Bachelor Computer Science

Developing and Verifying Methods to Search for Hidden Instructions on RISC Processors

Michael Göebel

Supervisors:
Dr. E. van der Kouwe & Dr. K. F. D. Rietveld

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
www.liacs.leidenuniv.nl

06/07/2019

Abstract

This thesis describes two methods to scan RISC chips for the presence of undocumented instructions. We developed a memory cage method and a ptrace method. These methods scan the entire instruction search space while maintaining control. The developed program writes an instruction to memory, executes it and performs analysis based on the result. Any undocumented instructions are logged, and after the scan has finished the results are analysed. Through a verification using QEMU, we show that the memory cage method is capable of finding hidden instructions. The memory cage method performs better than the ptrace method. No undocumented instructions were found on the Cavium ThunderX, Ampere eMAG 8180 and AL73400 AWS Graviton. However, we did find some hidden instructions on the QEMU emulator.

Contents

1 Introduction

People often are mistrusting of software, and for good reasons. Software is sometimes used to exploit its users or vulnerabilities in the software can form a security threat. Because of this a lot of research has been done on finding ways to verify that software is safe to use. Rather curiously less attention is given to the hardware that executes these programs, many people simply assume that it works correctly and that there are no security vulnerabilities in hardware. But hardware security has over the past few years become more of a hot topic because of a number of high profile hardware vulnerabilities such as Meltdown [11] and Spectre [8], or more recently the Zombieload MDS attacks [13].

There are also so called Halt and Catch Fire instructions, which upon execution cause the CPU to cease functioning, often leading to a restart being required. A famous example of such an instruction is the F00F instruction [4]. This instruction caused a bug in the Intel Pentium, Pentium MMX, and Pentium OverDrive processors. This illegal instruction would normally cause an exception to be thrown but when used with the lock prefix it causes the CPU to lock up and halt execution.

These kinds of instructions are often not intentionally implemented by the chip manufacturers, more often than not these instructions are the result of negligence or hardware bugs. And because these instructions would not be publicly documented, they would be very hard to find for the general public. Because they would not be documented as legal instructions, assembly programmers and compilers would not intentionally use them when making programs. The only way such an undocumented instruction can normally be found is by someone intentionally executing illegal instructions.

Not much research has been done on developing methods to find these undocumented (hidden) instructions. However, such a method was developed by Christopher Domas for finding hidden instructions on x86 chips [6]. But no such research has been done on RISC processors. That is why the goal of this research is to answer the following research question:
Can a general method be developed to find hidden instructions on RISC processors, what is its quantitative performance, and can its accuracy be verified?

The method should work on multiple RISC ISAs (instruction set architectures). The program will work primarily on ISAs with fixed instruction lengths. The program that utilizes this method to find hidden instructions is written in C++ and will hereafter be referred to as 'the program' or 'the scanner' since it scans instructions. The program will also be used on hardware emulators to check whether an emulator accurately emulates the hardware or not. If the program finds different results between an emulator and the hardware that it is supposed to emulate, it would be possible for any program to know if it is being executed in an emulator or on actual hardware. It would only need to execute a certain instruction and depending on the result it could establish what it is being run on.

1.1 Contributions

This thesis provides the following contributions:

- A method to efficiently find hidden instructions;
- A program that is able to find hidden instructions in ARMV8-A aarch64 and RISC-V CPUs;
- A framework that can be extended to find hidden instructions on other RISC ISAs;
- A summary of the hidden instructions found on various hardware, and on an emulator.

1.2 Thesis overview

The remainder of this thesis is organized as follows. In Chapter 2 some concepts are explained that are necessary to understand the rest of this thesis. Chapter 3 takes a look at some related work. The overview in Chapter 4 gives a general explanation of how the program works. Chapter 5 explains the developed methods for scanning instructions. Chapter 6 contains all information on how the instruction scanners and the rest of the program were implemented. The experiments that were performed can be found in Chapter 7. The limitations of the developed methods are outlined in Chapter 8. The thesis is concluded in Chapter 9, where the conclusions can be found along with some examples of possible future work.

This research is done for the bachelor thesis at LIACS under the supervision of Dr. E. van der Kouwe and Dr. K. F. D. Rietveld. Most of the research and development was done in collaboration with R. Dofferhoff. R. Dofferhoff developed all RISC-V related components, the ptrace method and the normal instruction blacklist version. M. Göebel developed the result analysis methods and the low memory instruction blacklist. He also verified the memory cage method. All of the other components were made together. This includes the memory cage method, the signal handlers and the single instruction analysis.

2 Background

In this chapter some concepts will be explained which are important to understand the rest of the thesis. The precise meaning of an hidden instruction will be explained in Chapter 2.1. Signal handlers are extensively used in the developed methods, information on signal handling can be found in Chapter 2.2. Information on disassemblers and the main disassembler used in this thesis can be found in Chapter 2.3. The QEMU emulator is used to verify the developed methods, the basics of QEMU are explained in Chapter 2.4. Finally, some important features of RISC architectures are explained in Chapter 2.5.

2.1 Hidden Instructions

For this research it is important to precisely define when an instruction is considered to be a hidden instruction. Whenever this research uses the term hidden instruction, it refers to an instruction that is executed by the hardware, but is not documented as a valid instruction in the official ISA manual for the corresponding chip. The ISA manual, often called the architecture reference manual, explains every detail of an architecture. Most notably it contains the instruction set encoding which describes how each instruction is encoded and what its effect should be. For example the LDR instruction in the ARMV8-A reference manual has the entry shown in Figure 1. As can be seen the different fields in the LDR encoding are given a name, their meaning is also explained nearby in the manual. Encodings that do not correspond to a valid instruction are also often listed in architecture reference manuals, these are referred to as unallocated encodings. An example can be seen in Figure 2. Here the instruction '00001100001000000000000000000000' for example falls under the unallocated section in the third row of the table. Instructions like these are not supposed to execute on an ARMV8-A chip. If it executes even though the manual does not specify that it should, then it is a hidden instruction.

LDR (register)

Load Register (register) calculates an address from a base register value and an offset register value, loads a word from memory, and writes it to a register. The offset register value can optionally be shifted and extended. For information about memory accesses, see *Load/Store addressing modes* on page C1-157.

31	30	29	28	27	26	25	24	23	22	21	20	16	15	13	12	11	10	9	5	4	0
1	x	1	1	1	0	0	0	0	1	1	Rm		option		S	1	0	Rn			Rt
size								opc													

Figure 1: The LDR instruction specification from the ARMV8-A reference manual.

Loads and Stores

This section describes the encoding of the Loads and Stores group. The encodings in this section are decoded from *A64 instruction set encoding* on page C4-232.

