Setup of a bare-metal hypervisor on ARM

Alexander Muth

Dept. of Computer Science, TU Darmstadt
Laboratory in System and IoT Security

Abstract

This paper presents the step-by-step implementation of a bare-metal hypervisor based on the ARMv8 architecture running on a Raspberry Pi 4 Model B. Leveraging a combination of C and Assembly language, the hypervisor is designed to provide direct hardware control while maintaining a structured and clear codebase. The implementation process is divided into several tasks, progressively expanding the hypervisor's functionality. Key features include the setup and utilization of the UART interface, exploration of the exception model, and initialization and coordination of multiple processor cores. Additionally, the hypervisor demonstrates the loading and execution of external programs, showcasing its capability to serve as a stable execution environment for running programs close to the hardware. Furthermore, a proof-of-concept fuzzer, named host_uart_writer, is developed to send programs to the hypervisor via UART and receive execution results. While the current implementation lacks security features such as memory isolation, it provides a solid foundation for further development and optimization. Overall, this paper serves as a practical guide and a playground for exploring the hardware and hypervisor concepts.

1 Introduction

The continuous evolution of modern computer systems prompts the adoption of sophisticated strategies to harden them against security vulnerabilities. This necessitates a multifaceted strategy that encompasses various layers of defense. Among these layers is the microarchitectural level, which encompasses the processor. Detecting and mitigating vulnerabilities within the processor requires the execution of numerous instruction combinations. The resulting outcomes and processor state can then be compared against expected results, aiding in the identification of undocumented or faulty behavior. Such rigorous execution demands a controlled environment where programs can be sequentially executed, errors can be handled, and execution can occur as close to the hardware as possible to minimize interference with other code, such as kernel operations, thereby ensuring the integrity of findings.

This paper outlines the development of a bare-metal hypervisor tailored to provide such a controlled environment on the Raspberry Pi 4 Model B, featuring an ARMv8 processor. The focus of this paper is on a proof-of-concept implementation of such a hypervisor, serving as a playground for teaching purposes. The implementation is structured into eight tasks, each contributing to the incremental expansion of functionality. The final implementation enables the reception and execution of programs received over a wired connection on the Raspberry Pi, orchestrating their execution across separate processor cores, and transmitting the computed outcomes via the same wired connection.

Subsequent chapters will delve into the background and fundamentals of hypervisors, hardware fuzzing, and the utilized hardware, including the ARMv8 architecture. Following this, the paper will detail the concepts and specifics of the implementation. Lastly, the paper will conclude with key insights from the implementation and propose future enhancements for the hypervisor.

2 Background

This chapter provides an overview of the fundamentals of hypervisors, processor fuzzing, the ARMv8 processor architecture, and the hardware utilized in this paper.

2.1 Hypervisors

Hypervisors, also known as virtual machine monitor, play a crucial role in the creation and management of virtualized environments. They abstract the underlying hardware of the host system to the software on top and facilitate the concurrent execution of multiple isolated operating systems on the same hardware.

There are two types of hypervisors. Type 1 hypervisors, also known as bare-metal hypervisors, run directly on the host hardware, abstracting it for the software layers above. They enable efficient partitioning of hardware resources among multiple operating systems with minimal overhead. Type 2 hypervisors, also called hosted hypervisors, operate within another operating system alongside other processes. While they offer the flexibility to execute additional operating systems, they introduce more overhead due to their layered architecture.

The primary functions of a hypervisor include resource abstraction, isolation, and management. By abstracting physical hardware resources such as CPU, memory, storage, and network interfaces, hypervisors enable multiple virtual machines to coexist on a single physical computer, each operating independently of the others. Isolation ensures that each virtual environment remains insulated from one another, preventing interference and ensuring security and stability [1].

2.2 Processor Fuzzing

Processor fuzzing is a form of security testing aimed at uncovering vulnerabilities and undocumented behavior in the microarchitectural components of a processor. While traditional software fuzzing techniques target bugs and vulnerabilities in software components by subjecting them to various potentially flawed input data, processor fuzzing focuses on the components of the CPU and therefore targets hardware instead of software.

