Universiteit Leiden
The Netherlands

Opleiding Informatica

A Performance Evaluation of Platform-Independent Methods to Search for Hidden Instructions on RISC Processors

Rens Dofferhoff

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

BACHELOR THESIS

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

02/08/2019

Abstract

Undocumented and faulty instructions can pose a security risk, as shown in the past with the infamous f00f-bug. A scanner that searches for undocumented instructions can be a useful tool to help verify the secure operation of processors. This research proposes two methods for finding undocumented instructions on RISC processors. These methods attempt the execution of a single instruction and analyze the resulting signal information to determine if the instruction is seen as valid by the processor. The results are compared to the behavior specified by the ISA the processor implements. The resulting scanner program is used to scan multiple ARMv8 and RISC-V systems. Various flaws were discovered in the used disassemblers and the QEMU emulator. An undocumented instruction was found on a RISC-V chip. Within this research the performance of the resulting scanner including multi-core scaling behavior is analyzed. Both methods allow the scanning of 32-bit instruction spaces in less than a day. Good multi-core scaling is seen on systems with less than 16 cores.

Contents

1 Introduction

Computer systems are of paramount importance to our modern society. Secure operation of these systems depends on both software and hardware. Software is generally mistrusted, both software verification and malware detection are extensively researched topics. Hardware however often seems to get a pass. Many programmers and analysts used to treat processors like trusted black boxes. In recent years we have seen some serious hardware security vulnerabilities like Meltdown [11] and Spectre [10] that affect many modern processors.

This thesis is inspired by previous research done by Christopher Domas [4]. In this research a tool is built and used to scan x86 processors for undocumented and faulty instructions. Using this tool various such undocumented or 'hidden' instructions were found. These hidden instructions can pose serious security risks. For example Domas was able to find a previously unknown halt and catch fire instruction similar to the infamous f00f-bug on a x86 processor [3].

With the rise of mobile devices the usage of RISC processors has grown. RISC processors make up a large majority of smartphone processors and are beginning to make their way into the server market. A possible security vulnerability on these systems brought about by a hidden instruction could therefore have a large impact. A program to search for hidden instructions on RISC processors could provide a useful tool to find possible security vulnerability in these processors.

The goal of this research is to develop methods to scan RISC processors for hidden instructions. The aim is that these methods are generic and as such can be used on a myriad of different RISC instruction sets and processors. This research will describe the effectiveness and quantitative performance of the developed methods.

1.1 Contributions

This research will deliver the following items:

- A description and implementation of two methods to search for hidden instructions on RISC processors
- Scan results of multiple RISC processors based on the ARM AArch64 and RV64GC architectures
- A performance evaluation of the developed implementations of the instruction scanner

1.2 Thesis overview

This section contains the introduction of this thesis; Section 2 contains some background information needed to understand this thesis; Section 3 describes previous research related to this subject; Section 4 gives a high level overview of the developed programs components; Section 5 discusses the overall program design and methods developed in this research in depth; Section 6 gives

information on how the methods and design are implemented and what issues were encountered in their implementation; Section 7 shows the results of system scans and performance analysis; Section 8 describes the limitations on the methods, their implementations and any gathered results; Section 10 makes suggestions for further research.

This is a bachelor thesis, by R. Dofferhoff for the Leiden Institute of Advanced Computer Science, supervised by Dr. E. van der Kouwe and Dr. K.F.D. Rietveld. A lot of the work was done in collaboration with M. Göebel, this includes development of the memcage method, basic analysis stage and basic fetch stage. All ptrace and RISC-V related components were developed by R. Dofferhoff. M. Göebel developed all analysis tools and the low memory blacklist. M. Göebel also verified the memcage method. Appendix A contains a table with a more detailed division of labor. For the thesis of M. Göebel we refer to [7].

2 Background

In this section the necessary background knowledge needed to understand this thesis is discussed. Definitions and explanations of various terms surrounding instruction set architecture and exception handling are given. Our definition of hidden instructions is also provided.

2.1 Instruction set architecture

An instruction set architecture (ISA) is the interface between hardware and software. It defines the various attributes a processor implementing the ISA should have. Examples of such attributes include the instruction set that it should support and the state it must contain, like the size and number of general purpose registers. For software development an ISA is effectively a compiler target. Software that runs on one implementation of the ISA should also run on a different implementation of the same ISA. The instruction set is defined within the ISA. It is a set of operations called instructions and their binary encoding. Examples of instructions are integer operations such as add and subtract or memory operations such as loads and stores. A software program is a collection of encoded instructions that are executed sequentially. Operands of an instruction such as register numbers or immediate values are encoded along with the operation. An example of an instruction encoding is shown in Figure 1. Aside from the few fixed bits that define the operation there are fields within the encoding that specify its various operands. An encoded instruction is often referred to as an 'instruction' as well, context should make clear if the binary encoding or the operation itself is meant.

Instructions use certain addressing modes, these addressing modes define how the instruction operands are retrieved using data in registers and immediate data within the instruction encoding. Important terms within this research are register and program counter relative addressing. Using register relative addressing the address of an operand stored in main memory is determined by the value in a specified base register and a (signed) offset encoded in the instruction encoding. The address a of an operand in main memory, resulting from register relative addressing relative to register r and encoded immediate i, is given by $a = r + i$. Using register relative addressing the entire virtual memory space can be reached. Program counter relative addressing is similar except the base register is always the current program counter instead of a register specified using a register number in the encoding.

This research focuses on RISC processors. RISC ISAs are characterized by a fixed instruction length (sometimes hybrid encoding see Section 5.7). Virtually all RISC ISAs are loadstore architectures, meaning that the only instruction types with access to main memory are load and store instructions. In RISC ISAs the amount of functionality a single instruction encapsulates is often less than CISC

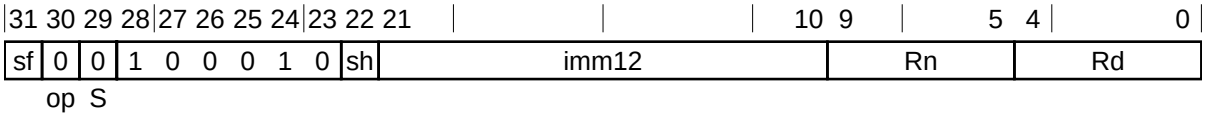

Figure 1: Encoding of the add immediate instruction in the ARM A64 instruction set

ISAs like x86. Examples of RISC ISAs are ARM, RISC-V, MIPS and PowerPC. RISC ISAs have been getting more prominent over the years with the rise of mobile devices.

2.2 Hidden instructions

ISA specification documents should list all instructions that a processor implementing that ISA is able to execute. These specifications explicitly define certain instruction values as illegal instructions, meaning they should not be recognized by the processor. The range of values that represent valid instruction codes almost never cover the entire space of possible values within the instruction encoding length. The undefined values should also not be seen as valid instructions by the processor. We define an instruction as a hidden instruction when the binary value is not explicitly defined to encode for a valid instruction by the ISA specification, but it is seen as a valid instruction by the processor.

