IAIK

SCIENCE ▪ PASSION ▪ TECHNOLOGY

TU Graz

Nikolaus M. Mutewsky

Undocumented Instructions in the Armv8-A Instruction Set Architecture

BACHELOR'S THESIS

Bachelor's degree programme: Computer Science

Supervisors

Catherine Easdon, MEng

Institute of Applied Information Processing and Communications
Graz University of Technology

Graz, August 2020

Abstract

Historically, faulting instructions and associated hardware bugs have proven that serious consequences may arise from incorrect behavior of hardware systems. Inspired by famous bugs like Pentium f00f and Domas' work on undocumented instructions in the x86 instruction set, we investigate undocumented instructions within the Arm instruction set in this thesis. Exhaustive fuzzing of the instruction set space and observation of the resulting CPU state on four CPU microarchitectures has revealed a significant amount of invalid instruction encodings that are decoded as valid instructions. All of these can be attributed to constrained unpredictable behavior as defined in the Arm instruction set architecture. In the process, several bugs in the disassemblers used and the virtual machine emulator QEMU have been encountered. We propose a novel fuzzing algorithm that enables the fast and failure robust traversal of the Arm instruction set space. This is achieved by using a low-level system disassembler to focus on instructions that are not recognized as valid to the system. We put multiple hardware implementations of the Armv8-A AArch64 instruction set under test and compare and report the experimental results.

Keywords:

Arm, Armv8-A, AArch64, A64, Undocumented Instructions, Instruction Fuzzing, Instruction Disassembly, Constrained Unpredictable Behavior

Contents

1 Introduction **5**

2 Background **7**
2.1 Instruction Set Architectures 7
2.1.1 Endianness in Armv8-A 8
2.1.2 Constrained Unpredictable Behavior 8
2.1.3 Instruction Layout 9
2.2 CPU-Architectures 10
2.2.1 Self-Modifiying Code 10
2.2.2 In Order Execution 10
2.2.3 Signals, Exceptions and Interrupts 10
2.2.4 Privilege and Exception Levels 11
2.3 Related Work 13

3 Threat Model **14**

4 Design **15**
4.1 Instruction Generation 15
4.2 Instruction Disassembly 15
4.2.1 Ground Truth and Accuracy of Disassemblers 16
4.2.2 Disassembler Result Parsing 16
4.3 Executing Unknown Instructions and CPU State 17
4.4 Multi-Core 17
4.4.1 Parallelization 17
4.5 Stack Corruption and Signal Stacks 18
4.6 Branches and Jumps 18
4.7 Register Zeroing 19
4.8 Memory Cage 19
4.9 Signal Handling 20
4.10 System State Recovery through Signals 21
4.11 Logging 21
4.11.1 Delayed Logging 23
4.11.2 Log Flushing 23
4.12 Log Analysis 23
4.13 Instruction Fuzzing Program Flow 23
4.14 Implementation and Features of the Fuzzing Tool 25
4.14.1 Argument parameters 25

5 Results **27**
5.1 Experimental Setup 27
5.2 Instruction Fuzzing Tests 27

6 Future Work **31**

7 Conclusion **32**

Bibliography **33**

1 Introduction

Ever since the advent of digital computers, bugs have been a part of software and hardware systems. While the first bug was an actual insect in the relay of the processing unit [1], nowadays, other causes are more likely. Today, when a bug is discovered, it is usually in software. Software bugs are easily patched through updates, often quickly over the internet. However, with hardware bugs, it is much harder, if not impossible, to patch after the chip has been manufactured. Additionally, a given CPU from a prominent vendor may be used across millions of devices, which makes a recall very costly. If hardware features, which are engineered to improve computing performance and are an integral part of the design of a system, are found to be the culprit of a hardware bug, the consequences can be devastating [2, 3, 4, 5].

Writing software nowadays is much easier compared to when computers were just beginning to emerge, since programmers using high-level programming languages do not need a deep understanding of the intrinsic properties of a CPU. Modern high-performance systems and software require powerful hardware that supports it. Thus, these systems often require very complex hardware implementations and instruction sets, which make hardware bugs much more likely. Introduced in 1985, Arm is a RISC Instruction Set architecture for CPUs. As opposed to the x86 instruction set, which has grown to be extremely complex through historical reasons, the Arm architecture aims to reduce complexity. However, even less complex Arm CPUs contain very sophisticated hardware, and often dozens of different architectural instruction set extensions. Multiple cores, shared resources like caches, and complicated execution parallelization make Arm CPUs wonders of modern engineering. Design flaws and hardware bugs may not immediately become apparent in design and production phases, since the ISA is defined by Arm and licensed to manufacturers, which use their specific implementations of the standard.

Due to the popularity of processors using the x86 instruction set, much research has been conducted focusing on its security focusing on undocumented instruction behavior as detailed by Collins [6] and Domas [7]. For the Arm and Armv8-A architecture in particular, however, there has been much less interest. This thesis focuses on the Armv8-A architecture specification and its implementations. Armv8-A, the successor of Armv7, is highly relevant because it is the first Arm architecture bringing fundamental changes to Arm CPUs, i.e., 64-bit operation, hardware cryptographic extensions, and advanced floating-point units. The simplified instruction sets, lower overall power consumption, and higher computational efficiency make Armv8-A CPUs a viable candidate for a wide range of devices currently used in millions of devices worldwide. Found in mobile devices like smartphones, tablets, laptops, IoT-devices, but also servers like the AWS Graviton, Arm CPUs make up 34% of all SoC's and over 166 billion Arm-based chips shipped in Q4 2019, based on the Arm Limited Roadshow [8]. The strong market pervasiveness

and widespread use of CPUs using the Armv8-A microarchitecture makes potential architectural or implementational bugs extremely dangerous.

As demonstrated by the f00f bug [6], instructions exist, that can wreak havoc on a faulty system. Therefore it is of great interest to find potentially fatal instructions on modern CPUs. Since Arm uses a reduced instruction set consisting of only 4 bytes or less per instruction [9], in a reasonable time, an exhaustive search of the instruction space is possible. This fact makes Arm architectures a candidate for exhaustive instruction fuzzing, as presented in this thesis.

Contributions

- Description of a detailed fuzzing algorithm to dynamically generate and execute instructions while preserving system stability.
- A tool that implements the fuzzing algorithm, enabling fast exhaustive instruction fuzzing on any AArch64 compatible CPU.
- Experimental results and evaluation of the undocumented instruction encodings on 4 CPUs.

Overview

In chapter 1, an introduction to instruction fuzzing is given. Relevant background information on CPU architectures, instruction sets, instruction decoding, signals, traps, exceptions, and interrupts is given in chapter 2. Related works in similar topics are reviewed in section 2.3. A threat model is given in chapter 3. The design of the fuzzing algorithm, considerations, limitations, and implementation details of the fuzzing tool are discussed in chapter 4. In chapter 5, the fuzzing test results are interpreted and discussed. Possible future work is reviewed in chapter 6. Chapter 7 contains a conclusion on the thesis.

2 Background

2.1 Instruction Set Architectures

An instruction set architecture (ISA), or architecture, defines an abstract model of a computer, usually comprised of supported data types, address formats, instruction sets, register- and memory layout, and state. Software written and compiled for a specific ISA can run on any processor supporting it. ISAs ensure backward compatibility for future hardware evolutions. To produce CPUs based on an ISA, manufacturers have to obtain a specific license.

An example of a current ISA family is x86, in use mostly in desktop and server hardware. In this market, it dominates over other ISAs in popularity. Due to historical reasons, x86 has grown very complex, with many additions and extensions added. For this reason, x86 is considered a family of ISAs rather than a specific one. Intel and AMD are the two leading manufacturers of x86 CPUs currently.

