Fredrik Strupe

Probing the Armv8-A ISA for Hidden Instructions through Processor Fuzzing

Master's thesis in Computer Science
Supervisor: Rakesh Kumar
June 2020

Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Fredrik Strupe

Probing the Armv8-A ISA for Hidden Instructions through Processor Fuzzing

Master's thesis in Computer Science
Supervisor: Rakesh Kumar
June 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Abstract

The increasing societal dependence on computer systems strengthens the need for verification and auditing of the systems in use, as a measure to ensure the absence of security issues and unwanted features like backdoors. For software, this is largely enabled by a wide range of available verification tools and a general drive towards open-source software. However, the underlying processor executing said software is more often than not regarded as a trusted black box, with little to no possibility for end users to verify the absence of secret functionality. Moreover, what research is available on such black-box processor verification mostly concerns the x86 instruction set architecture. In response to this, we present *armshaker*: a software-based fuzzer that can systematically uncover hidden instructions in the widely used Armv8-A instruction set architecture. Inspired by previous work, *armshaker* works by executing instructions that are undefined in the processor's reference manual and subsequently checking whether the execution result is as expected, with deviating behavior potentially indicating hidden instructions. Using this fuzzing approach, we identify multiple hidden instructions resulting from bugs in the commonly used QEMU emulator and the Linux kernel. However, none of the hidden instructions identified can be attributed to hardware, indicating that the tested processors do not include hidden instructions of the type identified by our approach. Nevertheless, the diversity of Arm processors available makes the presence of hidden instructions in certain models still a possibility. Consequently, we make our fuzzer freely available as open-source software to enable users to audit their own systems.

Sammendrag

Samfunnets økende bruk av datamaskiner forsterker behovet for å kunne verifisere og kontrollere systemer som tas i bruk, som et tiltak for å forsikre seg om at systemene ikke inneholder sårbarheter eller bakdører. For programvare tilrettelegges dette i stor grad gjennom allerede tilgjengelige verktøy og en generell tendens mot bruk av programvare med åpen kildekode. Den underliggende prosessoren som faktisk eksekverer denne programvaren regnes derimot ofte som en svart boks med få muligheter for sluttbrukere til å kunne verifisere at den ikke inneholder hemmelig funksjonalitet. Videre, tilgjengelig forskning innenfor slik verifikasjon tar stort sett for seg x86 instruksjonssettarkitekturen. Som et svar til dette presenterer vi *armshaker*: en programvarebasert fuzzer som kan systematisk avdekke skjulte instruksjoner i den mye brukte Armv8-A instruksjonssettarkitekturen. Inspirert av tidligere verk fungerer *armshaker* ved å eksekvere instruksjoner som er udefinert i prosessorens referansemanual og deretter sjekke om resultatet av eksekveringen var som forventet, der avvikende resultater potensielt indikerer skjulte instruksjoner. Ved bruk av en slik fuzzing-basert metode avdekker vi flere programvarefeil i den mye brukte QEMU-emulatoren og i Linux-kjernen. Imidlertid kan ingen av de skjulte instruksjonene vi identifiserer attribueres til maskinvare, noe som indikerer at prosessorene vi testet ikke inneholder skjulte instruksjoner av den typen som kan avdekkes gjennom vår metode. Dette utelukker likevel ikke at slike instruksjoner finnes i visse prosessormodeller, ettersom det finnes et stort antall ulike modeller. Vi tilgjengeliggjør derfor fuzzeren vår med åpen kildekode slik at brukere kan teste egne systemer.

Acknowledgement

I would like to thank my supervisor Rakesh Kumar for his valuable feedback and guidance during the course of this thesis. I would also like to thank Christopher Domas for his work on processor fuzzing, which served as the primary inspiration for this project.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Computers and digital systems have become an integral part of modern life, and something both individuals and societies as a whole rely on to a great extent. This augments the importance of cybersecurity in those systems, as a security breach can have potentially disastrous consequences. There has therefore been an increasing focus in recent years on cybersecurity measures like software testing and verification, penetration testing, new security standards, making code open-source and so on.

However, these measures largely focus on verifying software, with the underlying hardware that actually executes the software regarded as a trusted black box with few to no options for end user verification. While open-source hardware [2] would mitigate the issue somewhat, it is nowhere as common as open-source software and doesn't solve the problem of auditing already manufactured hardware. Instead, a great amount of trust is put into the hardware designers and manufacturers, in that the hardware functions exactly as documented with no hidden or undocumented behavior. This is different from software in that even closed-source software can be audited to some extent through binary analysis methods and black-box testing. Even when assuming no malicious intent though, security issues can still be present in hardware as a result of design mistakes [3, 4, 5] or manufacturing faults [6].

An example of such a processor-based security issue was made known by Christopher Domas in 2018 [7], when he revealed the presence of a secret coprocessor in a particular x86 processor model that could be accessed by executing a certain combination of machine instructions. The research was based on an earlier work of his where he developed a tool to search through the x86 instruction set and detect undocumented instructions [8] – that is, instructions that are not documented in the processor manual but still executes successfully on the physical hardware. However, apart from Domas's work, most published research on processor verification concerns correctness and formal verification – often from the designers' point of view – with no particular focus on security or hidden behavior. Furthermore, what research has been published with this focus primarily concerns the x86 instruction set architecture (ISA), without regarding other common ISAs like Arm and MIPS. Arm is a particularly interesting target because of its widespread use

– ranging from low-powered embedded devices to state-of-the-art super computers – with a particular prevalence in smart phones and a corresponding security impact potential.

In response to this, we develop an open-source processor fuzzer – codenamed *armshaker* [9] – targeting the latest version of the Arm ISA, namely Armv8-A. *armshaker* works in essence by exhaustively searching through the whole instruction space of the three instruction sets in Armv8-A (A64, A32 and T32), executing instructions that are undefined in the ISA specification. If an undefined instruction executes without faults, it is marked as a hidden instruction and logged for further analysis. As Armv8-A itself is an architectural specification, the fuzzer strictly speaking tests implementations of the ISA. It is therefore not limited to testing only physical processors, but can also fuzz virtual (emulated) implementations.

Formally, the research goal of this thesis is to investigate the presence of hidden instructions in Armv8-A implementations. To that end, we implement and apply a processor fuzzer with the intention of answering the following research questions:

Research Question 1 (RQ1)**.** Are there hidden instructions in particular implementations of the Armv8-A ISA?

Research Question 2 (RQ2)**.** Are there hidden instructions spanning all implementations of the Armv8-A ISA?

The distinction between the two research questions is important because of Arm's licensing model. Instead of manufacturing processors themselves, Arm (the company) licenses their processor designs to other companies which – depending on the license – can either produce them without change or with custom additions [10]. This implies that if a hidden instruction is found in all instances of a particular processor model or every processor model, it was likely a part of Arm's core design. On the other hand, if a hidden instruction is only present in a model from a particular manufacturer or third-party, it was likely introduced by the particular licensee or manufacturer. Naturally, this does not apply to implementations not originating from Arm, so RQ2 primarily concerns implementations based on Arm's designs – which encompasses the vast majority of available implementations, with the QEMU emulator being a notable exception.

1.1 Contributions

We make the following key contributions:

- **armshaker**: We design and implement a portable, open-source processor fuzzer – available on GitHub [9] – targeting the Armv8-A ISA, enabling a wide range of processors to be tested. *armshaker* automatically identifies divergent behavior of undefined instructions.

- **No Hardware-Based Hidden Instructions**: We run *armshaker* on a variety of Armv8-A-based systems, finding no hidden instructions that can be attributed to the underlying hardware processor. This indicates that the tested systems do not implement hidden instructions of the type identified by our fuzzing approach.

- **Software-Based Hidden Instructions**: We use *armshaker* to identify software-induced hidden instructions resulting from bugs in the Linux operating system kernel and the QEMU processor emulator. We also find bugs in the disassembler used, which is a part of libopcodes in GNU binutils.
- **Software Improvements**: We identify the root causes of the found software bugs and submit patches to Linux, QEMU and GNU binutils – enabled by all of them being open-source. Our patches were subsequently accepted and included in the respective projects.

1.2 Outline

The rest of the thesis is organized as follows. Chapter 2 covers the background information required to fully understand the research questions, fuzzer implementation and results. Specifically, it begins with an introduction to the Armv8-A ISA, followed by a definition of hidden instructions and how they relate to fuzzing and disassembly. Chapter 3 covers related work and how some of it influenced this project. In Chapter 4 the design and implementation of the *armshaker* fuzzer is described, first with an overview and subsequently in more detail. Chapter 5 presents the experimental methodology of the thesis, containing a description of the target systems to be tested and how the implemented fuzzer can be used on each of them. Next, in Chapter 6, the results from fuzzing the target systems are presented, together with an analysis of the results. Chapter 7 discusses the implications of the results in relation to the research questions and how they fit in a broader security perspective, together with some limitations of our approach. Finally, the thesis is concluded in Chapter 8.

Chapter 2

Background

To be able to identify hidden instructions, we must first establish exactly what a hidden instruction is and how it relates to the underlying ISA. An understanding of basic fuzzing approaches will also be beneficial in understanding the working principles of *armshaker*.

In this chapter we therefore first give an overview of the Armv8-A ISA – with a particular focus on the encoding of its three instruction sets (A64, A32 and T32) – and how behavior deviating from the ISA reference manual can be identified as stemming from hidden instructions. We then give a brief introduction to fuzzing, how it relates to our methodology, and how disassemblers can be used during fuzzing to identify deviations from the ISA specification.

2.1 Armv8-A ISA

Armv8 is the latest version of the Arm architecture, designed by Arm Holdings [11]. It uses a reduced instruction set computer (RISC) architecture, as apposed to a complex instruction set computer (CISC) architecture like in the x86 ISA. An essential part of RISC architectures is a smaller instruction set compared to CISC architectures, where each instruction typically is easier to implement in hardware and more applicable to pipelining, lowering the amount of processor cycles per instruction. Despite this, incremental changes over the years and an intention of using the architecture in everything from microcontrollers to supercomputers has made Arm a relatively extensive ISA.

As a way to cater to the different needs of such a diverse user base, Armv8 comes in three different architecture profiles, as denoted by the character following the architectural version and described as follows. The microcontroller (*M*) profile targets low-energy embedded systems; the real-time (*R*) profile targets real-time systems; and the application (*A*) profile targets mostly everything else where processing performance is a concern. The A profile is the profile we focus on in this work, and we therefore refer to the ISA as Armv8-A.

Armv8-A uses a 64-bit architecture, in contrast to the 32-bit architecture in earlier versions. However, to provide backward compatibility with Armv7-A, Armv8-A introduced the concept of *execution states* – of which there are two – each with their own instruction sets and processing environment:

- **AArch64**: The new 64-bit execution state, supporting the A64 instruction set.
- **AArch32**: The backward compatible 32-bit execution state, supporting the A32 and T32 instruction sets. The instruction set in use depends on a particular bit (the *T* or *Thumb* bit) being set or not in the Current Program Status Register (CPSR). This execution state is almost identical to Armv7-A, with some minor differences.

It is important to realize that an instruction set and instruction set architecture are not equivalent, but rather that an instruction set makes up a part of an instruction set architecture in addition to other aspects like the memory model. Consequently, even though Armv8-A has three different instruction sets, it is still only a single instruction set architecture.

2.1.1 Instruction Sets

In general, an instruction set specifies the interface for executing instructions on a given processor and is an integral part of the ISA. At the lowest level, this interface consists of certain bit patterns with corresponding functions, like arithmetic or memory operations. The mapping between bit patterns and operations as understood by the processor is decided by the processor designers, and documented for the users of the processor in the respective architecture's reference manual.

However, instructions in the form of different bit patterns are difficult for humans to keep track of. It is therefore customary for instructions to have certain textual mnemonics, which act as aliases for the bit patterns. These mnemonics then make up a part of the assembly language that can be used to program the processor, as further discussed in Section 2.4. While a mnemonic and bit pattern often can be considered to be equivalent, we sometimes want to make a distinction between them. In such cases we might call the mnemonic an assembly instruction or sometimes simply an instruction, while the bit pattern is called a machine instruction or instruction encoding.

There are in total three instruction sets available in Armv8-A, of which only a subset can be used at any given time depending on the current execution state. The most relevant part of the instruction sets to our work is the instruction encoding and how it relates to functionality in the processor, as opposed to the actual functionality available and how to use the instructions. Thus, we will consider the encodings of the three instruction sets next and discuss some of the differences between them.

A64

The A64 instruction was first introduced in Armv8-A and is used in the AArch64 execution state. It uses a significantly different instruction encoding compared to the instruction sets in earlier versions of the Arm architecture, as opposed to more incremental changes which had the been the case in earlier architectures. Nevertheless, most of the available instructions are the same as before. Furthermore, the instruction length is fixed at 32 bits like previous instruction sets.

Consider Figure 2.1 for an example of an instruction encoding in A64, namely for the `ADD (immediate)` instruction. The extended mnemonic would be `ADD Rd, Rn, #imm12`, where `Rd` and `Rn` represent registers and `#imm12` represents an immediate (constant) value, with the functionality being that `#imm12` is added to `Rn` and stored in `Rd`.

We can see that although `ADD (immediate)` represents a distinct operation, it will have many different instruction encodings depending on the values of its operands and options. The non-changing part of the encoding that uniquely identifies the particular operation is called the opcode (operation code) part, which in the case of Figure 2.1 are the bits marked with binary values instead of placeholder labels, namely bit 30 to 23.

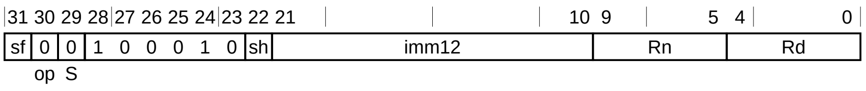


Figure 2.1: A64 encoding of the `ADD (immediate)` instruction, as documented in the Armv8 Architecture Reference Manual [11].

A32

A32 is one of the two instruction sets available in AArch32 (together with T32) and is in many ways similar to the Arm instruction set in Armv7-A and older versions. It is functionally similar to A64 and has the same fixed instruction length of 32 bits, but differs significantly in its encoding. Of particular importance is the presence of a condition code in most instructions, which indicates whether the instruction should execute or not depending on certain bits in the Current Program Status Register.

Comparing the A32 encoding of the `ADD (immediate)` instruction in Figure 2.2 with the A64 encoding of the same instruction in Figure 2.1, we can see differences in the opcode value, register indicator lengths (resulting from A32 having fewer available registers), operand ordering and the presence of the condition code.