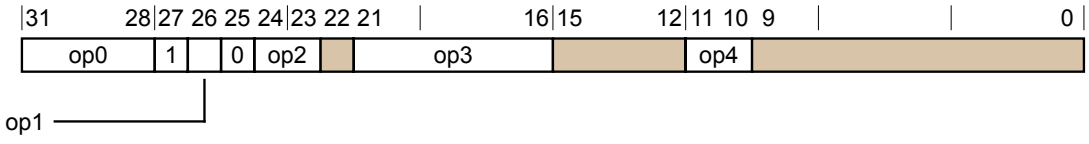

Table C4-5 Encoding table for the Loads and Stores group

Decode fields					Decode group or instruction page
op0	**op1**	**op2**	**op3**	**op4**	
0x00	1	00	000000	-	*Advanced SIMD load/store multiple structures* on page C4-247
0x00	1	01	0xxxxx	-	*Advanced SIMD load/store multiple structures (post-indexed)* on page C4-248
0x00	1	0x	1xxxxx	-	Unallocated.
0x00	1	10	x00000	-	*Advanced SIMD load/store single structure* on page C4-249
0x00	1	11	-	-	*Advanced SIMD load/store single structure (post-indexed)* on page C4-252
0x00	1	x0	x1xxxx	-	Unallocated.
0x00	1	x0	xx1xxx	-	Unallocated.

Figure 2: Part of the encoding table for the Loads and Stores group, taken from the ARM manual [9].

2.2 Signal handling

A signal is a kind of message delivered to a program by the kernel. This signal system is implemented by many operating systems. A signal can be generated by interrupts, exceptions or by an OS feature, but not every interrupt or exception causes a signal to be generated. Interrupts are created from interrupt requests (IRQs) by I/O devices or the CPU and exceptions are created as a result of instructions executed on the CPU. Exceptions that cause a signal to be generated can further be divided into faults and traps. Where the faults are often generated by mistakes in a program and the traps are intentionally generated by a programmer. Traps can for example be used for debugging purposes by generating a breakpoint trap.

When an exception is generated the kernel gains control, which first saves most of the registers on a special stack that only the kernel has access to. Then the kernel looks up what it should do when it receives a certain kind of exception. It could ignore the exception, terminate the program that caused it, or send a signal to the program that caused it. The receiving program has default signal handlers for most signals, but custom made signal handlers can also be supplied. This is

done using the sigaction system call [7]. In this research the SA_SIGINFO flag is supplied in the sigaction structure to make the signal handler receive more useful information. When this flag is supplied the handler must have the following signature:

```
void handler(int signo, siginfo_t* info, void* context);
```

The handler can use the signo and info parameters to determine what happened. The most infamous signo is SIGSEGV, which occurs when an address may not be accessed because it is not mapped or because of invalid permissions. The context parameter contains the context of the thread that generated the exception, in particular the values of the registers can be read and overwritten. If the register values are overwritten in the context parameter then the kernel will set all registers to the values contained in the context upon return from the handler. So for example, it is possible to overwrite the instruction pointer to resume execution at any desired address.

2.3 Disassembler

A disassembler takes as input machine language, the zeroes and ones representing instructions, and translates this into assembly language. A fictional example is shown below:

```
input:
0110101010011010
1101010111100100
0000011010110111

output:
ADD r1, r2, r1
B 4096
INVALID
```

Here the first two input lines are successfully translated while the third line does not represent a valid instruction. In this research the Capstone disassembler [3] is used. This is done by including the C version of the Capstone library. Capstone supports many different architectures including a large number of RISC architectures. The most important Capstone function is cs_disasm_iter, which is supplied with a single instruction. This function returns 1 if it was able to successfully disassemble the instruction and it is a valid instruction. The function returns 0 if the instruction is invalid. Version 4.0.1 of Capstone was used in this research.

2.4 QEMU emulator

QEMU is an emulator that can create and run a virtual machine [2]. It achieves this by performing dynamic binary translation to emulate a machine's processor. This translation is done by the tiny

code generator (TCG) in QEMU. The TCG translates blocks of the target architecture's code to an intermediate representation. Here a block is defined as a list of instructions starting after the previous block (or initially the first instruction) that ends after the next encountered branch, this is also known as a basic block. After the block has been translated into an intermediate notation, a couple of optimization steps might be performed, and finally the code is compiled for the host architecture.

QEMU has multiple modes it may operate in. The important mode for this research is system emulation. With system emulation a binary program for a different ISA is not simply cross-compiled for the host, but a whole computer system is emulated. QEMU can boot many different operating systems as the guest, and it can emulate various RISC instruction sets. Throughout this research the V3.10 release of QEMU was used.

2.5 RISC architectures

RISC (reduced instruction set computer) ISAs have a small and simple set of general instructions while CISC (complex instruction set computer) ISAs have an extensive and complex set of instructions. Most RISC ISAs also have fixed length instructions while the CISC ISAs often have variable length instructions. This makes it easy to find the start of an instruction on most RISC CPUs, the fixed length of instructions can simply be added to the program counter. However, some RISC ISAs have hybrid instruction lengths. The RISC-V ISA for example has instructions of a 16 bit length and 32 bit length. In these cases the length of an instruction can often be determined from a prefix in the instruction encoding. On CISC CPUs instructions and memory accesses do not have to be aligned in contrast to most RISC CPUs. If an unaligned memory access is done on a RISC CPU then the process is aborted or a slower method is used to still perform the memory access.

3 Related Work

This research was inspired by a similar previous research that scanned x86 chips for hidden instructions. This research was performed by C. Domas where he created Sandsifter, the x86 fuzzer [6]. The x86 ISA brings many challenges for the creation of a functional instruction fuzzer. The biggest challenge may very well be the enormous instruction search space that is a result of the variable instruction length in the x86 ISA. The longest possible instruction is 15 bytes, so in the worst case the search space is $2^{120}(1.33 * 10^{36})$ instructions.

This means that a method had to be developed to iterate only over the meaningful instructions in the x86 ISA. To this end Domas proposed a technique called tunneling where the length of each instruction is determined. Then the last byte of the instruction is incremented to form the next instruction. If the increment causes a different exception to be generated or causes the length of the instruction to be changed, the new instruction is incremented from its new end. When all 256 possible values for the last byte have been inspected, the previous byte of the instruction will be incremented from now on. This method quickly skips over less important fields of instructions such as immediate values.

The other big challenge of an instruction fuzzer is to regain control after executing a single instruction. This was solved by setting the x86 trap flag before executing each instruction. The trap flag generates an exception after executing a single instruction. This flag makes it easy to regain control after any instruction, even branches, which are otherwise difficult to deal with. To the best of our knowledge there is no prior research on the scanning of RISC instructions.

There is also the field of processor verification. Processor verification is about verifying that a processor, when given any random input, produces the expected output. Work has been done by A. Adir et al, where research was done on the generation of test programs for functional processor verification [1]. Their work thus verifies that valid instructions do what they are supposed to do, while this thesis focuses on verifying that invalid instructions do what they are supposed to do, which is not executing and on many CPUs generating some kind of illegal instruction fault. There are also other approaches to processor verification such as formal processor verification. This approach uses methods like theorem proving or model checking to verify the correct functioning of a processor. V.A. Patankar et al [12] applied formal processor verification to an ARM processor and found several bugs. The methods used in formal verification are very different from the methods used in this thesis but formal verification could also be employed to find hidden instructions on processors. While formal processor verification is very effective at finding bugs in hardware, it is often used with very specific verification goals or applied to smaller design blocks due to the involved complexity. In contrast, functional processor verification can be applied to bigger designs but often uses semi-random generated test programs which might not uncover all bugs.

Also related is the work done by R. Dofferhoff [5], he worked on the same framework used in this thesis and also applied it to the RISC-V architecture.