Modern processors consist of a multitude of interacting microarchitectural components, posing significant challenges for security researchers. Undocumented or faulty behavior of a processor may not necessarily impact system security negatively; however, in the worst case it can potentially expose sensitive data, making it necessary to identify and address such issues.

Processor fuzzing aims to uncover such security vulnerabilities by executing a multitude of instructions in various execution combinations on the processor. This involves varying not only the instructions themselves but also the initial contents of registers or memory and the timing of instruction execution. The input data aims to trigger undocumented, unusual, or unintended behavior. The fuzzing framework must detect abnormal CPU behavior, alerting the security researcher to investigate further. For instance, if only undocumented instruction combinations are executed, the CPU execution should always result in an error, indicating that the instruction does not exist. If this is not the case, undocumented behavior has been identified, which may not necessarily be security-critical but could potentially be [4].

With the increasing complexity of modern processors, fuzzing can be used to analyze these numerous and complex, opaque components and assess potential security vulnerabilities. However, efficient and intelligent tools for systematically generating input data and instructions, executing tests on the target hardware, and analyzing execution results are necessary for this purpose.

2.3 The ARMv8 Architecture

In this section, we provide both a general overview of the ARMv8 architecture and delve into specifics such as the exception model and the various types of registers.

ARMv8 denotes the eighth generation of the Advanced RISC Machines (*ARM*) processor architecture, predominantly utilized in mobile and embedded computing but increasingly employed in high-performance systems. It is a 64-bit Reduced Instruction Set Architecture (*RISC*), following the Load-and-Store approach [5]. The ARM architecture is renowned for its energy efficiency, scalability, and versatility, allowing its deployment in both embedded and high-performance sectors.

2.3.1 Exception Model

In contemporary computing systems, various modules operate with different privileges, including regular user programs, operating systems, and hypervisors. The ARMv8 architecture addresses this requirement through a hierarchical structure known as exception levels (*EL*).

Exception levels form a hierarchy of privileges ranging from EL0 to EL3, with EL0 being the lowest and EL3 the highest privilege level. Access can be made from higher levels (e.g., EL3) to lower levels (e.g., EL2), but the reverse is not possible. The architecture does not dictate which parts of the system must execute at a specific level. However, typically, user processes run at EL0, the operating system at EL1, and the hypervisor at EL2. EL3 is utilized by firmware and provides access to secure enclaves.

Transitions between exception levels can occur through several methods, such as the occurrence of an exception, return from an exception, processor reset, debug mode entry, or exit from debug mode. Exception occurrences lead to a transition to the same or a higher exception level. Conversely, returning from an exception can only transition to the same or a lower exception level. However, EL0 can only trigger exceptions to higher levels, making it unable to receive and handle exceptions.

Additionally, modern ARM processors feature execution states, which describe different modes of operation. AArch32 represents the 32-bit mode, compatible with older ARM architectures, while AArch64 is the 64-bit mode exclusive to ARMv8 and newer. The transition between execution states is only possible through a reset or exception level change. When transitioning from a lower to a higher EL, the execution state can remain the same or switch to AArch64. When transitioning from a higher to a lower EL, the execution state can remain the same or switch to AArch32. Thus, an AArch32 layer at EL0 can be hosted by an AArch64 layer at EL1, but not vice versa.

Exceptions in ARMv8 architecture can be synchronous or asynchronous events. Synchronous exceptions, triggered by instruction execution, are immediately handled during instruction execution. The ARMv8 architecture utilizes synchronous exceptions to implement kernel and hypervisor calls through specific instructions that trigger an exception, subsequently handled by the operating system or hypervisor. Asynchronous exceptions are triggered by external events, like a timer generating an interrupt after a specific duration. Interrupts in the ARMv8 architecture are therefore a form of asynchronous exceptions. Through masking, asynchronous exceptions can be ignored until unmasked, useful in critical sections where exceptions by interrupts are undesired.