2.3 Exceptions & signal handling

An exception is a synchronous interrupt signifying that an exceptional event has taken place within the processor. These events could be errors such as divisions by zero, but they can also be part of normal operation, such as a page fault exception for a memory page that is not yet mapped. When an exception occurs the processor will start executing a matching interrupt service routine set by the kernel. In case of the page fault this routine might resolve the exception by loading the needed page. Some exceptions of an erroneous nature, such as the one signifying a division by zero cannot be resolved by the service routines. In these cases the kernel will pass the exception in the form of a POSIX signal to the process. This starts the signal handler of the process that is set for the specific signal type. These signal handlers may try to resolve the situation that caused the exception and restart from that point. Signals are also used as a form of inter-process communication, processes can send each other signals to communicate. Delivery of signals is mediated by the kernel. Different types of signals are differentiated by their signal number. These numbers are defined as constants such as SIGSEGV for a segmentation fault or SIGILL for an illegal instruction signal.

3 Related work

There has been little previous research on the subject of searching for hidden instructions. In the work of Domas [4] a scanner to search for hidden instructions on x86 chips was successfully developed. x86 is a variable length CISC ISA, instructions can have a length of 1-15 bytes, making the search space to large for an exhaustive search. To shrink the total search space a smart traversal strategy dubbed tunneling is employed. To use this method the length of instructions must be dynamically determined. The scanner is capable of determining instruction lengths without relying on the x86 ISA specification. This research differentiates itself by developing methods and implementations that work on multiple RISC ISAs instead of being focused on a single CISC ISA. The work Domas makes use of the x86 trap flag to single step instructions. The ptrace method discussed in Section 5.5 is similar, also relying on hardware based capabilities to single step through instructions. The ptrace method uses the ptrace system call instead of machine specific assembly to access these capabilities, thus it is able to function on any RISC ISA for which the kernel provides support for the needed ptrace calls. Not all ISAs make demands for such single step capabilities, such as RISC-V. The memcage method discussed in Section 5.6 relies on memory protection instead of hardware based single step capabilities. All modern processors provide the needed memory protection. The memcage method should therefore provides a more general way of scanning for undocumented instructions than provided by Domas and the ptrace method.

In [15] undocumented instruction detection is done as part of an extensive CPU security benchmark. The tool is claimed to work on the ARM and MIPS ISA. Little detail is provided on the operation of the tool and how it is protected from loss of control due to branch instructions or corruption of state due to stores. No results or performance analysis of the tool are provided. It is unclear if this tool supports hybrid length encodings but it does not seem so. The exact traversal strategy is not specified although it is claimed that the tool can cover the entire search space. This research aims to give extensive detail on the developed methods and operation of the scanner. We also provide performance results on the resultant scanner and the results of scanned systems. This research provides two separate methods for scanning instead of one and compares their relative performance. Our scanner supports ISAs with hybrid length encodings by being able to detect the correct length of instructions. This length detection mechanism can also be used to check if the processor recognizes the instructions lengths correctly.

Undocumented or faulty instructions can pose a security risk. This risk can be caused directly by the instruction itself, for example causing system instability like the f00f-bug [3], or the instruction can be part of a CPU backdoor. [6] speculates on the behaviour of a simple CPU backdoor and how such a backdoor may be used in an exploit. The simple backdoor presented can be accessed only when the CPU is in a specific state, when in this state the backdoor is used by executing an undocumented instruction. The existence of such a backdoor was shown by Domas in [5]. The instruction scanner described by Domas in [4] is used to uncover a hidden instruction only visible when a certain model specific register bit is set. The scanner developed in this thesis cannot detect such instructions directly, see Section 8. The hidden instruction acts as the start instruction to a CPU backdoor that can cause privilege escalation. Unless a backdoor has no specific state requirement and can be used by executing a single undocumented instruction, it cannot be directly detected by the scanner program developed in this research.

This research tests processors for flaws after they are released and focuses specifically on the existence of undocumented instructions. In the development stages of processors, manufacturers verify their processor designs. [14] describes a state of the art formal verification technique for ARM processors used in industry, that overcomes the scaling issues of formal verification methods. Formal verification techniques test all sorts of processor behaviour, not just the correctness of the decoder like the scanner in this thesis. To use formal verification, models need to be derived from the ISA specification. Modern processors and the ISA specification to which they should comply are very complex, [13] describes how the ARMv8 ISA is converted to machine readable format. The resulting machine readable materials are used to create Verilog models for formal verification in [14]. This research uses disassemblers as a ground truth for correct processor behaviour, these are often flawed. Usage of the Verilog models used in [14] could provide a better ground truth.

4 Overview

To search for hidden instructions a scanner program is developed. The scanner must check if arbitrary instructions are recognized by the processor as valid instructions. It must compare these results to the ISA specification. Just like in the research of Domas [4] a decision was made to scan from user space within a Linux operating system. This provides the benefits of user space libraries, kernel capabilities and easier portability between architectures. A drawback is that we limit ourselves to ISAs with support in the Linux kernel. Scanning from user space might also limit the scanners ability to resolve instructions requiring a higher privilege level, as discussed in Section 8.

The high level structure of the scanner program is shown in Figure 2. To make use of the multiple cores provided by many modern chips and the inherent parallelism of the task, the scanner program consists of one or more scanner units created and maintained by one manager process. Communication between the manager and scanner units is bidirectional. Each scanner unit writes its findings to its own output files.

