UNIVERSITY OF OULU

FACULTY OF INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

Jussi Kurikka

Testing Embedded Software in a Simulated Environment

Master's Thesis
Degree Programme in Computer Science and Engineering
January 2022

Kurikka J. (2022) Testing embedded software in a simulated environment. University of Oulu, Degree Programme in Computer Science and Engineering. Master's Thesis, 50 p.

ABSTRACT

In this master's thesis, a simulation environment that can be used to execute embedded software's unit tests is implemented. The purpose of the simulation is to make the development of the embedded firmware easier, cheaper, and faster. Also, the purpose is to make remote work easier by enabling unit test and integration test execution on a laptop.

This topic has been researched a lot before and many different solutions and tools exist for embedded system simulation. Some of these solutions are introduced in this paper. After the introduction, two of the solutions are implemented for one embedded system that uses monolithic firmware.

The solutions implemented are emulation based on the Unicorn emulator and a simulation with native execution on a PC. Each solution has advantages and disadvantages. But in this case, the native execution on a PC was better, as the test execution was two times faster than in Unicorn emulator and three times faster than in an embedded device. Native execution was also easier to implement than Unicorn emulator and could use free compilers like GCC and Clang. The biggest disadvantage with native execution was the low fidelity.

Keywords: Embedded software, simulation, QEMU, emulation, testing

Kurikka J. (2022) Sulautetun ohjelmiston testaaminen simuloidussa ympäristössä. Oulun yliopisto, tietotekniikan tutkinto-ohjelma. Diplomityö, 50 s.

TIIVISTELMÄ

Tässä diplomityössä tehdään simulointiympäristö, jolla voidaan ajaa sulautetun järjestelmän yksikkö- ja integraatiotestejä. Simulaation tarkoitus on tehdä sulautetun järjestelmän ohjelmistokehitys helpommaksi, halvemmaksi ja nopeammaksi. Lisäksi simulaatiolla saadaan tehtyä etätyöskentely helpommaksi, kun yksikkö- ja integraatiotestit voidaan ajaa kannettavalla tietokoneella.

Sulautetun järjestelmän simulointia on tutkittu paljon ja simulointiin on kehitetty monia eri ratkaisuja ja työkaluja. Osa näistä työkaluista esitellään tässä diplomityössä. Esittelyn jälkeen toteutetaan kaksi eri simulointi ympäristöä yhdelle sulautetulle järjestelmälle.

Toteutetut simulaatiot ovat: emulaatio joka tehdään Unicorn emulaattorilla ja simulaatio joka toteutetaan natiiviajona PC:llä. Molemmilla ratkaisuilla on hyvät ja huonot puolet. Mutta kokonaisuutena natiiviajo oli parempi tälle sulautetulle järjestelmälle, koska natiiviajo oli kaksi kertaa nopeampi kuin Unicorn emulaattori ja kolme kertaa nopeampi kuin sulautettu järjestelmä. Lisäksi natiiviajo oli helpompi toteuttaa kuin Unicorn emulaattori ja natiiviajossa voitiin käytettään ilmaisia kääntäjiä kuten GCC ja Clang. Huonoin puoli natiiviajossa oli se, että natiiviajon tarkkuus ei ollut kovin hyvä, eikä sillä näin ollen pystynyt testaamaan kaikkia asioita koodista.

Avainsanat: Sulautettu ohjelmisto, simulaatio, QEMU, emulaatio, testaus

TABLE OF CONTENTS

FOREWORD

Writing this master’s thesis has been a big challenge for me. I have learned a lot of new things when doing this thesis. This thesis was done while I was working for Elektrobit, so I want to thank Elektrobit for giving me the opportunity to do this thesis. From Elektrobit I want to thank my technical supervisor Gabriel Byman and Juha Mäki-Asiala for good ideas. From the University of Oulu, I want to thank my second supervisor Miguel Bordallo Lopez and I especially want to thank my main supervisor professor Olli Silven for great advice and feedback.

Oulu, 18.2.2022

Jussi Kurikka

ABBREVIATIONS

ISA	instruction set architecture
CPU	central processing unit
JIT	just-in-time
AOT	ahead-of-time
SoC	system-on-chip
ISS	instruction set simulator
RISC	reduced instruction set computer
CISC	complex instruction set computer
VLIW	very long instruction word
MMU	memory management unit
MPU	memory protection unit
OS	operating system
HDL	hardware description language
AES	advanced encryption standard
GPU	graphics processing unit
ASLR	address space layout randomization
I/O	input/output
ROM	read-only memory
RAM	random-access memory
OTP	one-time programmable
ECC	error correction code
BSP	board support package
HAL	hardware abstraction layer
API	application programming interface
HSM	hardware security module
TRNG	true random number generator
ECDSA	elliptic curve digital signature algorithm
NVIC	nested vector interrupt control
GDB	GNU Debugger
IR	intermediate representation
LLVM	low level virtual machine
IPC	Inter-process communication

1 INTRODUCTION

Testing an embedded device's firmware on a physical board can be slow and difficult. What makes it slow is that the firmware binary must be written to the embedded device's chip and thus the device must be connected to a PC and a debugger must write the binary to the chip and this can take a lot of time. Also, the number of embedded devices can be limited. On top of this, the price of the debugger, compiler license, and hardware can be thousands or tens of thousands of euros, so it would be useful to have other ways to test the embedded software. One way of doing this is running the tests in a simulated environment on a PC.

Having the ability to test embedded software on the same machine as the code is developed, would be beneficial when working in an office, but when doing remote work, it is even more important. And as remote work has become more common, has other software development fields like web development become more compelling because they are easier to do remotely (as there isn't a need for a debugger nor a physical embedded board). So, it would be useful to improve embedded firmware development by making remote work easier, and a simulation environment is one way of doing so.

Many different test types can be used to test the validity of software. Examples of these include unit testing, integration testing, acceptance testing, performance testing, stress testing, and fuzz testing. Some testing must be done on a real embedded system e.g., acceptance testing, performance testing, and stress testing should be done on a real embedded system, but other testing like unit testing, integration testing, and fuzz testing can also be done in a simulated environment. This master's thesis concentrates on how firmware's unit tests and integration tests can be executed on a PC.

Embedded devices are many times developed as hardware/software co-design. In hardware/software co-design both hardware and software can naturally be tested in a simulated environment. But if the hardware is already finished or it is developed by a different company then only the software needs to be tested. For the implementation part of this master's thesis, the hardware has already been developed by another company so only the testing of the software is investigated.

In this master's thesis, two environments that simulate embedded device's hardware and can execute the embedded system's software are developed. The first solution is based on an emulator and the second is based on native execution on a PC. The main goal of the simulation environment is that it would be possible to develop and partly test the firmware on the same PC environment and thus make the developing process easier.

This master's thesis has the following structure: Chapter 2 explains how software can be executed on different platforms on a general level. Chapter 3 explains what challenges there are in cross-platform execution for embedded devices. And after that Chapter 4 explains different techniques of how firmware can be executed on a PC. Chapter 5 explains how the two simulation environments for this master's thesis were implemented. Chapter 6 covers the evaluation of the implementations. Finally, Chapters 7 and 8 discuss the whole topic and conclude this master's thesis.

2 CROSS-PLATFORM CODE EXECUTION

When software is executed in a simulated environment it means that the software is executed on a different platform than that originally intended. Software that can be executed on multiple platforms is called cross-platform software. There are many techniques for how cross-platform software can be implemented. To get an understanding of these techniques, the general ways of executing the same software on multiple platforms are explained in this chapter. While also the terminology related to the topic is introduced.

2.1 Terminology

This part introduces the basic concepts and terminology that are related to simulation, cross-platform software, and other concepts that are important for this master's thesis. Note that some sources can use different meanings of these terms, but in this section commonly used definitions that also make sense logically are introduced.

2.1.1 Simulation

Simulation means the imitation of some real-world system in an artificial environment that replicates the behavior of the real-world system. This definition is general; thus, the term simulation can mean many different things depending on the context. In the context of this paper, simulation means the imitation of a computer system's hardware on another computer system. The word simulation is used to mean all possible techniques that can be used to imitate the computer system. The system that is simulated is called a target system and the system that does the simulation is called a host system. Figure 1 shows the definition of simulation in the context of this master's thesis.

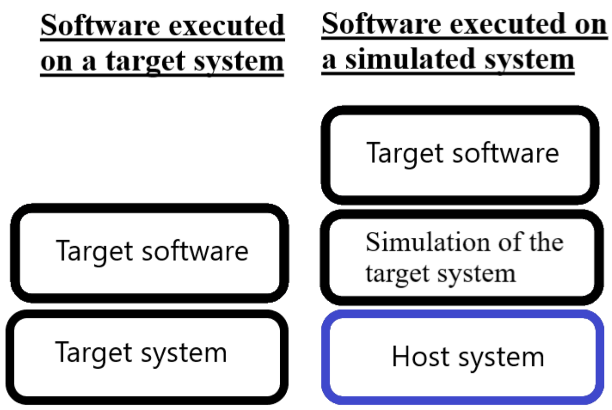

Software executed on a target system
Software executed on a simulated system
Target software
Target software
Simulation of the target system
Target system
Host system

Figure 1. Meaning of simulation in the context of this master's thesis.

2.1.2 Emulation vs Virtualization

Emulation and virtualization have similar meanings i.e., simulation of a computer system that is separate from the host system that is executing it. The difference between the terms is the purpose of the simulation.

For emulation the purpose is to accurately reproduce the behavior of another computer system e.g., the same binaries that were compiled for the target hardware should work directly on an emulator. Most of the time an emulator simulates a different platform than the host system, however it can also be the same platform as the host system as long as the host system is not used directly. The emulator's internal behavior doesn't have to be similar to the target system, it is enough that the behavior of the emulator is similar to the target system.

The purpose of virtualization is to make separate environments (instances) of the host system. The instances are isolated from other instances and the host system. Software/hardware that does virtualization is called "hypervisor". In virtualization, the target software can use the host system's resources directly (e.g., there isn't a translation of the machine code instructions) and this means that the target system and host system must be the same platform. Note that the hypervisor can still control how the target software can use the host system's resources, to make the isolation of the instances possible, but the resource that the target system uses are not simulated with software.

The term virtual machine means virtualization and/or emulation of a computer system. A virtual machine can be made with a hypervisor, an emulator, or with a combination of the two e.g., some parts of the virtual machine are virtualized, and some parts are emulated with software.

2.1.3 Full Virtualization and Full Emulation

The terms virtualization and emulation can mean imitation of the whole computer system or just some parts of it. The terms full virtualization and full emulation are subsets of virtualization and emulation. Full virtualization and full emulation mean that everything from the computer system is imitated. For example, for full virtualization, this means that it is possible to run OS on the hypervisor, and for full emulation, this means that the same binaries that were compiled for the target hardware should work directly on the emulator. For this to work, all peripherals of the target hardware have to be simulated. In this thesis, the purpose of the simulation is to execute embedded system's tests on a PC environment and not to make a new instance of a PC, so the needed technique is full emulation. To make it clear that emulation is the technique used, the term virtual machine is not used in this paper and only terms emulation/emulator are used.

2.1.4 Operating-system-level Virtualization

Operating systems have different ways of creating isolated environments that are somewhat similar to virtual machines. A container is an OS-level virtualization technique that can be used to make an isolated instance of the OS that seems like a real computer system from the point of view of a program. Containers are different from virtual machines because virtual machines can use any OS whereas a container always has the same hardware and OS as the host system. A process can also be thought of as very basic OS-level virtualization that makes it seem that the program uses the whole system (the program can use the whole address space). A process and a container are similar concepts from the point of view of the OS, but the difference between them is that a process has less isolation than a container and a container is a group of processes.

2.1.5 Emulation vs Simulation

Simulation is a general term, that means the imitation of a system. Hence, simulation can be used to mean emulation, virtualization, or something else. In this paper, simulation is used to mean any imitation/modeling of a computer system. Figure 2 shows how simulation terms are related to each other. Note that in some contexts the term simulation has a more specific meaning i.e., an imitation of the computer system but with fewer details than an emulator has. An example of this is the iPhone simulator and the Android emulator. The Android emulator simulates the whole system including the memories and the CPU instruction set. The iPhone simulator, on the other hand, does not simulate iPhone's processor, disk drive, nor memory constraints.

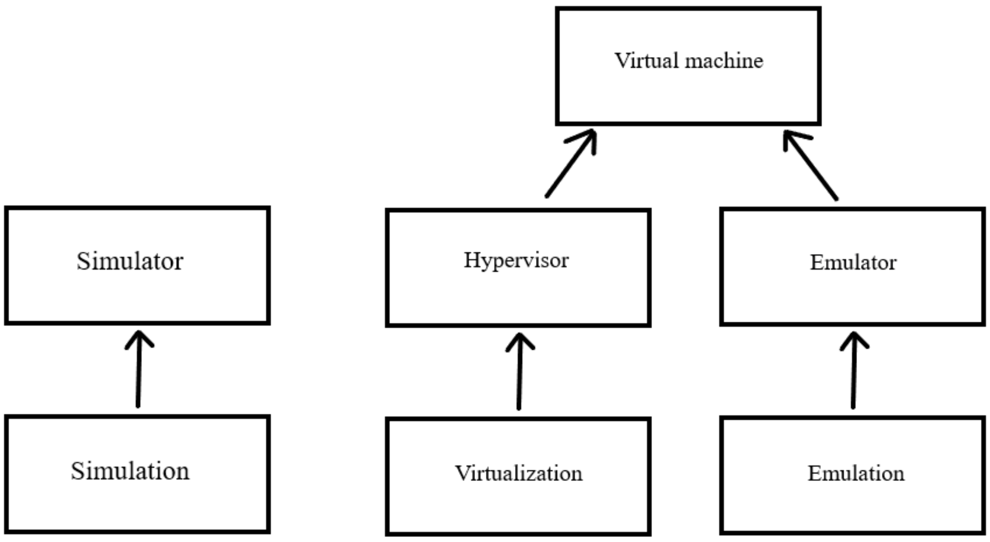


Figure 2. Diagram of simulation terminology.