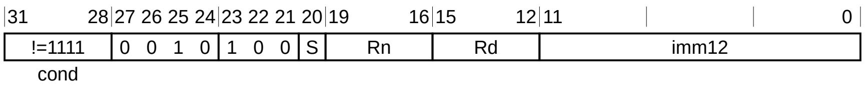


Figure 2.2: A32 encoding of the `ADD (immediate)` instruction, as documented in the Armv8 Architecture Reference Manual [11].

T32

T32 is the other instruction set available in AArch32 and is equivalent to the Thumb instruction set in Armv7-A; in other words, the Thumb instruction set was renamed to T32 in Armv8-A. The primary difference between T32 and A32 is that T32 uses a variable-length instruction encoding, as opposed to a fixed-length one. This is intended to reduce code size and improve cache utilization, especially for resource-constrained systems. T32 instructions are expanded to the equivalent A32 instructions in the processor's instruction decoder at runtime, and otherwise share the same execution environment.

The Thumb instruction set originally used a 16-bit fixed-length encoding, which resulted in each instruction having a length of one half-word as the architectural word size in all Arm architectures is 32-bit. However, because of the limited instruction space, an extension was later added to support certain 32 bit long instructions, making the instruction set variable-length. This was implemented by having the upper five bits of a half-word indicate whether an instruction is 16-bit or 32-bit, and in the case of it being 32-bit, a second half-word is fetched, making up the lower half of the 32-bit instruction. Specifically, a 32-bit T32 instruction is indicated by the upper five bits in the upper half-word having a value greater than or equal to `0b11101` (where the `0b` prefix denotes a binary number).

The 16-bit T32 encoding of the `ADD (immediate)` instruction can be seen in Figure 2.3, with the corresponding 32-bit T32 version in Figure 2.4. Note that some T32 instructions like `ADD (immediate)` have both 16-bit and 32-bit encodings, while others are exclusively 16-bit or 32-bit.

15	14	13	12	11	10	9	8 6	5 3	2 0
0	0	0	1	1	1	0	imm3	Rn	Rd

Figure 2.3: 16-bit T32 encoding of the `ADD (immediate)` instruction, as documented in the Armv8 Architecture Reference Manual [11]. Note that an alternative encoding also exists where `Rn` and `Rd` is the same register, allowing for a larger immediate value.

15	14	13	12	11	10	9	8	7	6	5	4	3 0	15	14 12	11 8	7 0
1	1	1	1	0	i	0	1	0	0	0	S	!=1101	0	imm3	Rd	imm8
												Rn				

Figure 2.4: 32-bit T32 encoding of the `ADD (immediate)` instruction, as documented in the Armv8 Architecture Reference Manual [11]. It can be identified as a 32-bit instruction by the upper five bits of the upper half-word (`0b11110`) being greater than `0b11101`. Like the 16-bit version, an alternative encoding also exists.

2.1.2 Unallocated Instructions

The total size of an instruction set's instruction space is determined by the number of available bits in the instruction encoding. Since A64 and A32 both are fixed-length instruction sets with 32-bit instructions, their resulting instruction spaces have a size of $2^{32} = 4,294,967,296$ instructions. In other words, the maximum amount of unique instruction encodings is a little over four billion. The instruction space of T32 is a bit smaller because of its variable-length instructions, having a size of $\frac{29}{32} \cdot 2^{16} + \frac{3}{32} \cdot 2^{32} = 402,712,576$ instructions. The fractions come from whether the upper five bits of the upper half-word indicate a 16-bit or 32-bit instruction.

An important observation is that none of the instruction sets utilize *all* of the available instruction space. Rather, there are blocks of unallocated instructions in each instruction space, resulting from their encoding not being assigned a function in the Armv8 Architecture Reference Manual. These unallocated blocks vary from spanning whole instruction groups (like in Figure 2.5) to particular operand values or options for certain instructions

– for example, the `ADD (shifted register)` instruction in A64 is unallocated if the 2-bit shift option is set to `0b11`.

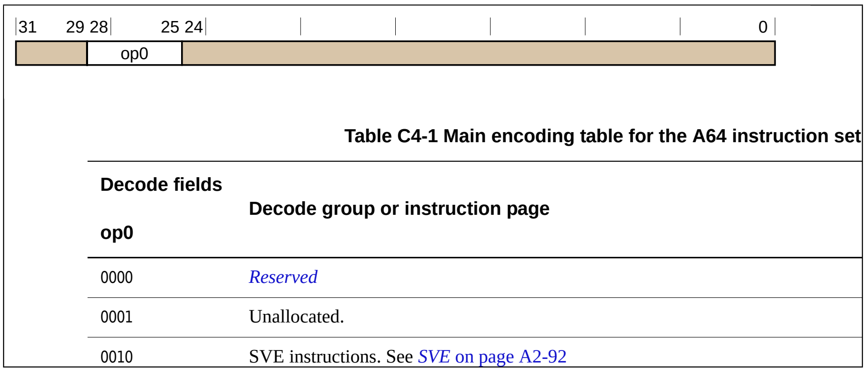

31
29 28
25 24
0
op0

Table C4-1 Main encoding table for the A64 instruction set

Decode fields op0	Decode group or instruction page
0000	*Reserved*
0001	Unallocated.
0010	SVE instructions. See *SVE* on page A2-92

Figure 2.5: Beginning of a table in the Armv8 Architecture Reference Manual documenting the highest level of the A64 instruction set encoding [11]. We can see that all instructions with the bit range [28:25] set to `0b0001` are unallocated.

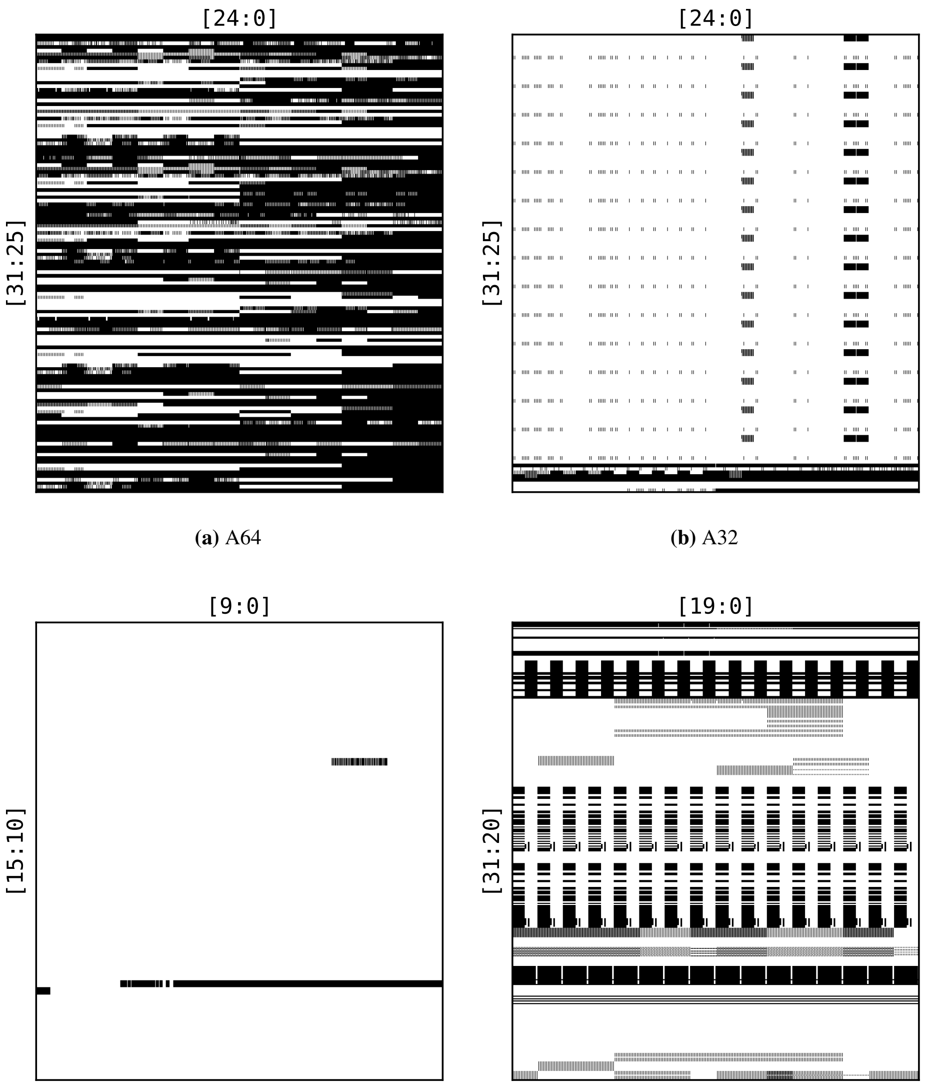
A question then arises on how many instructions actually are unallocated. If we plot the instruction space of the different instruction sets as two-dimensional maps and highlight unallocated instructions, we get something like the maps in Figure 2.6. The details of the figure is not important; rather, the takeaway is that we can see that a significant portion of the total instruction space is unallocated. In fact, 64.3 % of the instructions in A64 are unallocated, with the proportions being 12.2 % for A32 and 31.6 % for T32. The reason for the big difference between A64 and A32 is primarily that the condition code in the A32 encoding effectively duplicates most of the defined instructions, reducing the amount of unallocated instructions. T32 in turn has a higher percentage than A32 stemming from its lack of condition codes, but lower than A64 as it has a smaller instruction space. Strictly speaking, these percentages come from the number of undefined instructions as determined by the *libopcodes* disassembler (more details in Section 2.4), but it is a reasonable estimation.

Unallocated instructions as defined by the Armv8 Architecture Reference Manual can be further divided into the following three classes:

- **UNDEFINED**: Generates an undefined instruction exception upon execution. All unallocated encodings not belonging to a particular instruction are in this class, like the block in Figure 2.5.

- **UNPREDICTABLE**: The resulting state after execution is unpredictable and cannot be reasoned about. In practice, most UNPREDICTABLE instructions in Armv8-A are CONSTRAINED UNPREDICTABLE.

- **CONSTRAINED UNPREDICTABLE**: The execution result is constrained to a list of possible outcomes, which depends on both the instruction being executed and the particular ISA implementation. One outcome is generating an undefined instruction exception, but others like executing as a no-operation are also possible. In contrast to



(a) A64 **(b)** A32

(c) 16-bit part of T32 **(d)** 32-bit part of T32

Figure 2.6: Instruction space maps with black marks indicating unallocated instructions. The maps have a certain number of bits increasing from top to bottom along the y-axis indicating the most significant bits in each instruction – corresponding somewhat to the opcode portions of the respective instruction set encodings – with the remaining instruction bits increasing left to right along the x-axis. Note that the maps are not completely accurate because of their small scale, but does give an adequate overview.

the UNDEFINED class, instructions in this class usually stem from an invalid combination of instruction operands or options, in addition to incorrect *should-be-one* and *should-be-zero* bits. For example, the `STR (immediate)` instruction in A64 (storing a register value at a memory address contained in another register) is CONSTRAINED UNPREDICTABLE if the value and address registers are the same register.

Strictly speaking, the Armv8 Architecture Reference Manual refers to the above classes (in SMALL CAPITAL letters) as particular behaviors that can be assigned to instructions (both allocated and unallocated), as opposed to unallocated instruction classes per se. For example, disabling a particular instruction encoding at runtime will make it UNDEFINED, even though it might otherwise be defined in the specification. Still, we refer to a group of instructions sharing one of these behaviors as an instruction class. To make a distinction between the UNDEFINED instruction class and the more loosely defined term of undefined instructions – that is, instructions without a particular definition – we use the "UNDEFINED" notation when referring directly to ISA reference manual, and "undefined" otherwise. Nevertheless, the two can in most cases be regarded as interchangeable.

In A32 and T32, the majority of instructions in the CONSTRAINED UNPREDICTABLE class stem from incorrect should-be-one (SBO) and should-be-zero (SBZ) bits. These are bits in an instruction encoding that *should* have a particular value, but is not UNDEFINED if the value is incorrect, but rather CONSTRAINED UNPREDICTABLE. While this might seem like a peculiarity, it does make separating UNDEFINED and CONSTRAINED UNPREDICTABLE instructions slightly more complicated, as we will see later. An example of an instruction with SBO/SBZ bits can be seen in Figure 2.7.

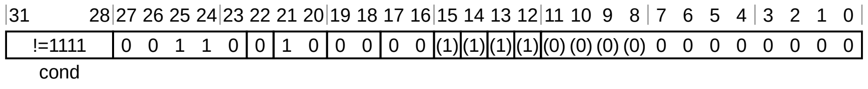


Figure 2.7: A32 encoding of the `NOP` instruction, as documented in the Armv8 Architecture Reference Manual [11]. (1) and (0) indicate SBO and SBZ bits, respectively.

Another subtlety is that a subset of the UNDEFINED instructions class is the *permanently* UNDEFINED instructions. These differ from the other UNDEFINED instructions in that they are guaranteed by the ISA specification to always result in an undefined instruction exception when executed, instead of as a consequence of being unallocated. In other words, they can be thought of as being defined as undefined. Yet another subset within the permanently UNDEFINED instructions is the instruction encodings mapping to the `UDF` instruction mnemonic, which can be used in assembly language to trigger an undefined instruction exception. The intention behind these instructions is that operating systems or other software running on the system can trap these instructions and use them for something else – like implementing a custom instruction – without affecting the behavior of defined instructions.

The takeaway of all of this is that there is not a clear distinction between defined and undefined instructions, as some instructions that are defined in the Armv8 Architecture Reference Manual can result in undefined instruction exceptions when executed, depending on their definition. This is one of the main complicating factors of detecting hidden instruction as defined next.

2.2 Hidden Instructions

The instruction set architecture for a processor makes up its user-facing interface, and is what enables it to be programmed externally. Consequently, the known capabilities of a processor depends heavily on its ISA specification, as it otherwise can be considered as a black box whose inner workings are unknown. One therefore has to trust the ISA documentation to faithfully reflect the actual functionality of a given processor, such as what instructions it supports. However, what if some supported instructions are omitted from the documentation? In that case, the processor can be said to have a number of *undocumented* or *hidden* instructions – terms that we regard as interchangeable in this work.

In other words, we can in general terms define a hidden instruction as a particular instruction encoding that successfully executes on a processor without generating an expected fault, while at the same time not being officially documented or documented as unallocated or non-functional. Such instructions were historically common in the 1970's and 1980's in older processors like the MOS 6502 [12] and Zilog Z80 and were a consequence of simplifications in the processor design, but became less common as manufacturing techniques improved, enabling a higher amount of transistors and wires at a smaller scale. Still, examples of hidden instructions in modern processors do exist, but then as a result of bugs or intentional features as opposed to side-effects of design simplifications. A famous example is the Pentium F00F bug [3] in some Intel processors in the 1990's that caused a complete system halt upon execution of certain instructions. Another example is a hidden instruction in certain VIA x86 processors that activates an undocumented coprocessor, as revealed by Domas [7]. However, to the best of our knowledge, there have been no public reports of hidden instructions in Arm processors.