To handle exceptions effectively, processors rely on exception vectors (*EV*) stored in the exception vector table (*EVT*). Each EV corresponds to a specific type of exception and contains the code to execute when the exception occurs. It is 32 words (128 bytes) in size, sufficient for storing a branch instruction that jumps to the respective exception handler (*EH*). The EVT includes multiple EV, 16 in total, for various exception types, such as synchronous, IRQ, FIQ, and SError, with multiple entries differentiating based on the current exception level and execution state. Each EL (excluding EL0 as it cannot receive exceptions) has its own EVT, with its memory position set by specific registers.

When an exception occurs, the specific EH is invoked by the EV to address the error. The exception handler can rectify the error or terminate the program, typically after saving the processor status for later restoration. After handling the error, the processor status is restored, allowing the program to continue its execution [5].

2.3.2 Registers

ARMv8 processors provide various types of registers for different purposes.

The most commonly used and most-known registers are the general-purpose registers. The ARMv8 architecture offers 31 general-purpose registers, each 64 bits wide, addressed as x0 through x30. Alternatively, to access the lower 32 bits, they can be referenced as w0 through w30. General-purpose registers are accessible at all exception levels. Typically, x29 is used as the frame pointer, and x30 serves as the link register. Additionally, ARMv8 processors offer 32 vector registers, each 128 bits wide, primarily used for Single Instruction Multiple Data (*SIMD*) instructions.

In addition to the general-purpose registers, ARM processors feature various special registers utilized for example to configure the CPU. These special registers have the suffix *_ELx*, where *x* denotes the minimum EL required to access the register. For instance, *SP_EL0*, *SP_EL1*, *SP_EL2*, and *SP_EL3* hold the stack pointer (*SP*) for different exception levels. Accessing SP_EL1, for example, requires the current EL to be at least EL1. The current stack pointer can be accessed via the SP register. Unlike ARMv7, in ARMv8, the program counter (*PC*) is a special register rather than being hold in a general-purpose register. The Saved Processor State Register (*SPSR_ELx*) stores the processor status upon taking an exception, while the exception link register (*ELR_ELx*) stores the return address to resume execution after handling the exception. The exception syndrome register (*ESR_ELx*) contains information about the cause of an exception. The hypervisor configuration register (*HCR_EL2*) is used for configuring exceptions and other features, such as the virtualization of the hypervisor exception level (EL2). The base address of each EVT is stored in the *VBAR_ELx* register [5].

2.4 Used Hardware

The target hardware for the developed hypervisor is a *Raspberry Pi 4 Model B* single-board computer. To facilitate communication between the development or fuzzing machine and the hypervisor running on the Raspberry Pi, the *Raspberry Pi Debug Probe* is utilized.

The Raspberry Pi 4 Model B features a *Broadcom BCM2711* processor comprising four *Cortex-A72* cores of the ARMv8 processor architecture. In addition to various interfaces such as USB, USB-C, micro-HDMI, and Ethernet, it provides a Micro-SD card slot serving as mass storage, from which an operating system or hypervisor can be loaded. The model used here is equipped with four gigabytes of RAM [3].

To establish a connection between the Raspberry Pi and the development machine, the Raspberry Pi Debug Probe is used. The probe can be connected via USB to the development machine and via the General Purpose Input/Output (*GPIO*) pins to the Raspberry Pi [2]. This allows bidirectional data exchange between the two computers using the Universal Asynchronous Receiver-Transmitter (*UART*) protocol.

3 Implementation

This chapter presents the basic principles and functionalities of the hypervisor implementation. Firstly, the programming languages and tools used for the implementation are introduced, followed by the implementation itself.

The implementation is divided into eight tasks to progressively extend the functionality of the hypervisor. Tasks zero to two represent the absolute minimum to start the hypervisor and become familiar with the architecture. Tasks three to five provide the foundational elements necessary for receiving, storing, and executing programs. Task six consolidates the previous concepts into a program, and task seven supplements the entire system with exemplary error handling.