4 Overview

A scanner program was made to find hidden instructions. This scanner can be run on RISC systems which use a Linux OS to check if there are any instructions that execute while they should not execute according to the ISA manual. The scanner executes instructions and analyzes whether they successfully executed or not. This is compared to what the ISA manual specifies for that instruction encoding.

The scanner program starts with a manager thread which manages and initializes several worker threads. The worker threads scan instructions in a loop. This loop consists of five phases, see Figure 3.

During the fetch phase the next instruction that needs to be executed is retrieved. When doing this a blacklist containing instructions that should not be executed is consulted. The blacklist is a necessity because certain instructions will always cause a crash that cannot be easily prevented. An example would be the aarch64 MSR instruction moving an invalid value into the EL0 Thread Pointer/ID Register, this causes the program to crash before the register can be restored to a valid value.

In the write phase the fetched instruction is written to a predetermined address in memory. Then the scanner executes that instruction by jumping to that memory address. After execution of the instruction a signal is guaranteed to be delivered to the scanner. This does not need to happen immediately but it should happen as fast as possible and the kind of signal delivered should indicate whether the instruction was executed or not.

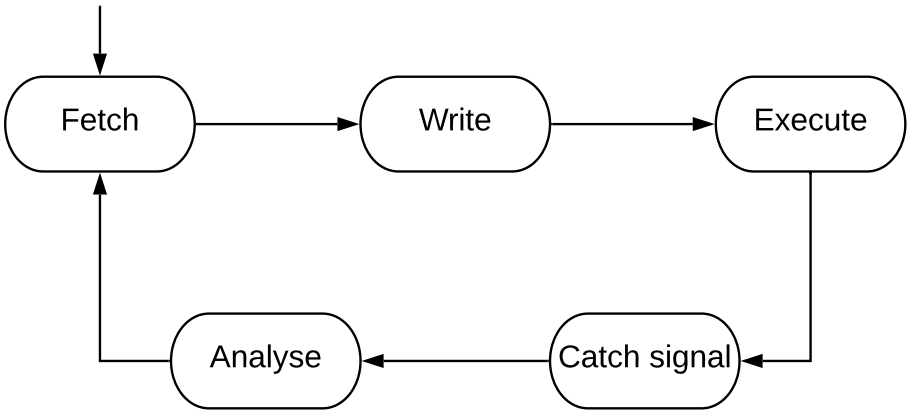

Figure 3: The five phases of the instruction scan loop.

In the next phase the signal gets caught by a signal handler. And in the final phase the generated signal is inspected to determine if the instruction was valid or illegal. This result is compared to a ground truth and a discrepancy indicates that either the ground truth is incorrect or that the instruction is a hidden instruction. If the instruction executed but the ground truth indicates that it should not, then it is classified as a hidden instruction. If the instruction did not execute but the ground truth indicates that it should, then it is classified as a disassembler fault. In both cases the offending instruction is logged to an output file, but this research focuses on the hidden instructions.

After all worker threads are done the main program also terminates. At that point several output files containing performance results and hidden instructions have been generated, which can be used to verify that an instruction that was marked as hidden is indeed undocumented in the ISA manual.

5 Design

In this Chapter the design of the scanner program is explained. The scanner program starts with a main thread which is called the manager. This research has produced the memory cage and ptrace methods, each method has its own manager and the correct manager is selected based on command line arguments. The manager creates scanner/worker threads which analyse the instructions. It also registers the appropriate signal handlers for each signal. The manager evenly divides the total amount of instructions that need to be scanned over the worker threads, all instructions are scanned because the search space is only 2^{32} instructions. The manager then starts the scanners and starts a timer that will invoke a callback every n seconds. In the callback the manager inspects the scanners and performs some logging. When the scanners are started they enter an instruction scan loop, it starts with the first instruction in its assigned range. The memory cage and ptrace methods must guarantee return of control by making sure that after each instruction is executed, a signal will be delivered to the scanner. If a scanned instruction is interesting it will be logged to a file for later analysis. An instruction is interesting if it executes when it should not, or does not execute when it should. A ground truth is needed to determine what each instruction should do. When a scanner has scanned all the instructions in its assigned range, it terminates itself. When all scanners have terminated the manager thread performs some final logging and then the entire program is terminated. Both the setup of the scanners and some aspects of the instruction scan loop depend on some architecture dependent parameters and functions.

The biggest problems that the scanner program faces are found in Chapter 5.1. Two solutions in the form of the memory cage and ptrace methods are explained in Chapter 5.2 and Chapter 5.3 respectively. The method for obtaining a ground truth can be found in Chapter 5.4. Finally, the design for result analysis can be found in Chapter 5.5.

5.1 Program Survival

To not lose control over the execution flow of the scanner and prevent corruption of critical program data it is important to set all general purpose registers (GPRs) to zero. This is necessary because otherwise these registers could contain a wide range of values, and when these registers are used by write or jump instructions, the write or jump could be performed at any memory address. A random jump can cause a lot of random code to be executed, which could harm the program state, or could cause an invalid instruction to be executed, which would result in the wrong kind of signal being delivered to the analysis phase. However, setting all GPRs to zero brings with it a couple of interesting problems. The ABI for some ISAs have the stack pointer, thread pointer and global pointer as GPRs. When these are all set to zero, the program cannot use the normal stack, thread local storage and global variables. This makes it impossible to do almost anything so it was necessary to acquire a stack and recover the thread pointer and global pointer.

The valid stack pointer can be acquired by setting up an alternate stack using the sigaltstack system call and using the SA_ONSTACK flag when setting up the signal handlers. Before entering the signal handler the kernel now uses the alternate stack for storing the signal number, signal info,

and context. The signal handler can now use the stack as usual.

The thread and global pointers are slightly more difficult to reacquire. When using mmap to allocate the alternate stack, more memory is allocated than is used for the stack. Before scanning the first instruction, in the entry handler, the thread pointer and global pointer are stored at a certain offset to the alternate stack pointer, in the extra allocated space. The address d where the pointers are stored is given by $d = i - i \mod (4 * pageSize)$ where $i = stackPointer + 4 * pageSize$. The result of this calculation is insensitive to small variations in $stackPointer$ and thus always returns the same address. The registers are stored by using inline assembly to retrieve the values of these registers, and subsequently writing these values to the calculated address. Whenever these pointers need to be restored after scanning an instruction, they can now be restored by calculating the same address and using inline assembly again to set the thread and global pointers.

To maintain control over the execution flow, return of control must be guaranteed after the execution of an arbitrary instruction. Assuring return of control is not difficult for most valid instructions. A simple method would be placing a branch to a certain routine behind the instruction that is executed. This method works for most data processing instructions and also for some load and store instructions. However, this method would not work for branches since those could jump to random memory addresses. A method needs to be developed that can guarantee return of control after executing any instruction and the method should also provide a way to make a distinction between instructions that did and did not execute. Because invalid instructions cause a signal to be generated it was decided to develop methods that also guarantee that valid instructions cause a signal to be generated. It is then possible to determine whether an instruction has executed or not based on the type of signal. The memory cage and ptrace methods are two solutions to guarantee that a signal is generated shortly after executing any instruction. Both methods are needed to cover a wider range of ISAs and systems that can be scanned. The ptrace method only works on systems where the ptrace system call is implemented with single stepping support and the memory cage method requires alterations for some ISAs.