As this work focuses on Armv8-A processors in particular, we can apply the previously discussed ISA details to the general definition above to produce the following definition of hidden instructions in Armv8-A.

Definition 1 (Hidden Instruction)**.** A hidden instruction in an Armv8-A implementation is an instruction that according to the Arm Architecture Reference Manual for Armv8 i) belongs to the UNDEFINED class of instructions and ii) does not generate an undefined instruction exception when executed on the particular implementation.

Notably, the definition does not include the UNPREDICTABLE and CONSTRAINED UNPREDICTABLE instruction classes. The reason for this is that although those instructions *may* generate an undefined instruction exception when executed, whether they actually do so or not is up to the particular implementations to decide. As such, their behavior is deliberately implementation defined and can't really break or contradict the official ISA specification.

Another point worthy of note is that the definition does not concern any defined instructions (i.e., not UNDEFINED) or their documented behavior. This means that we don't consider a defined instruction deviating from its defined behavior to be a hidden instruction, although it certainly is an inaccuracy in the documentation. Such issues relate to the broader field of processor verification and is outside the scope of this work.

2.3 Fuzzing

Software testing is an important aspect of verifying the proper functioning and stability of an application, with many different techniques in existence ranging from unit testing of particular software components to application-wide system testing. A particular testing technique is *fuzzing* – typically performed by a *fuzzer* – which is based on the concept of providing an application with random (and often invalid) input data and subsequently recording whether the application behaved unexpectedly, like whether it crashed or not [13]. Unexpected behavior for certain input values might be indicative of a bug in the processing of that value, which can then be further analyzed and fixed. Because of its reliance on exceptions like crashes, memory leaks and different program failures, the method is often used to find incorrect handling of edge cases and potential security issues in applications resulting from such exceptions.

Albeit a simple concept, the technique has shown to be effective in finding bugs in a wide range of applications [14], some of which have been critical from a software security point of view. Different fuzzing methods exist, ranging from the simpler cases of providing purely random or semi-random input data (like randomizing a field in an otherwise structured input), to more elaborate techniques like mutating the input data based on code execution paths taken in the target application. What particular method to use primarily depends on the desired results and the characteristics of the fuzzing target, like what kind of inputs it takes and to what extent it can be analyzed during execution.

While fuzzing is mainly used for testing software applications, the technique can be extended to other areas too, with the fundamental requirement being that the system to be tested has to take some input that will subsequently change the system's state in an observable manner. Translating this general specification to the field of processor fuzzing, the input can be instructions provided to the processor (in the form of bit patterns of a given length), with the observable state change being changes in register values, memory values and generated exceptions.

When testing a system – either through fuzzing or other techniques – one would ideally like to have 100 % test coverage, meaning that all possible test combinations have been tested. This is not feasible in many cases, as the test space will be too big to be searched exhaustively within a reasonable time frame. To combat this, certain assumptions can be made about the input to be generated, in a way such that even though the coverage of the total search space is low, the coverage of the actually relevant or interesting test cases is high. As an example, instructions in x86 are variable-length and can have a length of up to 15 bytes [15], making it infeasible to test all possible enumerations. However, using the observation that changes to the opcode part of an instruction often changes its length (as perceived by the processor), while operand changes seldom does, the fuzzer can be directed by skipping over parts of the search space that do not change the instruction length. With the presumption that what we're actually interested in is opcode enumeration (as opposed to operand enumeration), this greatly speeds up the search while still covering much of the interesting parts of the search space. This particular method was devised by Christopher Domas and used in his x86 fuzzer [8].

2.4 Disassembly

Disassembly is an integral part of our fuzzer, where it is used to identify undefined instructions. A similar technique is also used for instruction filtering, which will be discussed in Chapter 4. In this section we therefore give an overview of what disassembly is, how a disassembler works internally and how it can be used to find undefined instructions.

A processor fundamentally operates on binary numbers and has a binary machine language interface, with no concept of the human-readable code most often used to program it. Because writing code directly in machine language quickly becomes very laborious, it is customary to have a human-readable equivalent to the given machine language, namely an assembly language. The name stems from the act of converting the human-readable form to the machine language form, called program assembly and performed by an assembler. In practice, most compiled programming languages are first compiled to assembly language (or a similar low-level representation) and then assembled.

Likewise, converting an instruction in a machine language representation to the human-readable assembly representation is called disassembly and performed by a disassembler. As a general rule, the disassembler performs the inverse operation of an assembler (cf. Figure 2.8), though most assembly languages contain some additional features not present in the assembled code, like labels and macros, intended to make assembly programming a bit easier. Disassemblers are commonly used for static analysis of compiled programs [16], for example with the intention of reverse engineering an application or evaluating performance-critical code sections.

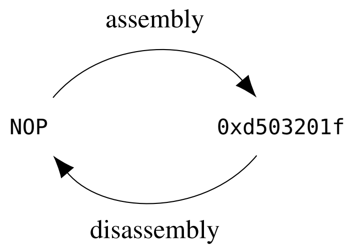
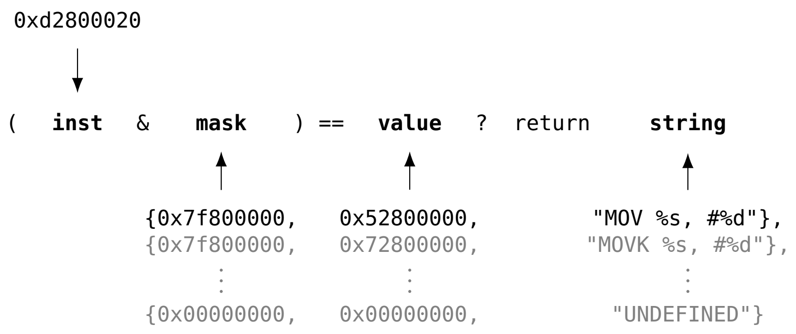


Figure 2.8: Relation between an assembly language `NOP` (no operation) instruction mnemonic and its corresponding machine language representation (in hexadecimal, denoted by the `0x` prefix) in the A64 instruction set.

Seeing as there generally is a one-to-one mapping between machine and assembly language instructions, most disassemblers work by successively taking a bit string from the program to be disassembled (representing a single machine instruction) and using it to retrieve the assembly representation from a pre-made lookup table. To reduce the lookup table into a manageable size, a bit mask is applied to the machine instruction before the lookup, which makes the table entries independent of the instruction's operands – essentially isolating the opcode part of the machine instruction. When the correct assembly instruction has been fetched, the operands can be extracted from the machine instruction directly and inserted into the assembly instruction, with the operand sizes and positions being hard coded in the disassembler. If no match is found, the instruction is marked as undefined. See Figure 2.9 for an illustration of this process.



```
0xd2800020

( inst & mask ) == value ? return string

{0x7f800000,   0x52800000,     "MOV %s, #%d"},
{0x7f800000,   0x72800000,     "MOVK %s, #%d"},
     ⋮              ⋮                ⋮
{0x00000000,   0x00000000,     "UNDEFINED"}
```

Figure 2.9: The machine instruction input `0xd2800020` (`MOV X0, #1` in A64) gets a mask from the lookup table applied to it (using a bitwise AND) – effectively removing the two operands `X0` and `#1` from the instruction – before being compared with the instruction value in the lookup table. In the case of a match, the corresponding assembly representation (with generic operands) is returned. Otherwise the process repeats for the next table entry, until the catch-all entry at the end is reached.

The entries in the disassembler's lookup table are made from reading the reference manual of the respective ISA and constructing corresponding masks, values and disassembly strings for each instruction. As such, using a disassembler on a single instruction can be compared to an automated reference manual lookup, replacing the need for looking up an instruction by hand. This also implies that if a particular instruction encoding is not present in the disassembly table, then it likely is undefined or unallocated according to the reference manual. Consequently, disassemblers can be used as a source of truth on whether an instruction is defined or not, and by extension to identify hidden instructions as defined in Definition 1.

Chapter 3

Related Work

There is not much prior work on processor fuzzing for detecting hidden instructions, apart from a few important publications. Nonetheless, related fields like general fuzzing, processor verification and emulator testing share many similarities to processor fuzzing. Processor security has also gained more attention recently as a result of the Meltdown [4] and Spectre [5] vulnerabilities. Accordingly, in this chapter we go through the publications available on processor fuzzing together with relevant work from the related research fields.

The inspiration for this thesis came from Chrisopther Domas's work on an x86 fuzzer (codenamed *sandsifter*) [8] and his subsequent usage of the fuzzer in uncovering a hardware backdoor in a particular VIA x86 processor [7]. Although the identified backdoor already appeared to be documented [17, p. 82], it still proved the efficacy of using fuzzing in uncovering security issues in processors. More importantly, his work on *sandsifter* and the techniques used made significant contributions to the field of processor fuzzing, which until then had seen little research except for more general research in the field of processor verification.

Processor verification is concerned with verifying the correct functioning of processor implementations and is a critical aspect of developing modern processors, as their complexity makes the introduction of bugs unavoidable. Such verification is generally done using either formal methods – like using mathematics to prove a specification's correctness – or more commonly functional methods like executing instructions and verifying that the result is correct. The maturity of the field combined with a goal in many ways similar to ours makes it an important source of information.

Of especial relevance to our work is the work of Reid et al. [18, 19] on verification of Arm processors. Using a sort of hybrid approach between formal and functional verification, their method revolves around parsing the ISA specification to generate executable code snippets whose execution results are then compared with the expected result as described by the same specification. Additionally, they test the execution of randomly generated instruction streams [20] – a method widely used in the industry [21] and very much similar to fuzzing. While it does seem like they test some undefined instructions, it is not clear whether they tested the whole instruction space as their focus is more on the correct execution of defined instructions.

Another closely related field to processor verification is processor emulator testing, which focuses on finding potential discrepancies between software-implemented processor emulators and their corresponding hardware implementations. In their seminal paper, Martignoni et al. [22] describe a fuzzing framework for testing processor emulators based on the idea of detecting state differences in the execution traces of randomly generated instruction streams. Even though the paper's focus is on emulator testing, it does highlight the efficacy of instruction-based fuzzing.

In relation to emulator testing, there is a relatively large amount of available research on emulator detection [23] for Arm-based processor emulators [24, 25, 26], primarily stemming from its relevance to malware analysis on the Android platform, which is mostly used by smart phones housing Arm processors. More specifically, much of the research is concerned with how applications can detect the presence of emulators in order to evade dynamic analysis, which can be discovered through various emulator testing techniques. Particularly relevant to us from this area is a publication by Sahin et al. [26], where they compare execution traces of randomly generated Armv7-A instruction sequences between the QEMU emulator and a more accurate emulator developed by Arm. The approach used is based on fuzzing, but like most of the other works discussed there is not a particular focus on detecting hidden instructions.

Finally, during the later stages of the work on this thesis a set of theses by Göebel [27] and Dofferhof [28] were published, describing how they searched for hidden instruction in Armv8-A and RISC-V processors. The theses share many similarities with the methodology used in this thesis, most notably with regard to the fuzzer implementation. However, their work differers from ours in only testing the A64 instruction set (in addition to RISC-V), having a bigger focus on fuzzing performance and with differing end results. Like us, they found no hardware-based hidden instructions in A64, but they did detect a number of hidden instructions in QEMU – all of which appear to have been fixed in the most recent release of QEMU.

Chapter 4

Fuzzer Implementation

We use a fuzzing approach in order to answer the research questions posed in the introduction on the existence of hidden instructions. Accordingly, a fuzzer is needed to do the fuzzing in an automated manner. However, existing solutions are either made for very particular use cases – like Domas's x86 fuzzer [8] – or focus on testing software as opposed to hardware, and with that provide a set of functionality different from what we require. Consequently, we implement a fuzzer for our specific use case of fuzzing Armv8-A processors, supporting all three instruction sets available in Armv8-A. In this chapter we describe the implementation of our resulting fuzzer, starting with an architectural overview before detailing its constituent parts.

4.1 Overview

Our fuzzer is written in C and intended to run on the Linux operating system kernel [1], both of which are widely supported on most Armv8-A-based systems. Having the fuzzer run on Linux not only eases development as a result of the features provided by Linux, but also enables a wide range of systems to be fuzzed without requiring any special hardware setup or software configuration. In the same manner, we have taken care to use as few external dependencies as possible and generally avoid operating system features not universally available, to increase operability across platforms.

Conceptually, the execution flow of *armshaker* can be divided into a set of repeated stages, as shown in Figure 4.1. First, some initialization code is run before entering the main fuzzing loop. Upon entering the loop and on every successive iteration, a new instruction to be tested is generated. The generated instruction is then disassembled using an external disassembler, and the resulting disassembly is checked. If the disassembly was successful – implying that the instruction is defined – the loop is cut short and continues with the next instruction. Otherwise – in the case of it being undefined according to the disassembler – it is run through a filter intended to remove certain false positives, primarily caused by CONSTRAINED UNPREDICTABLE instructions. If the instruction passes the filter, it is finally executed, and its result is checked. If an illegal instruction signal

(`SIGILL`) was received, then the loop is repeated as this is the expected behavior. However, if another signal or no signal at all was received, the instruction is marked as a hidden instruction and subsequently logged, before moving on to the next instruction.