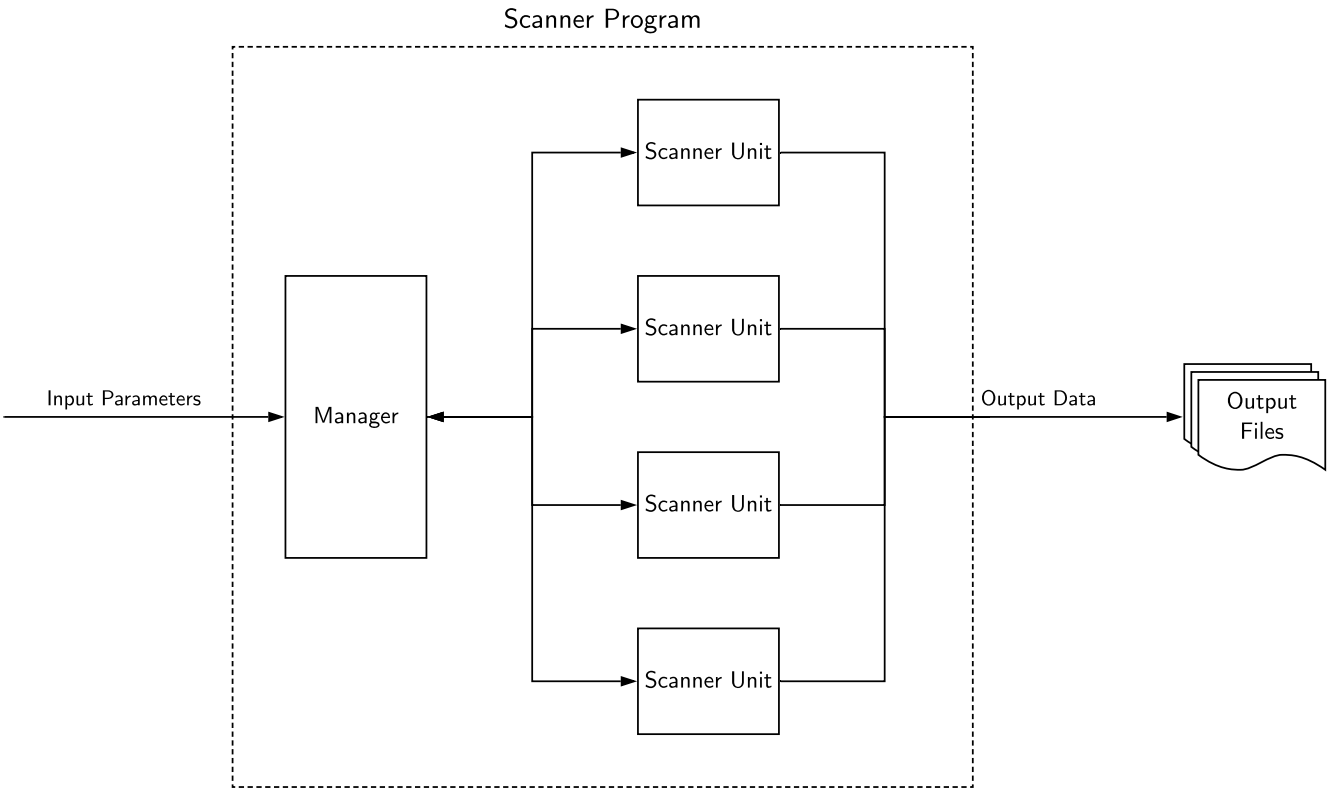

Figure 2: Overview of the scanner program

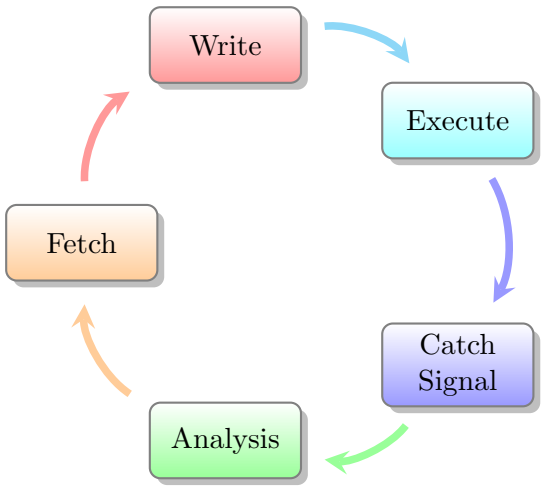

Figure 3: Stages of a scanner unit

Each of the scanner units is assigned a range of instructions to test by the manager. The scanner units operate following the stages shown in Figure 3. These stages are used within two different methods, the ptrace method described in Section 5.5 and the memcage method described in Section 5.6. These methods must satisfy the following requirements:

1. Guaranteed exception for every attempted instruction execution, in order to regain control
2. Valid instructions and invalid instructions should result in different exceptions
3. The program state must be protected

The operation of the scanner units depends on the requirement that every attempted execution of a test instruction results in an exception. The generated exception in turn must result in a signal being delivered to the scanner process. If the instruction that is being tested encodes for a jump or branch instruction its execution could result in jumps to arbitrary code and would result in loss of control. The exception requirement causes a signal to be delivered after every attempted execution. This signal can be used to regain control as shown in Sections 5.5 and 5.6. Using the signal the program can continue to the analysis step.

If the instruction being tested encodes for a load or store instruction, corruption of the program state can result. Care must be taken to protect program state. Discussion of this problem can be found in Section 5.1.

The instruction space can only be scanned if it is sufficiently small. Most RISC instruction sets have fixed instruction lengths of 2 or 4 bytes. These instruction sets fall within the scope of the described methods. Mixed instruction lengths can also be supported, see Section 5.7.

In the analysis stage the signal information is used to determine if the processors recognized the instruction as a valid instruction or not. The analysis stage uses a ground truth that reflects the instruction set architecture specification to check if the instruction should be a valid instruction. A disassembler is used as the ground truth within this research. A disassembler is a piece of software

that translates binary machine code into assembly language. It is able to reverse the encoding defined by the ISA specification.

The fetch stage determines the next instruction to be tested. This research is limited to exhaustive search of the entire instruction space. The code is written in a way such that different analysis and fetch stages can easily be added and used. This allows different types of analysis to be done and results to be gathered. It also allows for different ways to traverse the instruction space. This property is used to support the hybrid encoding of the compressed instruction extension of RISC-V, see Section 5.7.

5 Design

The scanner program is a user space program consisting of a manager process creating and controlling multiple scanner unit processes, as shown in Figure 2. The manager and scanner units communicate through shared memory. The manager creates and initializes the scanner units, divides labour, creates output files, starts the scanning operations and handles hang detection as discussed in Section 5.6.2.

The scanner units execute the stages shown in Figure 3. These steps are followed within two different methods described in Sections 5.5 and 5.6. These two methods guarantee the occurrence of an exception for every instruction that is tested no matter if it is valid or not. The resulting signal is used to regain control after an attempt to execute a test instruction.

5.1 Program state protection

Execution of legal instructions can alter program state. Scanner units must account for this by recovering a good register state after each attempted instruction execution. Scanner units should also guarantee that store instructions do not overwrite parts of memory where program variables are stored. To accomplish this all general purpose registers are set to zero before the execution of a new instruction is attempted. By setting all general purpose registers to zero, store instructions using register relative addressing will end up in the lower parts of the scanners memory space where they will cause a harmless segmentation fault that is later handled.

The offset in register relative addressing is usually rather limited in RISC instruction sets because of limited instruction sizes. For example in RISC-V offsets of stores lay between $[-2048, 2047]$ bytes. Because virtually all RISC instruction set architectures are load-store architectures, store instructions are the only instructions that can overwrite program variables. By setting the general purpose registers to zero before the test instructions are executed the program variables are protected.

5.2 Analysis stage

The analysis stage must determine from signal information if the processor recognized an instruction as a valid instruction. It must compare this result to a ground truth representing the ISA specification to determine if the instruction is hidden. All kinds of different analysis can also be done in this stage, like determining instruction length as shown in Section 5.7.

Attempting to execute an undefined or illegal instruction will result in an exception. This exception will lead to a signal being delivered to the offending scanner unit. The signal number of this signal will equal SIGILL. If the scanner unit receives any other signal this means that the processor recognized the instruction as valid.

The ground truth used in this research is a disassembler. In the analysis stage the instruction that is currently being tested is passed to the disassembler. If the disassembler is unable to decode the

instruction value this should mean that it does not encode for a valid instruction according to the ISA specification.