An alternative to x86 is the Arm ISA family, which has several different versions. Armv8-A, often called Armv8, is the 8th major version of the Arm architecture, the successor to Armv7. It offers features such as optional 64-bit operation, and cryptographic and advanced floating-point hardware. Arm architectures, and Armv8-A especially, are used in the majority of smartphones, tablets, SoCs and other mobile devices. Arm, in contrast to Intel and AMD, designs and licenses designs of hardware components as well as ISAs. They then sell manufacturing licenses to third party chip producers.

Execution States in Armv8-A

An execution state in Armv8-A is defined by the register widths, supported instruction sets, significant aspects of the exception model, Virtual Memory System Architecture, and the programmers model. Armv8-A has two execution states, AArch64 and AArch32. In AArch64, 64-bit addresses and registers are used, whereas in AArch32, the 32-bit equivalent is used [9].

Instruction Sets in Armv8-A

AArch64 uses the A64 instruction set and supports 64-bit addressing, while the AArch32 supports 32-bit addressing and the A32 and T32 instruction sets, which are compatible with earlier versions of Arm architectures. This thesis focuses on the A64 instruction set. The A64 instruction sets uses fixed-length 32-bit instruction encodings [9].

2.1.1 Endianness in Armv8-A

This section describes the relationship of endianness in Armv8-A and the implementation. Endianness refers to the order of bytes in memory. In little-endian, bytes with the least significance are stored in lower addresses. Bytes with the most significance are stored in higher addresses, vice versa for big-endianness. The endianness of instructions and data can differ in Armv8-A architectures.

Instruction endianness

Instructions are always little-endian on Armv8-A [9].

Data endianness

Data endianness can be configured on the fly separately for every exception level [9].

2.1.2 Constrained Unpredictable Behavior

Armv8-A defines architectural constraints on unpredictable behaviors. Unpredictable behavior describes features of the architecture that software must not use. Only behavior from a selected range is permittable. This selected range is called Constrained Unpredictable Behavior. Depending on the manufacturer, the behavior may differ between hardware implementations, while still fulfilling the architectural constraints, as described by the Arm Architecture Reference Manual [9].

An example of Constrained Unpredictable Behavior for the LDP instruction is given in figure 2.1.

LDP

For a description of this instruction and the encoding, see *LDP* on page C6-994.

CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `(t == n || t2 == n) && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs a load using the specified addressing mode, and the base register is set to an UNKNOWN value. In addition, if an exception occurs during such an instruction, the base register might be corrupted so that the instruction cannot be repeated.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value `0x0`.

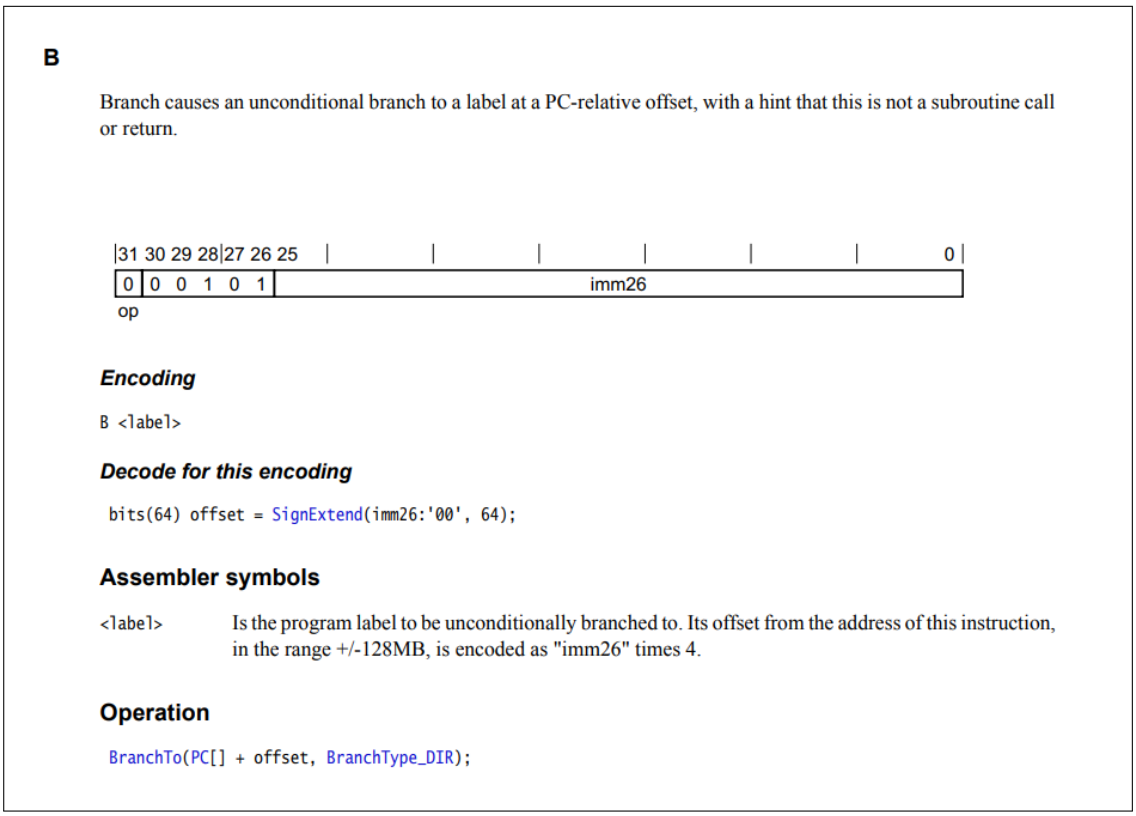
Figure 2.1: Excerpt from the Arm Architecture Reference Manual on Constrained Unpredictable Behavior of the LDP instruction [9].

2.1.3 Instruction Layout

A64 instructions have a fixed length of 32 bit. All instructions consist of a bit-pattern specifying the instruction type and optional fields for arguments. An example of the B (branch) instruction from the Arm Architecture Reference Manual [9] is given in figure 2.2.

In the branch instruction, bits 31-26 specify instruction opcode, and bits 25-0 specify the branching offset relative to the address of this instruction as *imm26* times 4.

For A64 instructions, there exist a number of different opcode and parameter layouts. Some similar instruction families include Branching, Store-/Load, and Dataprocessing. Due to the inconsistency between different instruction encodings' opcode and parameter layouts, it is infeasible to develop an algorithm incorporating the instruction encoding layout.



B

Branch causes an unconditional branch to a label at a PC-relative offset, with a hint that this is not a subroutine call or return.

31	30	29	28	27	26	25	...	0
0	0	0	1	0	1	imm26		
op								

Encoding

```
B <label>
```

Decode for this encoding

```
bits(64) offset = SignExtend(imm26:'00', 64);
```

Assembler symbols

<label>	Is the program label to be unconditionally branched to. Its offset from the address of this instruction, in the range +/-128MB, is encoded as "imm26" times 4.

Operation

```
BranchTo(PC[] + offset, BranchType_DIR);
```

Figure 2.2: Excerpt from the Arm Architecture Reference Manual on the branch instruction [9].

2.2 CPU-Architectures

This section dives deeper into the hardware-specific concepts used in hardware implementations of CPUs relevant to this thesis.

2.2.1 Self-Modifiying Code

Modern CPUs use a multitude of caches and cache levels to achieve high computing performance. For fuzzing, instructions are dynamically generated, written to memory, and executed. Caching presents a challenge to the fuzzing procedure and, more generally, dynamic code execution.

Data and instructions are cached independently of one another in Armv8-A in caches called, respectively, the D- and I-cache. Every processing unit has a private L1 cache, which consists of one D- and one I-cache. The processor only ever directly executes instructions from the I-cache and reads data from the D-cache. It is not possible to read data from the I-cache or execute instructions from the D-cache.

The cores share one L2 cache.

When the CPU writes data to memory, it is cached in the D-cache first, under the assumption that it will be accessed soon again. The actual memory and the I-cache, however, still contain old data. If the processor then executes instructions from that address, it will use old instructions from the I-cache, which is not what we intended to do.