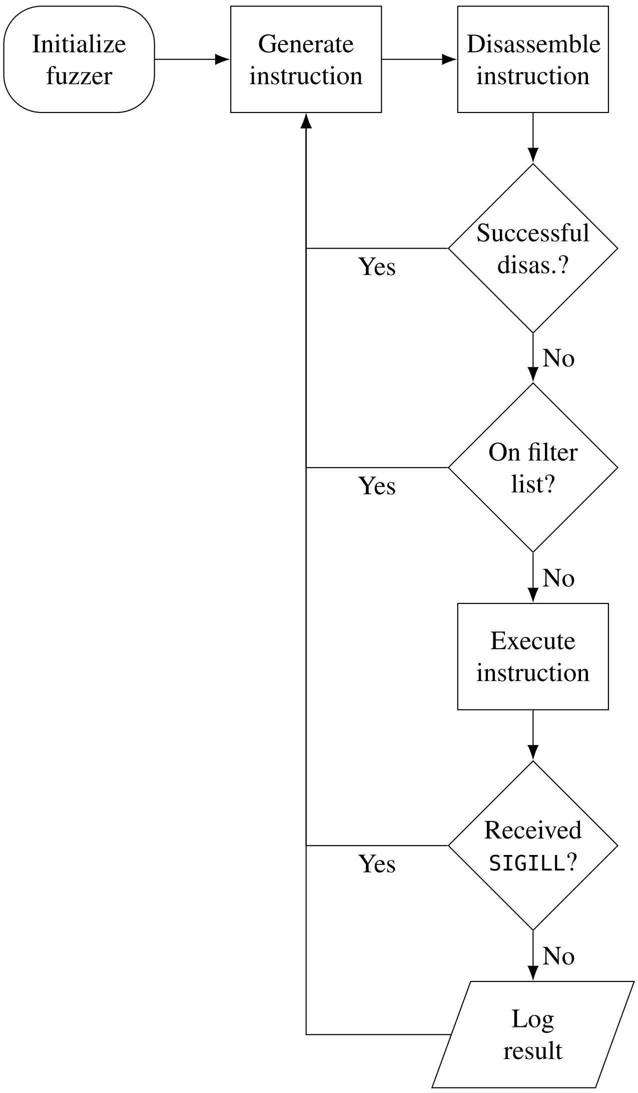


Figure 4.1: Execution flow of the fuzzing loop.

Although the fuzzing loop is conceptually simple, there are quite a few details that have to be taken into account in order to get a reliable fuzzer able to achieve both high test coverage and precision – all while supporting three different instruction sets. In the following sections we go through the implementation details of the different stages in the above fuzzing loop, highlighting the most important parts.

4.2 Initialization

The initialization stage does primarily two things. First, it processes the options passed to the fuzzer on the command line and sets the respective internal configuration variables. Then, it sets up the infrastructure needed for the following stages. This involves configuring the disassembler, clearing log files, initializing the instruction execution environment and so on. More details on the initialization particular to certain stages are provided in the description of the respective stages where relevant.

In contrast to the other stages, the initialization stage is run only once. This means that only program-wide initialization is done in this stage, while more local initialization like partially resetting the execution environment is done in the respective fuzzing loop stages.

4.3 Instruction Generation

In order to test the available instruction sets, we need to first generate the particular instructions to be tested. Since instructions have a 32-bit fixed-length encoding (16-bit for certain T32 instructions), this amounts to simply creating a bit string of the respective length with a certain combination of bits, and sending it to the next stage in the fuzzing loop.

However, as mentioned in Section 2.3, an exhaustive fuzzing search is often infeasible because of the size of the search space. Still, we have also seen that the search space size for each instruction set in Armv8-A – equivalent to the instruction space size – has an upper bound of $2^{32} \approx 4.3 \cdot 10^{9}$ instructions, which is not infeasibly large for modern processors. This means that we can in fact exhaustively search the whole instruction space – covering all possible instruction encodings – within a reasonable time frame, and is consequently what we do. As such, the fuzzer has a relatively simple instruction generation logic, linearly incrementing an initial instruction encoding until a terminal instruction has been reached. In the case of an exhaustive search, the initial instruction encoding would be `0x00000000` (all zeros), while the terminal instruction would be `0xffffffff` (all ones).

Since one also might want to test a non-contiguous subsequence of the instruction space – for example, operand variations on a particular instruction – masked increment is also supported. That is, when a search mask is applied, only bits in the instruction encoding matching the particular mask are incremented, with the remaining bits left unchanged. This option is particularly useful when analyzing results from an exhaustive search.

Another use of masked increment is for the variable-length T32 instruction set. Recall that T32 instructions with the upper five bits being lower than `0b11101` are 16-bit, while the remaining ones are 32-bit. To capture both lengths in a single search, all instructions are stored in a 32-bit variable internally, with 16-bit instructions having the lower two bytes set to 0 – equivalent to the lower half-word being zero. When incrementing an instruction, the upper five bits are first checked, and if they indicate a 16-bit instruction, a masked increment is performed on the upper half-word, leaving the lower half-word unchanged. For 32-bit instructions on the other hand, no mask is applied. This ensures a smooth transition from 16-bit to 32-bit instructions.

4.4 Disassembly

If the generated instructions were executed directly, we would in effect execute every single instruction in the whole instruction space. However, since we are only interested in executing undefined instructions, we want to skip generated encodings that correspond to defined instructions. Moreover, even if we did execute every instruction, we need to be able to verify whether an undefined instruction exception was expected or not for the particular instruction. Both of these problems are handled by the disassembly stage in the fuzzing loop.

In this stage, the generated instruction is passed to an external disassembler and the result is checked. If the disassembly is successful – indicating that the particular instruction is in fact defined – the remainder of the loop is skipped and the fuzzer moves on to the next instruction. Otherwise, in the case of the disassembler failing to disassemble the instruction – indicating an undefined instruction – the instruction is sent to the filtering stage. Since the disassembly is done before executing the instruction, it implies that every instruction that eventually ends up being executed is expected to be undefined, effectively solving both of the problems mentioned above at once.

As discussed in the background, using a disassembler in this way makes it a source of truth for checking whether an instruction is defined or not – like an automated reference manual lookup – with the fundamental assumption that it is a faithful representation of the reference manual. However, this is in fact a *false* assumption for even the most commonly used Arm disassemblers. There are several reasons for this, with the most common ones being bugs in the disassemblers, unsupported ISA extensions and inaccuracies in the interpretation of the ISA specification.

For instance, our initial implementation used the *Capstone* disassembler [29] for disassembly, which is a framework using disassemblers from the LLVM project in the back-end. Unfortunately, a lack of full Armv8-A support combined with large amounts of instructions incorrectly being marked as undefined made it unsuitable for our needs. Whether this was caused by Capstone itself or the use of an older LLVM back-end was not investigated. Instead, we opted for the Arm disassembler in *libopcodes* – a part of the GNU binutils project [30] – which is the same disassembler employed by the widely used *objdump* utility. Specifically, the code for the Arm disassembler and its internal dependencies were extracted from libopcodes and bundled with our fuzzer – as permitted by its license – to avoid having to separately compile and install libopcodes on every system to be tested.

The disassembler in libopcodes had a high accuracy compared to Capstone, possibly related to the fact that Arm was actively contributing to its development, but was still not without faults. Some instruction disassemblies were simply incorrect and caused by bugs, but those were few and could be readily fixed given the open-source nature of the disassembler. The biggest contributing factor to inaccuracies was related to the disassembly of CONSTRAINED UNPREDICTABLE instructions, as discussed in Section 2.1.2. More specifically, instructions with incorrect SBO/SBZ bits would be marked as undefined by the disassembler, when we want them instead to be marked as defined or unpredictable. This happens because these bits are included in the instruction value for many entries in the disassembly table, effectively making them a part of the instruction opcode.

Whether marking CONSTRAINED UNPREDICTABLE instructions as undefined is a bug or not is arguable, as the usage of such instructions should generally be avoided and might very well result in undefined instruction exceptions when executed. However, many instructions in this group can be executed without faults in practice, which makes them incorrectly being marked as hidden in our case. As such, we want to make a distinction between instructions in the CONSTRAINED UNPREDICTABLE and UNDEFINED groups. Since these discrepancies in practice are caused by incorrect SBO/SBZ bits in most cases, it was proved to be sufficient to filter out these particular instructions as discussed in the next section.

Finally, there is the question of whether *permanently* UNDEFINED instructions as described in Section 2.1.2 should pass as undefined in the disassembly stage or not, as these instructions can in some sense be thought of as being defined to generate undefined instruction exceptions when executed. However, recall that a subset of the permanently UNDEFINED instructions map to the `UDF` instruction mnemonic, which makes it more logical for software that wish to trap permanently UNDEFINED instructions to use encodings with the `UDF` mnemonic instead of an encoding with no mnemonic. We therefore consider `UDF` instructions to be defined like any other instruction, while the remaining permanently UNDEFINED instructions are considered as undefined and passed to the next stage.

4.5 Filtering

Arising from a wish to skip certain instructions in spite of them being marked as undefined by the disassembler, an extra filtering stage was added after the disassembly stage. In this stage, the instruction to be tested is compared against a set of instruction filters and skipped if it matches any of them. Filtering single instructions or small instruction groups like this is relatively simple as individual rules can be added manually, and is what was done for certain inaccuracies in the disassembler.

However, manually creating filters for instructions with incorrect SBO/SBZ bits is laborious and error-prone, as a significant amount of disparate instruction groups in A32 and T32 contain such bits. As a solution, a mechanism similar to disassembly with disassembly tables as discussed in Section 2.4 was developed – based on code from libopcodes – but with the addition of a should-be-bit mask in each table entry. When iterating through the table, the inverted should-be-bit mask is applied to both the instruction mask and value, and if a match is made, the instruction with the should-be-bit mask applied is compared to the one without. If the two don't match, it indicates the presence of incorrect SBO/SBZ bits as the instructions would otherwise be identical.

An example of such a filter table entry can be seen in Figure 4.2. The entry is similar to the disassembly table entries in Figure 2.9, but with the addition of a new should-be-bit (SB-bit) mask. Although it would be possible to change the existing instruction masks instead of introducing new should-be-bit masks, adding a separate mask has some benefits. First, it makes the job of manually converting encodings in the ISA reference manual to should-be-bit masks quicker and less error-prone. Second, it makes it possible to separate instructions with incorrect SBO/SBZ bits from otherwise undefined instructions.

When an instruction has passed through the filter without being rejected, it is fairly certain that the instruction is in fact undefined according to the ISA specification, as opposed to a defined instruction falsely identified as an undefined one. Consequently, we can move on to executing the instruction.

$$\{\ \underbrace{\texttt{0x0320f000}}_{\text{Instruction value}}\ ,\ \underbrace{\texttt{0x0fffffff}}_{\text{Existing mask}}\ ,\ \underbrace{\texttt{0x0000ff00}}_{\text{New SB-bit mask}}\ ,\ \underbrace{\texttt{"nop\%c"}}_{\text{Disassembly}}\ \}$$

Figure 4.2: Filter table entry for the `NOP` instruction in A32. We know from the Armv8 Architecture Reference Manual entry in Figure 2.7 that the second least significant byte in the encoding consists of should-be-bits, which correspond to the bits set in the should-be-bit mask.

4.6 Execution

In the execution stage, the given instruction is executed on the processor to check whether its execution results in an undefined instruction exception. If an undefined instruction exception indeed is generated – manifested as the executing process receiving an illegal instruction signal (`SIGILL`) from the kernel – the fuzzing loop is repeated as receiving such a signal is the expected behavior for undefined instructions. On the other hand, if a different or no signal is received – indicating a hidden instruction – the instruction is sent to the next stage for logging.

Executing an arbitrary instruction can have a wide range of different side-effects, potentially corrupting the fuzzer itself. It is therefore important to isolate the instruction execution from the rest of the fuzzing process to the extent where the probability of an instruction corrupting the fuzzer's execution environment is sufficiently low. Thankfully, this is greatly simplified by the fact that we only execute expectedly undefined instructions as opposed to all possible instructions, which means that as long as proper signal handling is in place, only side-effects from hidden instructions will potentially affect the system. Nevertheless, some protections should be in place such that hidden instructions can be executed without corrupting the fuzzer. For this we support two different execution techniques, namely *page-based* and *ptrace-based* execution, each with their own level of isolation and corresponding pros and cons, as discussed next.

4.6.1 Page-Based

The first execution isolation technique, which we call *page-based execution*, works by executing the instruction in the same process environment as the fuzzer, but from a dynamically allocated memory page with some boilerplate code. This is the same method used by Domas [8] in his x86 fuzzer. First – in the initialization stage of the fuzzer – an executable memory page is allocated on the heap with the `mmap()` system call in Linux, with some boilerplate code loaded onto it for for storing and restoring register values on entry and exit, resetting register values before execution, and a placeholder for the instruction to be tested. Next, a signal handler is set up, triggering every time a signal is received from the kernel and containing code to store the signal in a particular variable when triggered. Then, in every iteration of the fuzzing loop, the instruction to be tested is written over the placeholder in the executable page, the page is jumped to and subsequently executed before jumping back to the main loop. Whether the execution triggered the signal handler or not can then be determined by reading the variable set by the signal handler on every trigger, representing the signal value in the case of a trigger and zero otherwise. If the value of the signal variable is anything other than the value of `SIGILL`, it indicates a hidden instruction. For an illustration of this process, see Figure 4.3. Note that each timeline in the figure corresponds to a particular code region in the same process.

The isolation in this technique is primarily provided by the boilerplate code and having the executable page separate from the rest of the fuzzer code. Specifically, storing and restoring all register values mitigates corruption of register values that should stay unaltered according to the calling convention used, and resetting all register values to zero before execution somewhat reduces side-effects while also ensuring deterministic results. Furthermore, having the boilerplate code and test instruction in an executable page allocated on the heap while keeping the fuzzer code in read-only pages reduces the risk of corrupting the fuzzer's address space.

A somewhat obscure but important detail to be aware of for this method to work is that the first-level instruction and data caches in the Arm architecture are *not* coherent, meaning that an update to either of them does not necessarily propagate to the other. This means that self-modifying code like described above – that is, repeatedly writing to and executing a page – might lead to only the data cache being updated, with the instruction cache containing an older instruction. The result is then that the fuzzer will not execute the most recent instruction, but rather whatever instruction was in the page when the instruction cache was last updated, which will effectively skip over some instructions.

An illustration of this can be seen in Figure 4.4. The processor core first loads the instruction to be executed (`MOV R0, #42`) from main memory, which will update the instruction cache. It then at a later point overwrites the instruction with a new one (`MOV R1, #999`), which will only update the data cache. Since the instruction and data caches are not coherent, a subsequent fetch from the same location in the instruction cache will return the old instruction instead of the new one, which is not what we want. The solution to this problem is to write the new data cache contents back to main memory and invalidate the relevant location in the instruction cache, such that the instruction cache is updated on the next instruction fetch. This can conveniently be done with the `__clear_cache()` function in GCC, which under the hood executes a set of cache management instructions.

While the page-based execution method works in most cases and performs well, the isolation it provides is relatively weak. For example, any hidden instruction changing the program counter or certain system registers will corrupt the fuzzer if executed. While this rarely happened in practice for our tests, it was in fact an issue for certain instructions (as discussed in the results). As a solution to this problem – combined with a wish to make the fuzzer work in as many scenarios as possible – a second method was implemented, based on process tracing functionality available in Linux.