2.1.6 High-level vs Low-level Languages

A high-level programming language means a programming language that has strong abstraction from the details of the computer. High-level languages don't use platform-specific information within the code. Instead, they use high-level abstractions to instruct a computer. For this reason, it is easy to use high-level level languages to write code for many different platforms. Most of the commonly used languages are high-level languages e.g., C language, Python, Java, JavaScript, etc. High-level languages always need to be translated before they can be executed.

In contrast to high-level languages, there are low-level languages. These low-level languages use ISA (instruction set architecture) directly and thus are machine-dependent i.e., the code only works in the target computer architecture. For example, machine language is a low-level language that only consists of instructions that control the CPU directly and thus is the lowest possible language. Machine language is a numerical language i.e., the instructions are only binary or hexadecimal numbers and therefore it is difficult for a human to read. Assembly language is one level higher than machine language and it consists of commands that are close to machine code instructions; for this reason, assembly language is more human-readable than machine language. Low-level languages generally do not need compilation as they are written directly for the target architecture.

2.1.7 Compilation vs Translation

The general term for changing code to some other language is called translation. The term "translation" can be used regardless of whether the translation is happening from a high-level language to a low-level language, from a low-level language to a high-level language, or between similar levels. If the translation is done with a compiler from a high-level language to a low-level language the word "compilation" can also be used. Decompilation means translation from a low-level language to a high-level language. Examples of translations are presented in Figure 3.

Some sources use the term "recompilation" in the context of run-time translation between the same-level languages, but this usage is not compatible with the definition of compilation (translation from a high-level language to a low-level language) so in this paper, only the term "translation" is used instead of recompilation.

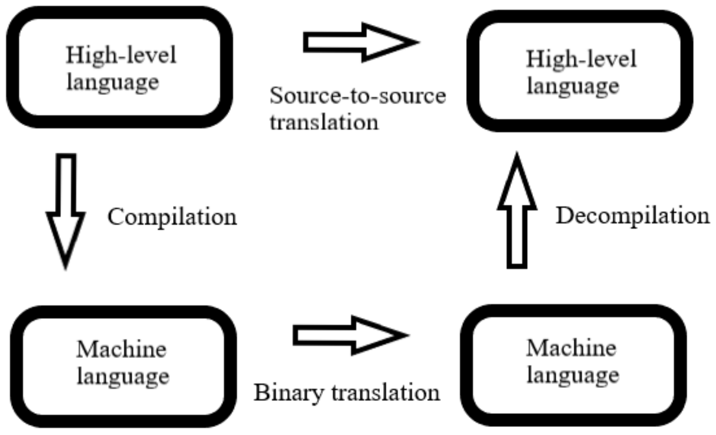


Figure 3. Examples of language translations.

2.1.8 Binary Files and Source Files

In general, a binary file means any file that is not a text file. In other words, a binary file is meant to be read by a computer. In this thesis, a stricter definition of the term "a binary file" is used i.e., any file that only consists of machine code that can be executed by a processor.

Source file means any file that only has code in high-level language so it cannot be executed directly. This also means that the source file must be translated before it can be run on a processor.

2.1.9 Dynamic Translation and Static Translation

Programming language translations can be divided into dynamic and static translation. Dynamic translation can also be called JIT (just-in-time) and it means that translation happens during the run time, examples of this are JIT compilation and an interpreter. Static translation can also be called AOT (Ahead-of-time), and it means that translation happens before the execution, for example, AOT compilation is a static translation method.

2.1.10 Compiler vs Interpreter

A compiler and an interpreter both do the same thing i.e., translate high-level language commands to machine code commands that can be executed on a CPU. As fundamentally a compiler and interpreter do the same thing, the difference between the two can be difficult to define. But in general, the difference between the two is that the interpreter reads one code line at a time and then executes the command with machine language commands that are already part of the interpreter itself, in other words the source code is not changed to another language, whereas a compiler first translates code to machine language and the resulting machine language is then saved to memory. After compilation, the code is then executed directly from the memory. Note that the execution can happen later (ahead-of-time compilation) or at the same time as the compilation is done (just-in-time compilation). Another difference is that a compiler in general translates code in larger parts. Also, a compiler may do optimizations, whereas an interpreter doesn't.

The basic implementation of an interpreter is very simple i.e., it reads one line and then executes the command thus there is no need to cache anything. Implementation of a compiler can be more complicated. Note that the difference between the two is somewhat arbitrary as an interpreter can be thought of as doing compilation one line at a time without doing optimization and then running the "translated" machine code directly without saving the resulting machine code to memory. Figure 4 illustrates the difference between a compiler and an interpreter.

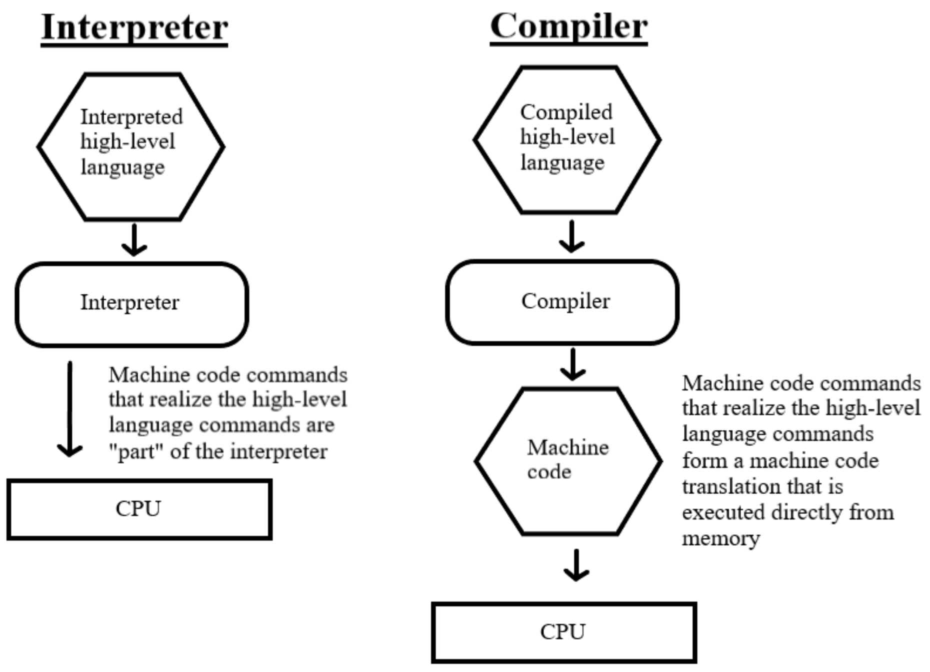

Interpreter
Interpreted high-level language
Interpreter
Machine code commands that realize the high-level language commands are "part" of the interpreter
CPU
Compiler
Compiled high-level language
Compiler
Machine code
Machine code commands that realize the high-level language commands form a machine code translation that is executed directly from memory
CPU

Figure 4. Difference between an interpreter and a compiler.

2.2 Introduction to Cross-platform Software

Every computer architecture can only execute code in its native machine language. This means that every architecture must have its own code and the code cannot be executed on another platform. To get around this limitation, usually the code is written in some high-level language that is later translated to the native machine language.

Methods for translating code to multiple platforms can be divided into two main categories. The first category is compiling the code to the target platform's machine code ahead of time. The second category is having the code in some other language (high or low-level language) than the target platform's machine code and translating it to machine code during run time.

2.2.1 Ahead-of-time Translated Cross-platform Software

The most straightforward method for cross-platform software is compiling code directly to a native machine language. With this method, every architecture will have a different binary. For this method to work the code must be structured in a way that allows it to be easily compiled for different target platforms. In general, this is achieved by conditional compilation i.e., dividing codebase into a platform-independent part and a platform-dependent part [1]. Most of the time the platform-dependent part is relatively small, and this method is easy to do, but if the platform-dependent part is big then every platform might need separate code that provides similar functionality (although this would not really be "cross-platform software"). Also, if a codebase doesn't have a platform-independent part, then it is possible to use different compilers to directly compile binaries from the same common code. Figure 5 illustrates these three possibilities.

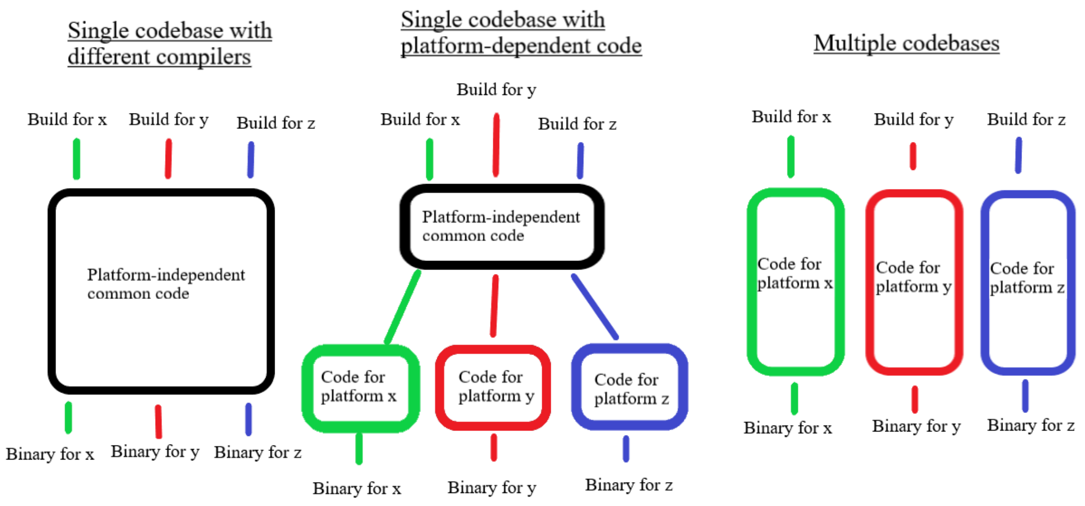


Figure 5. Different structures of a codebase.

2.2.2 *Just-in-time Translated Cross-platform Software*

The second method for implementing cross-platform software is to leave the code in some other language than the target platform's machine language and then translate it during run time. Emulation uses just-in-time translation to execute binaries that were made for a different platform. Also, many programming languages use just-in-time translation. For example, Java has bytecode that is just-in-time compiled or interpreted to target machine language just before the code is executed. All scripting languages use this method for example, JavaScript and Python are translated during run time. These examples are translations from a high-level language to machine code, but the same method works also between different machine languages. For example, an emulator or an ISS (instruction set simulator) translates code from machine code to other machine code. Figure 6 shows examples of how code can be translated starting from high-level languages. Note that all cross-platform software methods rely on some form of translation that will produce machine code for the target platform. The only differences between cross-platform software implementations are the time of translation (dynamic vs static), number of translations, translation methods, optimizations, and abstractions.

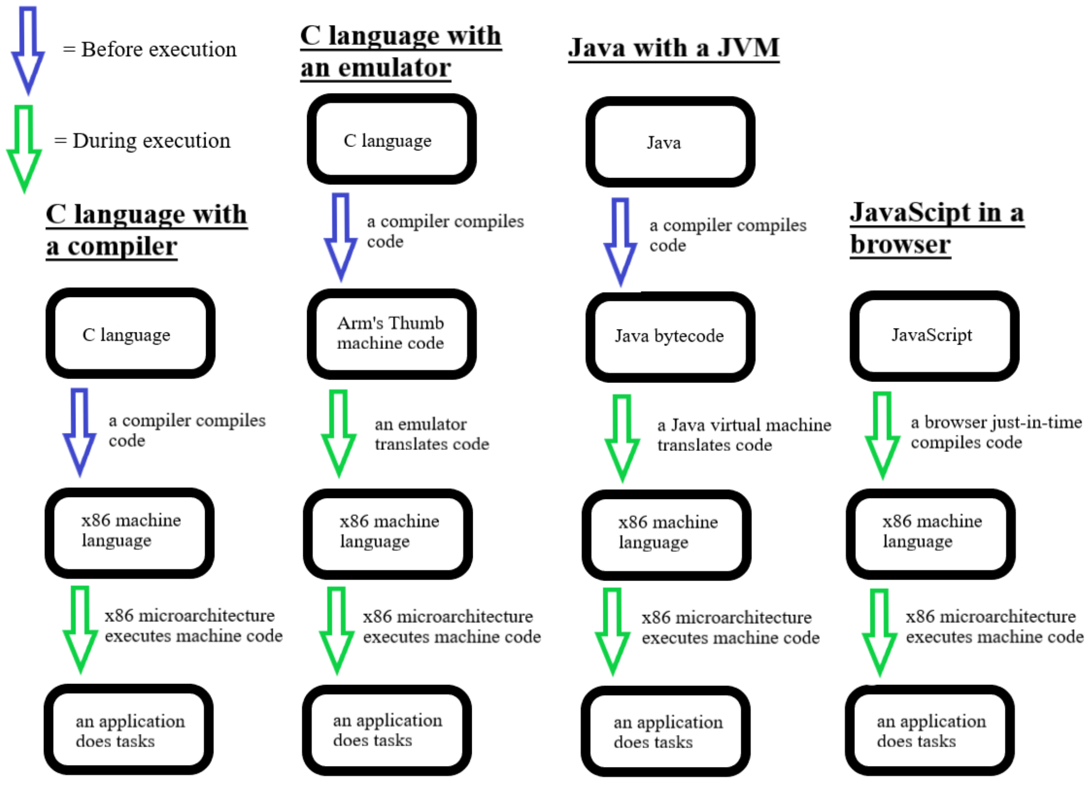

= Before execution
= During execution
C language with a compiler
C language
a compiler compiles code
x86 machine language
x86 microarchitecture executes machine code
an application does tasks
C language with an emulator
C language
a compiler compiles code
Arm's Thumb machine code
an emulator translates code
x86 machine language
x86 microarchitecture executes machine code
an application does tasks
Java with a JVM
Java
a compiler compiles code
Java bytecode
a Java virtual machine translates code
x86 machine language
x86 microarchitecture executes machine code
an application does tasks
JavaScipt in a browser
JavaScript
a browser just-in-time compiles code
x86 machine language
x86 microarchitecture executes machine code
an application does tasks

Figure 6. Examples of how high-level languages are translated to machine code instructions.