Discrepancies between results of the signal number analysis and the disassembler will be logged in output files. Discrepancies fall within two cases:

1. Valid according to disassembler & not valid on processor
2. Invalid according to disassembler & valid according to processor

The first case is marked as a disassembler fault in the output by the basic analysis stage. Assuming the scanner program does not contain faults this case can be caused by:

- The disassembler does not properly represent the ISA specification
- The processor is violating the ISA specification by not implementing a mandatory instruction
- The disassembler assumes an instruction set extension not implemented by the processor

The second case is marked as a hidden instruction in the output by the basic analysis stage. Assuming the scanner program does not contain faults this case can be caused by:

- The instruction represents a hidden instruction
- The disassembler does not properly reflect the ISA specification
- The disassembler is unable to decode an instruction set extension that the processor implements

The basic analysis stage used in this research only determines if an instruction is hidden or not, but other types of analysis can be added. The code is designed such that new analysis stages can be easily used instead of the basic one. For example one could try to analyze if the instruction was a load instruction since the analysis stage also has access to register state at the time the exception occurred.

5.2.1 Resolving privileged instructions

Attempting the execution of a privileged instruction in a unprivileged mode will result in a signal with signal number SIGILL, just like undefined and illegal instructions. The scanner program is executed from within user space. Because of this it is executing in a unprivileged mode. The scanner program can still detect the existence of instructions on a higher privilege level. In the analysis stage we can differentiate undefined and privileged instructions by means of the si_code that is delivered along with the signal number of the signal. Undefined instructions will result in si_code ILL_ILLOPC while privileged instruction will result in si_code ILL_PRVOPC.

5.3 Instruction fetch stage

In the instruction fetch stage the next instruction to test is determined. This research limits itself to exhaustive search. In the basic fetch stage the next candidate instruction is the previous instruction value plus 1. The fetch stage also checks if candidate instructions match any instruction that is blacklisted. See Section 6.2 for more information on the necessity and implementation of the blacklist. Just like the analysis stage the fetch stage can easily be changed. One could try to more intelligently traverse the instruction space based on the results of previous instructions or any other kind of heuristic. The RISC-V implementation of this stage also leads to an exhaustive search but uses instruction length information gathered in the analysis stage to skip over unnecessary instructions, see Section 5.7.

5.4 Manager

The manager is a separate process from the scanner units. The manager is responsible for creating and starting all scanner units. It also kills the program when the entire scan is done. The scanner units and manager communicate using shared memory pages. In the current implementation the manager is awoken every second by an alarm signal. It will then continue to check up on all the scanner units. When awoken the manager might log some performance statistics of the scanning process. Within the memcage method described in Section 5.6, the manager is tasked with checking the progress of the scanner units and resolving hangs, see Section 5.6.2.

The manager is responsible for the division of labour among the scanner units. Within this research the manager makes a simple static division of the range of possible instructions among the scanner units before scanning starts. More complex dynamic division of work could be implemented by communication through the shared memory.

5.5 Ptrace method

The scanner units should generate an exception for every instruction that is tested no matter whether it is valid or not. The ptrace method guarantees this exception by using single step functionality provided by the ptrace system call. This system call is often used by debuggers for setting breakpoints and tracing system calls. It allows the binding of two processes in a tracer/tracee pair. The tracer can observe and alter the tracee. By using ptrace calls the tracer may alter and read the memory and register state of the tracee. The tracee will be stopped upon signal delivery and tracer will be notified of this stop. The tracer can then proceed to alter the tracee and handle the signal that was received by it.

Using the ptrace call the tracer can order a stopped tracee to single step the next instruction. The ptrace single step call will cause the tracee to restart and arranges the tracee to throw an exception on entry and exit of every instruction executed. The exception causes the tracee to be stopped. The tracer will be notified that the tracee has stopped and can inspect the signal it received and the state of the tracee at the point of the exception.

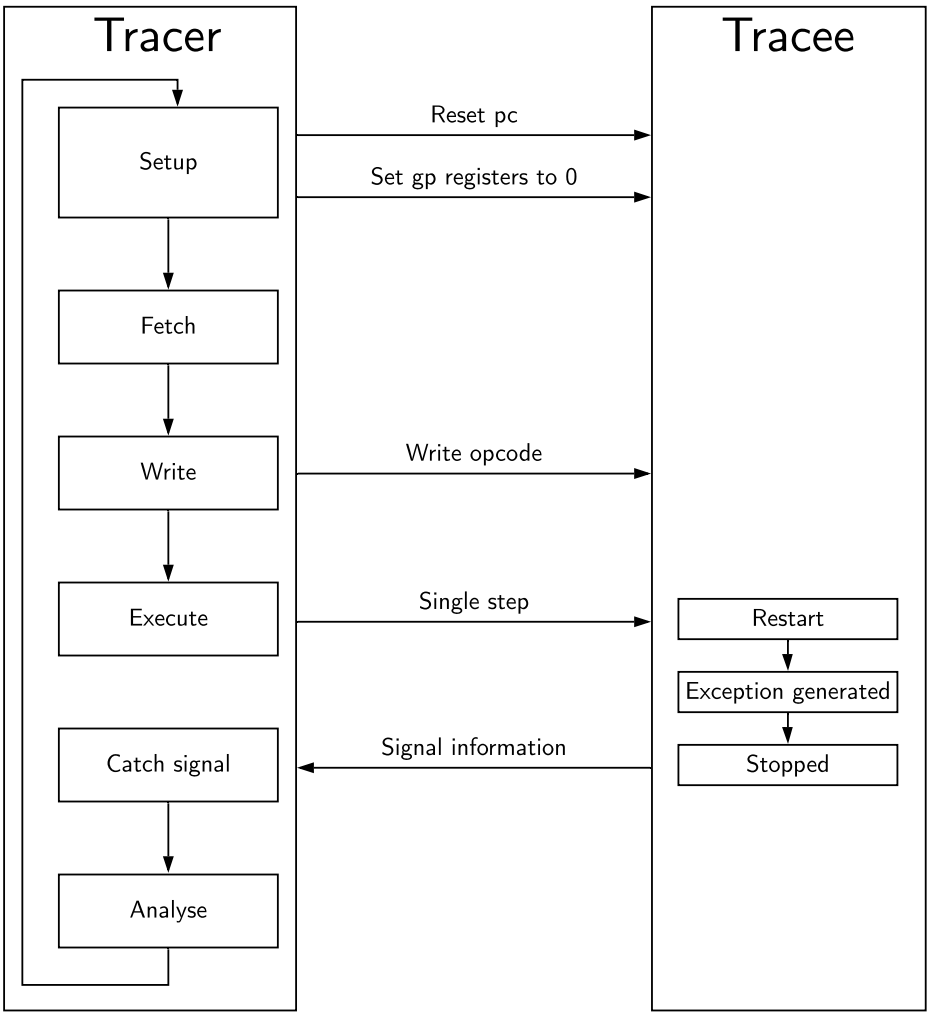

Figure 4: Overview of the ptrace method scanner unit

If the instruction being tested is legal it will execute. If its execution does not cause an exception on its own, a SIGTRAP signal will be delivered after its execution due to single step mode being enabled in the tracee. If the instruction is undefined an illegal instruction signal will be delivered. No matter the instruction type the tracer will receive notice and will start the analysis stage.