5.2 Memory Cage

The memory cage method relies on memory protection to guarantee that a signal is generated after any injected instruction. It allocates a big block of memory that has a size of twice the `maxArchOffset` variable from the architecture settings plus the size of a page. In order to ensure that every instruction can be executed without losing control over the execution flow or causing harm to program data, the `maxArchOffset` should correspond to the biggest write or branch offset relative to the program counter. This block of memory has the layout seen in Figure 4. So in the middle lies the page containing the instruction that needs to be scanned. This page contains that particular instruction at the very end of the page, this causes the scanner to enter the block of protected memory as soon as possible, optimizing performance. The rest of the page is filled with the `fillerInstruction` from the architecture settings. This filler instruction should cause a signal to be generated as soon as possible. The filler instructions can be reached if the injected instruction is a backwards branch.

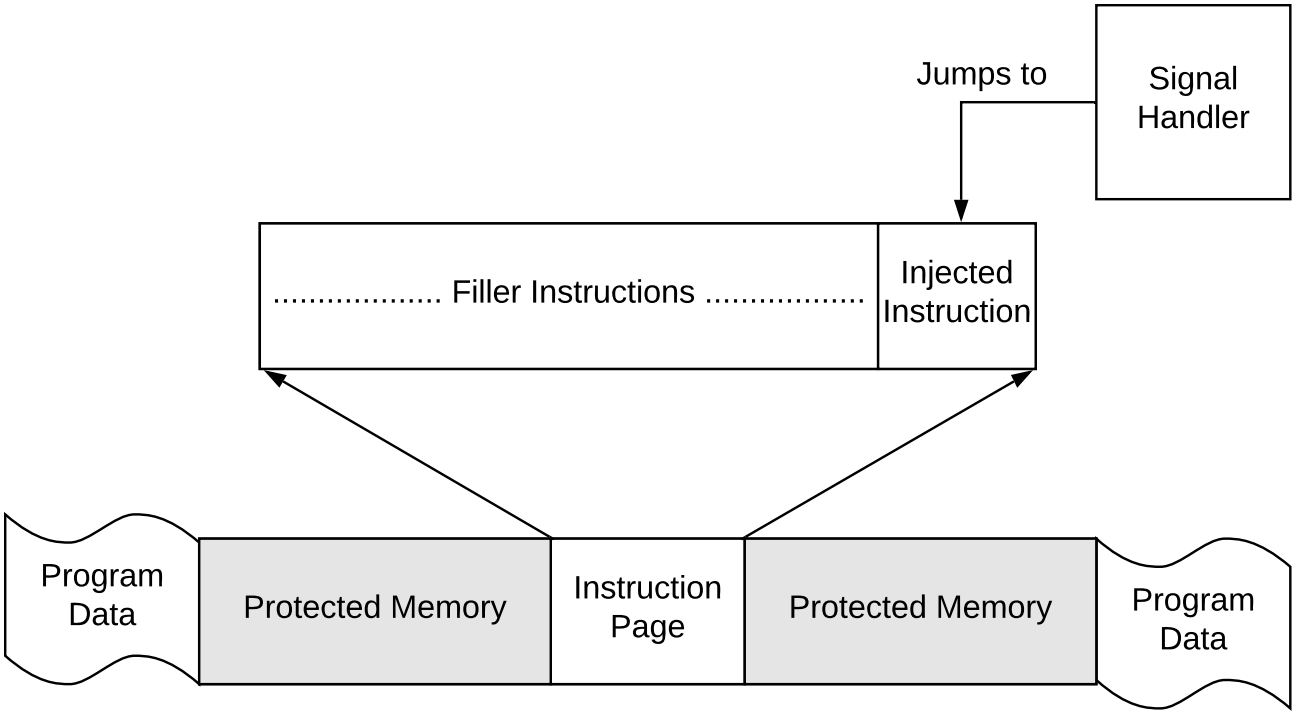

Figure 4: The memory layout of the memory cage scanner.

This page is surrounded by blocks of protected memory, each with a size of `maxArchOffset` and with read, write and execute permissions disabled. These blocks of memory serve multiple purposes. First they guarantee that a signal is generated regardless of the injected instruction. After any valid non-branching non-exception-generating instruction is executed the next instruction is fetched by the CPU, this instruction lies in protected memory, causing a SIGSEGV to be delivered because of the invalid permissions. If a branching instruction is executed it has an offset that is smaller or equal to `maxArchOffset`, as a result it jumps to protected memory or to somewhere within the page containing the injected instruction. This will result in a signal being generated unless the jump causes an endless loop to occur, this issue is handled in Chapter 5.2.1. The only remaining case is a GPR relative jump, due to the measures explained in Chapter 5.1, this jump can only have address zero as its base address. All supported ISAs have only small immediate fields for GPR relative jumps, as a result such a relative jump with zero as its base address can only reach addresses which belong to the zero page. This zero page is guaranteed to not be mapped, again resulting in a SIGSEGV.

The other purpose of the protected memory blocks is to protect important program data that is needed for the continued correct functioning of the scanner. Without the memory blocks a

write could overwrite the program data, but because the writes have an offset smaller or equal to `maxArchOffset` the program data cannot be reached.

In conclusion, if an instruction executes without generating an exception then a SIGSEGV is generated and if an instruction does not execute a SIGILL is generated. This means that the memory cage method guarantees that a signal is generated and that control over the execution flow is regained.

5.2.1 Hang issue

Some instructions cause the scanner to become stuck and do not generate a signal. The simplest example is an instruction that jumps to itself, the protected memory will never be reached and control would never be regained. This issue cannot be resolved by the scanner and thus needs to be handled by the manager. A callback is called every n seconds for the manager. In this callback the manager can inspect each scanner and check if the current instruction being scanned is causing a hang. To this end the amount of instructions executed by the scanner at the the previous time-step is compared to the current amount. If the amount has not increased then the scanner has been busy with the current instruction for at least n seconds, and a hang has thus occurred. The manager then sends a special signal to the scanner in question, causing a hang signal handler to be executed. This handler logs the offending instruction, performs the usual analysis, and continues with the next instruction.

5.3 Ptrace

The ptrace method was developed by R. Dofferhoff and the implementation details can be found in [5]. This method uses the ptrace system call to let a tracer process control a tracee process. The advantage of this method is that it works on any system that supports the ptrace system call and single-stepping.

5.4 Ground Truth

For the ground truth the Capstone disassembler is used, see Chapter 2.3. If the disassembler manages to disassemble an instruction it is indicated that the instruction is valid. The instruction is presumed to be invalid/illegal if the disassembler cannot disassemble the instruction.

5.5 Result Analysis

After the program has finished there are a couple of output files that can be analysed. There is one performance log that lists for each time-step how many instructions each scanner thread has

scanned during that time-step. There is also one result log for each scanner unit, which contains one line for each hidden instruction and disassembler fault. That line specifies what type of instruction it is (hidden or disassembler fault), the value of the instruction, the signal number that was received, and a si_code which gives more information on what caused the signal.