The processor may update the I-cache and memory at some point, but we can not rely on this behavior if we want to execute our generated instructions. This cache invalidation needs to be triggered manually. Therefore after writing the instruction to the D-cache, we flush the D-cache to memory and invalidate the I-cache before executing the instruction. When the processor attempts to fetch the instruction from the invalidated I-cache, it fetches the updated instruction from memory. Therefore the executed instruction at the memory address is under all circumstances the instruction that we wrote to that address, as described by the Arm Developer Article on caches and self-modifying code [10].

GCC offers the function *__clear_cache* to achieve the D-cache flush and I-cache invalidation.

2.2.2 In Order Execution

Armv8-A does not explicitly define an execution order. The hardware implementation is free to implement either in- or out-of-order execution.

2.2.3 Signals, Exceptions and Interrupts

Exceptions are generated or raised when the CPU encounters an error state, such as a division by zero, invalid instruction, or unprivileged memory access. Execution is stopped, and control of the executing thread is handed over to a higher privileged execution state, usually the operating system or supervisor. The higher privileged level then may issue a

signal to the process or handle the exception differently. AArch64 defines exceptions as the "modification" of sequential instruction execution in regular operation [11].

Interrupts commonly are also referred to as interrupt signals. They are generated asynchronously to normal execution, for example, from hardware or peripherals. They behave similarly to exceptions.

Signals are generated by the operating system and sent to the relevant process. Depending on the type, a handler can be registered to handle the signal. An example is the *SIGSEGV* signal generated by Linux for memory-related faults. *SIGKILL* is an example of a signal that can not be handled by a process.

When executing invalid instructions, an invalid instruction exception may be raised, which hands control over to the operating system. The operating system then calls the registered handlers (if any) of the affected process with the execution context, processor state, and signal code. The program can then handle the signal and return to regular operation.

A complete description of the differences and specifics of signals, exceptions, and interrupts is given in the Arm Developer Documents on AArch64 Exception and Interrupt Handling [11].

2.2.4 Privilege and Exception Levels

This section describes privilege levels in processors and how they relate to the instruction fuzzing process.

Modern software expects and relies on CPU features such as privilege and execution levels. To securely implement multi-level software, the concept of privilege levels is necessary.

In Armv8-A there exist two types of privilege, memory and register access privilege, as well as four exception levels [12]. Switching between different levels of privilege is only possible on exceptions, which is why they are referred to as exception levels. The four exception levels are demonstrated by figure 2.3.

Theoretically, instruction fuzzing is possible on all exception levels. However, of most interest and use is instruction fuzzing on the application level, for several reasons:

1. Attack Vectors: An undocumented instruction that causes system instability but is only executable on, for example, hypervisor level has a small attack surface because most systems only allow execution on the application level for regular users.

2. System stability: Fuzzing instructions can cause instability of the fuzzing process. If the fuzzing process runs on the application level, in the worst case, it crashes, while the rest of the system remains stable. If the fuzzing process runs on the OS or hypervisor level, it causes much greater, possibly permanent damage to data, software, and hardware.

The privilege levels serve as additional protection of the system from unwanted effects. For these reasons, the fuzzing process runs only with the lowest privilege at the application level.

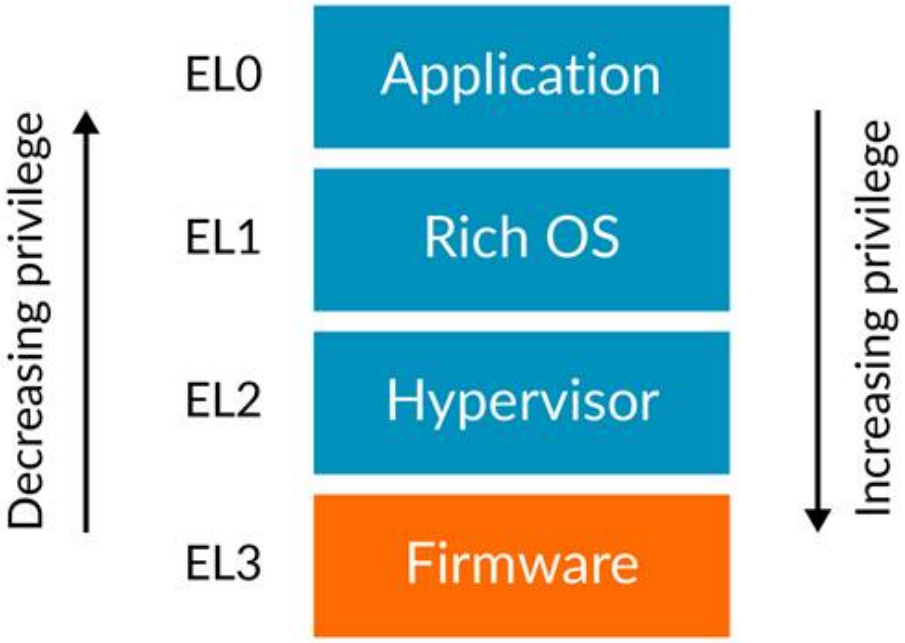


Figure 2.3: Exception Levels from the Arm Website [12]

2.3 Related Work

An anonymous user of the comp.os.linux.advocacy forum made significant progress in the world of instruction fuzzing by finding the f00f bug on an Intel CPU in 1997, which was later described in more detail by Collins [6]. They found an instruction that leads to a halt-and-catch-fire state on the Intel P5 architecture. In this context, halt-and-catch-fire means that the CPU gets stuck and stops executing instructions, only a hard reset can recover it.

Domas' [7] work on instruction fuzzing CPUs using the x86 instruction set has made large progress in instruction fuzzing. Their tool, called Sandsifter, manages to fuzz the x86 instruction set space efficiently and fast. Unfortunately, they did not release any investigation or implementation for the Arm architectures.

These famous works showcase that hardware bugs are a severe problem in previous, but also modern CPUs.

Li et al. [13] claim to have developed an efficient fuzzing method that works faster than Sandsifter, but unfortunately, no implementation or further results have been published.

Zhu et al. [14] propose an algorithm that enables fast instruction fuzzing on several microarchitectures, however, no implementation has been published.

Dofferhoff [15] and Göbel [16] offer an exciting approach to fuzzing RISC-CPUs with performant methods. Their *memory cage* method has been adapted and successfully used in this paper. They focus on evaluating the performance of the methods for fast instruction fuzzing and including valid instructions encodings in their tests. Their approach differs in the use of a manager thread and relies primarily on the memory protection of the CPU, whereas this thesis uses the BRK instruction as failsafe. As ground truth, they utilize the disassembler Capstone instead of the more reliable libopcodes-based one, and for verification, they utilize the QEMU virtual machine emulator, which we found to have numerous bugs, see section 5.2.

3 Threat Model

An undocumented instruction may cause any behavior with respect to physical laws. This behavior includes but is not limited to locking the CPU and leading to thermal runaway, burning internal processor fuses, or damaging peripheral hardware as demonstrated by Dadvar and Skadron [17] and Miller [18].

However, modern CPUs have independent power management units that prevent thermal runaway by issuing a hard reset when core temperature exceeds a dangerous limit.

Besides physical damage, it is entirely possible that sleeping malicious programs, hidden in the microcode level within the CPU, wait to be awakened by undocumented instructions. This code may be put there intentionally by the CPU designer, manufacturer, by software updates to the microcode or as demonstrated by Koppe et al. [19] by malicious software on the OS level. Even though Arm uses the RISC instruction set that does not necessitate a complex and patchable ROM-based microcode level, some features of their instruction decoding may still be vulnerable.

Infiltrated microcode is a severe security risk since it invalidates any guarantees on confidentiality, integrity, and availability of the affected system. In this case, the undocumented instruction is not the direct cause of the damages, but rather the trigger.

While an undocumented instruction may not directly grant native code execution, a process may be able to interact with other processes on the same core or CPU due to locality. This can be fatal on cloud servers, where execution and hardware are shared with a significant number of other users.