Figure 4 shows the design of a scanner unit under the ptrace method. The scanner in this method consists of a tracer/tracee process pair. All the stages shown in Figure 3 are present. The tracee starts off as a stopped process. This allows the tracer to alter the tracee. Execution of the test instructions happens within the tracee process. The tracer sets the program counter of the tracee to the executable memory location where the instructions will be written to. It will set all the tracees general purpose registers to zero using the ptrace system call to protect program state discussed in Section 5.1. The tracer will then start the fetch stage and write the resulting instruction in an executable page within the tracees memory space. It will then restart the tracee using the ptrace single step call and wait for the tracee to be stopped again. The tracer will receive the signal information that caused the tracee to stop and pass it to the analysis stage. It will start the entire process again until it exhausts all instructions it was assigned by the manager.

Each ptrace system call will cause a trap into kernel mode. This trap is pure overhead. To limit this overhead care was taken to limit the amount of ptrace calls. The tracer and tracee are run within the same virtual memory space. Because of this we do not need a ptrace call to write the next instruction for testing. It could also limit any context switch overhead from tracer to tracee depending on kernel implementation.

The ptrace method cannot be used on all RISC ISAs. Ptrace single stepping is dependent on on-chip debug capabilities. Some ISAs make no demands when it comes to these capabilities. This results in some ISAs having no support for ptrace single stepping within the kernel. Some systems do have the capabilities but just lack kernel support. For these systems a custom kernel could be developed. A similar method where single stepping is forced by use of a well defined interface with a custom kernel module was also suggested but dropped due to time constraints. This is left as a suggestion for future work see Section 10. The ptrace method has been shown to work on the ARMv8 ISA.

5.6 Memcage method

To provide a more general alternative to the ptrace method, the memcage method was developed. Instead of relying on hardware based single step capabilities, the memcage method operates using memory protection capabilities provided by all modern processors. The memcage method guarantees an exception for every instruction by placing the instruction on the end of an executable page followed by a page that is not executable, as shown in Figure 5. If the instruction is undefined it will generate the usual SIGILL signal. If it encodes for a legal instruction and its execution does not cause an exception on its own, the instruction following it lays in a memory page that is not executable. Attempting to execute this instruction will result in a segmentation fault. This guarantees an exception for all instructions that are not jumps or branches.

As discussed in Section 5.1 all general purpose registers are set to zero to protect the program state. By setting the general purpose register to zero the possible addresses a branch or jump instruction using register relative addressing could end up are limited to the lowest few pages of the memory space. Only the text section of the memory space contains executable pages. By setting the registers to zero this section of the process memory space cannot be reached by register relative branches and jumps. By writing the instructions far enough away from the text section of the process, program counter relative jumps and branches will also be unable to reach the text section. Just like store instructions their offsets in RISC instruction sets are limited. Although the offsets of unconditional jumps are usually larger than those of store instructions, they do not cover significant portions of the virtual memory space. For example program counter relative unconditional jumps in RISC-V can have a maximum offset of 1 MiB. By careful placement of the test instruction in memory and setting the general purpose register to zero, jump and branches cannot reach the text section. To guarantee correct placement two blocks of non executable memory, which shall be referred to as guards, are allocated before and after the instruction page. The size of the guards is equal the maximum program counter relative offset a jump or branch can have according to the ISA specification. Any program counter relative jump or branch can never reach any memory beyond the guards. The setup is shown in Figure 5.

The memcage method allocates one executable page to write the instruction to. This page will be

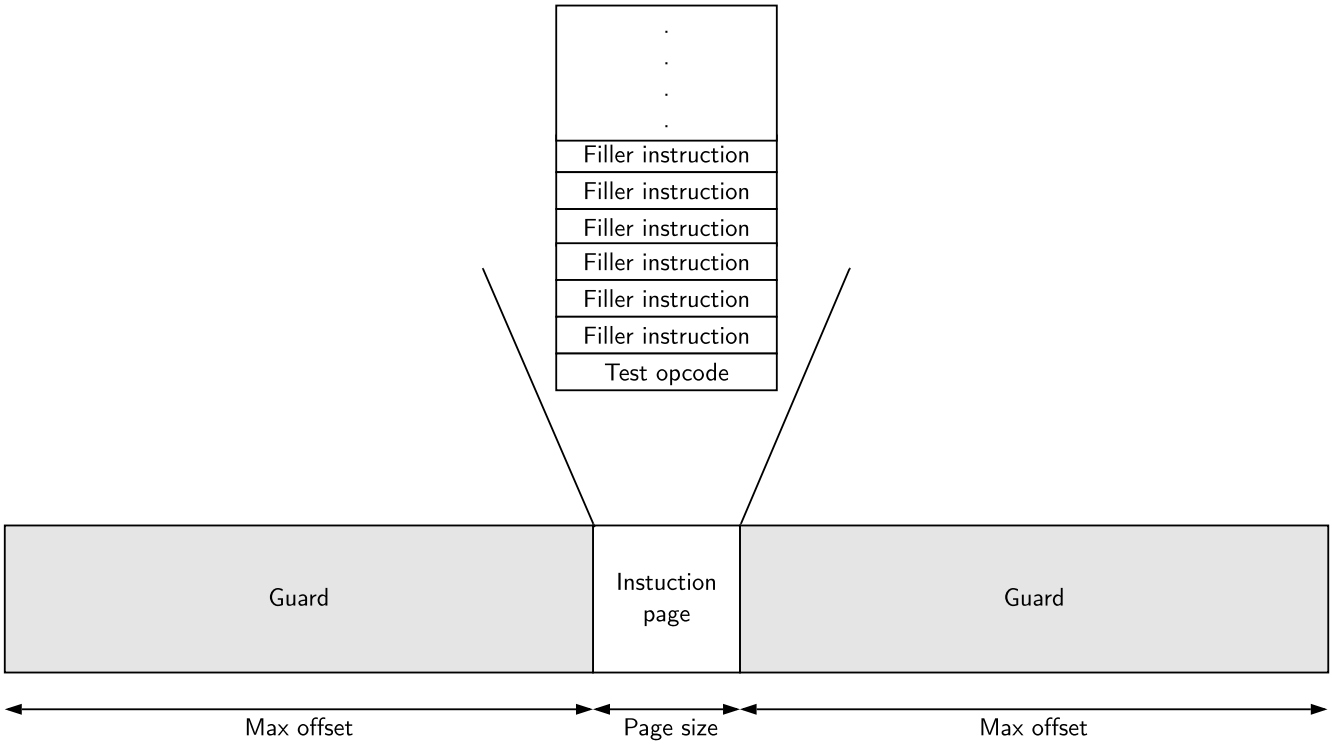

Figure 5: The memory construct to protect program state and guarantee an exception. Grey fields are non executable.

named the instruction page. The existence of the instruction page causes a problem that needs to be solved. Program counter relative jumps and branches could jump to the other locations on the instruction page and execute the values written there, possibly resulting in unintended consequences. For example the instruction being tested is a jump instruction that jumps 3 instructions lengths back. The program counter could now point to an instruction that does not encode for a legal instruction. This instruction does lay in a executable page so the resulting signal value will be SIGILL. Because an illegal instruction signal results from the jump instructions execution, the jump instruction will wrongfully be seen as an illegal instruction by the analysis stage. To resolve this the rest of the executable page is filled with a filler instruction that directly causes any exception other than SIGILL. For example a jump instruction to a page that is not executable or an instruction that directly causes a trap. The instruction page is shown in Figure 5. Now jumps and branches to instructions within the instruction page will result in signals that are different from illegal instructions.