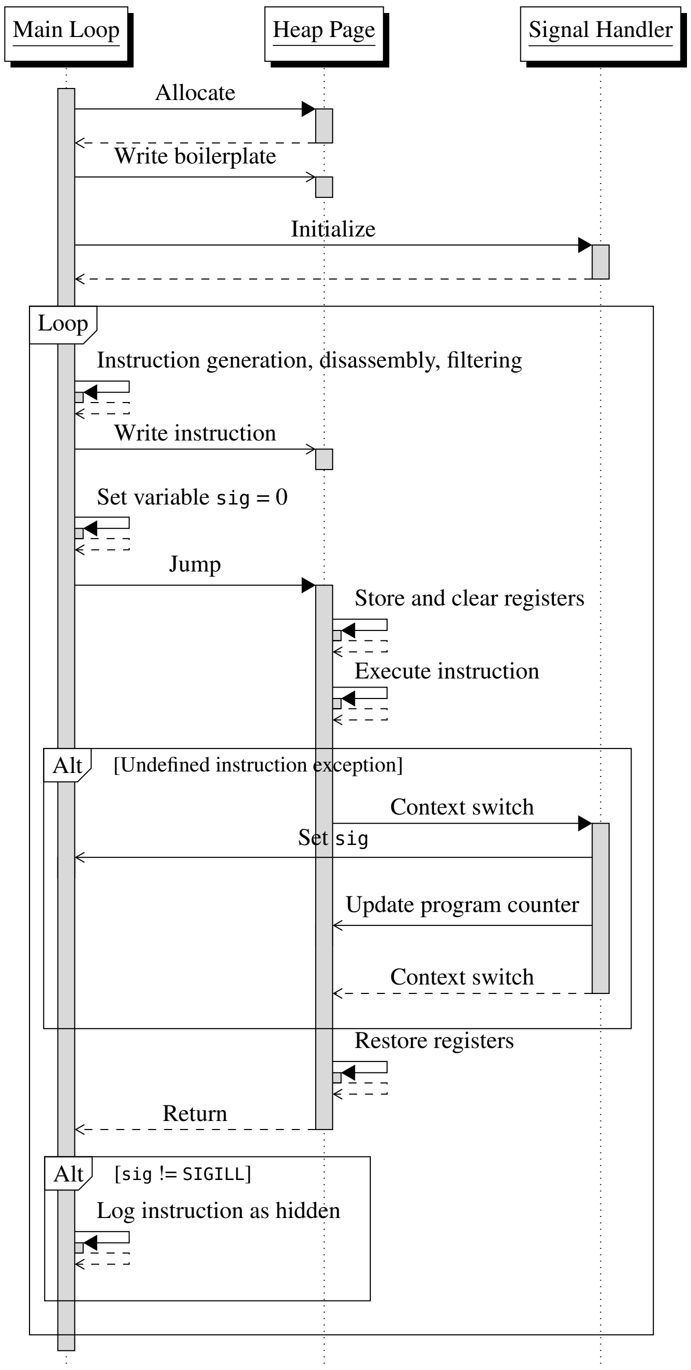


Figure 4.3: Page-based execution.

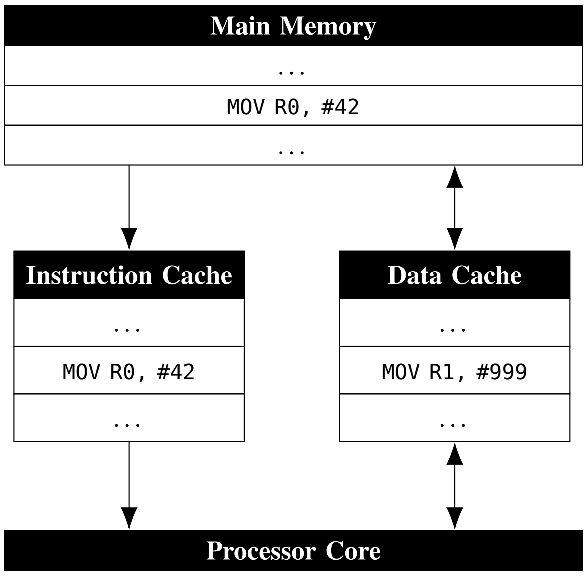


Figure 4.4: Cache coherence issue.

4.6.2 Ptrace-Based

The second execution isolation technique, which we call *ptrace-based execution*, is as the name suggests based on the `ptrace()` system call in Linux. With ptrace, a process (the tracer) can trace another process (the tracee), enabling the tracer to alter the tracee's register content, memory content and execution flow. Such tracing is often utilized by debuggers and instrumentation tools, but is also useful in our particular case. Concretely, the technique works by letting a separate child process execute the instructions to be tested, with the main fuzzer process using ptrace to modify the child's state on every iteration of the fuzzing loop. This means that if an instruction corrupts the child process, its state can simply be reset or the process restarted without interrupting the fuzzer.

The ptrace-based execution technique is illustrated in Figure 4.5 and works more specifically as follows. First – in the initialization stage – the fuzzer uses the `fork()` system call to create an identical copy of itself, which becomes the child process to be traced. The child process then jumps to a function with an infinite loop like in Listing 4.1, made with inline assembly code to ensure precise control over the generated code. The loop contains three instructions: a breakpoint instruction that stops execution until resumed by the tracer (A64 and A32/T32 use different breakpoint instructions, hence the preprocessor directive), a placeholder instruction that will be replaced with the instruction to be tested, and a branch instruction that jumps back to the breakpoint instruction.

```
void child_loop(void)
{
    asm volatile(
            "1:         \n"     // Label 1
#ifdef __aarch64__
            "   brk #0  \n"     // A64 breakpoint
#else
            "   bkpt    \n"     // A32/T32 breakpoint
#endif
            "   nop     \n"     // Instruction placeholder
            "   b 1b    \n"     // Jump backward to label 1
            );
}
```

Listing 4.1: The child process's execution loop.

After the initialization and at the beginning of every iteration of the main fuzzing loop, the child will be in a stopped state and open for modification by the main process. The main process then resets all of the child's registers (including the program counter), writes the instruction to be tested at the memory location of the instruction placeholder, and resumes the child process. The child will subsequently continue to the new test instruction and execute it. If executing the instruction generates a signal (like `SIGILL`), the child will stop and notify the main process. Otherwise, it will continue to the branch instruction, jump back to the breakpoint instruction and stop again (also notifying the main process). The main process can then retrieve the generated signal to determine whether a `SIGILL`

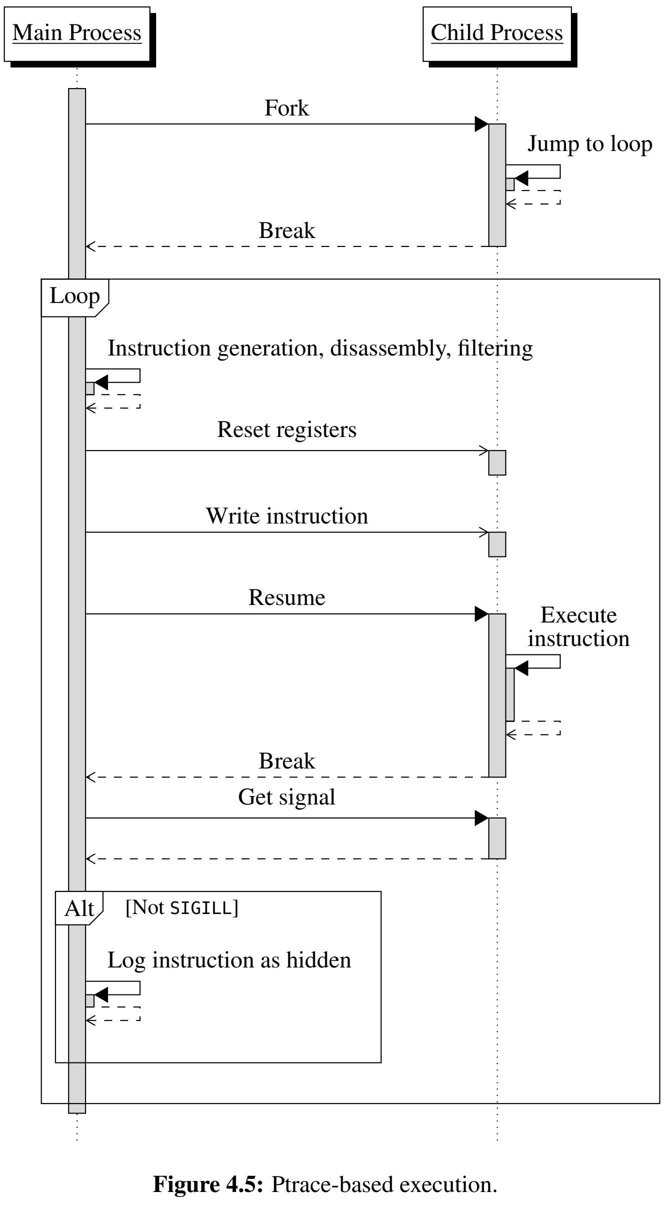


Figure 4.5: Ptrace-based execution.

signal was generated or not and take appropriate action, before continuing with the main fuzzing loop.

The ptrace-based execution method provides great isolation between the fuzzer and the instruction execution environment, owing to the inter-process isolation implicitly provided by Linux. It also has the added benefit of making it easier to set and retrieve the state of the system before and after executing an instruction, which can help in analyzing the behavior of hidden instructions. However, the frequent use of `ptrace()` system calls adds significant overhead to the total execution time of the fuzzer, with the instruction per second performance becoming an order of magnitude lower on most of the systems tested. Consequently, ptrace-based execution is intended to be used primarily when page-based execution has proved to fail for a given system or when analyzing a subset of the instruction space.

4.7 Logging

If the execution of an instruction resulted in anything other than a `SIGILL` signal, the instruction is marked as hidden and subsequently logged. This is done in the final logging stage and intended to both convey the presence of hidden instructions (assuming no false positives) and to give an overview of the state changes caused by executing the respective instructions.

An example of the log format used can be seen in Listing 4.2. The log uses a plain comma-separated values (CSV) format, with each line corresponding to a single instruction. The first entry in each line indicates the instruction, with the second entry indicating the group it belongs to (*hidden* is the only group implemented). Then follows the signal generated by executing the instruction, represented as a numeric signal value corresponding to the values in the Linux manual (seen by running the command `man 7 signal`), with a zero indicating no signal generated. After the signal value a variable number of entries follows, each representing a register value change in the state before and after executing the instruction, in the format `<register name>:<value before>-<value after>`.

```
eaa0b650,hidden,0,pc:116ca-116c8
eaa0b658,hidden,0,pc:116ca-116c8,cpsr:30-230
```

Listing 4.2: Excerpt from a log file generated by the fuzzer.

As a working example, consider Listing 4.2 again. We can see that none of the instructions generated a signal (indicated by the signal value being zero), and that the program counter (`pc`) decreased as a result of the branch instruction following the test instruction, as seen in Listing 4.1. However, for the second instruction the value of the Current Program Status Register (`cpsr`) changed, indicating that this instruction had an observable side-effect as opposed to just executing as a `NOP` instruction. Further analysis can then be done by hand, like correlating encodings and looking up the ISA reference manual for instructions similar in behavior or encoding. In the case of a large amount of different hidden instructions detected, it might be helpful to also use some kind of automatic instruction classification tool like what Domas used in his rosenbridge project [7], but that proved to not be necessary in our case. More information on our analysis methods can be found in the results.

4.8 User Interface

Interaction with the fuzzer is done through a command-line interface as shown in Listing 4.3. In that particular example, the fuzzer filters away disassembler inaccuracies as indicated by the `-f1` option; full documentation of available options can be found in Appendix A. We can also see that the fuzzer prints some status information, which is periodically updated.

```
$ ./fuzzer -f1
insn: 0x00583000, checked: 33536, skipped: 5500160, filtered: 245760,
hidden: 0, ips: 55387
```

Listing 4.3: Example of running the fuzzer from the command line.

However, the status information shown is rather limited and not particularly readable. More critically though, the fuzzing is done sequentially without exploiting the fact that executing a given instruction is fully independent from executing any other instruction. In other words, great performance improvements can be had by running multiple fuzzing processes simultaneously. As a solution to these two issues we implemented another text-based interface in Python – which we call the fuzzer front-end – that uses the fuzzer command-line interface behind the scenes. An example of this front-end in action can be seen in Figure 4.6.

The front-end automatically starts a number of fuzzer processes corresponding to the number of processor cores available on the running system – giving each process its own instruction range to fuzz – and periodically aggregates status data from each of them. Both the raw and aggregated information can then be seen in a text-based user interface. Additionally, statistics like the number of instructions per second tested (`ips`) and estimated time of completion (`eta`) are shown. The idea is that the front-end can be used for wide and potentially time-consuming searches, while the command-line interface is used for analyzing single instructions or smaller instruction ranges as it supports a wider range of options intended for analysis.

```
╔═ Summary ════════════════════════════════════════════════════════════════════════════════╗
║ checked:   1,425,408                           ips:       225,175                        ║
║ skipped:   175,416,192                         progress:  4.495%                         ║
║ filtered:  16,198,784                          elapsed:   0.27hrs                        ║
║ hidden:    0                                   eta:       5.1hrs                         ║
╚══════════════════════════════════════════════════════════════════════════════════════════╝

╔═ Worker 0 ════════════════════════════════╗  ╔═ Worker 1 ════════════════════════════════╗
║ insn:      02e42000                       ║  ║ insn:      42d68000                       ║
║ cs_disas:  N/A                            ║  ║ cs_disas:  N/A                            ║
║ opc_disas: rsceq r2, r4, #0               ║  ║ opc_disas: sbcsmi r8, r6, #0              ║
║ checked:   356,352                        ║  ║ checked:   356,352                        ║
║ skipped:   44,098,784                     ║  ║ skipped:   43,205,856                     ║
║ filtered:  4,049,696                      ║  ║ filtered:  4,049,696                      ║
║ hidden:    0                              ║  ║ hidden:    0                              ║
║ ips:       57,338                         ║  ║ ips:       55,736                         ║
╚═══════════════════════════════════════════╝  ╚═══════════════════════════════════════════╝
╔═ Worker 2 ════════════════════════════════╗  ╔═ Worker 3 ════════════════════════════════╗
║ insn:      82ef5000                       ║  ║ insn:      c2d7a000                       ║
║ cs_disas:  N/A                            ║  ║ cs_disas:  N/A                            ║
║ opc_disas: rschi r5, pc, #0               ║  ║ opc_disas: sbcsgt sl, r7, #0              ║
║ checked:   356,352                        ║  ║ checked:   356,352                        ║
║ skipped:   44,831,968                     ║  ║ skipped:   43,279,584                     ║
║ filtered:  4,049,696                      ║  ║ filtered:  4,049,696                      ║
║ hidden:    0                              ║  ║ hidden:    0                              ║
║ ips:       57,937                         ║  ║ ips:       54,164                         ║
╚═══════════════════════════════════════════╝  ╚═══════════════════════════════════════════╝
```

Figure 4.6: Fuzzer front-end showing information during an exhaustive search with four separate fuzzer processes (workers).