2.2.3 *Intermediate Representations*

One way of doing cross-platform software easier is to use an intermediate representation (IR) when compiling software. An intermediate representation is a low-level language that can be used when translating from a high-level language (front-end) to machine code (back-end). When doing compilation with an IR, first a high-level language is translated to an intermediate representation and after that, the intermediate representation is translated to machine code. An IR makes it easier to add support for new front-end languages and back-end languages because only translation between the IR and the new language is needed. Also, an IR makes optimization easier, as optimization can be done on the IR and the same optimization can be done regardless of what front-end or back-end language is used.

An IR can be used in emulators or compilers. An example of an IR compiler is the LLVM (low level virtual machine). It can translate from many front-end languages (e.g., C, C++, Rust, Fortran, etc.) to the LLVM IR and from the IR it can translate to many back-end languages (e.g., X86, X86-64, PowerPC, ARM, Thumb, RISC-V, etc.). Figure 7 shows how the LLVM’s front-end, IR, and back-end are related to each other.

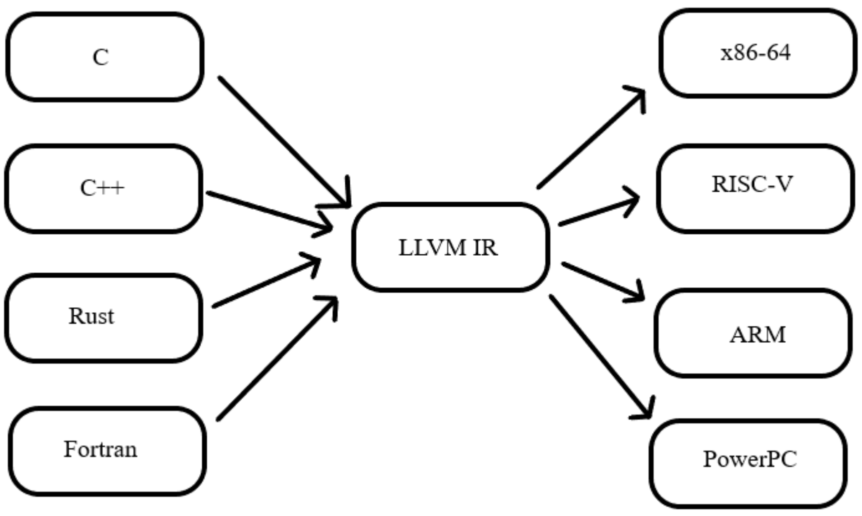


Figure 7. Illustration of LLVM toolchain's compilation process.

2.3 Instruction set architectures

An ISA (instruction set architecture) is an abstract model of the CPU (central processing unit). An ISA can also be called "computer architecture". An ISA defines what instructions, registers data types, addressing modes, memory consistency, etc. a CPU must support, and thus it defines what kind of machine code the computer platform uses. With an ISA abstraction, it is possible to have different implementations of the computer architecture from different manufacturers in a way where the same software binaries can be executed on the different manufacturers' machines. Implementations of ISAs are called microarchitectures. For example, AMD and Intel have different microarchitectures but it is possible to execute the same binaries on them because they both provide the same ISA abstraction.

An ISA layer can also be thought of as an interface between software (machine code) and hardware (microarchitecture). An ISA is needed to make sure that software and hardware are compatible with each other. Today there are many different ISAs in use, examples of these include x86, MIPS, PowerPC, ARM, RISC-V, TriCore, etc. Note that the term "instruction set" means almost the same as ISA, but the difference is that instruction set only means instructions whereas ISA means instructions and other things in computer architecture.

ISAs can be classified into sub-categories. The most common categories are CISC (complex instruction set computer) and RISC (reduced instruction set computer) architectures. And then there are some rarer categories, for example, VLIW (very long instruction word). In CISC architecture instructions are more complex and one instruction can do many operations. In contrast, RISC architecture's instructions are simpler and can only do one operation per instruction. RISC architecture also typically has more registers and is load-store architecture i.e., memory accesses must use their own instructions and most of the other instructions only use registers.

Because RISC architecture is simpler, it usually uses less energy and therefore it is commonly used in embedded and mobile devices.

Note that execution of the commands defined by an ISA is a complex process. The CPU's microarchitecture does many different things when executing the instructions. For example, a CPU might use microcode to "interpret" ISA commands before executing them to make the implementation of complex instruction sets easier. If microcode is used it can be thought that there is an interpreter implemented in the hardware that interprets ISA instructions into a simpler form.

2.3.1 x86 Architectures

The x86 is a backward compatible family of CISC ISAs that are used in modern PCs. The first version of x86 was introduced in 1978 for Intel's 8086 microprocessors. The first x86 ISA was only 16-bit architecture. Then in 1985 a 32-bit extension of the x86 ISA was introduced with the 80386 microprocessors. The 32-bit architecture can be called IA-32, i386, or x86-32. In 1999 AMD introduced a 64-bit extension of the x86 architecture that is called x86-64 and later Intel also started using the same x86-64 architecture.

Over the years x86 has had many new extensions like MMX, SSE1, SSE2, SSE3, SSSE3, SSE4a, SSE4.1, SSE4.2, AES, CLMUL, and SHA; these extensions are instruction sets that add new instructions but don't change existing instructions, so backward compatibility has always been preserved. X86 is the common term for all these architectures regardless of if they are 16-bit, 32-bit, 64-bit, or regardless of what combination of the extensions they support.

The long history of x86 with preserved backward compatibility makes x86 unnecessarily complicated. For example, in x86-64 the same accumulator register is mapped as 5 different registers: 8-bit AL (accumulator low), AH (accumulator high), 16-bit AX (accumulator extended), 32-bit EAX (extended accumulator extended), and 64-bit RAX (register accumulator extended). The lower bit versions of the register are mostly needed for backward compatibility. Figure 8 shows an example of an x86 register. The complexity of the x86 can be easily seen.

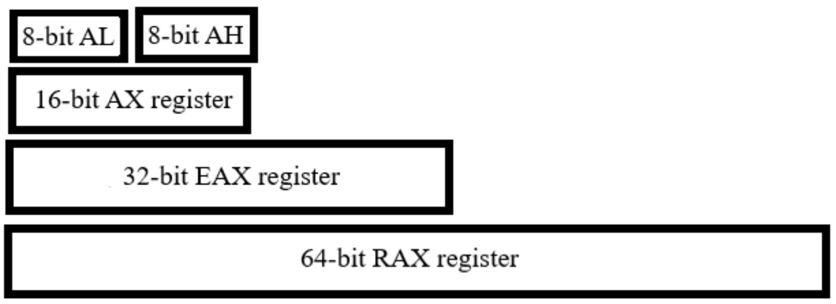


Figure 8. Example of x86-64 accumulator register mapping

2.3.2 ARM Architectures

ARM has a family of RISC ISAs that has low power usage and low price; therefore, ARM processors have become popular in smartphones, embedded devices and as part of SoC (system-on-chip) designs. As of 2021, more than 180 billion ARM chips have been produced, thus making ARM architecture the most common architecture by a wide margin [2].

The ARM architecture has many different ISAs that are for different use cases and in general ARM ISAs are backward compatible inside the same profiles. Even though the ARM ISAs are backward compatible the other parts of embedded devices (e.g., peripherals) are usually not backward compatible, so usually low-level embedded devices need binaries that cannot be used in other target machines. More general devices like smartphones can have better backward compatibility especially if they use a HAL (hardware abstraction layer).

The categorization and naming convention of the ARM ISAs is somewhat complicated. Examples of ARM ISAs are ARMv1, ARMv6-M, ARMv7-M, ARMv7-A, ARMv7-R, ARMv8-R, and ARMv8.2-A. The architectures are divided into different profiles that tell the designed use case of the ISA. The last letter of the ISA's name tells the profile of the ISA. For example, -A (application profile) is for powerful general-purpose systems with an MMU (memory management unit). The A profile is used in many devices, for example, smartphones use the A profile. -R (real-time profile) is for high-performing processors that contain an MPU (memory protection unit) and are used in safety-critical environments. While finally -M (microcontroller profile) is for microcontrollers, and it has good support for interrupts, good integration possibility into an FPGA, and it is designed for very low power usage.

Microarchitectural implementations of the ARM ISAs are called cores. For example, core names starting with Cortex-A (e.g., Cortex-A57) are either ARMv7-A or ARMv8-A architectures, and cores starting with Cortex-M (e.g., Cortex-M3) are either ARMv7-M or ARMv8-M architectures. Older legacy cores don't have a profile and don't use the name "Cortex" e.g., ARM60 core uses ARMv3 architecture. Table 1 shows examples of ARM cores.

Table 1. Examples of ARM cores

Architecture	**Core bit width**	**Core**	**Profile**
ARMv1	32 bits	ARM1	Classic
ARMv3	32 bits	ARM6 ARM7	Classic
ARMv6-M	32 bits	ARM Cortex-M0 ARM Cortex-M0+ ARM Cortex-M1	Microcontroller
ARMv7-M	32 bits	ARM Cortex-M3	Microcontroller
ARMv7-A	32 bits	ARM Cortex-A5 ARM Cortex-A7 ARM Cortex-A8	Application
ARMv7-R	32 bits	ARM Cortex-R4 ARM Cortex-R5 ARM Cortex-R7	Real-time
ARMv8-R	32 bits	ARM Cortex-R52	Real-time
ARMv8.2-A	64/32 bits	ARM Cortex-A55 ARM Cortex-A75 ARM Cortex-A76	Application

2.3.3 ARM Instruction Sets

ARM architectures use different combinations of instruction sets and extensions depending on the architecture version and implementation. The most important instruction sets are regular 32-bit ARM instruction set, 64-bit AArch64 instruction set, and thumb instruction set.

The 32-bit ARM is a basic ARM instruction set that is used in ARMv7 and older architectures. ARMv8 architectures also has a mode for 32-bit instruction set called AArch32 that is almost the same as the ARMv7's 32-bit instruction set but has some extra instructions. Also, in ARMv8 the AArch32 profile is optional, so an ARMv8 device might not support 32-bit code at all, depending on the implementation.

The 64-bit ARM instruction set, AArch64 was introduced for ARMv8 architecture. The AArch64 is optional for the ARMv8, so it is also possible to have ARMv8 with only 32-bit support. AArch64 architecture is not backward compatible with the 32-bit architecture (although like mentioned earlier there is an optional mode for AArch32 that is compatible with older 32-bit instructions).

On top of the basic 32-bit ARM instructions set, the 32-bit ARM architecture has a second instruction set called Thumb. RISC architectures requires a lot of instruction to do tasks (CISC would need fewer instructions to do the same tasks) which leads to high memory usage. Thus, to combat this high memory usage the Thumb instruction set was developed. The Thumb instructions are only 16 bits long and therefore need less memory. Each Thumb instruction has a corresponding 32-bit ARM instruction that does the same action, so it is possible to do the same things with regular ARM instructions as with Thumb instructions. The first version of the Thumb only had 16-bit instructions which was later extended with some 32-bit instructions. The extension is called Thumb-2.

Machine code is interpreted differently in regular ARM and Thumb e.g., machine code 0x020091E0 is interpreted as "adds r0, r1, r2" in normal ARM, but it is interpreted as "movs r2, r0 and b #0x128" in Thumb ("adds r0, r1, r2" in Thumb is 0x8818). The processor knows which one to use by a mode that tells is ARM or Thumb used currently. The mode can be changed with the least significant bit of addresses e.g., the `BX` instruction changes the mode depending on is the least significant bit of the address 0 or 1. Some cores like Cortex-M only uses the Thumb instruction set and cannot execute normal ARM instructions at all.

3 CHALLENGES IN EMBEDDED PLATFORM SIMULATION

Executing code made for an embedded device on another platform is a complicated problem that does not have one solution that works for all use cases. Many techniques to solve the problem have been developed, but all have some shortcomings. This chapter explains why executing embedded software on a pc is such a challenging problem and what differences there are between a PC platform and an embedded platform.

3.1 Embedded Device Classification

Embedded devices are an extremely diverse class of computers and usually a binary made for one target can only be run on the target hardware. There are many ways how embedded devices can be classified, but from the point of cross-platform code execution, it makes sense to divide the devices based on the complexity of the OS (operating system). The paper "*What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices*" introduces a good classification based on the OS complexity [3]. This thesis uses the same classification. In this classification, the devices are divided into three classes: type-I general-purpose OS-based devices, type-II embedded OS-based devices, and type-III devices without an OS-abstraction.

Type-I devices are the most powerful and thus they have a proper OS. For example, devices in this class may have a full Linux kernel and OS. This means that the OS and the application have separate code and binaries.