A virus exploiting other existing vulnerabilities to gain execution rights may be able to take out specific targets with a hardware-damaging halt-and-catch-fire instruction.

Since Armv8-A is a pervasive CPU architecture spanning across many fields such as consumer- and IoT-devices, cloud servers, and scientific equipment, a fault in the architecture would have a very far-reaching outcome, if exploited.

4 Design

4.1 Instruction Generation

This section describes the method used for systematically generating instructions in the fuzzing procedure.

Instructions will be referred to with their 8-character long equivalent opcode in hexadecimal notation in big-endian order. Alternatively, the abbreviation for the instruction may be used. For example, 0x20d40000 is the opcode equivalent for the BRK instruction. The BRK instruction raises an exception that is handled by the kernel. If a user signal handler for *SIGTRAP* signals is registered, it will be called by the kernel with signal info and exception context.

Instructions in Armv8-A are always 32 bit long and in little-endian order [9]. This results in

$$2^{32} = 4294967296$$

possible instruction encodings. Therefore, an exhaustive search of the instruction space is possible in a reasonable time. Starting from the instruction 0x00000000, the algorithm generates instructions incrementally until all possible instruction encodings are tested.

A different approach, such as random generation, is possible. In tests, this has proven to be very useful for quickly getting results on the stability of the fuzzing tool under development and on undocumented instructions. Random generation uniformly tests the instruction space, whereas incremental in-order generation may take a long time to reach a particular instruction family or instruction subspace.

When writing the instruction to memory, the little-endianness of instructions in Armv8-A architectures needs to be considered. The four-byte instruction is split into four one-byte chunks and written to memory, starting with the least significant byte on the lowest address location, to the most significant byte on the highest address location.

4.2 Instruction Disassembly

This section gives an overview of different disassemblers and disassembly strategies in Armv8-A instruction fuzzing.

Since the goal is to find undocumented instructions, documented instructions are not tested in the instruction fuzzing procedure. This speeds up the process of testing considerably since a large number of documented instructions can be skipped. Additionally, stability is improved because many documented instructions like branches enter system states that can be difficult to handle and recover from. Therefore, a way to test a generated instruction for validity before execution is needed.

Instruction disassembly presents a solution. After generating an instruction, the instruction encoding is analyzed with a disassembler. The disassembler represents the ground truth, which decides if a given instruction is valid or not. To achieve reasonable run times for a full fuzzing procedure, this procedure must be accessible through an API. When the disassembler does not recognize an instruction, it is executed, and the resulting system state is analyzed. If the instruction is recognized, this means that it is documented, and we move on to generating the next instruction, skipping the current one.

4.2.1 Ground Truth and Accuracy of Disassemblers

In theory, a perfect disassembler with 100% accuracy would be the preferred system for instruction disassembly. 100% accuracy is defined such that the disassembler reports a valid instruction for every documented instruction encoding, and an invalid instruction for every undocumented instruction encoding. Vixl [20] is a A64 assembler and disassembler maintained by Arm. Due to it being less widely used than libopcodes we chose to use libopcodes.

Libopcodes is the GNU disassembly framework used for many Linux distributions. It provides very accurate disassembly results for AArch64 and is used as base for popular tools such as objdump. Unfortunately, it is not intended to be used outside of internal Linux distributions and therefore not documented well. Adaption to the use in this project was a significant hurdle and took significant amounts of time. Nonetheless, it is used as the main disassembler in this project. Even after implementing a disassembly function that outputs if an instruction encoding is valid or not, it still relies on parsing the string output by the lower-level function for a substring. No additional information, in contrast to x86 disassembly, except the equivalent assembler string, is returned. It is kept up to date by the maintainers and has full support for Arm architectures. No direct bugs concerning the disassembly of AArch64 instructions were encountered while using libopcodes.

Capstone is a widely used open-source multi-architecture open source disassembler. Initially, some tests were performed with Capstone; however, it was quickly realized that it does not have the desired 100% accuracy. Bugs in Capstone result in the false reporting of invalid instruction encodings. This causes valid instructions to be executed, causing process instability and crashes of the fuzzing process.

4.2.2 Disassembler Result Parsing

The generated instruction is passed to the disassembly checking function. The libopcode disassembly function is obtained by fetching the appropriate dissassembler for AArch64 and calling it with a disassembly info struct containing the instruction and other system-relevant data as a parameter. The libopcodes disassembler only reports a string result for each disassembled instruction. This string either contains the assembler code of an instruction if it is valid, or the substring *UNDEFINED* or *undefined* in case of an invalid

instruction. The string is checked for this occurrence, and the disassembly checking function returns *true* if it does not contain the substring.

4.3 Executing Unknown Instructions and CPU State

Executing unknown instructions comes with a few risks. An undocumented instruction can theoretically cause any behavior of the CPU . As described in the following chapters, several precautions and techniques need to be employed to prevent loss of control flow in the fuzzing process.

To determine if an undocumented instruction is of interest, the architectural state needs to be analyzed before, during, and after its execution. This presents a significant challenge. One solution is to analyze the register state of the CPU.

Since an undocumented instruction should not change the CPU state, the register state before the execution should be precisely equivalent to the register state after the execution of the instruction under test. This is not a trivial problem since dumping the register state means writing to memory. Writing to memory always entails an address or stack offset present in one of the registers. This, unfortunately, changes the register state. Due to this contradiction, it is not possible to compare all registers before and after execution.

In practice, this was not an issue, since no undocumented instructions that execute were found. A more elaborate analysis of the CPU's architectural state is possible using performance counters, as demonstrated by Domas [7]. This is out of the scope of this project and reserved for future work.

4.4 Multi-Core

Most modern CPUs contain more than one core in the same processor. They often share caches and other resources. When executing an instruction under test, the CPU state is vital to achieving reproducible results. If the fuzzing process thread was switched out of context and rescheduled on another core between the setup of the CPU state and the instruction's execution, the behavior is undefined.

Some CPUs have different core architectures on a single processor. For example, the Nvidia Tegra TX2 CPU used in the tests, has 2 Denver and 4 Cortex A-57 cores [21]. To specifically target a core architecture, locking to a single core is essential. Therefore the fuzzing process is locked to a single core.

4.4.1 Parallelization

Instruction fuzzing on multi-core processors is possible and desirable since it allows parallelization of the fuzzing procedure over several cores if the execution is locked onto a specific core, as described in section 4.4. By splitting the instruction set space by the number of cores available and assigning each fuzzing process an equal range of instructions

to test, this theoretically allows for a speed-up by the number of available cores. This method was deemed out of the scope of this project and therefore not implemented.

4.5 Stack Corruption and Signal Stacks

Any executed instruction may cause a write to a memory location. Many documented write instructions use addressing relative to the stack pointer, where a register value determines the relative offset. If the relative address is within the current thread's stack, it can overwrite memory on the stack and crash the process. Thus, it is beneficial to keep all runtime-critical variables in the data segment and not on the stack. Since the stack memory is located in a much higher address space than the data segment, relative writes cannot reach the data segment.

However, modern operating systems and compilers employ various measures for preventing stack corruption, which results in the process crashing, even if no runtime-critical variables were overwritten. ASLR [22] and GCC's stack canaries are two primary protective measures. Disabling these measures is not always possible or practical, so to improve the fuzzing process' stability, signal stacks are used.

After executing an instruction, control flow is regained through signals as described in section 4.10. If the signal handler were to use the same stack as the thread that executes the instructions, the process might crash in the signal handler routine. Separate signal handler stacks are used to prevent this. Therefore even if the thread that executes instructions corrupts its own stack, the signal handler can regain control of the program and exit gracefully.

Both absolute writes and relative writes can be mitigated by using no runtime-critical variables on the stack, separate signal handler stacks, and register zeroing described in section 4.7 and the memory cage in section 4.8.