The only truly reliable way of verifying that a logged instruction is indeed a hidden instruction is by manually going through the ISA manual. However, a typical run of the program produces millions of results, way too many to process manually. To reduce the amount of results, similar instructions have to be grouped together or should be filtered out together. This should be possible because the operand fields of hidden instructions typically do not matter. For example, 32768 (2^{15}) hidden instructions would be found if the instruction has 15 bits for operands.

A large portion of the results on the inspected chips turned out to be constrained unpredictable instructions. These kind of instructions are allowed to have several different implementations across chips according to the manual, ranging from not executing to writing unknown values to memory or registers. Instructions are often only constrained unpredictable if the operand fields form specific combinations, an example from the ARMV8-A manual is shown in Figure 5. A separate method had to be developed to filter these from the output due to their complex nature. As can be seen the LDP instruction is constrained unpredictable if $t1$ or $t2$ is equal to n and n is not equal to 31. Constrained unpredictable instructions end up in the result log because the used ground truths assume that they should not execute. This results in a discrepancy between the observed behaviour and the expected behaviour.

LDP

For a description of this instruction and the encoding, see *LDP* on page C6-880.

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

Figure 5: An example constrained unpredictable instruction.

6 Implementation

This Chapter contains the implementation details for several important program features. The scanner program needs to support multiple ISAs, how this is achieved is explained in Chapter 6.1. Chapter 6.2 contains info on how the manager thread initializes the worker threads. Chapter 6.3 explains why the program has a cache incoherency problem and how this problem was resolved. Some instructions should never be executed because of their unwanted side effects, Chapter 6.4 contains the implementation of an instruction blacklist which is used in the fetch phase to guarantee that those instructions will never be executed. The scanner program needs several signal handlers to work, these can be found in Chapter 6.5. Chapter 6.6 explains what we do with the results of a scan.

6.1 Architecture Dependent Settings

The methods developed in this research are intended to work on many different ISAs. Realistically, this cannot be realized without specifying some details about the architecture such as the size of an instruction. Each architecture has its own folder in the `arch` folder, which should contain the `ArchProperties` and `ArchFunctions` header and source files. The variables listed below need to be defined for each different ISA in their `ArchProperties.h` file, their assigned values stem from the aarch64 implementation and serve as an example.

```
static const bool archSingleStep = true;
static const bool archSingleStepPtrace = true;
static const bool ssHang = false;
static const int maxArchOffset = 128 * 1024 * 1024;
static const int maxPcRelativeWriteOffset = 0;
static const size_t instructionSize = 4;
static const uint64_t numInstructions = (uint64_t)1 << (instructionSize * 8);
static const int pageSize = 4096;
static const instr_t fillerInstruction = 0x16000000;
static const int numGPRegs = 31;
static const int regBytes = 8;
static const int registerFileSize = numGPRegs * regBytes;
extern Blacklist blacklist;
```

Their usage will be explained in the rest of this Chapter when necessary. There are also some functions that need to be implemented in the `ArchFunctions.cpp` file. The disassembler initialization and usage for example differ between the ISAs and require custom functions. The function that sets all GPRs to zero also needs to be specified because many ISAs use a differently formatted context struct. Finally, the functions that store and restore the global and thread pointers are also different for each ISA because inline assembly is required to perform these actions.

6.2 Initialization

The manager performs the initialization for the scanners. Each scanner has some data that it needs to perform its duties. This data consists of variables to: communicate with the manager, keep track of the instructions it needs to execute, and make the disassembler work. This data is allocated using mmap with the MAP_SHARED flag because it needs to be shared between the manager process and the scanner process. The manager divides all instructions between the scanners by assigning each a continuous range of instructions, where each range has approximately the same size. The manager also opens three output files for each scanner, one for reporting results, one for reporting hanging instructions and one for debugging purposes. The memory cage manager also sets up a memory region for the scanner as shown in Chapter 5.2 using mmap with the MAP_PRIVATE flag because only the scanner needs access to it. When the initialization is done the manager sends the SIGUSR1 signal to each scanner process using the kill system call. This signal will cause the scanners to start working as explained in Chapter 6.5. As mentioned before a callback needs to be invoked every n seconds so the manager can inspect the scanners and perform logging, this is implemented as a signal handler that gets invoked whenever a SIGALRM is generated. The SIGALRM is generated using the alarm system call.

6.3 Self-Modifying Code and Cache Coherence

The scanner executes self modifying code since it writes the next instruction it wants to scan to a memory location to which it will then jump. This causes a problem on chips with a split data- and instruction cache (Modified Harvard Architecture). The problem is cache incoherence caused by the writing of the instruction which updates only the data cache. When the scanner then jumps to the memory location it might execute the old instruction and not the new one because the instruction cache still contains the old one for that memory address. This problem was resolved by clearing the part of the instruction cache containing the old instruction before jumping to the memory location. The `__builtin___clear_cache` function provided by GCC takes care of this by executing machine specific instructions to clear a specified range in the instruction cache.

6.4 Blacklist

Two variants of the instruction blacklist were implemented. The first put each single instruction that needs to be blacklisted in an STL unordered_set. The unordered set is internally implemented as a hash map, and when its max_load_factor is decreased to 0.5 the resulting hash map is sparse. This implementation has an high memory consumption but allows for a very fast membership test. The other implementation stores instructions together with a don't-care-mask as a pair in a STL set. Each instruction with don't-care-mask represents many single instructions. This implementation has a low memory consumption but the membership test is not as fast as the one of the other implementation.

The low memory consumption version was necessary because some architectures require millions of

instructions to be blacklisted. Perhaps the worst architecture in this aspect is aarch32, where a lot of instructions can modify the program counter in undesirable ways and every instruction has four bits for conditional execution. Consequentially there were 40 instruction encodings to be blacklisted with on average 23 don't care bits. This results in a total of $40 * 2^{23} = 3.35 * 10^8$ instructions that need to be blacklisted. The first method would need $4 * 3.35 * 10^8$ bytes (approximately 1GB) to blacklist all instructions, which was a problem since not all hardware that was scanned had that much memory available. The second method needs only $4 * 2 * 40 = 320$ bytes to blacklist all of the instructions.

6.5 Signal Handlers

The signal handlers that are made need to be registered using the sigaction system call, see Chapter 2.2. Besides the SA_SIGINFO flag the SA_ONSTACK flag is also supplied to sigaction. This flag causes an alternate stack to be used when a signal handler is invoked, see Chapter 5.1 for why this is needed. The alternate stack is allocated by the manager using mmap and made available using the sigaltstack system call. All signal handlers used are explained below.

6.5.1 Alarm Handler

The alarm handler is only used by the manager and is invoked if a SIGALRM gets delivered to the process. Every n^{th} time the handler is invoked it writes to the performance log how many instructions each scanner has scanned since the last time it was logged, the total number of instructions scanned since the last time is also logged. Then the shared memory of each scanner is inspected to check for hanging instructions, if a scanner is currently stuck on a hanging instruction then a SIGUSR2 is send to that scanner. Finally it checks if all scanners have stopped because they are done, if this is the case then the process is exited.