Type-II devices are mid-level devices that use a simpler OS. The OS is made for embedded devices, but the application and the OS are still separated.

Type-III devices are the least powerful devices which don't have a separate OS. In a type-III device, the OS and the application are compiled into the same binary. Software for type-III devices is also called monolithic firmware. The OS used in a type-III device is usually as simple as possible providing only a minimal amount of functionality. For example, the OS can be a simple task switcher that changes tasks periodically.

Note that in this master's thesis only a type-III embedded device is considered during the implementation. Hence, most of this master's thesis is written from the point of view of type-III device simulation.

3.2 Differences Between Computer Platforms

Computer platforms have many different designs that make it impossible to make binaries that would work on all the individual platforms directly. This section introduces some of the issues that can cause problems when running the same software on different platforms. This section is mainly written from the point of view of trying to run embedded software on a PC platform.

3.2.1 Computer Architecture (Instruction Set Architecture)

The most important difference between computer platforms is the instruction set architecture. A PC uses an x86 ISA, whereas embedded devices usually use RISC

ISAs like ARM, PowerPC, MIPS, or SPARC. This means that the binary files for different platforms use completely different instructions thus it is mandatory to translate them at some point if the same binary is executed on multiple platforms.

3.2.2 Hardware Accelerators

Embedded devices usually aren't powerful, so sometimes they have hardware accelerators to speed up certain tasks. The hardware accelerators are specialized electronic circuits designed using HDL (hardware description language) and many times they cannot be changed after the chip has been produced. They are designed to only do one thing. For example, an embedded device can have an AES (advanced encryption standard) accelerator that can perform AES encryptions and decryptions faster than AES implemented with software. PCs can also have some task-specific accelerator components like GPU (graphics processing unit), but GPUs are different from an embedded system's accelerators because a GPU has a lot more complexity, and it can be programmed to do different things. For a simulation environment, hardware accelerators can be difficult to simulate, but it must be done somehow if the code uses hardware accelerators.

3.2.3 Endianness

Another difference in computer platforms is endianness. Endianness means the order of bytes in a word. Or in other words, it tells in what order bytes will be read/written in word-sized accesses to memory. Endianness can be little-endian or big-endian. A little-endian system stores the least-significant byte at the smallest address, and a big-endian system stores the most significant byte at the smallest address. Figure 9 shows the difference between little-endian and big-endian systems. A PC uses little-endian byte ordering, whereas an embedded device can use little-endian or big-endian byte ordering.

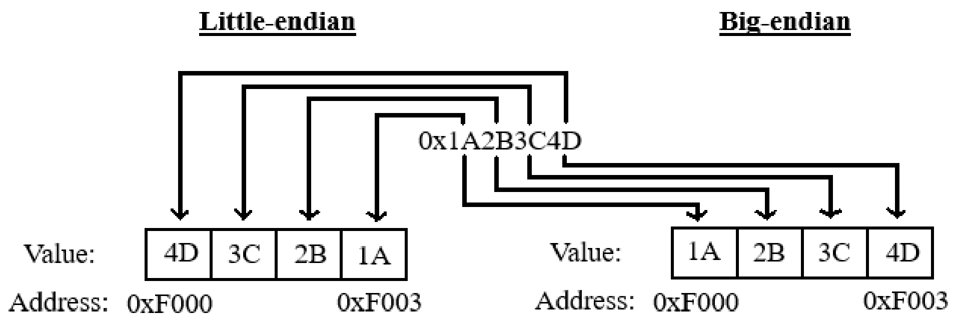


Figure 9. Example how a 32-bit integer is stored in little-endian and big-endian systems.

3.2.4 Interrupts

Interrupts are events that tell a processor that it needs to temporarily stop the current execution of the code and change code execution to a handler that will handle the triggered interrupt. Interrupts can come from three places i.e., from the hardware, the software, or the processor itself. Interrupts from hardware are asynchronous (they can arrive at any time regardless of what the processor is doing). For example, a timer, a keyboard, or a mouse can send asynchronous interrupts to a processor. Interrupts from software and processor are synchronous interrupts i.e., they come from the processor, and they happen at the same interval with the processor's clock cycle. Interrupts from a processor are called exceptions and they happen if something is wrong in the execution of the instruction e.g., division by zero or illegal opcode. The last interrupt type is software interrupts where the interrupt is invoked by an instruction. These interrupts are mainly used for system calls. For example, in x86 INT and syscall instructions invoke interrupts. Different platforms use different instructions and have different peripherals so naturally they will have different interrupts and interrupts must be handled somehow in cross-platform code execution.

3.2.5 Operating Systems

Different operating systems provide many kinds of services such as system calls, virtual memory, context switching, etc., and in general, binaries only work in one operating system. PCs usually use Windows or Linux as the OS. Whereas embedded systems can be type-I (e.g., Linux), type-II, or type-III (monolithic firmware with a simple OS) Type-I embedded devices are more similar to a PC environment so software for a Type-I device can be easier to run on a PC, as it might be enough to compile the application directly for a PC's OS or use emulator's user-mode emulation. But for a type-III embedded device, the whole monolithic firmware binary that includes an OS and an application must be run on a PC, or the behavior of the embedded device's OS must be replicated somehow in the PC.

3.2.6 Memory and Memory Maps

There are two ways how CPUs can access peripherals. They are memory-mapped I/O (input/output) and port-mapped I/O. Port-mapped devices have a separate address space for the peripherals and the peripheral addresses cannot be accessed directly from the code. Instead, port-mapped devices access peripherals with special instructions e.g., IN and OUT instructions in the x86 architecture are used for I/O operations. Memory-mapped devices expose all the resources on the same memory map so that they can be used directly from the code. Or in other words, peripheral's registers have addresses that can be read or written directly with C language (or other languages) the same way as RAM. PC's x86 architecture uses port-mapped I/O and most embedded devices use memory-mapped I/O.

Resources on a memory map can be different kinds of memories, registers, I/O, etc. As platforms have different peripherals, naturally memory maps between systems will also be different. Different memory maps can make platforms incompatible with each other even if the platforms use the same ISA. This is because memory addresses are used directly from instructions and thus binaries won't work on other platforms that have different memory maps.

A PC's Memory Map

A PC uses port-mapped peripherals which means that user address space is reserved for the application and most peripherals are accessed through kernel space with system calls. A PC has many processes running at the same time. To separate them every process has its own address space, and the kernel has its own address space. This separation is done with virtual memory. Virtual memory can be thought of as an abstraction that makes it seem that the process is the only one using the system. Virtual memory is implemented with the OS and MMU and virtual memory addresses don't directly map physical resources (e.g., register, flash memory, etc.). Many modern systems also use ASLR (address space layout randomization) that randomly arranges addresses for the stack, the heap, etc. every time the program is executed thus making the virtual memory map random. This means that hardcoded addresses are not really used on a PC, whereas on an embedded system hardcoded addresses are common.

Embedded Systems' Memory Maps

Embedded systems have complicated memory maps. Many embedded systems don't have virtual memory, and this means that system uses physical memory addresses directly from the code. The memory map in an embedded system is divided into different address ranges that correspond to different resources. Most of the addresses are empty and accesses to them will cause a bus error. Other addresses can have accesses to registers, Flash memory, ROM (read-only memory), RAM (random-access memory), OTP (one-time programmable) memory, hardware accelerators, I/O, etc. Memory addresses can be read-only, write-only, or allow both.

Most embedded systems have a lot of registers that can be used to configure the chip or to read information about the system. Registers are also used in I/O operations, and they can also be used to control different tasks like writing to Flash memory. Most embedded system chips have different memory maps thus making them a diverse group of devices that cannot execute the same binaries.

On some embedded devices the code is run directly from flash whereas some devices may be similar to a PC i.e., code is first copied to RAM and then executed from RAM. These different memory types e.g., Flash, ROM, RAM, OTP memory, and registers can be challenging to simulate when running embedded software on a PC, but in many cases, it may be sufficient to simulate all memory types as RAM. However, this simplification would not give the same behavior as when running on a real embedded system e.g., writes to read-only memory would behave differently in the simulation compared to the embedded system.

Another challenge for a simulation is that in a real embedded system writes to Flash memory use their own complicated logic circuits. Typically, these writes are controlled by registers and the methods of writing to Flash memory can be completely different between devices. Embedded devices might also use ECC (error correction code) memory that is not typically used in a PC meaning that the real target hardware can have ECC errors that can be difficult to replicate on a PC. Figure 10 shows examples of PC's virtual memory and a memory map of an embedded device that doesn't use virtual memory. Note that on a real embedded device there would be a lot more memory areas than in the figure; especially there would be a lot of registers that could be in any location of the memory map.

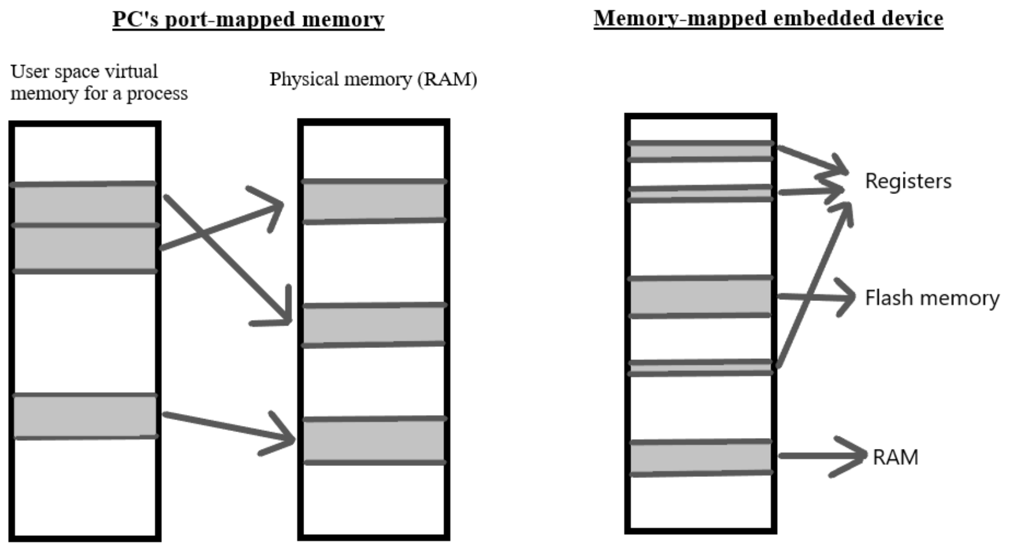

PC's port-mapped memory
Memory-mapped embedded device
User space virtual memory for a process
Physical memory (RAM)
Registers
Flash memory
RAM

Figure 10. Examples of user-space memory maps on a PC and an embedded system.

4 EMBEDDED PLATFORM SIMULATION TECHNIQUES

There are countless ways to run embedded software on a PC environment. In this chapter different techniques of how an embedded system can be simulated on a PC are introduced. In this master's thesis, it is assumed that hardware has been designed beforehand and that it functions correctly. Therefore, only testing of embedded system's software is discussed. Especially, the ways of executing unit and integration tests on a PC are introduced, although the same simulation techniques can also be used for other types of testing like fuzz testing.

4.1 Fidelity

When deciding how to simulate an embedded system it is important to decide how accurate the simulation must be. This degree of accuracy is called fidelity. High fidelity means that the simulation simulates accurately the low-level details of the real hardware while low fidelity means that only the high-level behavior of the simulator is similar to the real hardware, but low-level details are not the same.

Different parts of the simulated system may have different levels of fidelity. That means that the instructions (CPU), the memory, and other parts of the system can be simulated with different levels of detail. For example, in a game console emulator the instructions might be emulated with high fidelity, but the sound card might only be emulated on very low fidelity e.g., externally the simulated sound card behaves the same as the real target system while internally the implementation is completely different.

In this paper, the same categorization of instruction fidelity is used as in the paper "*Challenges in Firmware Re-Hosting, Emulation, and Analysis*" [4]. In the paper's categorization, there are seven levels. The levels are black box, module, function, basic block, instruction, cycle, and perfect instruction fidelity.

Black box fidelity means that the system behaves similarly to the real system, but the internal implementation is completely different from the real target system. Module fidelity means that internal implementations of the modules are made differently than in the real target system e.g., the OS could be implemented differently. Function level fidelity means that functions will be accurately simulated, but functions' internal implementations may be different in the simulation than in the real target system. Basic block fidelity means that every basic block will be simulated, but internally they may execute different instructions than in the real target system. Instruction fidelity means that all instructions are executed in the simulator without skipping or combining instructions. Cycle level fidelity means that CPU's instruction cycles are simulated meaning that timings will be similar to the real target system. Perfect fidelity means that a simulator behaves the same way as the real target system would and all details are simulated. Table 2 shows all levels of fidelity.

Table 2. Different fidelity levels of a CPU simulation

Fidelity level	**Description**
Black box	Low-level details are not simulated, only high-level behavior is similar to the real system
Module	All modules are simulated, but internal details of the modules are not simulated
Function	Functions work the same way as in the real system, but internal implementation can be different
Basic block	All basic blocks will be simulated, but they can be executed differently than in the real system
Instruction	All instructions are simulated on the simulator, but they can have different timings
Cycle	CPU's cycles are simulated and thus timings will be correct
Perfect	Simulator behaves exactly the same way as the real system

When testing embedded software in a simulator, most details of the code should be preserved, so the fidelity should be at least basic block level or instruction level for the majority of the simulation. But for some parts of the simulation, it can be enough to only have module or function level of fidelity. For example, an embedded system's OS could be replaced with another OS module for a PC and some functions could be replaced with some other functions to make simulation easier on the PC. Even if some modules/functions are replaced other parts of the code should be simulated at the basic block/instruction level fidelity. If hardware is also tested at the same time as software i.e., hardware/software co-design, then simulation's fidelity might need to be on the cycle level.