4.6 Branches and Jumps

This section describes the effects of executing branching instructions in the fuzzing process. When a branch instruction is executed, several things can happen:

1. It branches to an unmapped area of memory.
2. It branches to mapped memory containing data.
3. It branches to mapped memory containing code.
4. It branches to the same address as the executed instruction.

To regain control flow after the execution of a branching instruction, it is important to mitigate these. In case 1, no additional measures need to be taken, since this generates an exception and is caught in the signal handling routine as described in section 4.10.

Case 2 is of no concern since code execution is by default disabled on data pages. It will generate an exception, and the signal handler will recover program control.

Case 3 presents a problem that is solved in two aspects. If the instruction is an absolute branch, it uses the address contained in a register. Register zeroing as described in section 4.7 prevents this. Branching to address 0 always raises an exception, because the zero page is not mapped in memory. Relative branches are mitigated by the memory cage method described in section 4.8.

Case 4 causes an uncontrolled loop since the instruction will always branch back to itself. A method for preventing this is implementing a watchdog thread, which analyzes the state of the executing thread. If the state does not change for a certain amount of time, meaning the executing thread is looping, then the watchdog thread can shut down the executing thread and regain control of the fuzzing process. We did not implement the presented solution, as we did not observe any branch loops in practice in any of the fuzzing tests.

4.7 Register Zeroing

For various reasons as described in sections 4.6 and 4.5, zeroing registers is necessary. Immediately preceding the execution of the instruction under test, all registers are set to 0. It is implemented in inline assembly to assure that no compiler optimizations rearrange the code. However, not all registers are actually zero at the point of execution of the instruction.

To execute the instruction, which lies on a separate special page in memory as described in section 4.8, the memory address where the instruction under test lies needs one register. Theoretically, this could cause unwanted behavior. However, we encountered none such issues in the fuzzing tests. The added benefit of zeroing most registers is that all address related instructions are limited to the address 0x0 in their operation. Since address 0x0 is never mapped, this always raises an exception, and the fuzzing process can recover.

4.8 Memory Cage

Originally described by Dofferhoff [15], the memory cage method offers an elegant way to prevent uncontrolled branches and loss of fuzzing process control. Instructions that use relative addressing may cause memory corruption or a loss of control over the fuzzing process. Figure 4.1 represents the schematic layout of the memory cage method used.

To mitigate this issue, an area of read-, write- and execution-protected pages are mapped into memory. These pages pad the page on which the instruction under test lies. This area is made larger than the maximal relative addressing offset of all relative address instructions. The instruction set determines the maximal relative addressing offset. For AArch64, the branch instruction has the largest possible relative offset address of all instructions with ±128MB [9]. All memory-related, relative-address instructions will inevitably access memory within this area, which raises an exception, which in turn enables the gain of control flow.

The instruction under test may access memory on the same page the address where the instruction lies. To avoid indeterministic behavior, we initialize the memory on that

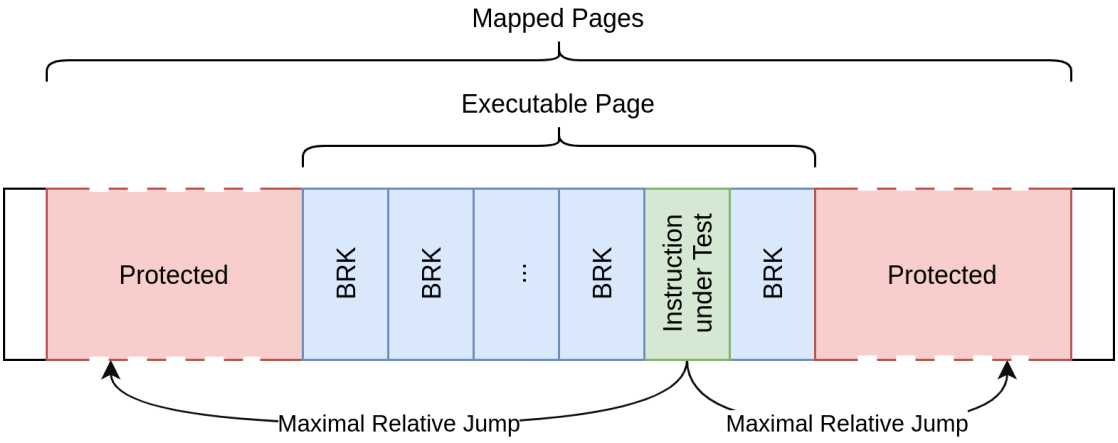


Figure 4.1: Memory Cage adapted from Dofferhoff [15].

page with BRK instructions before executing an instruction. Writes and reads will cause no issues since we reset the page content on every instruction test. Branches within the page will inevitably execute a BRK instruction, and therefore the signal is again trapped in the fuzzing process.

For a page of length n bytes, the 4-byte instruction under test is located on address $n-8$. On location, $n-4$ immediately following the instruction under test is a BRK instruction. This ensures that when we execute the instruction, the BRK instruction generates a signal, and we gain control of execution.

4.9 Signal Handling

This section describes the handling of signals generated by exceptions within the fuzzing process. Executing an unknown instruction causes a loss of control over the fuzzing process since the instruction can behave as a branch, continuing code execution in a different location. To regain control flow, the fuzzing process relies on the exceptions generated by the instruction under test itself or by the padding instructions and pages.

The memory cage 4.8 and register zeroing 4.7 techniques ensure that under all circumstances, the fuzzing process will trap due to a BRK, or raise an exception. In the initialization phase of the fuzzing process, we register a signal handler for all error signals. Signals are handled according to the algorithm detailed in 4.9. This avoids catching signals which were not directly generated by an exception from testing an instruction.

The variables $handler\_has\_run\_before$ and $currently\_executing$ need to be of an atomic type to ensure that all variable state writes and reads are not affected by rogue instruction behavior. Both are global variables due to potential stack corruption described in section 4.5.

Generally logging to stdout or files within a signal handler is not desirable, since both relevant $printf$ and $fprintf$ functions are not $async-signal-safe$ [23]. We can mitigate this by writing log information to variables, and logging them to their respective

handles once back in the fuzzing thread. In practice, this was observed not to be an issue or affect the fuzzing process and, therefore, not implemented.

Algorithm 1 Pseudocode for the signal handler

```
1: function HandleSignal(SignalContextInfo)
2:     if handler_has_run_before then
3:         Log Double Exception
4:         Deregister Signal Handler
5:     else
6:         handler_has_run_before ← true
7:         if not currently_executing then
8:             Log Fault Outside Execution Phase
9:             Deregister Signal Handler
10:        end if
11:    end if
12:    Save SignalContextInfo
13:    return to Fuzzing Control Flow
14: end function
```

4.10 System State Recovery through Signals

Shortly before the program executes the instruction under test, we set the signal reentry point by the *sigsetjmp* macro provided through *setjmp.h*. The underlying function returns 0 when it is called directly, leading to the program's instruction execution branch. After the execution of the instruction, when the exception is raised, and the signal handler handles the resulting signal, as described in section 4.9, code execution continues at the point where the *sigsetjmp* macro has been called the first time, this time returning a non-zero value. The program continues by analyzing the signal code and logs the results.

4.11 Logging

The results of a fuzzing test are saved as log files containing entries for all executing instructions. In the header of the file, the command used to execute the tool is included.

For instructions that generate an illegal instruction exception (*SIGILL* in Unix signals), only every 2^{16}-th entry is logged due to log file size considerations. *SIGILL*s are of little interest since they are the expected behavior for executing illegal instructions.

When the fuzzing process is running, the log file or *stdout* is, therefore still updated regularly. This serves the purpose of identifying potentially hanging fuzzing processes by manually checking the log files. For this reason, some SIGILL generating instructions are still included in the log files, serving as a control for fuzzing process health. These are later removed in log file sanitation.

A more elegant solution with a separate control and fuzzing process was considered but not implemented due to time restrictions, and since the described method was sufficient for the tests performed.

Any instruction which is not recognized by the disassembler but generates any other signal than a *SIGILL* is logged.