Figure 6 shows the design of the scanner units using the memcage method. In the scanner unit process a signal handler is set for all necessary signals. Each attempted execution of an instruction will cause an exception. This exception ensures that the main handler will be called, returning control so the scanner can start analysis and prepare the next iteration. A machine dependent context structure is passed to the signal handler along with the signal number. Within this context structure resides the register state at the point of the exception. This context will be restored when the signal handlers returns. After the steps shown in Figure 3 are taken the context structure is

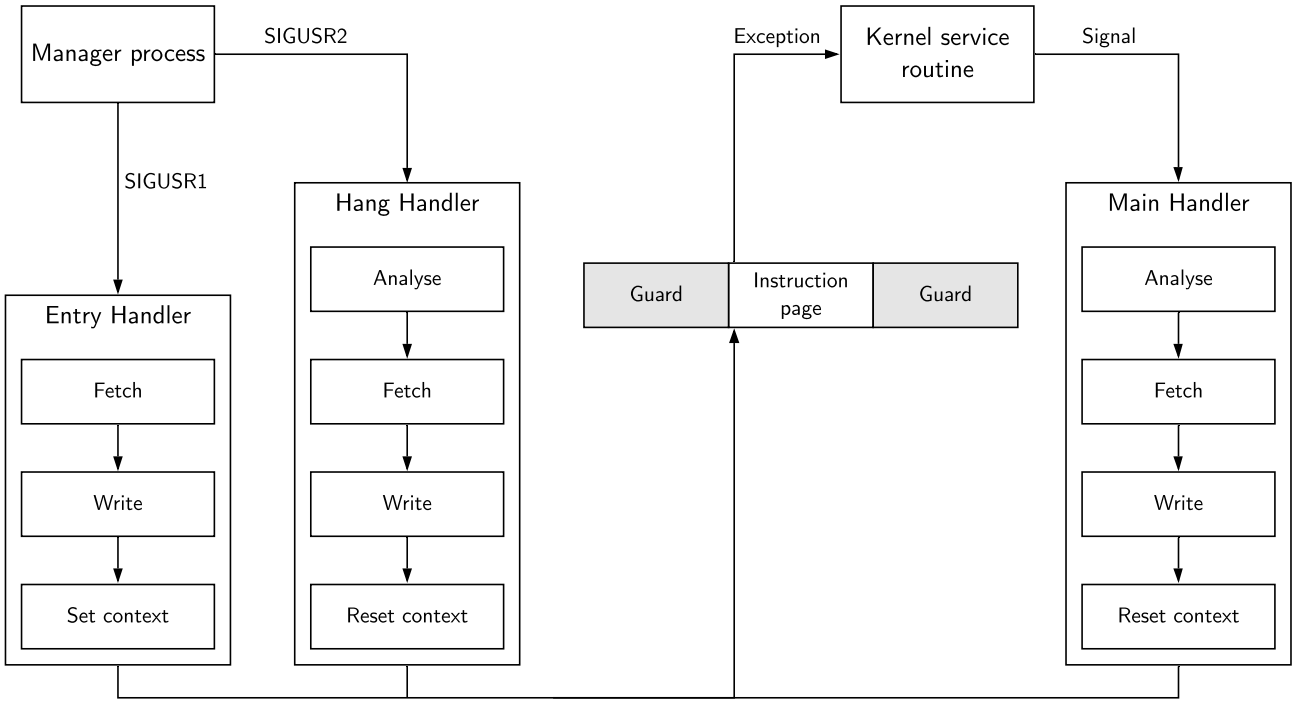

Figure 6: Overview of the memcage scanner unit

altered such that upon return of the signal handler all general purpose registers will be zero and the program counter is reset. Apart from the main signal handler there is also an entry handler that writes and jumps to the first instruction to start the scan loop. The entry handler started by the manager by sending an SIGUSR signal to the scanner process. The hang handler is discussed in Section 5.6.2.

5.6.1 Program state recovery

When starting a signal handler the kernel assumes a good stack pointer. The stack pointer is a general purpose register defined in the ABI for almost all ISAs. If a good stack pointer value is not set before the kernel tries to write the signal handler parameters on stack, it will write them to address zero or any arbitrary value loaded by the tested instruction. This will most likely cause an unwanted segmentation fault and crash the program. To resolve this sigaltstack() is used to set an alternative stack address for signal handling. The kernel will load the address in the stack pointer register on delivery of signals. Using this kernel feature a good stack pointer value is recovered before entry to the handlers.

The ABI associated with the ISA will often designate certain functions to general purpose registers, like it being the stack pointer, pointer to thread local storage or a global pointer. For all these registers good values need to be recovered. These values or references to these values can not be stored in global variables because the global pointer is set to zero or any other arbitrary incorrect value. We let the entry handler store good known values of these register in a memory location in front of the alternative stack address. The alternative stack address is the only known good value

at the start of the signal handler since it is reset by the kernel upon entry. Using the alternative stack pointer in a machine dependent function using some inline assembly, good values for any important registers can be recovered.

5.6.2 Hangs

Some instructions may form self contained infinite loops. Program counter relative jump and branch instructions add a (scaled) offset to the program counter. When this offset is zero it causes this instruction to be executed infinitely. This causes the scanner unit to hang because it cannot progress to the next instruction. In the memcage method the scanner unit manager must resolve such hangs. When the manager is awoken it will check the progress of all units. If it detects that a unit hangs it will send a user defined signal to the unit. Each scanner unit has a special signal handler installed for this specific signal. Within this handler the instruction causing the hang is logged and the analysis stage is started. As usual a new instruction is fetched and written so the scan loop may continue. The hang handler is shown in Figure 6. To make hang detection possible scanner units write progress updates to memory shared with the manager.

5.7 Hybrid length encoding

Some RISC ISAs have instruction sets with a hybrid length encoding meaning that an encoded instruction can have a few different lengths. Examples of such instruction sets are ARM THUMB2 or the RISC-V C extension. In both these instruction sets the instruction encoding can be 16 or 32 bits wide. The first 16 bits fetched by the processor determines the length of the instruction, see Figure 7. To support the RISC-V C extension new analyzers and fetch stages were developed for the memcage method.