4.2 Emulation

Emulation is a common way to run binaries that are compiled for another target system than the host system. An emulator must be able to replicate every hardware component of the target system to be able to execute the binaries completely. Emulators are used often to emulate old game consoles and they are also used to run application binaries that were compiled for different systems e.g., an emulator can execute embedded system's binaries on a PC.

There are many things that emulators have to simulate. The two most important are the simulation of the CPU and the memory map. An emulator could even be described as a program that does translations on top of a memory map. In addition to translation and a memory map, an emulator should simulate an MMU, timers, interrupts, etc. Hardware accelerators and registers are part of a memory map and the behavior of these should be imitated with software when the registers' addresses are accessed, although in some situations (e.g., in fuzz testing) it is not necessary to simulate all behavior because many registers don't affect the test results in a meaningful way and the registers only have use in real hardware. In other words, the needed fidelity of the emulation depends completely on what is the goal of the testing.

4.2.1 Translation Techniques of Emulators

There are many ways how translation can be done on an emulator. In this subsection, common ways of implementing translation are discussed.

The easiest way of implementing a CPU emulator is with an interpreter. An interpreter can have a very basic structure (e.g., a switch statement that has cases for every instruction that the target architecture supports), and every case implements the target instructions with host instructions so that the behavior will be similar. In practice, this means that the emulator needs to execute multiple host instructions to emulate one target instruction. The emulator must also keep track of the emulated CPU's state in software because the instructions' results depend on the CPU's internal state (e.g., state of the registers). [5]

Another common way of emulating a CPU is with dynamic binary translation. Dynamic binary translation is a similar concept to JIT compilation, however as the translation happens between binaries the term "JIT compilation" should not be used. In "dynamic binary translation" the translation is done on a basic block level and some optimization can also be used. So, the resulting host code will need fewer instructions to execute a basic block than an interpreter would require. The drawback of this method is that the translation process might need more clock cycles than an interpreter would, however the translated blocks can be cached. Caching allows that a basic block only needs to be translated when the basic block is executed for the first time. And because the majority of programs spend most of the execution time to execute the same basic blocks in loops, therefore dynamic binary translation is faster than interpreters in most cases. Although an interpreter might be faster in parts of the code that are executed only once. [5]

The third method for CPU emulation is threaded code. Threaded code is somewhere between an interpreter and dynamic binary translation. Threaded code is not that common nowadays [5] so details about this concept are skipped in this thesis.

Some emulators may use a combination of the techniques e.g., an emulator might first use an interpreter when a basic block is executed for the first time, but if the basic block is executed multiple times the emulator will utilize dynamic binary translation and cache the results. In this way the emulator gets the best sides from both techniques i.e., fast speed when executing code that is rarely needed and fast speed for code that is executed many times in loops.

It's good to note that all these emulation methods are fundamentally doing the same thing i.e., translation. Only the implementation of the translation varies between the different methods.

Static Binary Translation

Static binary translation is another way of doing CPU "emulation" [5]. The idea behind static binary translation is similar to dynamic binary translation. The difference is that the whole target's binary is translated before execution and saved as a new host binary. Because static binary translation can use the same binaries and compilers as target hardware, and there is no need to install an emulator or a virtual machine, static binary translation could be considered the best method for cross-platform CPU "emulation". However, because of a variety of problems (e.g., recognition of data and code areas, self-modifying code, etc.), static binary translation is difficult to implement. Therefore, it is not a practical method for embedded software simulation. There are only some cases where static binary translation has been done successfully e.g., StarCraft port to the ARM architecture and Super Mario Bros port for the x86 architecture [6].

4.3 Existing Emulators

There are many emulators which can be used to execute embedded software. Some of these emulators are open source while others are commercial emulators. In this section emulators QEMU, Unicorn, OVPsim, Avatar2, VPSim, and Simics are introduced. Countless other emulators and instruction set simulators could also be used in the emulation, but most of them have drawbacks e.g., no support for peripheral emulation.

4.3.1 QEMU

QEMU (Quick EMUlator) is free and open-source software that can emulate computer systems and do virtualizations. QEMU offers many different modes of operation. They are user-mode emulation, full-system emulation, and virtualization [7]. The user-mode emulation can emulate individual Linux programs that were compiled for a different ISA e.g., ARM Linux binary can be run on an x86 Linux. In the virtualization mode, QEMU can run KVM and Xen virtual machines. With the full-system emulation mode, QEMU can emulate the whole target system including peripherals, a memory map, timers, etc. In the full-system emulation mode, QEMU can run whole operating systems or bare-metal applications. For example, with the full-system emulation mode, it is possible to run a binary that is compiled for an ARM embedded device.

The benefit of QEMU is that it uses dynamic binary translation for CPU emulation and thus it achieves good performance. As QEMU uses dynamic binary translation, the QEMU's instruction fidelity is on the basic block level.

The problem using QEMU for embedded software emulation is that embedded devices are a very diverse class of devices and QEMU only supports a really small set of embedded devices. However, as QEMU is open source it can be modified to support new devices. Unfortunately, QEMU's codebase is complex making it difficult to add custom peripherals. For example, adding a small ROM area to the memory map might need changes to 9 files of which around 90 lines of code have to be modified [8].

QEMU uses tiny code generator (TCG) to do the translations between different languages. The tiny code generator has a front-end and a back-end. The front-end translates the target language (e.g., ARM) to a TCG microcode that is an intermediate representation language. Then the back-end translates the microcode to a host language (e.g., x86). The intermediate language increases portability and thus it is easier to add new front-end and back-end languages to QEMU. [9]

4.3.2 QEMU and SystemC

SystemC is a set of C++ classes and macros which can be used to make an event-driven simulation of the target system. SystemC can be used alone, or it can be used together with an emulator like QEMU to simulate the whole embedded system. If used with QEMU, SystemC can simulate peripherals and possibly some communication busses. While QEMU can simulate the CPU and possibly some of the peripherals. There are multiple ways how this can be done e.g., QEMU can be a SystemC module that emulates CPU and SystemC does other hardware simulations, or SystemC modules can be directly connected into QEMU as the peripherals [10]. The implementation of a SystemC and QEMU simulation can be difficult and time-consuming especially if the developers don’t have previous experience with SystemC. If the purpose is to only test software, SystemC might make an unnecessarily detailed simulation of the hardware. Nevertheless, dual QEMU and SystemC simulations have been implemented in countless papers and they can be useful simulation methods [9, 10, 11, 12, 13, 14, 15].

4.3.3 QEMU Forks

QEMU has readily available only a limited number of embedded boards that it can emulate. But as QEMU is open source many developers have done their own forks from QEMU, adding new embedded device emulation models. So, if QEMU doesn’t have a specific embedded device model, support for it may exist in some fork. For example, “The xPack QEMU Arm” [16] is a QEMU fork that has better support for bare metal Cortex-M boards than the normal QEMU. It also has readily available debugging support with Eclipse, which is important if the simulation is used in software development. Often real-world ARM designs have custom parts like specific hardware accelerators, thus in these cases even “The xPack QEMU Arm” doesn’t support everything that is used in an ARM chip, and as it is based on QEMU it would be difficult and time-consuming to add custom peripherals.

4.3.4 Unicorn

Unicorn [17] is a CPU emulator framework that is based on QEMU. It uses the same “tiny code generator” to do dynamic binary translation as QEMU, but almost everything else is removed. This means that the Unicorn emulator only emulates the CPU and the memory map. Unicorn can run any raw binary code without any context i.e., it doesn’t need a whole system image as QEMU would need.

Unicorn is written in C-language, but it has bindings to many other languages e.g., Python, so it can be used from many languages directly. The support for good languages like Python makes Unicorn easy to use. Unicorn supports instrumentation and hooks for different memory accesses like execution, read, and write accesses, which makes it easy to add custom peripherals to the memory map. For example, it is possible to add an AES accelerator to some address by adding a hook to it and then implementing the functionality of the AES accelerator with a Python function.

With Unicorn, it is possible to dynamically change a firmware's function with a function that is implemented on Unicorn. This can be done by intercepting the firmware's function when it is executed and then executing a replacement function on the Unicorn and after the replacement function is finished the execution continues without executing the firmware's function. This feature can be useful when emulating complex hardware behavior like a Flash memory write module. For example, a function that controls the Flash controller can be intercepted, so that the logic for Flash memory writes doesn't have to be implemented on the memory map.

There are two versions of Unicorn. Unicorn 1 was released in 2015 and it was based on the 2015's version of QEMU. Unicorn 1 got new updates throughout the years, but it was never updated with the new changes that were introduced to the QEMU mainline. Unicorn 2 was released in October 2021. Unicorn 2 is written from scratch based on the new QEMU version (QEMU 5) and Unicorn 2 is also backward compatible with Unicorn 1.

4.3.5 OVPsim

OVPsim (Open Virtual Platform) [18] is a multiprocessor emulator that can run production binaries that were compiled for the target embedded system. OVPsim is free for non-commercial usage but requires a license for commercial usage. OVPsim has free models for different processors, platforms, and peripherals. For example, there are processor models for ARM, MIPS, PowerPC, RISC-V, etc. It is also possible to create one's own models for processors and peripherals with OVPsim's APIs. As OVPsim is a multiprocessor emulator, OVPsim can be used to emulate multiprocessor systems. For example, it has been used to simulate parallel computing platforms [19] and hardware/software co-designs [20].

4.3.6 Simics

Simics [21, 22] is a full-system simulator that can be used to run embedded software in a simulation environment. Simics can emulate a wide range of ISAs like Alpha, x86-64, ARM, MIPS, PowerPC, etc. It can also simulate other parts of the system with models. Simics can run code in forward and backward directions and thus it is possible to do reverse debugging. Simics has been used to simulate a multitude of different embedded systems, for example, a system used in a satellite [23] and many other use cases [24].

4.3.7 Avatar2

Avatar2 [25] is a multi-target dynamic analysis framework for embedded system's firmware that can be used to execute and analyze embedded system's binaries. Avatar2 isn't an emulator on its own rather it uses other emulators like QEMU and

Unicorn as the modules that do the actual emulation. Avatar2 can also connect the emulators to dynamic analysis tools and provide external memory representation with the possibility to add custom peripherals. Additionally, Avatar2 can dynamically change the target during code execution i.e., an emulator might run most of the code but when peripherals are accessed, the state of the execution is transferred to the real embedded device and after that, the emulator can continue the execution. With this capability, it is easier to execute and analyze complex embedded systems, because there isn't a need to emulate all peripherals. The bad side of this transfer between the emulator and target hardware is that it slows down the code execution significantly. Better execution speed can be achieved if Avatar2 is configured to not use the real embedded device but instead to use Avatar2's peripheral models that may be written with Python.

4.3.8 VPSim

VPSim [14] is an emulation framework that uses QEMU and SystemC. VPSim can be used to make fast prototypes of the hardware. VPSim executes all CPUs (emulated with QEMU) and peripherals in a SystemC context. This means that all accesses to peripherals are visible to the SystemC which makes debugging and profiling easier. The difference between VPSim and other QEMU/SystemC frameworks is that VPSim is only compiled once and after which it can be configured with a dynamic front-end. The Front-end uses Python and it can be used to easily define the system that will be emulated. Also, the front-end uses XML to communicate the configured platform to the back-end VPSim, so it is possible to change the Python front-end to some other front-end language/application. VPSim has good debugging support with GDB and "VPSim Monitor" that can be used to inspect the executed code.

4.3.9 Other Emulators

There are also countless other emulation methods. In this subsection, some of those are introduced. Many of these emulation techniques do not emulate the peripherals precisely. Instead, they stub them such that the emulation doesn't stop thus these methods don't give the same behavior as the real embedded device. An example of this kind of emulation is P2IM [26]. P2IM automatically models the I/O behavior of the peripherals. P2IM only models the behavior of the registers and considers the actual peripheral as a black box (the actual peripheral behavior is not modeled at all). This means that the behavior of the peripheral is not the same as in the real embedded device, this however allows that the code execution will not stall or crash. With this kind of emulation, you can get most (around 80%) of the target system's test cases passing. This kind of emulation is not sufficient for this thesis, as the purpose of this thesis is to get all unit tests to pass. But P2IM may be enough for other testing e.g., fuzz testing.

Another example of an embedded system emulation is FIRMADYNE [27] it can emulate a type-I embedded firmware by extracting a file system from the firmware and then using a pre-built Linux kernel to execute the application on QEMU.

4.4 Partial Emulation

Another way of simulating embedded systems is partial emulation. Partial emulation is almost the same as normal emulation, but the difference is that in full system emulation binaries compiled for the target hardware can be used directly, whereas in partial emulation the software is changed in a way that makes it easier to run on an emulator. The easiest way of changing software is to do it before compilation. But if it is not possible to edit the code before compilation e.g., only binary is available. It is still possible to replace functions dynamically as long as the emulator supports it. In this case the emulator needs to know the start address of the function to know when the function is called and needs to be replaced.

An example of partial emulation is an emulator that doesn't have a specific hardware accelerator and therefore the hardware accelerator is then replaced with a software implementation that provides the same operation as the hardware accelerator. This kind of simulation has a module or a function level fidelity for the replaced part and higher fidelity (e.g., basic block or instruction level) for the other modules. Figure 11 shows an example of how an AES accelerator can be simulated with partial emulation and full system emulation. In the example, full system emulation can use the same binary as target hardware while partial emulation requires a binary that has a software implementation for the AES.