Chapter 5

Methodology

With the fuzzer implementation in place, we can experimentally investigate the research questions posed in the introduction. Since both questions extend beyond single ISA implementations, it's crucial to test a wide range of implementations in the form of different processor models and manufacturers in order to get reliable results. Consequently, we begin this chapter by listing the target systems that were tested, with a discussion of why they were chosen. Next, we discuss the fuzzer configuration for the various systems and how some of them require different options as a result of implementation-defined behavior.

5.1 Targets

While one ideally would like to test as many systems as possible, we are limited by the hardware available at our disposal. Fortunately, Armv8-A is widely used in consumer electronics and thus not particularly difficult to procure. In addition to hardware-implemented processors we also target a selection of Armv8-A emulators, which can be regarded as software implementations of the ISA.

The full list of target systems can be seen in Table 5.1. The first column lists the particular target systems, followed by the respective chipset manufacturers in the second column. The chipset manufacturer is relevant because it usually indicates the manufacturer of the processor, with potentially differing implementation details. Finally, we see the processors models used for the particular systems. Even though they all implement the same Armv8-A ISA, different models have different microarchitectures (corresponding to the ISA implementation) which makes it desirable to test different models.

Each of the target systems can be briefly described as follows. The Raspberry Pi 4 Model B and Orange Pi Lite 2 are single-board computers, essentially concentrating a complete computer into a single credit card-sized circuit board. The Huawei P8 Lite and Motorola Moto G5S are both Android-based smartphones. The Oculus Quest is a head-mounted display (virtual reality headset) that runs Android and is very similar to a smartphone, the user interface notwithstanding. QEMU [31] is a widely used open-source emulator capable of emulating a wide range of systems, many of which are Arm-based. In our particular tests we use version 5.0.0 and emulate a Raspberry Pi 3, but the underlying Armv8-A implementation is shared across all emulated systems and thus the particular system is not that important. Finally, we have the Arm Base Fixed Virtual Platform (FVP) which is a proprietary emulator developed by Arm, intended to be used for software development. There are various FVPs available emulating different processor models, but the one we use is the Base FVP that emulates a generic processor as opposed to a particular model.

System Model	**Chipset Manufacturer**	**Arm Processor Model**
Raspberry Pi 4 Model B	Broadcom	Cortex-A72
Orange Pi Lite 2	Allwinner	Cortex-A53
Huawei P8 Lite	HiSilicon	Cortex-A53
Motorola Moto G5S	Qualcomm	Cortex-A53
Oculus Quest	Qualcomm	Cortex-A73
QEMU 5.0.0 (RPi3)	N/A (virtual)	Generic Armv8-A
Arm Base FVP	N/A (virtual)	Generic Armv8-A

Table 5.1: Target systems.

5.2 Fuzzer Configuration

For all the target systems, we use *armshaker* to perform an exhaustive search of all three instruction sets (A64, A32 and T32) with filters enabled for disassembler inaccuracies. This amounts to simply first compiling the fuzzer with `make` and then running the fuzzer front-end like in Listing 5.1, where `armshaker.py` is the name of the fuzzer front-end executable. The start and end terminals of the search will automatically be set to `0x00000000` and `0xffffffff`, respectively.

```
$ make
...
$ ./armshaker.py -f1
```

Listing 5.1: Starting a multiprocessed exhaustive search with the fuzzer front-end.

Notably, starting the fuzzer like this requires access to a command-line interface which is not readily available on Android systems. Nevertheless, Android allows developers to execute native code (as opposed to Java bytecode) through its Native Development Kit, which in turn can be used to implement terminal emulators. We therefore use such a terminal emulator application to run our fuzzer – specifically, we use the open-source application Termux [32].

There is another issue that manifests itself in the ISA implementations used in the Oculus Quest and QEMU, which is caused by a particular implementation-defined behavior in Armv8-A – that is, behavior that can legally differ between implementations of the same ISA. Recall that many instructions in A32 are conditional, meaning they won't be executed if the specified condition doesn't match certain status bits in the Current Program Status Register. A particular implementation-defined behavior concerns what to do in the case of executing an undefined instruction with an unmatched condition. On most of the tested implementations, executing such an instruction results in an undefined instruction exception regardless of the condition. However, it is also legal for implementations to skip the instruction like if it was a defined instruction with an unmatched condition, essentially executing it as a `NOP` instruction instead. The latter behavior – which is seen in the Oculus Quest and QEMU – results in the fuzzer generating a lot of false positives as it expects all undefined instructions to fail. As a solution to this, we introduce an option (`-c`) that for every instruction makes the fuzzer set certain bits in the Current Program Status Register such that the condition of the current (undefined) instruction is matched, making it always execute. This allows for full instruction space coverage without reducing the accuracy of the results.

Finally, a feature of Armv8-A is that a kernel running in the AArch64 execution state (using the A64 instruction set) can execute user-space applications in the AArch32 execution state, enabling them to use the A32 and T32 instruction sets. This means that if we have a kernel running in AArch64, we can test all three instruction sets on the same running system. However, this is not the case the other way around – i.e, a kernel running in AArch32 cannot execute A64 applications. For our particular targets, this is an issue for the Motorola Moto G5S as the Android distribution it uses runs on a 32-bit kernel, which makes us unable to test the A64 instruction set on this particular device. All other devices have 64-bit kernels available, and thus do not have this issue.

Chapter 6

Results

In this chapter we present and analyze the results obtained through fuzzing the target systems. We begin by listing the number of hidden instructions marked for each system, followed by an analysis of the respective instructions. We also discuss a novel disassembler bug that was discovered during the development of the fuzzer and fixed before fuzzing the target systems.

6.1 Fuzzer Output

The number of hidden instructions found for each target system can be seen in Table 6.1. We can see that all systems had some instructions marked as hidden by the fuzzer, with some systems having more hidden instructions than others. Note however that one instruction corresponds to a unique instruction encoding, meaning that both the opcode and operands are included. Consequently, an instruction with variable operands or options may be marked multiple times.

Target	**Hidden Instructions**		
	A64	**A32**	**T32**
Raspberry Pi 4 Model B	0	15	0
Orange Pi Lite 2	0	15	2,184
Huawei P8 Lite	0	15	0
Motorola Moto G5S	N/A	15	736
Oculus Quest	0	15	2,184
QEMU 5.0.0 (RPi3)	0	69,647	69,632
Arm Base FVP	0	45	738

Table 6.1: Hidden instructions detected in the target systems.

After obtaining the raw results, the next step is to determine the exact behavior and root cause of the hidden instructions. To do this, we start with the log output generated by the fuzzer and use methods like looking for patterns shared between hidden instructions, encoding similarities to existing instructions as documented in the Armv8 Architecture Reference Manual, execution side-effect analysis, inspection of the underlying operating system or emulator source code and so on. The results of these efforts are discussed next.

6.2 Hidden Instructions

Our root cause analysis were largely successful and managed to account for all of the hidden instructions listed in Table 6.1. We found that all instructions except for the ones in QEMU can be attributed to bugs and backward compatibility measures in the Linux kernel. For QEMU, the hidden instructions are a result of bugs in its underlying Arm instruction decoder. In other words, none of the hidden instructions can be attributed to the underlying hardware. Nonetheless, it does show that hidden instructions as perceived by the user of a system can be induced by the software running on the system.

The revealed bugs and how they induce the hidden instructions are described next. To the best of our knowledge, all of the identified bugs have been unknown until found and subsequently reported as a result of our fuzzing efforts.

6.2.1 Linux

Most of the hidden instructions found can be attributed to three bugs and one backward compatibility measure in Linux, all of which are related to instruction hooking or trapping done by the kernel. Understanding the root cause of the bugs requires some knowledge of how Linux handles undefined instructions, which is described next.

When an executed instruction causes the processor to generate an undefined instruction exception, execution is transferred to the kernel which can then perform actions based on the exception generated. In the case of undefined instruction exception, Linux compares the instruction that caused the exception to a set of undefined instruction hooks, which when matched indicates that some particular action should be performed in the form of a function call – effectively *trapping* the instruction. An example of such an undefined instruction hook can be seen in Listing 6.1, where the masks and values are used for matching much like the previously discussed disassembly procedure. Critically – as we will see – the masks and values are always 32-bit unsigned integers. If none of the hooks match, Linux will instead send a `SIGILL` signal to the offending process before transferring execution back to it.

Hooks like these have different purposes, but are primarily used for debugging and instruction emulation by Linux when using the Arm architecture. The debugging hooks enable certain `UDF` instructions to be used as breakpoints or for other debugging procedures like tracing, where `UDF` is an instruction in Armv8-A that is defined as permanently UNDEFINED with the intention of allowing operating systems to hook on such instructions without affecting the execution of other instructions, as discussed in the background. The emulation hooks are used to provide support for certain deprecated instructions without executing them directly in hardware, which is done by executing an equivalent stream of instructions instead.

```
// Source file: arch/arm/kernel/ptrace.c
static struct undef_hook thumb2_break_hook = {
        .instr_mask     = 0xffffffff,
        .instr_val      = 0xf7f0a000,
        .cpsr_mask      = PSR_T_BIT,
        .cpsr_val       = PSR_T_BIT,
        .fn             = break_trap,
};
```

Listing 6.1: Example of an undefined instruction hook in the Linux source code [1].

All of the hidden instructions induced by Linux are the result of incorrect masks being used in various undefined instruction hooks. Specifically, the masks used for emulating the deprecated `SETEND` (set endianness) instruction, certain breakpoint traps in A32 and T32 and a set of Uprobe traps (providing tracing support) all have too wide masks, making Linux match more instructions than intended. The details of each of these bugs are as follows.

T32 SETEND Emulation

A particular instruction that Linux can emulate is the `SETEND` instruction, which is deprecated in Armv8-A. The instruction can be used to set the endianness of the system – indicating the byte-order used when operating on multi-byte variables – but was superceded by setting a bit in the Current Program Status Register instead. For T32, the encoding for `SETEND` is exclusively 16-bit and equals `0xb650` for the little-endian option and `0xb658` for big-endian.

If we consider the T32 hook for `SETEND` in the Linux source code – shown in Listing 6.2 – we can see that that the instruction value and mask seem to match what we expect, and the hook only matches when the T bit is set as enforced by the pstate mask and value (corresponding to the CPSR). However, recall that the masks and values are 32-bit unsigned integers, regardless of the instruction set used. When combined with the fact that T32 is variable-length, we can see that a 32-bit T32 instruction will get the upper half-word masked out because of the leading zeros in the highlighted mask. Since the hooks are always compared against undefined T32 instructions regardless of their length, this essentially makes all undefined 32-bit T32 instructions with the lower half-word equal to the 16-bit encoding of `SETEND` work as `SETEND` instructions, regardless of the upper half-word.

As an example, consider the T32 instruction encoding `0xeaa0b650`. This instruction is a 32-bit instruction as indicated by the upper five bits. However, looking up this particular encoding in the Armv8 Architecture Reference Manual reveals that it is unallocated. Nevertheless, Linux will execute it as a `SETEND` instruction, since applying the instruction mask to it yields `0xeaa0b650 & 0x0000fff7 = 0x0000b650` – which matches the instruction value in the hook – while it also has the T bit set, passing the pstate test.

This incorrect matching accounts for all of the 2184 hidden instructions found in T32 for the Orange Pi Lite 2 and Oculus Quest. The reason that only these systems are affected and not the others is that `SETEND` emulation has to be enabled when compiling the kernel – or by setting the `/proc/sys/abi/setend` file to 1 at runtime – which were the case only

```
// Source file: arch/arm64/kernel/armv8_deprecated.c
static struct undef_hook setend_hooks[] = {
    ...
    {
        /* Thumb mode */
        .instr_mask     = 0x0000fff7,  // Should be 0xfffffff7
        .instr_val      = 0x0000b650,
        .pstate_mask    = (PSR_AA32_T_BIT | PSR_AA32_MODE_MASK),
        .pstate_val     = (PSR_AA32_T_BIT | PSR_AA32_MODE_USR),
        .fn             = t16_setend_handler,
    },
    {}
};
```

Listing 6.2: `SETEND` hook in the Linux source code [1] with comments added.

for these two systems. The other systems were not affected simply because they didn't have `SETEND` emulation enabled. Furthermore, the bug only affects 64-bit Linux kernels (running in AArch64), as the 32-bit versions don't support `SETEND` emulation.

Luckily, all that is needed to fix the bug is to correct the highlighted mask. Specifically, it should be changed to `0xfffffff7`, which makes it include the upper half of the instruction value. Further inspection of the Linux source code reveals that these bits are always zero for 16-bit T32 instructions, which makes the new mask fix the bug without affecting the intended behavior. We submitted a patch with this fix to the Linux kernel which was subsequently included in both the official mainline kernel and older versions with long-term support.

T32 Breakpoint Traps

We saw above how a too wide mask (that is, a mask that captures too many instructions) can cause certain 32-bit T32 instructions to be unintentionally matched, when the intention was to only match 16-bit instructions. The exact same problem is present for certain breakpoint hooks too, where a breakpoint is an instruction that generates a `SIGTRAP` signal upon execution. In particular, consider the code in Listing 6.3, which is used to make the `UDF #1` instruction (with encoding `0xde01`) work as a breakpoint instruction. Since the underlying mask variable is 32-bit, the mask is essentially equivalent to `0x0000ffff`, which has the same problem as for `SETEND` emulation with matching undefined 32-bit T32 instructions where the lower half-word is equal to `0xde01` regardless of the upper half-word, making them work as breakpoints too.

Notably, this bug is only present in the 32-bit version of Linux, as the 64-bit version uses a direct comparison without masking instead. It therefore only appears on the Motorola Moto G5S and Arm Base FVP where it accounts for 736 of the hidden instructions. When testing the Arm Base FVP we used 32-bit and 64-bit kernels separately as opposed to running everything with a 64-bit kernel, which is why the bug appears for the Arm Base FVP but not the other systems with 64-bit kernels.

Like the `SETEND` bug, this bug is fixed by extending the instruction mask of the hook to

```
// Source file: arch/arm/kernel/ptrace.c
static struct undef_hook thumb_break_hook = {
    .instr_mask = 0xffff,  // Should be 0xffffffff
    .instr_val  = 0xde01,
    .cpsr_mask  = PSR_T_BIT,
    .cpsr_val   = PSR_T_BIT,
    .fn         = break_trap,
};
```

Listing 6.3: T32 breakpoint hook in the Linux source code [1] with comments added.