The logs use a CSV format; the length of an entry depends on the verbosity parameter. If logging is non-verbose, one entry (line) of the log consists of the instruction as 4 bytes in hexadecimal notation without prefix, separated by a comma. Following the instruction, the resulting signal code is included. The result can be one of: {*NO EXCPT, EXCPT signal, RAN (TRAP)*}, where *signal* is the unix signal code.

Examples of the log are given in figures 4.3 and 4.2.

```
./opcodetester

MSB, byte 2, byte 1, LSB, Result
00,00,00,00,EXCPT 4
00,01,00,00,EXCPT 4
00,02,00,00,EXCPT 4
00,03,00,00,EXCPT 4
00,04,00,00,EXCPT 4
```

Figure 4.2: Example of non-verbose logging output of the fuzzing tool. Intermediate instruction encodings for all values of LSB and byte 1 are skipped in non-verbose mode for reasons explained in section 4.11.

```
./opcodetester -v
using verbose mode

LSB, byte 1, byte 2, MSB, MSB, byte 2, byte 1, LSB, Result
00,00,00,00,00,00,00,00,EXCPT 4
01,00,00,00,00,00,00,01,EXCPT 4
02,00,00,00,00,00,00,02,EXCPT 4
03,00,00,00,00,00,00,03,EXCPT 4
04,00,00,00,00,00,00,04,EXCPT 4
05,00,00,00,00,00,00,05,EXCPT 4
```

Figure 4.3: Example of verbose logging output of the fuzzing tool. Every instruction encoding is logged with both little-endian and big-endian order.

4.11.1 Delayed Logging

Depending on the exception that the execution of the instruction under test generates, some instructions are skipped while others are logged as described in section 4.11. Because the logging is strictly appending, logging the instruction hexadecimal opcode and its resulting exception signal code is only performed **after** the instruction has been tested, and control flow has been regained. This might cause issues if we can not regain control flow or the fuzzing process crashes since the instruction will not be logged in this case. Nonesuch issues were observed in practice. We performed the tests with sequential instruction testing, so even if this error case would have happened, the offending instruction can be easily identified through numerical order.

4.11.2 Log Flushing

To ensure that all tested instructions are correctly logged to the log files in case of crashes or termination of the fuzzing process, it is necessary to flush the file streams. Both *stdout* and the log file are flushed after every instruction test. Therefore the log files are guaranteed to contain all tested instructions and their resulting signals up to the point of process termination.

4.12 Log Analysis

When we conducted the experiments, the log format of the program did not use comma separation, so we used a bash script, utilizing the Linux text parsing and formatting tools *sed* and *awk* to convert the logs to a format specified in section 4.11.

For fast, automated log analysis, we developed a tool for checking instructions through a set of constrained unpredictable behavior rules. The tool's input is a list of instruction encodings, with bytes separated by commas from most significant byte to least significant byte. Every instruction encoding is checked for constrained unpredictable behavior as specified by the Arm Architecture Reference Manual [9]. We implemented all constrained unpredictable behavior rules that were necessary to determine the cause for the tested instructions that did not generate illegal instruction exceptions.

Only constrained unpredictable behavior which relates directly to the instruction encoding can be checked with this method. For address-related constrained unpredictable behavior such as misaligned addresses for specific instructions, a different method would need to be employed, since it depends on the execution context.

In practice, we observed no errors due to this fact.

4.13 Instruction Fuzzing Program Flow

The schematic program flow of the instruction fuzzing algorithm is detailed in figure 4.4. The instruction fuzzing program flow follows the algorithm described below:

1. Lock the executing thread to a specific CPU core as detailed in section 4.4.

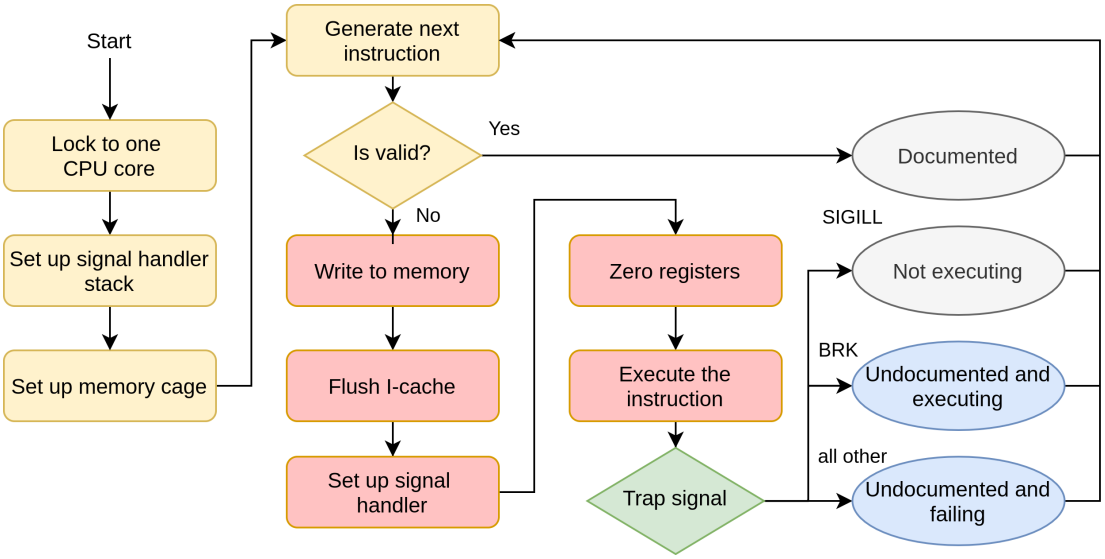


Figure 4.4: Instruction Fuzzing Program Flow

2. Set up the stack of the signal handling thread for reasons explained in section 4.5.
3. Map the pages for the memory cage as described in section 4.8.
4. Generate an instruction as described in section 4.1.
5. Check the generated instruction with a disassembler as detailed in section 4.2.

   If the generated instruction is documented, continue with step 4.

   If the generated instruction is not recognized, continue with step 6.
6. Write the instruction to the executable page within the memory cage described in 4.8.
7. Flush the instruction cache as described in 2.2.1.
8. To return control flow from trapped signals later in the execution, set the signal handler reentry point according to 4.10 for reasons detailed in 4.9.
9. Zero the CPU registers as detailed in section 4.7.
10. Execute the instruction.
11. Trap the resulting signal and analyse it as detailed in 4.9.
12. Hand control flow back to the fuzzing thread, where the instruction and exception information are logged according to 4.11.1.

4.14 Implementation and Features of the Fuzzing Tool

We implemented the fuzzing algorithm detailed in chapter 4 as a tool for fuzzing Armv8-A processors supporting the AArch64 instruction set. It is statically compiled with the necessary libraries and written in C to enable maximal portability over various systems.

Executing the tool will generate a folder *log* in the same directory, where the logs of the fuzzing run will be written to. Log files are date- and timestamped with the starting time of the fuzzing run.

A full fuzzing run can be started by executing the binary without arguments:

```
> ./opcodetester
```

4.14.1 Argument parameters

All parameters except the Display Help can be used in any combination with another.

Display Help

For a full list of all supported arguments, start the binary with:

```
> ./opcodetester -h
```

Initial Instruction

By default the tool starts fuzzing with the instruction opcode 0x00000000. This can be changed with the parameter $-c$ *opcode* where *opcode* is any unsigned 4 byte hexadecimal value with prefix $0x$ notation, e.g. 0x01234567.

```
> ./opcodetester -c opcode
```

Core Locking

By default the tool locks to CPU core 0 for fuzzing. To select a different core, use the parameter $-k$ *core*, where *core* is an integer in $[0, n)$ where n is the number of available cores on the CPU under test.

```
> ./opcodetester -k core
```

Number of Instructions to Test

To control the number of instructions to test, the parameter $-n$ *count* can be used. Here *count* is an integer in the interval $[1, 2^{32})$. In sequential mode (see section Random Instruction Testing in section 4.14.1), the fuzzing process will exit if either *count* instructions have been tested, or the highest possible instruction encoding 0xFFFFFFFF has been reached. In random mode exactly *count* instructions will be tested until exit.

```
> ./opcodetester -n number
```

Random Instruction Testing

By default, instruction testing is sequential, which means that after testing instruction 0x00000000, the next instruction tested will be 0x00000001. Sequential execution continues until instruction 0xFFFFFFFF is tested, or the limit set by $-n$ *count* as described in section 4.14.1 is reached. To change sequential execution to random instructions, use the parameter $-r$.