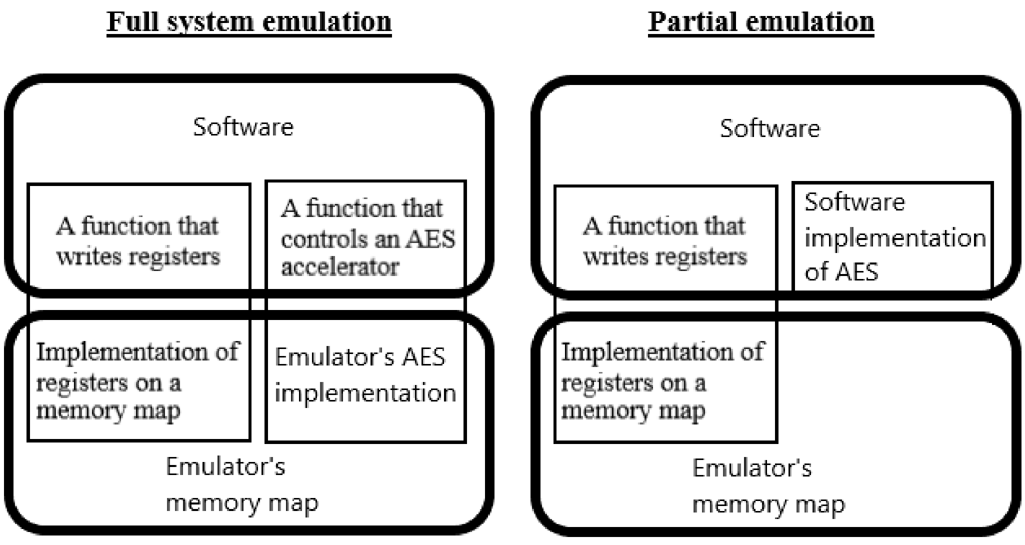

Full system emulation
Partial emulation
Software
A function that writes registers
A function that controls an AES accelerator
Implementation of registers on a memory map
Emulator's AES implementation
Emulator's memory map
Software
A function that writes registers
Software implementation of AES
Implementation of registers on a memory map
Emulator's memory map

Figure 11. Difference between full system emulation and partial emulation that uses software stubs.

Another example of partial emulation that can use the same binaries that are used for a real target embedded device is HALucinator [28]. It does partial emulation by dynamically changing the HAL (hardware abstraction layer) during the emulation to replacement functions.

The downside of partial emulation is that all code is not executed. For example, in the Figure 11 the function that controls the AES accelerator is not executed at all and new code is introduced with the software replacement function. However, if the replaced function is an insignificant part of the whole software, then this isn't a big problem.

The upside of partial emulation is that an emulator doesn't need to be modified even if the emulator doesn't support all custom parts of the hardware.

4.5 Native Execution

One way of executing embedded software on a PC is to compile the code directly for a PC and use a normal PC's process to run it. This works because an emulator just translates the instructions on run time, so it is somewhat similar to the situation where the source code is translated directly to the host machine language. Figure 12 illustrates how native execution is similar to emulation. In the figure ARM compiler's compilation and emulator's translation are replaced with direct compilation from source code to x86 instructions.

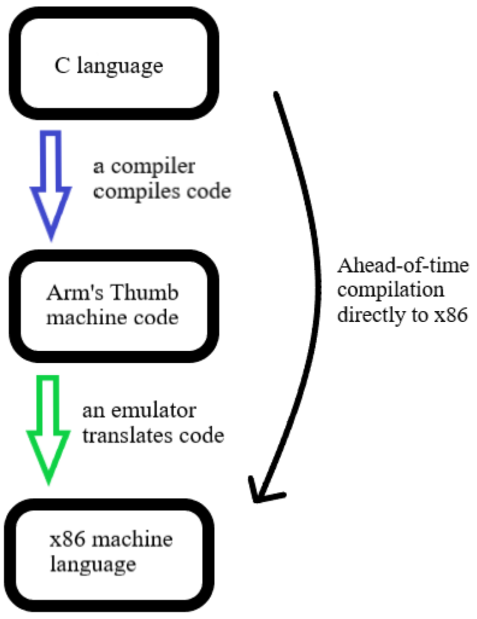
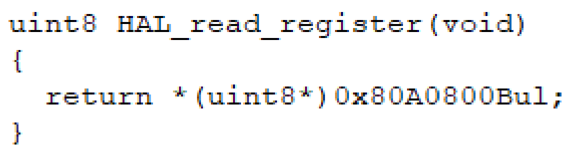


Figure 12. Example how emulator can be skipped with direct compilation to x86. The ARM ISA abstraction layer is completely skipped.

This direct translation can be enough for some high-level software for type-I embedded devices that use Linux. However, if the memory map is accessed directly from the code, then the translation of the instructions is not enough. For that kind of situation also the memory map has to be handled somehow. This can be done the same way as is done for partial emulation i.e., replacing and/or removing all accesses to the specific memory addresses. But as an embedded device's code can have many direct accesses to the memory map, the code has to be written in a way that makes it possible to replace the memory accesses easily.

To make it easy to replace direct memory accesses, the code should have platform-dependent code that can be easily replaced with code for other platforms. This can be easily achieved with a BSP (board support package) and/or a HAL. These two terms can have somewhat different definitions based on the source and sometimes they are used interchangeably. But in general, a BSP means hardware abstraction for the OS, and it also includes other stuff that makes it possible to execute code with specific hardware. While a HAL on the other hand means an API (application programming interface) for hardware abstraction e.g., an API for registers and hardware accelerators. A HAL can be part of the OS, or it can be part of the application. Figure 13 shows a simple HAL function for memory access in a bare metal system (type-III). Software that has a BSP and a HAL and can be compiled for two platforms is called dual-target software [1], and if the software supports more than two targets it can also be called multi-target software.



```
uint8 HAL_read_register(void)
{
  return *(uint8*)0x80A0800Bul;
}
```

Figure 13. Example of simple C language HAL function that reads a register.

A downside of native execution is that compiler will be different than for a real target. This means that if intrinsic functions (built-in functions) are used, the x86 compiler might not have them or they might have different names. To overcome this, all intrinsic functions must be replaced with some other implementations. Also, the embedded system’s assembly code has the same problem that it will not work for x86 and thus it has to be replaced with x86 assembly code or some other implementation written in a high-level language. Another problem with different compilers is that the compilers might have bugs and/or different behavior for undefined parts of the language thus the compiled machine codes might behave differently depending on which compiler was used.

An upside of native execution is that most compilers for x86 are free, and developers know how to use them. Whereas an embedded system’s compilers might need a license that can only be used by one developer at a time and the license might cost thousands or tens of thousands of euros. Also, for native execution, there isn’t a need for emulators or hardware thus it is easy and fast to run native code on a development machine without a need to study new tools.

Another positive thing about native execution is that many software development/analysis tools only work on an x86 PC and most of the tools are free. Examples of the tools include AddressSanitizer, LeakSanitizer, ThreadSanitizer, and MemorySanitizer which are used to instrument the code to find bugs. Also, with native execution, it is possible to use many dynamic analysis techniques like Valgrind, fuzz testing, symbolic execution, and taint propagation that can be impossible or difficult to do on a real embedded device.

Another advantage of native execution is that test execution will be fast. When running tests on a real embedded device, a lot of time is wasted on programming the Flash memory, booting hardware, etc. Therefore, test runs can take hours. But on a PC, tests can be executed a lot faster, and this makes it possible to use better development techniques like test-driven development [1, 29].

Figure 14 shows an example of how embedded software can be run on a PC. The behavior of the OS should stay the same, but the OS itself might require modifications, and the BSP layer must be changed for the PC. On an embedded system the BSP can have a lot of hardware-specific things like assembly code. So, for a PC they must be replaced with new code. The new BSP/OS can for example use Linux system calls to make sure that the OS will have the same behavior as the OS in the embedded device. The HAL must also be changed with software stubs that imitate the original functions. Note that not all code is similar between the real binary and the binary for native execution, but this isn't necessarily a problem. Because the application part of the software can be tested with native PC execution and afterward the whole firmware can be tested with a real embedded device.

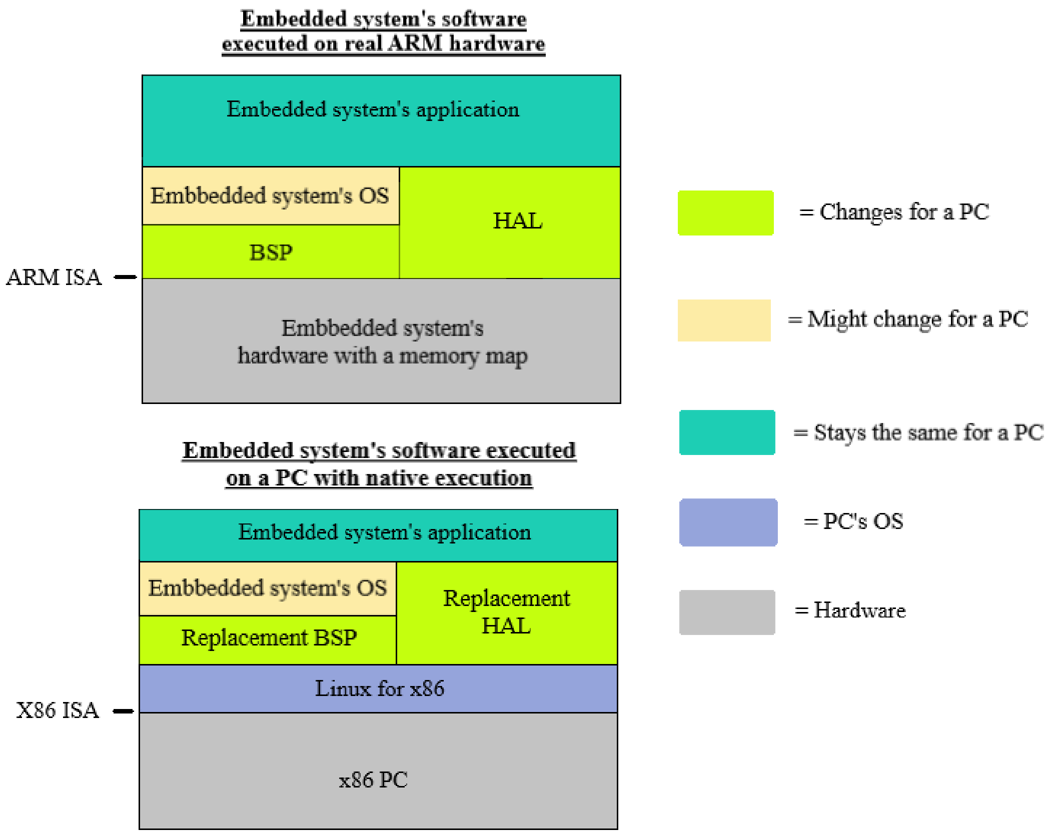

Embedded system's software executed on real ARM hardware
Embedded system's application
Embbedded system's OS
BSP
HAL
ARM ISA
Embbedded system's hardware with a memory map
Embedded system's software executed on a PC with native execution
Embedded system's application
Embbedded system's OS
Replacement BSP
Replacement HAL
Linux for x86
X86 ISA
x86 PC
= Changes for a PC
= Might change for a PC
= Stays the same for a PC
= PC's OS
= Hardware

Figure 14. Example of an embedded system's software stack when run on a PC.

5 IMPLEMENTATION

For this master thesis's two simulations of an embedded system were implemented. In this chapter, the simulated embedded system is described, and the two simulations are explained. The first simulation was done with Unicorn emulator and the second simulation was done with native PC execution.

There were many good options for the simulation. But in the scope of this thesis, it was not possible to try all methods. So, it was decided to do two implementations of the simulation. The customization of the QEMU would have been an interesting solution, but it would have been a challenging task because QEMU is not designed, to be edited to include custom peripherals. Other emulators e.g., Avatar 2, OVPsim, and Simics could also have worked, but all had some problems like the need for a license or lack of sufficient information on how to use them. The Unicorn emulator was free and could be used with Python and the Unicorn's API was also easy to use, so it was decided to use the Unicorn emulator as the first simulation method.

As the first method was emulation, it was decided that native execution would be the second simulation method because it would be a good comparison for an emulator. The downside of the native execution was that it cannot execute platform-dependent code. But as most unit tests only test common application code, the native execution method would be acceptable. Also, the native execution is fast at code execution, so it was likely that native execution would be useful in the development by allowing more frequent test execution.

5.1 Simulated Embedded System

In this section, details of the embedded system that was simulated are introduced. Both implementations i.e., emulation and native PC execution simulated the same embedded system. The system has an asymmetric multi-core CPU, that consists of many symmetric (identical) cores that use TriCore ISA. The symmetric TriCore cores are called "host". The CPU also has one asymmetric core that uses ARM ISA. The ARM ISA core is used to implement an HSM (hardware security module) and is therefore called "HSM".

The firmware application is mostly executed on the HSM's core, but it also includes a simple API layer on the host side that is used to call functions on the HSM side. A communication between the host and the HSM is also included in the firmware application.

The HSM has access to all memories and registers, but the host side can only access host side memories and it cannot access HSM's memories. Communication between HSM and host is done with special registers that both can access. The communication registers are called the "bridge".

The HSM codebase can be compiled for different targets and the targets have different features. The differences include different memory sizes and different amounts of hardware accelerators.