`0xffffffff`, which captures only the one intended instruction. We also submitted a patch with this fix to the Linux kernel which was subsequently included in the official mainline kernel.

A32 Breakpoint Traps

The two bugs above applied only to T32, with A32 and A64 unaffected. However, similar effects exist for breakpoint hooking in A32 too. Specifically, consider the hook code in Listing 6.4, which makes the `UDF #16` instruction in A32 – with encoding `0xe7f001f0` – work as a breakpoint. As we can see, the first four bits of the instruction encoding are masked out, corresponding to the bits in the condition code and indicating that the instruction is to be matched regardless of the condition. Yet, the Armv8 Architecture Reference Manual explicitly states that `UDF` is *not* conditional, meaning that having a condition code other than `0xe` is unallocated. In other words, the instruction mask and value should seemingly be `0xffffffff` and `0xe7f001f0`, respectively. This affects both 64-bit and 32-bit kernels, and accounts for 15 of the hidden instructions in A32.

However, a further inspection of related parts in the Linux source code indicates that this behavior is intentional and not the result of some mistake. This was confirmed through correspondence on the Linux Kernel Mailing List [33], where it was made apparent that the behavior is in fact a backward compatibility measure for Armv6 and earlier versions, where instruction encodings corresponding to conditional `UDF` instructions were legal. Furthermore, even though the conditional `UDF` instructions are unallocated in Armv8-A, they still belong to the permanently UNDEFINED instruction class, so trapping them is not really incorrect as that is what the class is intended to be used for.

Still, when describing the disassembly stage of the fuzzer in Section 4.4 we established that we expect permanently UNDEFINED instructions without a corresponding `UDF` mnemonic to not be trapped – not because it's illegal, but rather that hidden instructions very well could be placed in this space. The detected instructions also pass our definition of hidden, as they belong to the UNDEFINED class of instructions. However, their presence cannot be regarded as incorrect or breaking the ISA specification. This highlights that a hidden instruction stemming from the ISA implementation is not necessarily equivalent to one induced by software when considering user-level correctness of the ISA.

```
// Source file: arch/arm/kernel/ptrace.c
static struct undef_hook arm_break_hook = {
    .instr_mask = 0x0fffffff,  // Should be 0xffffffff
    .instr_val  = 0x07f001f0,  // Should be 0xe7f001f0
    .cpsr_mask  = PSR_T_BIT,
    .cpsr_val   = 0,
    .fn         = break_trap,
};
```

Listing 6.4: A32 breakpoint hook in the Linux source code [1] with comments added.

Uprobe Traps

Finally, there is a bug in Linux related to a set of Uprobes hooks. Uprobes is a user-level tracing feature offered by Linux – similar to ptrace – with two `UDF` instructions used for breaking and single-stepping. In A32, the instructions have the same behavior as the breakpoint instructions above in that the condition code is ignored as indicated in the first two highlights in each of the structs in Listing 6.5, which together account for an additional 30 hidden instructions. But as discussed above, this cannot be considered a bug.

However, there is an actual bug in this code that affects T32. Namely, the status register masks (`cpsr_mask`) do not include the T bit, which makes the hooks apply to T32 too (in addition to A32). This is incorrect as the `UDF` instruction encoding differs between A32 and T32, and by pure coincidence two unallocated T32 instructions (which are not in the permanently UNDEFINED instruction space) are matched, turning them into tracing instructions. Furthermore, the Uprobes source code mentions that Thumb (T32) is not supported, indicating that it shouldn't be triggered from T32 in the first place.

```
// Source file: arch/arm/probes/uprobes/core.c
static struct undef_hook uprobes_arm_break_hook = {
    .instr_mask = 0x0fffffff,
    .instr_val  = (UPROBE_SWBP_ARM_INSN & 0x0fffffff),
    .cpsr_mask  = MODE_MASK,  // Should be (MODE_MASK | PSR_T_BIT)
    .cpsr_val   = USR_MODE,
    .fn         = uprobe_trap_handler,
};

static struct undef_hook uprobes_arm_ss_hook = {
    .instr_mask = 0x0fffffff,
    .instr_val  = (UPROBE_SS_ARM_INSN & 0x0fffffff),
    .cpsr_mask  = MODE_MASK,  // Should be (MODE_MASK | PSR_T_BIT)
    .cpsr_val   = USR_MODE,
    .fn         = uprobe_trap_handler,
};
```

Listing 6.5: A32 Uprobes hooks in the Linux source code [1] with comments added.

This bug accounts for the remaining two hidden instructions in T32 for the Arm Base FVP. Out of all the tested systems, it is only present in the Arm Base FVP simply because it was the only system with a kernel compiled with Uprobes support. It can be fixed by adding the T bit to the status register mask, which ensures that the T bit has to be unset for the hook to be matched. We submitted a patch with this fix to the Linux kernel that at the time of writing has yet to be reviewed.

6.2.2 QEMU

If we remove the hidden instructions induced by Linux in the QEMU result, we end up with both A32 and T32 having 69,632 hidden instructions. Analysis of the side-effects of the hidden instructions combined with reference manual correlation revealed that the hidden instructions correspond to `VMUL (floating-point)` and `VQDMULL` instructions – both used to perform vector (SIMD) multiplication. Further analysis of the QEMU source code revealed that the root cause of the hidden instructions is a pair of bugs in the Arm instruction decode logic in QEMU that makes it regard certain instruction encoding bits in `VQDMULL` and the floating-point variant of `VMUL` as instruction options when they in fact should be a part of the opcode. This essentially reduces the opcode bit count while increasing the option bit count, which in turn makes certain unallocated instructions map to `VMUL` or `VQDMULL` instructions.

For the `VMUL (floating-point)` bug, consider the encoding in Figure 6.1. It's apparent that bit 20 exclusively indicates the operand size option (`sz`). However, QEMU includes bit 21 as a part of the size option too, effectively executing the instruction even when bit 21 is 1, which is unallocated.

31 30 29 28 27 26 25	24	23	22	21	20	19 16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1 0 0 1	1	0	D	0	sz	Vn	Vd	1 1 0 1	N	Q	M	1	Vm

Figure 6.1: A32 encoding of the `VMUL (floating-point)` instruction, as documented in the Armv8 Architecture Reference Manual [11].

Likewise, consider the encoding in Figure 6.2 for the `VQDMULL` bug. For certain vector instructions, bit 24 is used for the `U` option, indicating whether the instruction operates on signed or unsigned numbers. For `VQDMULL` however, this bit should always be 0, with 1 being unallocated. In spite of this, QEMU still executes the instruction as normal when `U` is 1.

31 30 29 28 27 26 25	24	23	22	21 20	19 16	15 12	11 10 9 8	7	6	5	4	3 0
1 1 1 1 0 0 1	0	1	D	!=11	Vn	Vd	1 1 0 1	N	0	M	0	Vm
				size								

Figure 6.2: A32 encoding of the `VQDMULL` instruction, as documented in the Armv8 Architecture Reference Manual [11].

The bugs are present in both A32 and T32 as a result of the two sharing much of the same vector decode logic, with the main encoding difference being in the uppermost opcode bits. For `VMUL` the location of the bit triggering the bug is the same for both A32 and

T32. For `VQDMULL` on the other hand, the bit triggering the bug is bit 28 (alternatively, bit 12 of the upper half-word) in T32 as opposed to bit 24 in A32, but the effect is otherwise identical.

We identified the root cause of both bugs and submitted corresponding patches to QEMU. At the time of writing, the `VQDMULL` patch has been included directly in the official development version of QEMU, set to be a part of a future release. The `VMUL` patch on the other hand has been included indirectly by the QEMU developers as part of an ongoing refactoring of the relevant code.

6.3 Disassembler Bug

When discussing the disassembly stage of the fuzzer in Section 4.4, it was mentioned that some bugs in the disassembler used – that is, the Arm disassembler in libopcodes – lead to incorrect results, making the fuzzer end up falsely identifying certain instructions as hidden. Most of these are not clear-cut bugs, but rather inconsistencies in marking UNPREDICTABLE instructions, similar to the issue with instructions with incorrect SBO/SBZ bits. However, we did identify a more clear-cut bug related to the disassembly of conditional `VDUP` instructions, which we fixed before fuzzing the target systems. We describe this bug next.

In A32 and T32, most vector instructions either don't have a condition code or are UNPREDICTABLE if the condition code is anything other than `0b1110` (always execute), which in practice makes them unconditional. However, this is not the case for `VDUP` (that duplicates a general-purpose register into a vector register), which can legally be conditional. Yet, libopcodes failed to correctly disassemble conditional `VDUP` instructions, disassembling them instead as unconditional `VDUP` instructions or undefined instructions.

For a demonstration of this incorrect disassembly, consider Listing 6.6 which shows the result of disassembling a file containing `VDUP` machine instructions with different condition codes, using the objdump utility (which uses libopcodes internally). In contrast, the correct disassembly result can be seen in Listing 6.7. Comparing the two, it is apparent that only the last instruction disassembled by objdump (before the fix) is correct.

The problem here is in fact twofold. First, libopcodes marks some (but not all) conditional `VDUP` instructions as undefined, stemming from erroneous instruction masks in the internal disassembly table. Second, the machine instructions that actually are disassembled to `VDUP` instructions miss the condition in the resulting instruction mnemonic, simply because this was not implemented in libopcodes for vector instructions. Accordingly, we fixed these issues and submitted a patch to GNU binutils (which libopcodes is a part of), which at the time of writing has been included in the official development version of GNU binutils and set to be a part of a future release.

```
$ objdump -b binary -m arm -D vdups.bin
...
00000000 <.data>:
    0:   0e800b10        vdup.32 d0, r0
    4:   1e800b10           ; <UNDEFINED> instruction: 0x1e800b10
    8:   2e800b10        vdup.32 d0, r0
    c:   3e800b10           ; <UNDEFINED> instruction: 0x3e800b10
   10:   4e800b10        vdup.32 d0, r0
   14:   5e800b10           ; <UNDEFINED> instruction: 0x5e800b10
   18:   6e800b10        vdup.32 d0, r0
   1c:   7e800b10           ; <UNDEFINED> instruction: 0x7e800b10
   20:   8e800b10        vdup.32 d0, r0
   24:   9e800b10           ; <UNDEFINED> instruction: 0x9e800b10
   28:   ae800b10        vdup.32 d0, r0
   2c:   be800b10           ; <UNDEFINED> instruction: 0xbe800b10
   30:   ce800b10        vdup.32 d0, r0
   34:   de800b10           ; <UNDEFINED> instruction: 0xde800b10
   38:   ee800b10        vdup.32 d0, r0
```

Listing 6.6: Incorrect disassembly of conditional `VDUP` instructions.

```
00000000 <.data>:
    0:   0e800b10        vdupeq.32       d0, r0
    4:   1e800b10        vdupne.32       d0, r0
    8:   2e800b10        vdupcs.32       d0, r0
    c:   3e800b10        vdupcc.32       d0, r0
   10:   4e800b10        vdupmi.32       d0, r0
   14:   5e800b10        vduppl.32       d0, r0
   18:   6e800b10        vdupvs.32       d0, r0
   1c:   7e800b10        vdupvc.32       d0, r0
   20:   8e800b10        vduphi.32       d0, r0
   24:   9e800b10        vdupls.32       d0, r0
   28:   ae800b10        vdupge.32       d0, r0
   2c:   be800b10        vduplt.32       d0, r0
   30:   ce800b10        vdupgt.32       d0, r0
   34:   de800b10        vduple.32       d0, r0
   38:   ee800b10        vdup.32 d0, r0
```

Listing 6.7: Correct disassembly of conditional `VDUP` instructions.

Chapter 7

Discussion

Fuzzing the target systems revealed a number of different hidden instructions, as presented in the results. In this chapter we discuss how the obtained results relate to the research goal of this thesis, followed up by a discussion of the potential security implications of the findings and some limitations to our approach.

7.1 Research Goal

The research goal of this thesis was to probe for hidden instructions in Armv8-A implementations, as formulated with the two research questions in the introduction. Although an Armv8-A processor can be implemented either in software or hardware, the vast majority of implementations are hardware-based, with software implementations primarily being used as a compatibility or development tool when hardware is not available. Consequently, our primary focus were on fuzzing hardware implementations – especially because of their black-box nature.

As presented in the results, the fuzzing efforts did not uncover any hidden instructions that could be attributed to hardware. This indicates that there are no hidden instructions present in the processors tested that can be revealed by executing undefined instructions – see the Limitations section below for a further discussion on this. However, the instructions identified in QEMU can in fact be categorized as hidden if we follow the definition we formulated in Section 2.2 in the background. They also differ from the Linux bugs in that they are caused by the underlying ISA implementation – which happens to be software-based – as opposed to being caused by the software running *on* the implementation.

We can therefore say that the QEMU result answers Research Question 1 (concerning particular implementations) in the positive, even though the instructions were caused by bugs and naturally not intentional. Research Question 2 (concerning all implementations) on the other hand has been answered in the negative, simply as a result of at least one of the ISA implementations tested not having any identified hidden instructions.

However, the fuzzing results did reveal an interesting aspect of the relation between the underlying ISA implementation and the operating system or software running on the implementation. Namely, the operating system can induce hidden instructions in the system that from a user's perspective will be indistinguishable from hidden instructions based in hardware, as is the case for Linux.

7.2 Security Implications

We indicated in the introduction that hidden instructions can lead to security issues on the affected systems. At the same time, our fuzzing efforts revealed hidden instructions in QEMU and bugs in Linux resulting in software-induced hidden instructions. A natural question is then to what extent the found bugs can compromise the security of the affected systems. Accordingly, we discuss the security implications of the QEMU and Linux bugs next.

The bugs in QEMU allowed us to execute certain undefined instructions as `VMUL` or `VQDMULL` instructions. This can in fact be used by programs to detect that they are being emulated with QEMU, by checking whether executing the particular instructions result in undefined instruction exceptions or not. Such emulator detection is a common dynamic analysis evasion technique as discussed in Chapter 3, and is exacerbated by the fact that QEMU is by far the most commonly used Arm emulator and form the basis of several Android malware analysis solutions. Although there in practice are many other ways to detect emulation too – ranging from reading system information to detecting performance traits particular to emulators – only executing a single test instruction is significantly faster and stealthier. For a proof of concept of this technique using one of the hidden instructions, see Appendix B.