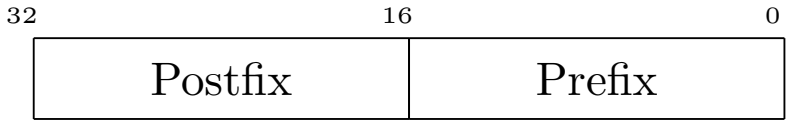

Figure 7: In a little endian system, the lower bits of an instruction encoding are prefixes since they are fetched first. For RISC-V C specifically the bits 0-16 can be seen as prefixes. Some of these prefixes encode for 16-bit instructions while others may encode for 32-bit instructions.

5.7.1 Instruction space traversal

In order to scan these types of instruction sets a new fetch stage needed to be developed. In the fetch stage we traverse the instruction space by iterating over 16-bit prefixes of possibly 32-bit wide instructions. These 16-bit prefixes are first scanned as 16-bit wide instructions. In the analyzer stages it is determined if the instruction was indeed 16 bits wide, as will be described in Section 5.7.2. If the length is determined to be incorrect the prefix is combined with a 16-bit postfix

and scanned again as a 32-bit wide instruction. The fetch stage will combine and scan the prefix with all possible 16-bit postfixes before moving on to the next prefix. Using this traversal method the instruction space is shrunk significantly if the instruction set contains a lot of valid 16-bit instructions. In the RISC-V C extension the instruction space is shrunk to nearly a quarter of the normal 32-bit instruction space.

5.7.2 Length detection

A simpler version of the length detection method described by Domas [4] is used. All prefixes are first executed as instructions with the smallest possible length. Instructions are aligned to the page boundary of the instruction page. If the length is incorrect an attempt to execute the instruction will cause an exception, since part of the instruction lies in memory that is not executable. The resulting register state and signal information can help us identify this situation, see Figure 8. For this method to work properly the neighbouring page must be readable and writable but not executable. The size detection is done in the analysis stage. If an incorrect size is detected this is communicated to the fetch stage.

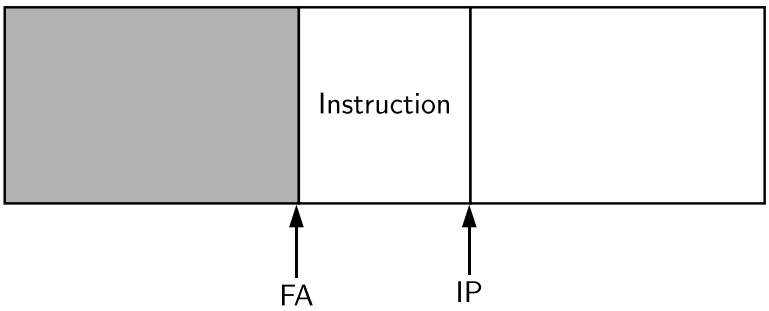

Figure 8: An incorrect size is characterized by the faulting address (FA) returned in the signal info struct pointing to the page boundary, while the instruction pointer (IP) has not moved.

6 Implementation

The program is implemented in C++ and compiled with g++ (built-in functions are used). Both the manager and the scanner units are separate processes. The manager runs in the process that is created at start up. The manager creates the scanner units using fork and has one or more shared pages with each of the scanner units. All information a scanner unit contains is defined in a C++ structure that is stored in the shared memory. This structure contains among other things the current instruction and data to let the manager track progress. All stages shown in Figure 3 can access the data in this structure. Using this structure data can be passed from one stage to the other. A pointer to the structure is stored in each scanner unit as a global variable. The manager keeps a list of all scanner units. The manager itself is implemented as a C++ class. Specific manager implementations for the memcage and ptrace methods inherent from this class. The main function parses user input and creates the appropriate manager class and orders it to start all scanner units. The scan can be limited to a certain range of instructions and the number of scanner units to be created can be set. The method to be used can be selected too.

6.1 ISA specific variables & functions

The arch folder contains a subfolder for every supported ISA. These folders must contain the ArchProperties and ArchFunction files. Within these files the necessary ISA specific functions, properties and type definitions the program needs must be implemented.

The following properties need to be described:

```
//Integer type large enough to hold an instruction
typedef uint32_t instr_t;
//Integer type large enough to hold register values
typedef uint64_t reg_t;

//Does ISA have single step capability?
static const bool archSingleStep = true;
//Does ISA have ptrace single step support?
static const bool archSingleStepPtrace = true;
//Can hangs occur even when using the ptrace method?
static const bool ssHang = false;

//Maximum offset a pc relative jump can have in bytes
static const int maxArchOffset = 128 * 1024 * 1024;
//Maximum offset of a pc relative write. Set 0 if not possible.
static const int maxPcRelativeWriteOffset = 0;

static const int numGPRegs = 31;
static const int regBytes = 8;
```

```
static const int registerFileSize = numGPRegs * regBytes;

//Is the ISA using a hybrid length encoding scheme
static const bool variableLengthEncoding = false;
//Smallest instruction size
static const size_t instructionSize = 4;
//Largest instruction size
static const size_t instructionSizeUC = 4;
static const reg_t MaxNumInstructions = (reg_t)1 << (instructionSizeUC * 8);

static const size_t fillerInstructionSize = instructionSize;
static const instr_t fillerInstruction = 0x16000000;

static const int pageSize = 4096;
extern Blacklist blacklist;
```

The following functions need to be implemented:

```
//Functions to alter context structs. Sets program counter and gp registers
void setState(mcontext_t* context, reg_t pc, reg_t regs[]);
void setStatePtrace(user_regs_struct* context, reg_t pc, reg_t regs[]);

//Functions for Disassembly
void initDisassembler(Injector* injector);
bool disassemble(Scanner* scanner, instr_t instruction, void* info);

//Functions to set and recover necessary registers
//on start of signal handlers in memcage method
void setRecoveryData(Scanner* scanner);
void recoverState();

//Function to flush I-cache
void clearCache(void* begin, void* end);
```

The given examples are for the ARM A64 instruction set. Within the ArchFunction files other functions such as architecture-specific fetch and analysis stages can be defined.

6.2 Blacklist

An ISA might contain instructions that should not be scanned. An example of such an instruction is ARM's BLX. This instruction changes the core's working instruction set. A blacklist was implemented so that such instructions can be skipped by the scanner units. Performance of the blacklist is very important because at least one search operation must be done for every instruction

that is tested. As described in Section 5.3 within the fetch stage a search operation is started for every candidate instruction. Two different implementations were made for the blacklist, the first one gives the best performance but uses a significant amount of memory, the second one is slower but uses much less memory resources. Different implementations of the blacklist can easily be added. The blacklist is defined in the ArchProperties file. Different implementations can be used here as long as they have the same interface.

The blacklist is initialized with a set of (value, don't-care mask)-pairs. A instruction may have multiple immediate fields at any point of the instruction. These fields can be marked by the don't care mask so they are neglected in comparisons. From these (value, don't-care mask)-pairs any blacklist implementation must fill its internal data structure. We will now discuss two implementations of the blacklist.