6.5.2 Entry Handler

The entry handler is invoked if a SIGUSR1 gets delivered. The entry handler is different for the ptrace method, this method does not need the handler to do anything, it just needs the signal to be delivered to start working. The memory cage version does do a couple of things. First of it sets the recovery data, see Chapter 5.1. Then it logs the start time to the result output file. The first instruction from the range that needs to be scanned is written to memory, the context of the scanner thread is altered for program survival, and the program counter in the context is set to the memory location containing the instruction that needs to be scanned. Finally it clears the instruction cache and returns.

6.5.3 Fault Handler and Hang Handler

The hang handler is invoked if a SIGUSR2 gets delivered. The fault handler gets invoked for any signal that does not cause one of the other handlers to be invoked. The hang handler is the same as the fault handler, but also writes the current instruction to the output file for hanging instructions and sets the signal number to SIGSEGV because the instruction has been executed. These two handlers are exclusive to the memory cage method. First of the state is recovered, see Chapter 5.1. Secondly, analysis is performed using the signal number and signal info. To this end a disassembler is handed the current instruction. If the signal delivered to the handler is SIGILL, the si_code in the signal info is ILLOPC, and the disassembler indicates that the instruction should execute, then a disassembler fault is logged to the results output file. If any other combination of signal and si_code is delivered and the disassembler indicates that the instruction should not execute, then a hidden instruction is logged.

After this analysis the next instruction is retrieved by incrementing the current instruction value with one. If the next instruction is blacklisted and is not past the final instruction that needs to be scanned, then it is incremented again until a non blacklisted instruction is found. At this point the scanner halts indefinitely if it has scanned all instructions in its assigned range. Otherwise the page containing the instruction is repaired by using memcpy to write the filler instructions again if the ISA allows program counter relative writes. This turned out to be faster than protecting the page against writes using the mprotect system call each time after writing the next instruction to execute. Then the next instruction is written to memory using memcpy. After that the registers in the context are set to zero, and the program counter is set the address where the instruction to scan is located. Finally, part of the instruction cache is cleared again before returning from the handler and scanning the next instruction.

6.6 Result Analysis

After the program has scanned all instructions, the output file contains all instructions that were marked as hidden instructions or disassembler faults. The goal of the result analysis is verifying that the hidden instructions in that output file are not documented in the ISA manual. Multiple methods for result analysis were attempted, most focused around grouping the instructions together to reduce the number of results that would have to be manually sifted through. First was hierarchical clustering and agglomerative clustering, these did not produce usable results because groups would end up as mixtures of several opcodes. The next method relied on the fact that instructions belonging to the same opcode will often appear sequentially in the output. It looks at the next two instructions in the output and calculates the difference in value. If this difference is a power of two then an operand value has most likely increased. The method iterates over the instructions in the output while sequential instructions have the same difference. If a different difference is found then the next instruction most likely belongs to another opcode, and the range that has been iterated over is replaced by a single instruction that represents the entire group. This process is repeated.

This method is accurate but was only able to put a small fraction of the results into groups. Upon

inspection of the remaining instructions it was discovered that these instructions all belonged to constrained unpredictable instructions. This makes sense because the relation between these instructions is quite complicated. A program was developed that can identify all instructions in the output that belong to a certain constrained unpredictable instruction. This program is supplied with the instruction's opcode, field names, and rules that specify when its behaviour is constrained unpredictable. This program was able to filter all constrained unpredictable instructions out of the results, significantly reducing the amount of instructions that have to be inspected.

7 Experiments

This Chapter contains the experiments that were executed using the developed scanner program. The first experiment in Chapter 7.1 verifies the memory cage scanner using QEMU. The results of scanning various systems can be found in Chapter 7.2. A scan of an emulated system is compared to the scanned hardware in Chapter 7.3.

7.1 Verification with QEMU

To verify that the program is always able to find hidden instructions we need to test it on certain instructions which are guaranteed to be hidden instructions. This cannot be guaranteed of any instruction on real hardware. However, a solution presents itself in the form of QEMU, see chapter 2.4. In order to verify our program can find hidden instructions, we can insert these in QEMU on purpose. Before the translation from target binary to intermediate code, it is possible to change the binary value of the instruction to another value. This is done in `qemu/target/$arch/translate.c`. So any instruction can be altered to act like any other instruction, in particular, an illegal instruction can be altered to behave like a valid instruction. A visual example can be seen in Figure 6.

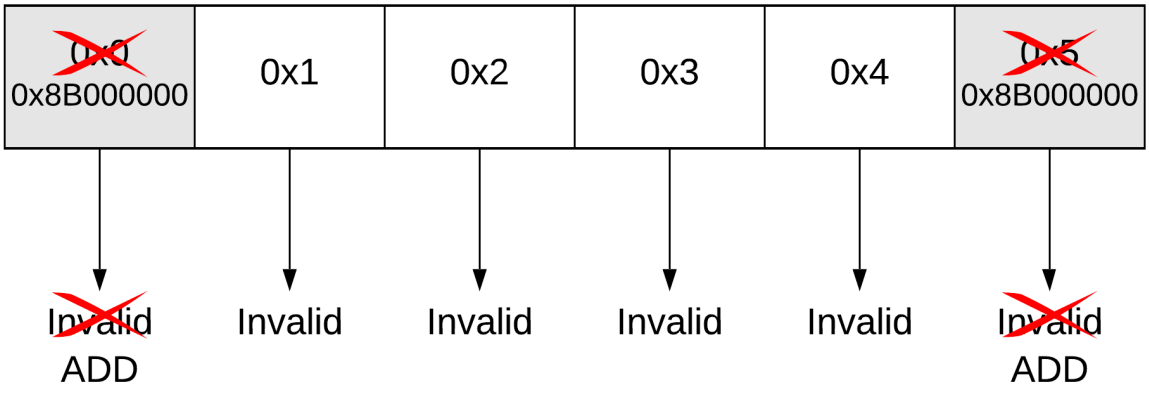

Figure 6: An example of invalid instructions being remapped to an ADD instruction.

Being able to find any hidden instruction is the same as being able to find a hidden instruction independent of the execution path in the signal handlers. So in order to verify the ability to find any hidden instruction it is necessary to test each execution path. On aarch64 this is done by altering the following illegal instructions to behave like an ADD instruction for the reasons mentioned:

- 0 - First scanned instruction, to test the entry handler
- 11&12 - 11 will be altered to behave like an hanging instruction, to test the hang handler
- 5 - Hidden instruction after illegal instruction
- 8&9 - Hidden instruction after hidden instruction
- 3560964065 - Hidden instruction after valid instruction
- 3569352676 - Hidden instruction after disassembly fault.

The last four bullets test all the execution paths in the fault handler. First alter the listed instructions in `qemu/target/arm/translate-a64.c` in the `disas_a64_insn` function by changing `insn` to an ADD instruction (0x8b000000) if `insn` has the value of one of the listed instructions, except 11 which is replaced with a B instruction (0x14000000). Then QEMU is compiled again and the qemu-system-aarch64 program is run. The scanner program is run with both the memory cage and ptrace methods on the ranges 0-16, 3560964060-3560964070, and 3569352660-3569352680 to cover all of the test cases. This is done with the `qemuVerification.sh` script.