To limit the scope of this master's thesis, it was decided that only the simplest version of the hardware is simulated i.e., the one with the smallest amount of memory and only one hardware accelerator. Also, only the HSM side of the firmware is simulated to make the scope more suitable for a master's thesis. This means that the host-side API and the communication between the host and the HSM are not simulated in this master's thesis. However, because the simulation of the full system might be required in the future only solutions that could simulate the whole system including the communication, are considered. Figure 15 shows the whole HSM system, and it also shows what part is simulated in this master's thesis.

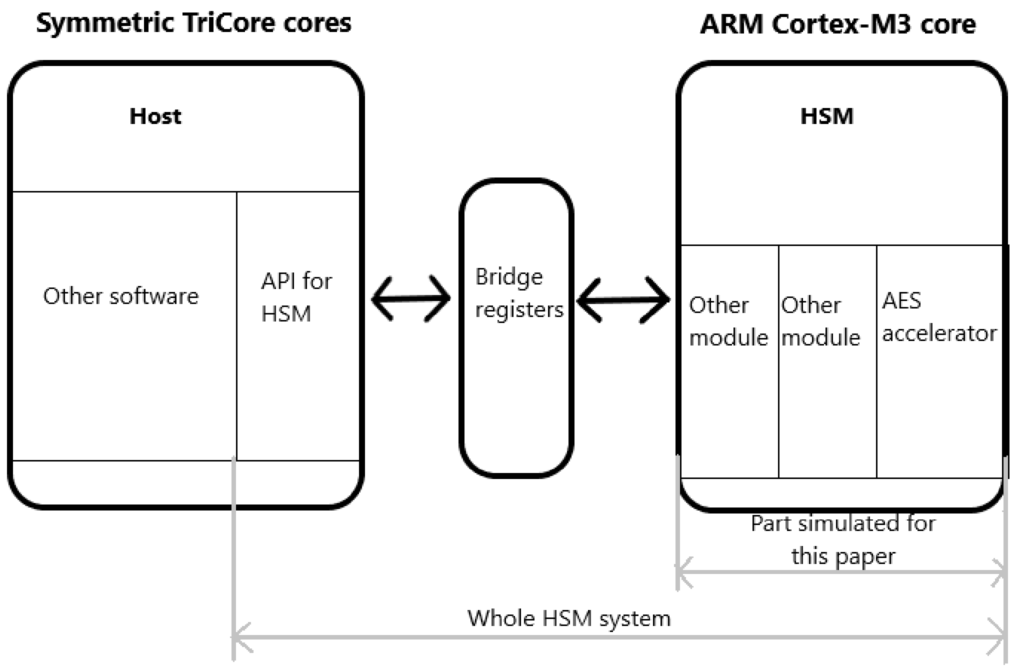


Figure 15. HSM on the embedded SoC that is simulated in this paper.

5.1.1 HSM

The embedded system that is simulated has a core that is used to implement a hardware security module (HSM) and the core is part of an SoC embedded device. The hardware security module is a physical module that is designed to store and manage cryptographical keys and do cryptographical operations. All operations i.e., key storage and cryptographical operations could also be done on a normal CPU. But embedded system's security is a lot better with the HSM because it limits physical access to the keys i.e., keys cannot be read/edited directly. Also, as the HSM has its own core, performance will be better because cryptographical operations can be done at the same time as other operations.

The HSM's core is based on an ARM's Cortex-M3 core thus the HSM uses ARMv7-M architecture and only supports Thumb instructions. Normal ARM instructions cannot be executed at all on the HSM (all Cortex-M cores only support Thumb instructions). An HSM has a different amount of hardware accelerators depending on the version of the hardware. All HSM versions have at minimum an AES accelerator, a TRNG (true random number generator), and a Flash memory controller that can write and erase Flash memory. Some HSM versions also have hash accelerators and ECDSA (elliptic curve digital signature algorithm) accelerators. To reduce complexity of this thesis, hash and ECDSA accelerators will not be simulated in this master's thesis.

5.1.2 HSM's Firmware

The firmware that is run on the HSM has an OS that uses a simple kernel and can only do basic task switching. The system is a type-III embedded system where the firmware application and the OS are compiled to the same monolithic binary.

The hardware accesses are separated to own files and functions, and they form a HAL that can be changed easily for different target hardware. The firmware application has different modules/drivers that control different parts of the system. For example, the firmware has modules for Flash memory, AES accelerator, and key storage.

5.1.3 Test Framework

The unit and integration tests for the embedded system are cross-compiled on a PC. The tests are compiled to a monolithic binary that includes the OS, the software modules needed for the tests, and many sub-test cases. The sub-test cases can be unit or integration tests. Due to memory constraints, all sub-test cases can't be compiled to the same binary, so the tests are divided into modules. In total there are 108 separate test modules. After a test module is compiled, it is written to the embedded system's Flash memory with a debugger after which the tests are executed. Once the tests are finished, the results are read with the debugger. In this master's thesis, these test modules are called "unit tests".

The normal firmware binary and test module binary are similar to each other. The only difference is that for the test module binary the unit and integration tests are included in the application part of the software. Therefore, when the word "firmware" is used in this master's thesis it can mean either the normal firmware or it can mean a test module.

5.2 Emulation

The first simulation done for this master thesis was emulation with the Unicorn emulator. The emulation is somewhat similar to what was done on the paper "*Cortex-M Simulator*" [30], but there are some differences. The Unicorn emulator's version was version 1.0.2. Unicorn 2 was not used because it was not yet released when the implementation part of this thesis work was done. The Unicorn emulator was used with Python because it was easy to use, and Python makes it easy to write custom peripherals for the emulator. In general, Python is a slow language, but because the underlying Unicorn emulator is written in C, and Python only acts as the binding on top of the C implementation, it follows that the execution speed is almost the same regardless is Python or C used.

It was relatively easy to emulate registers because most of the accesses to registers were not important for the emulation and it was enough to add the registers to the memory map. By default, the added areas were either RAM or ROM and had initialized value of 0 in each address location. This was enough to get the code to execute without errors, but a couple of the registers had to be initialized with some other values than 0 to get the correct behavior. All memory types of the real system i.e., Flash, RAM, ROM, and registers were emulated as either RAM or ROM in the emulator.

The AES accelerator was emulated by adding a hook to specific addresses on the memory map. The hook works in a way that if the defined memory address is accessed the code execution jumps to a substitute Python function. For example, in the case of the AES accelerator, the AES accelerator's address was hooked to a Python function, that implemented the AES algorithm with Python's cryptography libraries and wrote the results into the memory map. After the hook, the execution returned to the emulation of the firmware where now there was the correct AES output in the memory map and code execution could continue correctly.

Hooks were also needed to emulate TRNG, timer register, and for a module that can be used to write memory without using cache. All these implementations were straightforward to do just like the AES accelerator hook.

The Flash memory controller could have been added the same way as the AES was i.e., when the Flash memory controller's memory address is accessed, the execution jumps to a Python function with a hook. However, the logic of the Flash memory controller was complex in the target embedded device, therefore it was difficult to implement with a Python function. Thus, to simplify things, the Flash memory controller was emulated from one abstriction level higher i.e., the HAL functions that erase, read, and write the Flash memory was replaced with new functions implemented with Python code. This was done by adding a hook to the start address of the functions e.g., if the "write Flash function" would start at address 0x8000A000 then the hook would activate if the code was executed at the address 0x8000A000. After the hook was activated, the Python replacement function would write into the Flash memory (into the memory map). And after that, the execution was returned to the place before the write function was called by writing the program counter with the return address that was stored in the link register. Figure 16 shows examples of the hooks for "write Flash function" and for AES hardware accelerator.

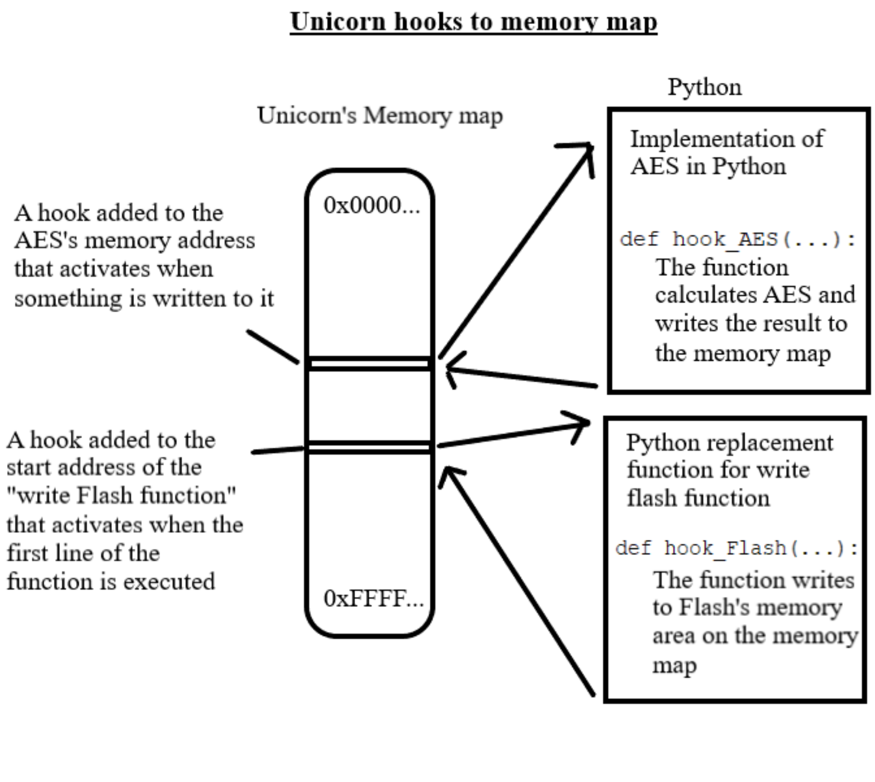

Unicorn hooks to memory map
Python
Unicorn's Memory map
Implementation of AES in Python
def hook_AES(...):
The function calculates AES and writes the result to the memory map
A hook added to the AES's memory address that activates when something is written to it
0x0000...
A hook added to the start address of the "write Flash function" that activates when the first line of the function is executed
0xFFFF...
Python replacement function for write flash function
def hook_Flash(...):
The function writes to Flash's memory area on the memory map

Figure 16. Examples of hooks in Unicorn emulation.

The biggest challenge with the Unicorn emulator was that the emulator doesn't support the emulation of interrupts. Unicorn isn't a full system emulator, so it doesn't have support for ARM's NVIC (nested vector interrupt control) that implements interrupts. The usage of interrupts was needed for the firmware's OS to do context switching. This problem was solved by two methods. The first method was running the emulator in a loop that executed only 7000 instructions, and then checking manually had an interrupt been triggered. The second method was adding a new global variable that indicated if interrupts were enabled or disabled in the code. The new global variable was the only mechanism that had to be added to the code for the emulation to work. So, in the end, the emulation wasn't 100% full system emulation. It could have been possible to do the emulation without the global variable; however, it would have needed editing of the Unicorn's source code, thus it was decided to do the emulation with this workaround.

5.3 Native PC Execution

The second simulation of the firmware was done by native PC execution i.e., compiling the code directly for an x86 PC and running it in a process. The existing codebase was used for many different target embedded devices, so it already was multi-target software and had a BSP and a HAL. This made it easy to add support for a PC by adding a new BSP and HAL for Linux. This multi-target software concept is similar to dual-target software that is described in the book "Test-Driven Development for Embedded C" [1]. The combination of the new BSP, the new HAL, and Linux's process can be thought of as a simulator that makes a simulated environment for the common application code.

The HAL and the BSP of the firmware were made by having different HAL and BSP files for every hardware. And when the software was compiled, the correct target system's files were used for the compilation.

The firmware's OS/BSP used a lot of assembly code, so some of it had to be replaced with new functions. The new functions mostly used the POSIX library's functions to handle signals and change context. It was surprisingly difficult to get the same behavior from the OS as the target embedded system's OS had. This was because Linux doesn't have good support for asymmetric preemptive user-level context switching and it is difficult to pause threads with one command and resume them later. This was a significant difference from the target embedded firmware where the OS scheduling could be done relatively easily with a combination of assembly language and C. In the end, it was possible to get almost the same OS behavior with the standard and the POSIX library functions.

The simulation of the interrupts was done with signals. For PC's process, signals are logically similar to what interrupts are for hardware i.e., they stop the execution from the outside and then they execute some function to handle the interrupt/signal. As signals are logically similar to interrupts, they were an almost perfect way of simulating interrupts. The embedded system's timers were simulated with POSIX timers using functions timer_create() and timer_settime().

The AES hardware accelerator was replaced with OpenSSL's AES functions. OpenSSL is part of the C compiler, so OpenSSL was easy to use directly without the need to install anything additional.

The flash memory that stored AES keys could be simulated with a big C array and new HAL functions that wrote, read, and erased from the array.

Most of the registers didn’t need to be simulated, as they didn’t affect how the code would execute. In the case of registers that affected code execution they were replaced directly with new HAL functions using basic C language commands.

Native execution needed different compilers and linker options, as the compiler for native execution was different than for the ARM target system. This could be done easily by adding new configuration files for the Linux target.

The embedded system used the same endianness as a PC i.e., little-endian byte ordering, so endianness didn’t affect the simulation.

6 EVALUATION

In this chapter, the implemented simulations are evaluated. The execution times between the real embedded device, Unicorn emulator, and native execution are compared. Also, the advantages and the disadvantages of all three execution methods are discussed. The comparison between the platforms is mostly done from the point of view of unit test execution.

6.1 Execution Speed

In this section, the execution speed of the unit tests is compared between the platforms. Due to differences between the implementations, some of the unit tests could not be executed on both Unicorn and native execution.