No other arguments are required for this parameter.

In random instruction testing, the instructions are generated by a pseudo-random number generator from *stdlib.h*.

Instructions may be tested multiple times. At most, 2^{32} instructions will be tested.

```
> ./opcodetester -r
```

Verbose Mode

To increase verbosity of logging, the parameter $-v$ enables more verbose logging. Specifically, valid instructions are included in the logs as described in section 4.11

```
> ./opcodetester -v
```

Execute Valid Instructions

To execute valid instructions, the parameter $-x$ disables the disassembly validity check.

```
> ./opcodetester -x
```

5 Results

5.1 Experimental Setup

We tested three devices with four different core architectures implementing the Armv8-A architecture. The Nvidia Tegra TX2 has a six-core CPU. Two cores belong to the *Denver* architecture developed by Nvidia themselves, and four cores with Cortex A-57 architecture. Nvidia's Denver architecture is especially interesting because they employ dynamic code optimization running transparent to execution.

As an example of a widely used Armv8-A device, we included the Raspberry Pi 3b+ in our test. The SoC has a Cortex A-53 CPU, which is less powerful than the Nvidia Tegra TX2.

The third CPU tested is an AWS Graviton2 Neoverse N1, which is a CPU used by Amazon Web Services in their EC2 M6g, C6g, and R6g instances as described on the AWS page [24].

These three devices represent different use-cases of Armv8-A CPUs such as SoCs, IoT-devices, and cloud servers.

We performed the fuzzing tests on the devices running different distributions of Linux.

Additionally, a random fuzzing trial was performed on QEMU 5.0.0 emulating the Armv8-A architecture.

5.2 Instruction Fuzzing Tests

While fuzzing, none of the systems experienced any stability issues in the fuzzing process, operating system, or hardware.

In Table 5.2 the four different core architectures, the fuzzing runtime, the number of executing instructions and the percentage of instructions which can be attributed to constrained unpredictable behavior are listed.

We wrote a bash script using the Linux tools *sed* and *awk* to analyze the log files. After finishing the tests, we improved the logging functionality of the fuzzing tool, and therefore less complex post-processing of the log files is necessary. Since a significant number of the instructions tested are logged as undocumented and executing, manual analysis of every single instruction was infeasible. In a first approach, basic filtering using *awk* was used to group instructions for manual inspection. However, this was found to be too inflexible. Groupings by hexadecimal digits were possible but not sufficient for further analysis, as the resulting groups spanned across multiple instruction families.

To solve this issue, we designed a program for analyzing instruction opcodes for constrained unpredictable cases, as described in section 4.12. After analysis of the logs with the program, all reported instructions were found to be constrained unpredictable behavior as described in section 2.1.2. All instructions initially reported by the tool as undocumented, either raised a segfault or illegal instruction exception and can be attributed to constrained unpredictable behavior as described by the Arm Architecture Reference Manual in section 2.1.2.

One example of an instruction encoding triggering constrained unpredictable behavior is the instruction **0x80c08020** that, when executed, results in an *SIGSEGV* signal being raised. This instruction is not recognized by the disassembler as the very similar valid LDARB encoding, because its should-be-one bits are not all 1. It is decoded as an LDARB instruction nonetheless and therefore generates a signal other than the illegal instruction exception. This is within the restrictions of the constrained unpredictable behavior.

Another example of an instruction encoding triggering constrained unpredictable behavior is the instruction **0x68d61ce0**. Its parameters evaluate to $imm7 = 0x2c$, $t2 = 0x7$, $n = 0x7$, $t = 0x0$. The Arm Architecture Reference Manual [9] defines that under these parameters, constrained unpredictable behavior applies as seen in figure 5.1

LDPSW

For a description of this instruction and the encoding, see *LDPSW* on page C6-994.

CONSTRAINED UNPREDICTABLE behavior

If the instruction encoding specifies pre-indexed addressing or post-indexed addressing, and `(t == n || t2 == n) && n != 31`, then one of the following behaviors must occur:

- The instruction is UNDEFINED.
- The instruction executes as a `NOP`.
- The instruction performs a load using the specified addressing mode, and the base register is set to an UNKNOWN value. In addition, if an exception occurs during such an instruction, the base register might be corrupted so that the instruction cannot be repeated.
- For execution at EL0 or EL1, when EL2 is implemented and enabled for the current Security state and HCR_EL2.TIDCP is 1, the instruction is trapped to EL2 with EC value `0x0`.

Figure 5.1: Excerpt from the Arm Architecture Reference Manual on Constrained Unpredictable Behavior of the LDPSW instruction [9].

The differing number of instructions reported as constrained unpredictable in the Nvidia Tegra Denver architecture compared with others highlights that the manufacturer of the chip may choose to implement a different behavior within the range of allowed constrained unpredictable behaviors of the same ISA. In this specific case, the other core architectures chose to raise an illegal instruction exception, whereas the Denver raises a segfault upon execution of the invalid instruction.

No instructions that immediately raised a TRAP signal were observed on any devices. Neither were any instructions observed that executed correctly and subsequently executed

the BRK protection instruction, also resulting in a TRAP signal.

The differing implementations of the constrained unpredictable behavior allow for some interesting hypothetical attack scenarios. An attacker may analyze the hardware through probing specific instruction encodings and statistical analysis of the resulting exceptions, matching it with known behavior of a set of tested platforms.

Especially concerning in constrained unpredictable behavior is the inclusion of invalid instructions that may execute as *NOP*. Even though software generally does not intentionally rely on the fact that invalid instructions generate an exception, it may still be possible that the same binary behaves differently in development, testing, and production.

If, for example, in development and testing, the software raises an exception due to invalid input data causing invalid instructions to be executed, it may pass checks if the safe state is to crash. Rolled out on production with a different Armv8-A implementation, the same binary running in the same instruction set on the same software may, however, silently continue execution with the same invalid input data.

Depending on the type of fault and software, this behavior can be exploitable in the form of bypassing data checks, possibly allowing for buffer overflows or similar, leading to full native code execution. Without a deep understanding of the ISA's inner workings, which most high-level software developers do not need under normal conditions, inconsistent behavior and potential for unintentional security flaws can find their way into well planned and realized software projects. This presents a severe problem in the design of the Armv8-A architecture.

Random fuzzing on QEMU Armv8-A emulation

We performed a random fuzzing test on the QEMU Armv8-A emulation using the latest QEMU 5.0.0 release compiled from source. Initially, we planned a full fuzzing trial. Unfortunately, it became quickly apparent that the emulation speed on QEMU would not allow for a full instruction set space fuzzing test within the time limitations of this thesis. Because emulation is so slow on QEMU, we settled on random fuzzing. Unfortunately, QEMU repeatedly and consistently crashed due to incorrect handling of invalid instructions. The crashes were so frequent that we deemed further testing and analysis of the bugs out of the scope of the project.

This particularly inconsistent behavior compared to physical Armv8-A hardware is a big flaw, and allows potential attackers to easily recognize whether code is running on an emulated or physical Armv8-A system. A piece of malicious software, that behaves benignly when tested in a virtual machine, but becomes active when running on physical hardware is now very easy to implement.

Since QEMU can be crashed through invalid instruction encodings executed inside the guest operating system, it is entirely possible that privilege level execution from guest to supervisor can be triggered through this undefined behavior.