All of the listed instructions were found. This means that the scanner can find a hidden instruction independent of its location in the search space. The ptrace method could not be verified using QEMU because it yielded wrong results. When running the program with the ptrace method every scanned instruction except the first yields a SIGSEGV signal with a fault address at the start of the instruction page. Even a breakpoint instruction yields a SIGSEGV instead of a SIGTRAP when executed using the ptrace method on QEMU. This means that only the first instruction to scan is executed, while the others are not executed in the future iterations of the ptrace instruction scan loop. This must be caused by an inaccurate emulation by QEMU since this problem does not present itself when the ptrace method is run on real hardware.

In conclusion, the ability of the memory cage method to find all hidden instructions on a processor was successfully verified by using QEMU. The same could not be verified for the ptrace method because of inaccuracies in QEMU.

7.2 Scanning Hardware

The program was run on several ARMV8-A chips with the aarch64 ISA. The results of a couple of the scanned systems can be found below. In this thesis we focus on cloud instances. Results for single board computers such as the Raspberry Pi 3B+ and Orange Pi PC 2 can be found in [5].

System	Memory cage IPS	Ptrace IPS
Packet c1.large.arm	1,326,964	378,351
Packet c2.large.arm	1,823,334	500,326
Amazon Graviton EC2 A1.4xlarge	533,871	705,460
qemu-system-aarch64	31,077	-

Table 1: Maximum instructions per second scanned with the memory cage and ptrace methods for each system over the first 50 million instructions that were scanned.

All results found were identical for the memory cage and ptrace method. All instances used the Ubuntu 18.04 OS. The optimal amount of cores for each method on each system was determined in [5], this was done by measuring the performance with varying amounts of threads. CPU affinity was set for the performance experiments on all systems except for the memory cage method on the Packet c2, which performed better without setting the affinity. For a summary of the performance results, see Table 1.

7.2.1 Packet c1.large.arm

The first scanned system was the Packet c1.large.arm instance [1]. This is a bare metal instance that runs on two Cavium ThunderX CPUs and has DDR4 ECC memory. The Cavium ThunderX is the only ARMV8.1-A chip that was scanned, the others are all ARMV8.0-A.

Performance This instance had a total of 96 cores available. For the memory cage method 48 threads turned out to be optimal for the program. With these 48 threads a maximum scan rate of 1,326,964 instructions per second was achieved. For the ptrace method only 28 threads were used and the method achieved a maximum scan rate of 378,351 instructions per second. So the memory cage was 3.51x faster than the ptrace method.

Results In total all the scanner units marked 12,600,320 instructions as hidden. This amount being far greater than the amount found on other chips is explained by this chip having an ARMV8.1-A architecture. The disassembler used supported only the ARMV8.0-A architecture and thus marked all new instructions in V8.1 as illegal, producing a lot of false positives as a result. To get rid of these V8.1 instructions a small program was made that filtered the results using a blacklist filled with all V8.1 instructions. After this program was applied there were 5,907,456 instructions remaining. Finally the constrained unpredictable instructions were filtered out using the program described in chapter 6.6, afterwards there were no instructions remaining. Thus there were no hidden instructions found on the Cavium ThunderX CPU.

[1] https://www.packet.com/cloud/servers/c1-large-arm

Decode for this encoding

```
integer d = UInt(Rd);
integer n = UInt(Rn);

integer size = LowestSetBit(imm5);
if size > 3 then UNDEFINED;
```

Figure 7: Decode operation of the INS (element) encoding.

7.2.2 Packet c2.large.arm

Another Packet instance, the c2.large.arm [2], was also scanned. This is also a bare metal instance, and it runs on the Ampere eMAG 8180 chip which has an ARMV8.0-A architecture.

Performance This instance had 32 cores,but only 23 threads were used by the program for the memory cage method, and 10 threads for the ptrace method. This was done because scanning with 23 threads turned out to be approximately twice as fast as with 32 threads, and the 10 threads for ptrace also performed better than with 32 threads. The memory cage method achieved a maximum scan rate of 1,823,334 instructions per second, while the ptrace method had a maximum scan rate of 500,326 instructions per second. So the memory cage method was 3.64x faster than the ptrace method.

Results A total of 6,103,040 instructions were marked as hidden. After filtering out the constrained unpredictable instructions there were 195,584 instructions remaining. All of these instructions fall under the DUP (general) and INS (element) encodings. All of these turned out to be false positives because the disassembler mistakenly thinks that these instructions should not execute. This might have been caused by a misinterpretation of the decode operation for this encoding, which can be seen in Figure 7. The instruction is undefined if the lowest set bit of `imm5` is greater than 3. In other words, the instruction is undefined only if `imm5` is equal to `0b10000`. After filtering out this disassembler mistakes there were no instructions left. So the Ampere eMAG 8180 also has no hidden instructions.

7.2.3 Amazon Graviton EC2 A1.4xlarge

The last scanned instance was the Amazon Graviton EC2 A1.4xlarge [3], which was hosted on a dedicated host. This was also an ARMV8.0-A chip.

[2] https://www.packet.com/cloud/servers/c2-large-arm/
[3] https://aws.amazon.com/ec2/instance-types/a1/

Performance The instance had a total of 16 cores, and the optimal amount of threads for the program was also 16 for both methods. These 16 threads cumulatively achieved a maximum scan rate of 533,871 instructions per second for the memory cage method. While the ptrace method usually has a lower scan rate compared to the memory cage method, it is faster on this instance. The ptrace method scanned up to 705,460 instructions per second. So the ptrace method was 32% faster than the memory cage method.

Results This time 2,957,312 instructions were marked as hidden. After filtering out the constrained unpredictable instructions there were 195,584 instructions remaining. Again, these all fell under the DUP (general) and INS (element) encodings. So in the end this CPU also has no hidden instructions.

7.3 QEMU Compared to Hardware on ARM

7.3.1 Performance

The unmodified version of the qemu-system-aarch64 binary, which provides full system emulation, was executed on a server containing the Intel Xeon E5-2630v3 CPU, and it was executed with 8 threads. The binary emulated the 'virt' machine with a cortex-a53 CPU. Debian 9.8 was used for the guest OS. Only the memory cage method was run on this virtual system because the ptrace method does not work on QEMU. A maximum scan rate of 31,077 instructions per second was achieved. It was thus far slower than any hardware that was scanned. The main reason for this is that QEMU does more than just binary translation, it emulates the entire system. As can be seen in Table 2 QEMU needs a lot of time to emulate the MMU and in particular the TLB. The top five most time consuming functions are all related to emulating the MMU. Most of the other functions in the table are related to the translation buffer where the translated binary code is stored.

Function	Amount of time
tlb_reset_dirty_range_locked	13.48%
tlb_set_page_with_attrs	10.76%
get_phys_addr_lpae	6.61%
tlb_reset_dirty	5.35%
tlb_flush_by_mmuidx_async_work	4.23%
qht_lookup_custom	3.70%
address_space_ldq_le	3.04%
tb_htable_lookup	2.66%
helper_lookup_tb_ptr	2.38%
cpu_get_tb_cpu_state	2.26%

Table 2: Top ten most time consuming functions in QEMU while running the memory cage scanner.