In addition to the hypervisor itself, the program *host_uart_writer* was developed for tasks six and seven. It runs on the development or fuzzing machine connected to the Raspberry Pi via UART over the Raspberry Pi Debug Probe and represents the proof of concept of a fuzzer.

3.1 Used Programming Languages and Tools

The hypervisor was programmed using the languages Assembly and C, while C++ was utilized for the proof of concept fuzzer (host_uart_writer).

Assembly is a language that exclusively utilizes processor instructions and is thus directly understood by the processor. It is used when maximum control over the hardware is required, as it allows direct manipulation of registers and precise control over the executed instructions. In this project, Assembly was primarily used for booting the hypervisor and accessing specific registers.

C is a general-purpose programming language suitable for both application development and system programming. Its direct access to system resources, such as memory, makes it suitable for hypervisor development. C enables structured and organized code, facilitating the realization of large and complex projects. While C is translated into processor instructions by a compiler, inline assembly code allows specific instructions to be embedded directly into C code, enabling the integration of complex assembly code into readable C code.

C++ was chosen for implementing the proof of concept fuzzer. While also supporting low-level programming, C++ provides a high level of abstraction and an extensive standard library, simplifying and accelerating the programming of complex systems.

CMake was used as the build tool for all tasks, including the fuzzer. It consolidates all necessary steps for compilation and linking into a single script, streamlining the build process and caching intermediate products to reduce future build times.

A linker script was employed to organize the binary file of the hypervisor and programs. This script informs the linker, which links the various binary files, about the ordering and location of different sections.

Due to the time-consuming nature of testing the hypervisor on the Raspberry Pi, Quick Emulator (*QEMU*) was utilized for testing. QEMU can emulate ARMv8 instructions and other necessary components of the Raspberry Pi, allowing testing on different machines with other processor architectures. However, adjustments to the GPIO base address were required depending on whether the hypervisor was executed in QEMU, which only provides a Raspberry Pi 3 profile, or on real hardware, achieved through a preprocessor directive. Additionally, QEMU, combined with the Embedded GDB Server, enables debugging of the hypervisor, allowing step-by-step execution and observation of variable and register states.

For UART communication from the development machine's perspective, the console program *Minicom* was used, while the C++ library *CppLinuxSerial* was utilized in the introduced fuzzer.

3.2 Task 0 - Hello World

This task serves merely to start the hypervisor and output data via UART.

Initially, the *MPIDR_EL1* register is read to obtain the ID of the current core. This is necessary because all cores except core #0 are to be put into an infinite loop. Core #0 overwrites the BSS section, which contains declared but uninitialized hypervisor variables, with zeros. After that it sets the currently used SP to a free memory address beyond the loaded data in memory, and calls the *main* function.

The *main* function is implemented in C and serves as the entry point for further operations. In the *main* function, UART is initialized first and then utilized to send various data types such as chars, strings, and numerical values in different formats via UART for testing purposes. Subsequently, an infinite loop is started, which echoes back all characters received via UART.

3.3 Task 1 - Exception Model

In this task, the exception model is explored, and all levels of the exception model are descended, testing the handling of exceptions at EL2 by a hypervisor call (*HVC*) instruction. All EL changes get reflected by an output via UART.

The startup code is similar to that in task 0, with the difference that, before configuring the SP, the use of the SP of EL0 (register SP_EL0), also from other EL, is permanently set via the *SPSel* register.

The hypervisor starts at EL2. To intercept a later exception, the EVT of EL2 needs to be initialized in the VBAR_EL2 register. This holds the base address of the EVT of EL2, as described in Chapter 2.3.1. An array with 16 entries, one for each EH, is declared, in which a function pointer to the EH can be stored at the corresponding position. When the address of an EH was stored, it is called by the EV upon an exception. If no EH is present for a corresponding exception, the type of exception, along with the ELR_ELx and ESR_ELx, is output via UART upon occurrence of the exception, and the core is put into an infinite loop.