The hidden instructions induced by Linux on the other hand are probably not as critical. Although they could be used to detect the presence of Linux and determine whether the Linux version in use was released before or after the corresponding bug fixes, much simpler methods for doing that exist, making the approach moot. However, the hidden `SETEND` instructions could potentially be used as part of an obfuscation scheme where the endianness of a program is changed at runtime using seemingly undefined instructions, making static analysis slightly more difficult.

On a related note, the issue disassemblers have with instructions with incorrect should-be-one/should-be-zero bits could also be used for obfuscation. For example, one could randomize the SBO/SBZ bits for each instruction in a program binary, making most of the instructions appear to be undefined when disassembled while still executing without faults. In fact, a proof of concept of this has already been made by Eric Davisson with his ARMaHYDAN tool [34].

7.3 Limitations

There are some limitations to the approach we have chosen in finding hidden behavior in processor implementations. Most importantly, the definition we use of hidden instructions only concern the direct execution of undefined instructions, without taking state variables like register contents into account. This means that our results don't rule out all kinds of hidden behavior, but rather only one method of hiding functionality in the processor.

For example, the hardware backdoor revealed by Domas [7] required a bit in a certain register to be set for the backdoor to be activated, although the complementary activation instruction required in that particular case was found through a method similar to ours. Other examples of methods for hiding behavior can be through interacting with certain memory addresses, executing particular streams of instructions, timing differences and so on. It is also conceivable that certain instructions have undocumented side-effects *in addition to* generating undefined instruction exceptions, which would not be detected by our fuzzer.

Another issue that can affect the instruction space coverage we achieve is a particular type of disasssembler inaccuracies. Through the use of filters we made sure that defined instructions are not mistaken as undefined ones, but not the other way around. That is, it is possible that bugs in the disassembler make certain instructions incorrectly disassemble to defined instructions when they in fact should be undefined and subsequently tested by the fuzzer. Although such instructions likely are low in numbers or don't exist – as they otherwise would have been noticed by users of the disassembler – their existence is still a possibility; proving otherwise would require verification of every entry in the internal disassembly table.

Finally, the number of tested systems is relatively low, despite our efforts to make it as high as practically possible. There are not many Armv8-A emulators available, so the emulator coverage is fairly good, but the number of Armv8-A implementations in the form of a combination of processor models and manufacturers far exceed what we have tested. We can therefore not fully exclude that Armv8-A-based systems with hidden instructions exist.

Chapter 8

Conclusion

The goal of this thesis was to investigate the existence of hidden instructions in Armv8-A processors, arising from a general lack of available research on security verification of black-box ISA implementations. To enable this investigation, we developed the *armshaker* fuzzer that can exhaustively search the instruction space of the A64, A32 and T32 instruction sets in the Armv8-A ISA and detect hidden instructions. The detection is done by executing presumably undefined instructions and then comparing the signal generated by the execution with the expected signal, utilizing techniques like instruction filtering and sandbox execution to achieve high search coverage, precision and stability. Using *armshaker*, we found two groups of hidden instructions in QEMU caused by a pair of decode bugs in its Arm implementation – which we subsequently fixed and submitted patches for – with the result of incorrectly executing certain undefined instructions as `VMUL` or `VQDMULL` instructions. We also identified various hidden instructions induced by the Linux operating system kernel – as opposed to the underlying ISA implementation – caused by the usage of incorrect instruction masks when trapping certain instructions. These instructions had side-effects ranging from debug breaking to the introduction of a novel 32-bit `SETEND` instruction in the T32 instruction set. However, despite fuzzing a variety of Armv8-A implementations in the form of different processor models and manufactures, we found no hidden instructions that could be attributed to the underlying hardware implementation of the tested systems.

Our *armshaker* fuzzer is open-source [9] and able to fuzz most Armv8-A-based systems capable of running Linux. The intention behind this is twofold. First, we want to let users audit their own systems for hidden instructions and raise awareness of the black-box nature of contemporary processors. Second, we hope that our efforts can serve as a basis for future work in the field of processor fuzzing. Next steps could be to extend the fuzzer to other instruction set architectures like MIPS or RISC-V, testing a wider range of systems, and developing techniques to detect more advanced types of hidden instructions.

Bibliography

[1] Linus Torvalds. *Linux (5.6)*. 2020. URL: `https://www.kernel.org/`.

[2] John R Ackerman. “Toward open source hardware”. In: *U. Dayton L. Rev.* 34 (2008), p. 183.

[3] Robert R Collins. “The Intel Pentium F00F Bug Description and Workarounds”. In: *Doctor Dobb’s Journal* (1997).

[4] Moritz Lipp et al. “Meltdown”. In: *arXiv preprint arXiv:1801.01207* (2018).

[5] Paul Kocher et al. “Spectre attacks: Exploiting speculative execution”. In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2019, pp. 1–19.

[6] Yuriy Shiyanovskii et al. “Process reliability based trojans through NBTI and HCI effects”. In: *2010 NASA/ESA Conference on Adaptive Hardware and Systems*. IEEE. 2010, pp. 215–222.

[7] Christopher Domas. “Hardware Backdoors in x86 CPUs”. In: *Black Hat USA* (2018).

[8] Christopher Domas. “Breaking the x86 ISA”. In: *Black Hat USA* (2017).

[9] Fredrik Strupe. *armshaker: Processor fuzzer targeting the Armv8-A ISA*. URL: `https://github.com/frestr/armshaker`.

[10] Andrei Frumusanu. *ARM Details Built on ARM Cortex Technology License*. 2016. URL: `https://www.anandtech.com/show/10366/arm-built-on-cortex-license`.

[11] *Arm® Architecture Reference Manual - Armv8, for Armv8-A architecture profile*. DDI0487E. Arm Limited. 2019. URL: `https://static.docs.arm.com/ddi0487/ea/DDI0487E_a_armv8_arm.pdf`.

[12] Michael Steil. “Reverse Engineering the MOS 6502 CPU”. In: *27th Chaos Communication Congress*. 2010.

[13] Ari Takanen et al. *Fuzzing for software security testing and quality assurance*. Artech House, 2018.

[14] Michal Zalewski. *American fuzzy lop*. URL: `http://lcamtuf.coredump.cx/afl`.

[15] *Intel® 64 and IA-32 Architectures Software Developer’s Manual*. Intel. 2018.

[16] Benjamin Schwarz, Saumya Debray, and Gregory Andrews. “Disassembly of executable code revisited”. In: *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.* IEEE. 2002, pp. 45–54.

[17] *VIA C3 Nehemiah Processor Datasheet*. Via Technologies Inc. 2004.

[18] Alastair Reid et al. “End-to-end verification of processors with ISA-Formal”. In: *International Conference on Computer Aided Verification*. Springer. 2016, pp. 42–58.

[19] Alastair Reid. “Trustworthy specifications of ARM® v8-A and v8-M system level architecture”. In: *2016 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE. 2016, pp. 161–168.

[20] H-P Klug. “Microprocessor testing by instruction sequences derived from random patterns”. In: *International Test Conference 1988 Proceeding @ New Frontiers in Testing*. IEEE. 1988, pp. 73–80.

[21] Shajid Thiruvathodi and Deepak Yeggina. “A random instruction sequence generator for ARM based systems”. In: *2014 15th International Microprocessor Test and Verification Workshop*. IEEE. 2014, pp. 73–77.

[22] Lorenzo Martignoni et al. “Testing CPU emulators”. In: *Proceedings of the eighteenth international symposium on Software testing and analysis*. 2009, pp. 261–272.

[23] Thomas Raffetseder, Christopher Kruegel, and Engin Kirda. “Detecting system emulators”. In: *International Conference on Information Security*. Springer. 2007, pp. 1–18.

[24] Thanasis Petsas et al. “Rage against the virtual machine: hindering dynamic analysis of android malware”. In: *Proceedings of the Seventh European Workshop on System Security*. 2014, pp. 1–6.

[25] Yiming Jing et al. “Morpheus: automatically generating heuristics to detect android emulators”. In: *Proceedings of the 30th Annual Computer Security Applications Conference*. 2014, pp. 216–225.

[26] Onur Sahin, Ayse K Coskun, and Manuel Egele. “Proteus: Detecting Android Emulators from Instruction-Level Profiles”. In: *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer. 2018, pp. 3–24.

[27] Michael Göebel. “Developing and Verifying Methods to Search for Hidden Instructions on RISC Processors”. Leiden University, 2019.

[28] Rens Dofferhoff. “A Performance Evaluation of Platform-Independent Methods to Search for Hidden Instructions on RISC Processors”. Leiden University, 2019.

[29] Nguyen Anh Quynh. “Capstone: Next-gen disassembly framework”. In: *Black Hat USA* (2014).

[30] The Free Software Foundation (FSF). *GNU Binutils*. URL: `https://www.gnu.org/software/binutils/`.

[31] Fabrice Bellard. “QEMU, a fast and portable dynamic translator.” In: *USENIX Annual Technical Conference, FREENIX Track*. Vol. 41. 2005, p. 46.

[32] Fredrik Fornwall. *Termux*. URL: `https://termux.com/`.

[33] Russell King and Robin Murphy. *Re: [RFC PATCH] arm: Don't trap conditional UDF instructions*. (Linux Kernel Mailing List). URL: `https://lkml.org/lkml/2020/5/13/1295`.

[34] Eric Davisson. "Undefining the ARM". In: *PoC* || *GTFO* Issue 0x19 (Mar. 2019). Ed. by Manul Laphroaig et al., pp. 17–20.

Appendices

Appendix A

Fuzzer Options

```
$ ./fuzzer -h
Usage: ./fuzzer [option(s)]

General options:
    -h, --help              Print help information.
    -q, --quiet             Don’t print the status line.

Search options:
    -s, --start <insn>      Start of instruction search range (in hex).
                            [default: 0x00000000]
    -e, --end <insn>        End of instruction search range, inclusive (in
                            hex). [default: 0xffffffff]
    -i, --single-exec       Execute a single instruction (i.e., set end=start).
    -m, --mask <mask>       Only update instruction bits marked in the supplied
                            mask. Useful for testing different operands on a
                            single instruction. Example: 0xf0000000 -> only
                            increment most significant nibble.

Execution options:
    -n, --no-exec           Calculate the total amount of undefined
                            instructions, without executing them.
    -x, --exec-all          Execute all instructions (regardless of the
                            disassembly result).
    -f, --filter <level>    Filter away (skip) certain instructions that would
                            otherwise be executed and might generate false
                            positives. Supports the following levels, where
                            each level includes the numerically lower ones:
                                1: Inaccurate disassemblies, primarily caused
                                   by SBO/SBZ bits.
                                2: Hidden instructions induced by Linux
```

```
                                   as a result of bugs or backwards
                                   compatibility measures.
    -p, --ptrace            Execute instructions on a separate process using
                            ptrace. This will generally make execution slower,
                            but lowers the chance of the fuzzer crashing in
                            case hidden instructions with certain side-effects
                            are found. It also enables some additional options.
    -c, --cond              On AArch32: Set the condition flags in the CPSR to
                            match the condition code in the instruction
                            encoding. This ensures that undefined instructions
                            with a normally non-matching condition code won’t
                            be skipped, as is the case in some ISA
                            implementations.

Logging options:
    -l, --log-suffix        Add a suffix to the log and status file.
    -d, --discreps          Log disassembler discrepancies.

Ptrace options (only available with -p option):
    -t, --thumb             Use the thumb instruction set (only available on
                            AArch32). Note: 16-bit thumb instructions have the
                            format XXXX0000. So to test e.g. instruction 46c0,
                            use 46c00000.
    -r, --print-regs        Print register values before/after instruction
                            execution.
    -z, --random            Load the registers with random values, instead of
                            all 0s. Note that the random values are generated
                            at startup, and remain constant throughout the
                            session.
    -g, --log-reg-changes   For hidden instructions, only log registers that
                            changed value.
    -V, --vector            Set and log vector registers (d0-d31, fpscr) when
                            fuzzing.
```
Listing A.1: Command-line options for the fuzzer.

```
$ ./armshaker.py -h
usage: armshaker.py [-h] [-s INSN] [-e INSN] [-c] [-w NUM] [-p] [-n]
                    [-f LEVEL] [-t] [-z] [-g] [-V] [-c]

fuzzer front-end

optional arguments:
  -h, --help            show this help message and exit
  -s INSN, --start INSN
                        search range start
  -e INSN, --end INSN   search range end
  -d, --discreps        Log disassembler discrepancies
  -w NUM, --workers NUM
                        Number of worker processes
  -p, --ptrace          Use ptrace when testing
  -n, --no-exec         Don't execute instructions, just disassemble them.
  -f LEVEL, --filter LEVEL
                        Filter certain instructions
  -t, --thumb           Use the thumb instruction set (only on AArch32).
  -z, --random          Load the registers with random values, instead of all
                        0s.
  -g, --log-reg-changes
                        For hidden instructions, only log registers that
                        changed value.
  -V, --vector          Set and log vector registers (d0-d31, fpscr) when
                        fuzzing.
  -c, --cond            Set cpsr flags to match instruction condition code.
```

Listing A.2: Command-line options for the fuzzer front-end.

Appendix B

QEMU Detection Proof of Concept

Listing B.1 shows a C program that when run detects whether it is being emulated or not, applicable to QEMU version 5.0.0 and lower. It works as follows. First, a function that sets a particular variable upon being called is installed as a signal handler for the `SIGILL` signal. Then an undefined instruction that maps to `VMUL` in QEMU is executed. In a correct ISA implementation, this will result in a `SIGILL` signal being sent to the process, which subsequently calls the signal handler and sets the `handler_activated` variable. In QEMU however, the undefined instruction will execute as a `VMUL` instruction instead, with no signal being generated. Consequently, if the `handler_activated` variable is not set after executing the undefined instruction, it means the current process is being emulated with QEMU.

```
#include <stdio.h>
#include <signal.h>
#include <ucontext.h>

volatile sig_atomic_t handler_activated = 0;

void sigill_handler(int sig_num, siginfo_t *sig_info, void *uc_ptr)
{
    handler_activated = 1;

    // Skip the illegal instruction
    ucontext_t* uc = (ucontext_t*) uc_ptr;
    uc->uc_mcontext.arm_pc += 4;
}

int main()
{
    struct sigaction s = {
        .sa_sigaction = sigill_handler,
        .sa_flags = SA_SIGINFO
    };

    sigfillset(&s.sa_mask);
    sigaction(SIGILL, &s, NULL);

    // 'VMUL.F32 D0, D0, D0' with bit 21 set.
    asm (".inst 0xf3200d10");

    if (handler_activated == 0) {
        printf("I'm being emulated.\n");
    } else {
        printf("I'm not being emulated.\n");
    }

    return 0;
}
```

Listing B.1: QEMU detection code.