7.3.2 Results

A full run on QEMU was not possible because several instructions caused segmentation faults that could not be handled by the scanner. For example a segmentation fault was caused by the instruction `11010101010110111101000001000000`, even though the manual indicates that this is an unallocated encoding which should have lead to an illegal instruction fault being raised (see page 237 of the ARMV8-A manual [9]). So the scanner crashes while executing a hidden instruction with unknown side effects. Even though a full run was not possible the majority of the search space was successfully explored.

The program marked 78,389,761 instructions as hidden. After removing the constrained unpredictable instructions from the results there were still 72,429,569 hidden instructions remaining. A vast increase compared to the zero hidden instructions on real ARM hardware. Due to time limitations not all 72 million results were inspected. However, a couple ranges of instructions were inspected to verify that they are really undocumented.

The first hidden instruction range in the output was `11111000100xxxxxxxxx11xxxxxxxxxx`, where an `x` indicates a don't care bit. This encoding led to page 266 of the manual [9] and it is a range of unallocated load/store register instructions. `xxx01011011xxxxxxxxx00xxxxxxxxxx` was the last range in the output, which led to page 284 of the manual, where the range is specified as unallocated add/subtract (extended register) instructions. Also notable was the instruction `11010100111111111111111111111111` which caused a hang to occur, the encoding is found on page 237 of the manual, which lists branch, exception and system instructions. This instruction is interesting because it shows that at least one of these hidden instructions has a significant side effect.

In conclusion, the QEMU aarch64 full system emulation binary does not properly emulate an aarch64 chip because 72,429,569 instructions execute on the emulator while these instructions should not execute on a proper aarch64 chip, as is dictated by the ARM manual.

8 Limitations

8.1 Complex Hidden Instructions

A hidden instruction was defined as an instruction that should not execute according to the manual, but does execute on the chip. In this research it is assumed that an instruction that does not execute can never be a hidden instruction. However, it could be the case that an instruction that normally does not execute, does execute when some unknown conditions are met. An example would be an instruction that is normally invalid and raises a SIGILL, but when a specific combination of flags is set in a status register, it does execute. The program developed does not consider these types of hidden instructions.

The program also assumes that receiving a SIGILL when trying to execute an instruction always means that the instruction was not executed. However, a hidden instruction could perform some operation and then throw an exception causing a SIGILL.

8.2 Faulty Ground Truth

If the ground truth that is used is not correct for every single instruction, it is possible that certain hidden instructions will not be found by the scanner. This is the case when the ground truth mistakenly indicates that an instruction should execute, while it actually should not. A ground truth is only completely reliable if it always indicates the same as the architecture reference manual. So the ground truth in this research could be improved by replacing it by a ground truth made by parsing the architecture reference manual. However, a parsable version of the manual is not available for each ISA. Although a perfect ground truth could for example be developed for aarch64 since an official XML version of the manual is available [10].

9 Conclusions and Further Research

This thesis was about developing a general method to find hidden instructions on RISC processors, measuring the quantitative performance of that method, and verifying the results. To this end two methods were developed, the memory cage method and the ptrace method. The memory cage method is a bit less general than the ptrace method because it works on multiple architectures, but it does not work on all architectures without making changes to the design. For instance, the memory cage method does not work on aarch32 because a prefetch abort exception is caused by fetching an instruction in non-executable memory after branching to the instruction to scan. The ptrace method works on any system where the ptrace system call is implemented and single stepping is supported.

Verifying that the program is able to find hidden instructions was done by modifying QEMU. The memory cage method was successfully verified and the ptrace method could not be verified because single-stepping with ptrace was not supported on QEMU.

The quantitative performance of both methods was inspected on multiple systems. The memory cage method often performed better than the ptrace method, achieving a roughly four times higher instruction scan rate on an average system. However, both methods could scan the entire instruction search space within half a day on all scanned systems, but running the program on the QEMU emulator took multiple days. Both methods always flagged the same instructions as hidden. The program found no hidden instructions on any of the hardware that was scanned. The scan of the QEMU emulator did find several hidden instructions.

For further research a perfect ground truth could be constructed from parsable ISA manuals. This would make the result analysis a lot easier because it removes a lot of false positives. Most false positives are constrained unpredictable instructions, the constructed ground truth would be able to indicate that an instruction is constrained unpredictable, removing them from the result log. It would also be interesting to research the effects of constrained unpredictable instructions on ARM CPUs and the effects of the hidden instructions that were found on the QEMU emulator.

References

[1] Allon Adir, Eli Almog, Laurent Fournier, Eitan Marcus, Michal Rimon, Michael Vinov, and Avi Ziv. Genesys-pro: Innovations in test program generation for functional processor verification. *IEEE Design & Test of Computers*, 21(2):84–93, 2004.

[2] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, volume 41, page 46, 2005.

[3] The ultimate disassembly framework. http://www.capstone-engine.org/. Accessed: 2019-05-06.

[4] Robert R. Collins. The pentium F00F bug. http://www.drdobbs.com/embedded-systems/the-pentium-f00f-bug/184410555. Accessed: 2019-06-17.

[5] Rens Dofferhoff. Developing methods to search for hidden instructions on RISC processors and quantifying their performance. *BSc Thesis Leiden University, LIACS*, 2019.

[6] Christopher Domas. Breaking the x86 ISA. *Black Hat, USA*, 2017.

[7] IEEE and The Open Group. The open group base specifications issue 7, 2018 edition. http://pubs.opengroup.org/onlinepubs/9699919799/. Accessed: 2019-06-17.

[8] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *2019 2019 IEEE Symposium on Security and Privacy (SP)*, Los Alamitos, CA, USA, may 2019. IEEE Computer Society.

[9] ARM Limited. ARM architecture reference manual ARMv8, for ARMv8-A architecture profile. https://static.docs.arm.com/ddi0487/db/DDI0487D_b_armv8_arm.pdf. Accessed: 2019-05-21.

[10] ARM Limited. Exploration tools - ARM developer. https://developer.arm.com/architectures/cpu-architecture/a-profile/exploration-tools. Accessed: 2019-05-09.

[11] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. Meltdown: Reading kernel memory from user space. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 973–990, 2018.

[12] Vishnu A Patankar, Alok Jain, and Randal E Bryant. Formal verification of an ARM processor. In *Proceedings Twelfth International Conference on VLSI Design.(Cat. No. PR00013)*, pages 282–287. IEEE, 1999.

[13] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. Zombieload: Cross-privilege-boundary data sampling. *arXiv preprint arXiv:1905.05726*, 2019.

Appendix A Division of labor

Task	Rens Dofferhoff	Michael Göebel
Memcage development	50%	50%
Development of signal handlers	50%	50%
Analysis stage	50%	50%
Fetch stage ARMv8-A	50%	50%
Blacklist hash set	100%	0%
Blacklist low memory	0%	100%
Ptrace development	100%	0%
RISC-V development	100%	0%
Instruction length detection	100%	0%
Memcage verification using QEMU	0%	100%
Result analysis programs	0%	100%
ARM server scan experiments	0%	100%
ARM SBC scan experiments	100%	0%
HiFive Unleashed scan experiment	100%	0%
QEMU ARMV8-A scan experiments	0%	100%
QEMU RISC-V scan experiments	100%	0%
SBC and server multi-core scaling experiments	100%	0%
Profiling on hardware	100%	0%
Profiling QEMU performance	0%	100%
Handler state recovery	0%	100%

Table 3: Division of labor