After initializing the EVT of EL2, an EH is set to handle a synchronous HVC instruction from EL1 in AArch64 execution mode. The installed EH checks the ESR_EL2 register to determine if the exception was triggered through a HVC instruction. If this is the case, the handler sends a customized message over UART and returns to continue execution. If the EH is not caused by a HVC instruction, it continues as if no EH was installed. After that, the hypervisor first descends to EL1. For this, the address at which the execution on EL1 should start is stored in ELR_EL2, and both HCR_EL2 and SPSR_EL2 are configured for the appropriate execution state (AArch64 in our case) and target EL (EL1). The exception return (*ERET*) instruction is then used to change the EL. Since this is technically a return from an exception, the EL can be decreased.

Now the processor is at EL1. Here, an HVC instruction calls the Exception handler of EL2, which was previously configured. During its execution, the processor is at EL2. By using an ERET instruction at the end of the EV, we return to EL1 and continue with the execution of the code after the HVC instruction.

Now, as previously done, we descend from EL1 to EL0, utilizing ELR_EL1 and SPSR_EL1 here, with HCR_EL2 not being set. On EL0, all received data via UART is echoed back in an infinite loop.

3.4 Task 2 - Registers

This task aims to output the different types of registers described in Section 2.3.2. However, not all existing special purpose registers are outputted as not all of them are needed, and the output can be customized as required.

To output the various types of registers, individual macros were implemented, as well as a macro that outputs all registers. For the register output to work, it is necessary that they are not pushed onto the stack beforehand, which is only ensured by macros or inline functions. Since the inlining of functions by the compiler is not always enforced despite the *inline* keyword, macros were chosen here. Macros are replaced directly with the corresponding source text during preprocessing.

Task 1 was expanded to additionally output the set of selected registers with every change of the exception level. To enable access to the SIMD registers at EL1 and EL0, the feature control register (*CPACR_EL1*) is manipulated.

3.5 Task 3 - From UART to Memory

In this task, data is received via UART, stored in memory, and then returned after a user defined number of input data. This process is essential for later receiving programs via UART from the fuzzer.

The incoming data is stored at a fixed address, located one megabyte behind the loaded program data in memory. Initially, a numerical value is obtained via UART. For this, the received string from UART is converted into an integer. This integer dictates the number of characters to be stored in memory.

All subsequent characters entered are then stored in memory until the user defined number of characters is reached. Subsequently, all stored characters are output via UART, cleared, and the data reading process is restarted.

3.6 Task 4 - Program execution

To execute programs received from the fuzzer later on, the execution of programs in memory must be tested. This involves knowing the entry address of the program and then continuing execution at this address using a branch and link register (*BLR*) instruction.

First, the program is built. It performs basic arithmetic operations on three input numbers and returns the results. The base address of this program is set using a linker script. To this address it will be loaded by the hypervisor later on. The program is essentially "hardwired" in this manner, which is enough for our proof of concept. The binary file of the built program can be converted into a char array within a C header file using the console program *xxd*, enabling the ability to be compiled into the *.data* segment of the hypervisor.

From the *.data* segment, the program is then loaded into memory in the hypervisor at the previously defined memory address. The address is cast into a function pointer in C code, accepting three integer parameters and returning an integer. Calling this function pointer executes a BLR instruction, which branches to the program stored in memory, performing the calculations on the three passed numbers. The resulting value is output via UART.

Subsequently, as in previous tasks, all characters entered via UART are echoed back via UART.

3.7 Task 5 - Multicore

Later-received programs should be executed on a core other than #0. To achieve this, the other cores must first be initiated, with the capability to pass them a memory address to start execution on. This task accomplishes that. Additionally, individual stack pointers are set up and used for each exception level and core, rather than just always using the SP of EL0 as before.

In the linker script, memory is reserved to hold the stacks of each core and each exception level (EL0 to EL2). Additionally, 64 bits are reserved for each core to hold the start address of a core's execution.

Initializing a core's stacks involves setting the registers containing the SP of the various ELs (SP_EL0, SP_EL1, and SP_EL2) to the respective addresses where the stack should reside. Core #0 initializes its stacks only once at the start of the hypervisor. The stacks of the other three cores are initialized whenever the respective core is about to execute a new function.