In the speed test, 12 unit tests were executed. The execution time includes compilation, time to open the debugger (only needed for the real embedded device), time used to write the unit tests to the Flash memory of the embedded device (only needed for the real embedded device), and the time needed for the execution of the test. For the real embedded device time needed to connect hardware and the time needed to cycle the power of the hardware was not counted to the total time. For all three methods, tests were compiled on a PC that used Linux. For the real embedded device and the emulator, tests were cross-compiled with ARM's embedded compiler (arm-none-eabi-gcc). Whereas Clang was used for the native PC execution to compile tests directly for the x86 architecture.

The fastest platform was a PC with a total time of 44 seconds. The second fastest platform was emulation with a total time of 88 seconds. The slowest was the real target embedded device with a total run time of 129 seconds. Table 3 shows the time needed to execute the 12 unit tests on the platforms.

Table 3. Execution times of the 12 unit tests on all three platforms

Platform	**Total execution time [s]**	**Average time for 1 unit test [s]**	**How many tests can be run in 1 minute [tests/min]**
Native execution on an x86 PC	44 seconds	3.7 seconds	16.4 tests/min
Unicorn emulator	88 seconds	7.3 seconds	8.2 tests/min
Real embedded device (ARM)	129 seconds	10.8 seconds	5.6 tests/min

Another thing to notice is that there were some differences in the results between the platforms depending on the test type. For example, stress tests (tests that do the same thing many times in a row to test the system under load) were really slow on the emulator when compared to the native PC execution or the real embedded device. This means that the run time of the tests might change depending on the test's type e.g., stress tests can be slow on an emulator. But in general, the order of unit test execution speed should be the same i.e., native execution on a PC is the fastest, a real embedded device is the slowest, and a good emulator without too high fidelity will be somewhere in between. Note that this order only holds if the time needed to open the debugger and writing unit tests to the Flash memory is counted for the real embedded device as they can be time consuming operations.

6.2 Advantages and Disadvantages of the Unicorn Emulation

The advantages of Unicorn emulation were that it was easy to use, free, and the code was open source. Because of the easy-to-use API and the Python bindings, it was easy to add custom peripherals and add hooks to the code. Also, the execution speed was quite good because the dynamic binary translation made the translation process fast.

The disadvantage of the Unicorn emulation was that the Unicorn emulator had some bugs that made the implementation difficult to do. Also, the lack of support of interrupts, made it difficult to replicate the hardware. With small improvements, Unicorn would be a powerful emulator for embedded device emulation, so it would be nice to see that the Unicorn emulator would be developed with full system emulation in mind.

Another problem with the Unicorn emulator was the lack of debugging possibility for the emulated code. This was the biggest problem of the Unicorn emulator because the purpose of the simulation was to enable the development and testing of the embedded software on a PC. In comparison, normal QEMU and "The xPack QEMU Arm" have good support for debugging. It would have been possible to add support for GDB (GNU Debugger) to the Unicorn, but this wasn't possible in the scope of this master's thesis.

6.3 Advantages and Disadvantages of the Native PC Execution

An advantage of the PC target was that it was easy to develop support for a PC. This was because Linux has good libraries for the C language e.g., standard C library, POSIX library, and OpenSSL. Also, most developers have experience with basic C language development on Linux, so every developer understands how to edit and use the code. Another good thing about the PC support was the ability to debug code with normal debuggers like GDB. Debugging was also made easier by the possibility of writing to standard output which is not possible when using a real target embedded device. Another advantage was the speed of the unit test execution. Executing unit tests on a PC was the fastest and the easiest of all three methods. With fast execution speed, it is possible for a developer to run tests more frequently and this makes better software development methods like test-driven development possible. Additionally, there wasn't a need to use and install emulators that can be difficult and time-consuming to use.

A significant disadvantage of the PC target was that it could not run 100% of the firmware's code. The real firmware's BSP and HAL were not executed at all on a PC and thus those parts of the code were not tested when unit tests were run. Also, the replacement of these code parts added a lot of new code that otherwise would not have been needed. Another disadvantage was that the firmware update feature had to be disabled from the builds. This was because the feature wrote new firmware directly to specific memory addresses and this would have been difficult (but not impossible) to implement for the native execution. Then an additional disadvantage was that a different compiler was used for the PC's binaries and thus it was possible to get different warnings/errors. And this meant that development might be a little bit slower, although this was more than compensated by the faster test execution.

In general, it can be said that the native execution is a good way of testing the logic and the algorithms of the embedded system’s platform-independent code, although native PC execution cannot test the HAL/BSP layers at all.

7 DISCUSSION

The results of this master's thesis are discussed from the point of view of how the implementations would work in software development. It is also discussed that would there have been better solutions than the implemented solutions.

Both solutions that were implemented in this paper i.e., native PC execution and emulation can be used to make the development of embedded firmware easier by making test execution easier and faster. Both solutions have their advantages and disadvantages from the point of view of test execution.

The native execution is quite easy to implement, and it is the fastest option when considering test execution. However, the low fidelity of the native PC execution can cause problems. For example, all code is not run from the original firmware and thus bugs in those parts of the code cannot be found with native execution testing. This means that the usefulness of the native PC execution completely depends on what kind of firmware is being tested. If the firmware uses a lot of custom hardware peripherals and/or has a lot of intrinsic functions and assembly code, then the native execution isn't that beneficial. However, if most of the software is common application code then the native execution might be the best solution.

Unicorn emulator has higher fidelity than the native PC execution, so it would have been a good addition to testing. However, the poor support for debugging led to the decision to not use it after this thesis. So, for this master's thesis native execution was the better solution. It is good to note that it is not generally true that native execution is better than emulation because in some situations an emulator would be better.

Unicorn emulator and native execution both have some disadvantages so possibly there could have been better solutions for this master's thesis simulation environment. Especially Simics would seem like a good way of doing the emulation as it supports emulation of custom peripherals. Also, the xPack QEMU Arm could have been used with partial emulation or it could have been edited to support the custom peripherals directly from the memory map. Unfortunately, there wasn't enough time to test any other solutions in the scope of this master's thesis.

In general, the best simulation method depends completely on what kind of firmware is being developed and on what kind of test needs to be executed. So, there isn't one simulation solution that is best for every situation. Instead, there are many different solutions. This is because embedded devices are a very diverse group of devices, thus it is only natural that there will be many different solutions to the simulation problem.

From the point of view of testing, it would be beneficial to use all three methods to run tests i.e., at first tests could be run on a PC then after that, tests could be executed in an emulator, and finally if both pass, tests could be run in a real embedded device. With this kind of testing strategy, tests could be run more frequently, and remote work would be easier, as native execution and emulator testing could be done on a laptop without a debugger or target embedded system hardware. And only when tests are run in a real embedded device would the tester need to be at the office or use a remote connection to the office. The downside of using all three methods is that it takes time to develop the PC and emulation environments, and the maintenance of the environments also needs resources.

8 CONCLUSION

In this master’s thesis, techniques for running embedded software in a simulated environment were introduced. The main purpose of this was to make embedded firmware development easier, faster, and cheaper by being able to execute unit/integration tests on the same computer as the firmware is developed.

Many different simulation methods and programs that could be used for simulation were introduced in this master’s thesis. And two of the simulation methods were implemented for one SoC embedded system. The implemented methods were Unicorn emulator and native execution on a PC. Based on this master’s thesis’ results both methods could be used in real testing, and both make remote work easier for embedded system developers. But the native execution on a PC was a better solution because it was 2 times faster than the Unicorn emulator, and it also had better support for debugging and it was easier to implement. There might have been some other emulator that could have been better than the Unicorn emulator but there wasn’t enough time to implement them. Even though the direct PC execution was better in this case, it is good to note that for other embedded systems other solutions could have been better. This is because embedded devices have different peripherals and different kinds of firmware and therefore there isn’t one solution that is the best in every situation.

In this master’s thesis, it was also found out that emulation and native PC execution don’t rule each other out, as they have different simulation fidelity, and both could be used in different stages of testing. However, neither method can completely replace testing on a real embedded system.

8.1 Future Work

After this thesis, the simulation environment can be extended to include more complicated HSM variants that have more hardware accelerators. Additionally, the simulation environment can be extended to include the whole HSM system i.e., host side API and communication between the host and the HSM. This could be done with an emulator or with the native PC execution environment. The emulator implementation would be more difficult as the host side uses TriCore, and it is not usually supported with emulators although QEMU has support for it. Implementation for the PC execution environment would be easier as the host side code could just be executed in a process or a thread and then the communication could be simulated with IPC (Inter-process communication) or in the case of threads the communication could be simulated with a basic C array.

9 REFERENCES

[1] Grenning, J. W. (2011). *Test Driven Development for Embedded C.* Pragmatic bookshelf.

[2] https://www.arm.com/company/news/2021/02/arm-ecosystem-ships-record-6-billion-arm-based-chips-in-a-single-quarter.

[3] Muench, M., Stijohann, J., Kargl, F., Francillon, A., & Balzarotti, D. (2018, February). What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices. In *NDSS*.

[4] Wright, C., Moeglein, W. A., Bagchi, S., Kulkarni, M., & Clements, A. A. (2021). Challenges in firmware re-hosting, emulation, and analysis. *ACM Computing Surveys (CSUR)*, *54*(1), 1-36.

[5] Moya, V. (2001). Study of the techniques for emulation programming. *Proyecto fin de carrera. Universidad Politécnica de Cataluña. España*, 21.

[6] https://andrewkelley.me/post/jamulator.html

[7] https://www.qemu.org/

[8] https://embeddedinn.xyz/articles/tutorial/Adding-a-custom-peripheral-to-QEMU

[9] Koppelmann, B., Messidat, B., Becker, M., Kuznik, C., Mueller, W., & Scheytt, C. An Open and Fast Virtual Platform for TriCore™-based SoCs Using QEMU.

[10] Monton, M., Portero, A., Moreno, M., Martinez, B., & Carrabina, J. (2007, June). Mixed sw/systemc soc emulation framework. In *2007 IEEE International Symposium on Industrial Electronics* (pp. 2338-2341). IEEE.

[11] Yeh, T. C., Tseng, G. F., & Chiang, M. C. (2010, April). A fast cycle-accurate instruction set simulator based on QEMU and SystemC for SoC development. In *MELECON 2010-2010 15th IEEE Mediterranean Electrotechnical Conference* (pp. 1033-1038). IEEE.

[12] Nakajima, K., Hieda, T., Taniguchi, I., Tomiyama, H., & Takada, H. (2012, December). A fast network-on-chip simulator with qemu and systemc. In *2012 Third International Conference on Networking and Computing* (pp. 298-301). IEEE.

[13] Delbergue, G., Burton, M., Konrad, F., Le Gal, B., & Jego, C. (2016, January). QBox: an industrial solution for virtual platform simulation using QEMU and SystemC TLM-2.0. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*.

[14] Charif, A., Busnot, G., Mameesh, R., Sassolas, T., & Ventroux, N. (2019). Fast virtual prototyping for embedded computing systems design and

exploration. In *Proceedings of the Rapid Simulation and Performance Evaluation: Methods and Tools* (pp. 1-8).

[15] Montón, M., Carrabina, J., & Burton, M. (2009, September). Mixed simulation kernels for high performance virtual platforms. In *2009 Forum on Specification & Design Languages (FDL)* (pp. 1-6). IEEE.

[16] https://xpack.github.io/qemu-arm/

[17] https://www.unicorn-engine.org/

[18] https://www.ovpworld.org/

[19] Agrawal, P. (2009). *Hybrid Simulation Framework for Virtual Prototyping Using OVP, SystemC & SCML A Feasibility Study* (Doctoral dissertation, Thesis, Indian Institute of Technology, 2009, 49p).

[20] Nita, I., Lazarescu, V., & Constantinescu, R. (2009, July). A new Hw/Sw co-design method for multiprocessor system on chip applications. In *2009 International Symposium on Signals, Circuits and Systems* (pp. 1-4). IEEE.

[21] Magnusson, P. S., Christensson, M., Eskilson, J., Forsgren, D., Hallberg, G., Hogberg, J., ... & Werner, B. (2002). Simics: A full system simulation platform. *Computer*, *35*(2), 50-58.

[22] https://www.windriver.com/products/simics

[23] Engblom, J., & Holm, M. (2006). A fully virtual multi-node 1553 bus computer system. *Data Systems in Aerospace*.

[24] Aarno, D., & Engblom, J. (2014). Software and system development using virtual platforms: full-system simulation with wind river simics. Morgan Kaufmann.

[25] Muench, M., Nisi, D., Francillon, A., & Balzarotti, D. (2018, February). Avatar 2: A multi-target orchestration platform. In *Workshop on Binary Analysis Research (colocated with NDSS Symposium)(February 2018), BAR* (Vol. 18).

[26] Feng, B., Mera, A., & Lu, L. (2020). P2im: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In *29th {USENIX} Security Symposium ({USENIX} Security 20)* (pp. 1237-1254).

[27] Chen, D. D., Woo, M., Brumley, D., & Egele, M. (2016, February). Towards Automated Dynamic Analysis for Linux-based Embedded Firmware. In *NDSS* (Vol. 1, pp. 1-1).

[28] Clements, A. A., Gustafson, E., Scharnowski, T., Grosen, P., Fritz, D., Kruegel, C., ... & Payer, M. (2020). HALucinator: Firmware re-hosting

through abstraction layer emulation. In *29th {USENIX} Security Symposium ({USENIX} Security 20)* (pp. 1201-1218).

[29] Grenning, J. (2007). Applying test driven development to embedded software. *IEEE instrumentation & measurement magazine*, *10*(6), 20-25

[30] Jakubík, T. (2020, September). Cortex-M Simulator. In *2020 International Conference on Applied Electronics (AE)* (pp. 1-4). IEEE.