Chip	Core	OS	Kernel Version	Runtime	Executing Instr.	% Constr. unpr.
AWS Graviton2	Neoverse N1	Ubuntu 19.04	5.0.0-1022-aws	2h 19m 42s	4 254 880	100
Raspberry Pi 3b+	Cortex A-53	Ubuntu 19.10	5.3.0-1028-raspi2	88h 46m 39s	4 254 880	100
Nvidia Tegra TX2	Denver2	Ubuntu 16.04.6	4.4.38-tegra	5h 21m 6s	4 611 232	100
Nvidia Tegra TX2	Cortex A-57	Ubuntu 16.04.6	4.4.38-tegra	6h 0m 34s	4 254 880	100

Table 5.1: Test device details and fuzzing test results.

6 Future Work

The fact that this project did not uncover any undocumented instructions does not disprove the fact that any exist. A serious limitation to the instruction fuzzing presented here is that no observations on architectural states such as performance counters and register values before and after the execution of instructions, except the resulting exceptions and signals, have been made. This could motivate an improved version of the fuzzing tool, incorporation of performance counters, and register values to detect architectural changes, even if instructions seemingly do not execute according to the exception.

A different approach investigating the exploitability of Armv8-A's constrained unpredictable behavior would be interesting from a security standpoint. This integral design feature of the ISA is flawed by design and presents room for potential exploits.

Another limitation of this work is the test sample size of $n = 4$ core architecture implementations. Testing a wide range of Armv8-A CPUs may result in several undocumented instructions being reported.

QEMU's behavior in the tests has shown that improvements are gravely necessary and raises the question of whether the faulty behavior is limited to QEMU, or if privilege level execution may be possible through invalid instructions. Furthermore, other virtual machines should be tested and compared to QEMU's results to establish if this problem exists on other implementations.

To motivate further work, the instruction fuzzing tool and the tool for checking logs on constrained unpredictable behavior of the reported instructions and the instruction disassembly code will be released as open-source on github [25].

7 Conclusion

Even though no undocumented instructions have been found on the Armv8-A architecture so far, it does not disprove the existence of hidden features in the microarchitecture. Neither does it disprove the existence of bugs or backdoors.

Since Armv8-A is a relatively recent ISA, Arm and hardware manufacturers learned from problematic cases of undocumented instructions found previously on other CPUs, and internally test CPUs not to contain any directly accessible undocumented instructions. This is especially easy with Armv8-A's small instruction space. However a hardware manufacturer may still incorporate different malicious instruction behavior after verification.

A definite limitation to this work is that a combination of (invalid) instructions may still trigger hidden features, back doors, or bugs, but it is much harder if not impossible to test meaningfully due to exponential increase of complexity. Instructions and features hidden in this matter would be missed with our fuzzing approach.

Much more concerning than undocumented instructions, however, is an intentional design component of Armv8-A. Constrained unpredictable behavior poses a serious security issue, since low-level behavior of a system implementing the Armv8-A standard may still behave differently on different CPUs. Software should not rely on specific constrained unpredictable behavior, as stated by the Arm Architecture Reference Manual [9]. However, we doubt that every high-level software developer knows about the inner workings of the Armv8-A ISA and the specific hardware implementations used in production when designing their software projects.

In combination with the flawed implementation of QEMU's Armv8-A emulation, it opens the door for a family of Armv8-A specific malicious programs. We propose that further work is necessary to establish definite results on the Armv8-A instruction set architecture's potential security flaws.

Bibliography

[1] Sharron Ann Danis. *Rear Admiral Grace Murray Hopper*. http://ei.cs.vt.edu/~history/Hopper.Danis.html. Accessed: 17.07.2020.

[2] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. "Spectre attacks: Exploiting speculative execution". In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2019, pp. 1–19.

[3] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. "Meltdown". In: *arXiv preprint arXiv:1801.01207* (2018).

[4] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. "ZombieLoad: Cross-privilege-boundary data sampling". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2019, pp. 753–768.

[5] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. "Rowhammer. js: A remote software-induced fault attack in javascript". In: *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer. 2016, pp. 300–321.

[6] Robert R. Collins. *The Pentium F00F Bug*. http://www.rcollins.org/ddj/May98/F00FBug.html. Accessed: 04.06.2020.

[7] Christopher Domas. "Breaking the x86 ISA". In: *Black Hat* (2017).

[8] Arm Limited. *Arm Limited Roadshow Slides*. https://group.softbank/system/files/pdf/ir/presentations/2019/arm-roadshow-slides_q4fy2019_01_en.pdf. Accessed: 26.08.2020.

[9] Arm. *Architecture Reference Manual*. https://static.docs.arm.com/ddi0487/fb/DDI0487F_b_armv8_arm.pdf?_ga=2.132578958.1898446815.1590764666-27691479.1590764666. Accessed: 04.06.2020.

[10] Arm Developer. *Caches and Self-Modifying Code*. https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/caches-and-self-modifying-code. Accessed: 24.08.2020.

[11] Arm Developer. *AArch64-Exception-and-Interrupt-Handling*. https://developer.arm.com/documentation/100933/0100/AArch64-Exception-and-Interrupt-Handling?lang=en. Accessed: 23.08.2020.

[12] Arm. *Privilege and Exception Levels*. https://developer.arm.com/architectures/learn-the-architecture/exception-model/privilege-and-exception-levels. Accessed: 10.07.2020.

[13] X. Li, Z. Wu, Q. Wei, and H. Wu. "UISFuzz: An Efficient Fuzzing Method for CPU Undocumented Instruction Searching". In: *IEEE Access* 7 (2019), pp. 149224–149236.

[14] Jianping Zhu, Wei Song, Ziyuan Zhu, Jiameng Ying, Boya Li, Bibo Tu, Gang Shi, Rui Hou, and Dan Meng. "CPU security benchmark". In: *Proceedings of the 1st Workshop on Security-Oriented Designs of Computer Architectures and Processors*. 2018, pp. 8–14.

[15] Rens Dofferhoff. "A Performance Evaluation of Platform-Independent Methods to Search for Hidden Instructions on RISC Processors". Aug. 2019.

[16] Michael Göbel. "Developing and Verifying Methods That Search for Hidden Instructions on RISC Processors". Aug. 2019.

[17] Puyan Dadvar and Kevin Skadron. "Potential thermal security risks". In: *Semiconductor Thermal Measurement and Management IEEE Twenty First Annual IEEE Symposium, 2005.* IEEE. 2005, pp. 229–234.

[18] Charlie Miller. "Battery firmware hacking". In: *Black Hat USA* (2011), pp. 3–4.

[19] Philipp Koppe, Benjamin Kollenda, Marc Fyrbiak, Christian Kison, Robert Gawlik, Christof Paar, and Thorsten Holz. "Reverse Engineering x86 Processor Microcode". In: *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 1163–1180. ISBN: 978-1-931971-40-9. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/koppe.

[20] Arm. *Arm Vixl*. https://community.arm.com/developer/ip-products/processors/b/processors-ip-blog/posts/announcing-vixl-a-dynamic-code-generation-toolkit-for-armv8. Accessed: 26.08.2020.

[21] Nvidia. *Jetson TX2 Module*. https://developer.nvidia.com/embedded/jetson-tx2. Accessed: 15.07.2020.

[22] PaX Project. *Adress Space Layout Randomization*. https://pax.grsecurity.net/docs/aslr.txt. Accessed: 23.08.2020.

[23] Michael Kerrisk. *Signal Safety*. https://man7.org/linux/man-pages/man7/signal-safety.7.html. Accessed: 17.07.2020.

[24] Amazon AWS. *AWS Graviton with ARM Neoverse Cores*. https://aws.amazon.com/ec2/graviton/. Accessed: 04.06.2020.

[25] Nikolaus M. Mutewksy. *Github Page for Instruction Fuzzing Tools*. https://github.com/nisky-dev/armv8-instruction-fuzzer.