Core #0 must initialize the other cores before executing the *main* function. For this purpose, the address of the instructions to be executed by each core must be written to a memory address predefined for each core. The entry point of each core is chosen to be the entry point of the hypervisor, which, based on the core's ID, makes a distinction between core #0 and all other cores. The behavior of core #0 has been described above, and now the behavior of all other cores follows.

The cores are intended to be started by other cores, specifically by the *main* function of our hypervisor. This is achieved by having the cores first wait for a send event (*SEV*) instruction by executing a wait for event (*WFE*) instruction. If they receive an event, they check a memory location designated for the core for an address to execute. If no address is given, indicating there are no instructions to execute, the core waits for the next event. If an address is given, the memory area is first set to 0 to avoid re-entry into the function in the next iteration. Then, the stacks of the various ELs of the core are initialized, and finally, a BLR instruction to the given address is executed. After the invoked instruction stream returns, the core waits for a new address to instructions to execute.

The hypervisor begins execution of the *main* function in C code on core #0. In this function, the other three cores are started successively, and the implementation from Section 3.3 is executed on each core. Each time, execution waits until the core reaches EL0 by setting an integer flag to 1. The *main* function of core #0 waits for the flag to be set before starting the next core. After all other cores have finished their execution, the code is executed again on core #0 as a final step.

3.8 Task 6 - Bringing it all together

In this section, the concepts from the three previous tasks are integrated into a hypervisor, which receives a program via UART, stores it to memory, executes the program on all cores, and returns the calculation result via UART. At the same time, the program host_uart_writer is implemented, which serves as proof of concept fuzzer and therefore sends the program from the development machine to the hypervisor via UART and receives the results.

3.8.1 Hypervisor

First, all cores are initialized as described in section 3.7. Core #0 waits for the input of the number of bytes of the program to be transferred. Then the previously parsed number of bytes is read and stored in memory. After receiving and storing the program in memory, Core #0 sequentially starts the other cores with a function that executes the program saved to memory and returns the calculation result via UART. Finally, the program is also executed from core #0. After returning the calculation results here as well, all entered characters are echoed back in an endless loop

3.8.2 Fuzzer

To avoid manually copying the program into the Minicom console every time to send it via UART and to demonstrate that programmatic input is very easy, the program host_uart_writer was written, which serves as a proof of concept for a fuzzer. It is executed on the development machine, which is connected to the hypervisor on the Raspberry Pi via UART through the Raspberry Pi Debug Probe.

The program first reads in a provided program, which is then sent to the hypervisor and should be executed by it. For testing purposes, the same program as in section 3.6 is used here.

After the program is read in, it is sent to the hypervisor via UART using the C++ library CppLinuxSerial. For this, a device file must be passed to the library (e.g., */dev/ttyACM0*), through which the debug probe can be addressed. If the hypervisor is run via QEMU, its UART input and output can be redirected to a pseudoterminal (e.g., */dev/pts/6*), through which communication for debugging can also take place. The hypervisor then reads in the program, stores it, and executes it as described in section 3.8.1. The returned results are then read from UART by the host_uart_writer and displayed on the console.

3.9 Task 7 - Exception handling

In this final task, the execution of multiple programs consecutively is demonstrated. Additionally, an exception handler is registered to catch all exceptions during program execution, handle them, and thus, despite, for example, faulty instructions, provide a result to the fuzzer and prevent the termination of the core.

3.9.1 Hypervisor

The hypervisor essentially works like the one from section 3.8.1. However, the reading and execution of the program are repeated continuously until a stop signal is received. This allows for the execution of more than one program. The program is always executed on core #1 and always written to the same location in memory. Thus, more than one program cannot be executed simultaneously.

An exception handler is installed for each of the 16 exception vectors, which outputs a simple error message via UART. This error message includes only the ELR_EL2 and ESR_EL2 registers but can be expanded if needed. After the error output, including the two registers above via UART, the core waits for a new program to execute, just as it would if terminated without errors. If no error occurs, the core waits for a new address to continue execution as in section 3.7.

