
  - `Purpose:` Allows running the x86 UEFI firmware in a virtual machine.
  - `URL:` `https://github.com/PSPReverse/vbox-ice`
- **x86-stub**
  - `Description:` Stub running on the x86 BSP executing requests forwarded from the PSP.
  - `Purpose:` Provides hardware access when running the x86 UEFI in a virtual environment.
  - `URL:` `https://github.com/PSPReverse/x86-stub`

Unicorn modifications

C

The following table gives the list of changes done to unicorn in the repository[1] in order to enable proper support for the PSP emulator:

Commit ID	Commit message
d8937580	Port MMIO branch from https://github.com/kitlith/unicorn to recent upstream unicorn
72d71a25	uc.c: Fix use-after-free if a hook is deleted in the callback (no idea why the available fixed wasn't merged upstream yet)
dab22aab	arm: Support reading adn writing the VBAR register by add UC_ARM_REG_C12_C0_0/UC_ARM_REG_VBAR
79d4b9fc	arm: Add new hook to get notified about cp15 writes and add a few more ARM registers for reading/writing using unicorn
e45b0636	Add flag to force ARM MMU to be disabled regardless of the bit in SCTRL
a785aa28	Make it possible to intercept all CP accesses by always emitting code calling the helpers
2f1ce1b7	More hacks for the PSP emulator • Make it possible to intercept CPSR writes • Enable EL3 (aka TrustZone) feature and define a few more cp15 registers so we can intercerpt reads and writes • Flush the TLB whenever a memory write hook was called to avoid stale entries in there. The memory write callback in the emulator will be used to track page table writes and it unmaps affected memory regions.
60cf78cc	Always give the complete CPSR value in the callback

[1] `https://github.com/PSPReverse/unicorn/tree/mmio`