3.9.2 Fuzzer

This task's proof of concept fuzzer is mostly the same as the previous one described in section 3.8.2.

The only difference is that it alternates between two programs to be tested and, in addition to the error-free program from before, also tests a faulty program. The faulty program executes an undefined (*UDF*) instruction, which by definition triggers an undefined instruction exception.

The output of the hypervisor, including all program outputs, is displayed on the console of the host_uart_writer.

4 Conclusion

This paper described the step-by-step implementation of a bare-metal hypervisor based on the ARMv8 architecture on a Raspberry Pi 4 Model B. By utilizing C and Assembly, the functionality was realized, with C being used for a higher, more structured, and clearer implementation wherever possible, and Assembly being used for precise control of the hardware.

The implementation was divided into multiple tasks, gradually expanding the functionality of the hypervisor. It can still be further extended and provides a clear playground for exploring hardware and hypervisor concepts. The implemented functionality includes the setup and use of the UART interface, the exception model, and the initialization and utilization of all processor cores. Additionally, loading and executing an external program were demonstrated with a simple implementation. The presented approaches illustrate how a hypervisor can serve as a stable environment for running programs close to the hardware with direct access to the processor and without interfering with code from a kernel.

Furthermore, the host_uart_writer demonstrated how a fuzzer could send data to the hypervisor via UART and receive the execution results or occurring errors. This fundamental principle can be used for a fuzzer that tests the processor with various programs for errors or undocumented behavior.

The implementation presented here can serve as a starting point for further development of the hypervisor. However, its main purpose is to demonstrate some basic concepts and serve as a playground to better understand the hardware in general, especially the ARMv8 architecture.

5 Outlook

The hypervisor developed in this work can still be further developed and improved in many aspects.

One major area that has not been addressed yet is security. Currently, the programs are executed at EL2 and can therefore access the memory of the hypervisor as well as other processes. Thus, there is currently no isolation. Introducing virtual memory can prevent this, where each process is only allowed to access its own memory, enforced by the processor and allocated by the hypervisor. This would also eliminate the need for programs to be written to a fixed memory address, as the virtual address space starts at address zero, allowing the program to be linked and started as if it had the entire RAM to itself.

Furthermore, the functionality is far from being exhausted. More complex exception handlers can be written that handle exceptions more comprehensively than the rudimentary ones implemented here for demonstration purposes. Additionally, scheduling could be introduced to allow more processes to run than cores available. However, this contradicts the idea of running fuzzing processes directly on the hardware without interfering with other code.

The architecture of the hypervisor could also be adjusted to build on a layer that abstracts the platform-specific differences, allowing it to be executed on any processor architecture with the same interface and code.

To enable proper fuzzing, it is conceivable to use a common format such as XML or JSON for the input and output of the fuzzer, which can be used to transmit a large, structured amount of data as needed.

Another improvement, which is not related to the hypervisor itself but to the development process, is to use the Joint Test Action Group (*JTAG*) methodology via the GPIO pins, instead of the used UART communication. This enables the ability to debug the hypervisor on the real hardware.

References

This work was created independently and was linguistically revised with the help of ChatGPT.

[1] https://www.redhat.com/en/topics/virtualization/what-is-a-hypervisor.

[2] https://datasheets.raspberrypi.com/debug/raspberry-pi-debug-probe-product-brief.pdf. https://datasheets.raspberrypi.com/debug/raspberry-pi-debug-probe-product-brief.pdf.

[3] https://datasheets.raspberrypi.com/rpi4/raspberry-pi-4-product-brief.pdf. https://datasheets.raspberrypi.com/rpi4/raspberry-pi-4-product-brief.pdf.

[4] Christopher Domas. Breaking the x86 isa. 2017.

[5] ARM Holdings Limited. *ARM Architecture Reference Manual - ARMv8, for ARMv8-A Architecture Profile*.