6.2.1 Blacklist using sparse hash set

A hash set can on average find elements in constant time. From the (value, don't-care mask)-pairs all instructions that equal the value when not considering the don't-care bits are generated. All these instructions are added to the hash set. This takes up a lot of memory resources if there are a lot of don't-care bits in the masks, but makes fast searches possible. The C++ STL unordered set is used as the hash set data structure. We lower the chance of hash collision by setting the maximum load factor to 0.5. This speeds up the search operation but uses even more memory.

6.2.2 Blacklist using list of (value, care mask)-pairs

This implementation uses significantly less memory. It transforms all the (value, don't-care mask)-pairs into (value, care mask)-pairs and stores them in a list. In the search operation the instruction being searched must be compared to every pair. The comparison operator applies the care mask to both the value and instruction and then checks for equality. For a non blacklisted instruction the search operation has a time complexity linear with the amount of blacklisted instructions.

6.3 Disassembler

The basic analysis stage described in Section 5.2 uses a disassembler to represent the ISA specification. For ARMv8 capstone 4.0.1 is used [1]. Some instructions within the ARMv8 specification are marked as 'constrained unpredictable' for certain field values. The results of these instructions can vary between implementations, but are constrained to a list of options such as: undefined, NOP, load unknown values, etc. When trying to disassemble these instructions capstone notes these as being undefined. On actual ISA implementations some of these instructions do execute and this leads to large number of values being marked as hidden instructions. To remove these from the output some filter programs were developed by M. Göebel [7].

For RISC-V capstone is used for all 4-byte wide instructions. For compressed instructions capstone was found not to be sufficient. For this reason all instructions of length 2 are disassembled using a

different disassembler, named the riscv-disassembler by M.J. Clark [2]. This disassembler seems to better reflect the ISA specification for the compressed instruction extension.

6.4 Self modifying code & Cache coherency

A non deterministic bug was encountered while developing the scanner on an ARM AArch64 system. The results of the program would vary between runs. The non determinism in the result was caused by a cache coherence problem. Unlike the x86 ISA most RISC ISAs make no guarantees about coherence between instruction and data caches. This ISA property must be taken in to account when writing self-modifying code, like the scanner program in this research. When a new instruction is written to the appropriate memory location its value is stored in the data cache. Stores do not invalidate the instruction cache entry so an old instruction value could still be present in the I-cache. When attempting to execute the newly written instruction the old value is fetched from the I-cache. If this old value is seen as an hidden instruction the new instruction might be wrongfully marked as a hidden instruction too. The situation is made visible in Figure 9. To solve the issue the new instruction value must be written back to memory from the data cache. Then the I-cache entry needs to be invalidated so the new value will be fetched from memory. We use __builtin___clear_cache() after writing new instructions to accomplish these actions. This compiler built-in function emits some platform specific instructions to ensure coherency for a range of memory addresses, if I-cache coherence is already guaranteed it does nothing. When the __builtin___clear_cache() function is not supported inline assembly can be used to ensure coherence instead.

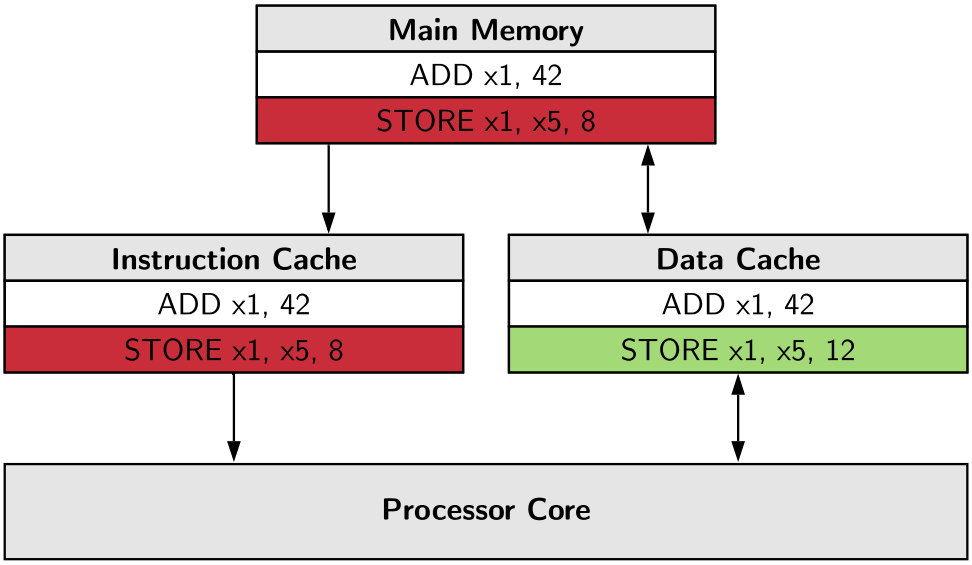

Figure 9: Visualization of incoherence between the data and the instruction cache. All new values are marked green. The old values are marked red. Caches can be made coherent by triggering a write back to memory and invalidation of the instruction cache entries, so the new value will be read from memory.

6.5 Self-attach ptrace

In earlier versions of the program the scanner units and manager were implemented as threads in a single process using the pthread API. This resulted in many problems and no tangible performance increase since process creation is trivial on total run time. The biggest problem was caused by the ptrace method and eventually lead to the decision to go with separate processes instead. The issue was caused by the tracer and tracee being threads within the same thread group. This makes it impossible to attach the tracer to the tracee using ptrace. This behaviour was allowed in the past and its removal is not well documented. There is a LKML thread describing the issue [8].

7 Evaluation

Within this section the ability to find hidden instructions on RISC-V systems using the memcage method is verified. The length detection method described in 5.7.2 is also verified for RISC-V. This section also contains a performance evaluation of the developed implementation of the instruction scanner. This evaluation includes the multi-core scaling behaviour. The scan results of multiple systems are discussed in Section 7.4. First descriptions of the test systems will be given.

7.1 Test system descriptions

Table 1 gives information on all systems used in tests. Orange PI PC2 shall be shortened to OPC2 and Raspberry Pi 3B+ to RPI3B+. Important to note is that the server systems c1.large.arm, c2.large.arm are bare metal servers and not shared systems. The AWS a1.4xlarge instance is run on dedicated host to ensure consistent performance.

7.2 RISC-V Verification

In this thesis, we will describe the verification of the memcage method on RISC-V systems. Verification of the methods and their implementations on ARMv8 AArch64 are described by the thesis of M. Göebel [7]. For the verification of the memcage method on RISC-V the same method is used.

Full Name	ISA	OS	Kernel	CPU (cores)
Orange PI PC2	ARMv8	Armbian (debian stretch)	4.19.38 -sunxi64	Allwinner H5 (4)
Raspberry Pi 3B+	ARMv8	Debian buster	4.19.0-5 -arm64	Broadcom BCM2837B0 (4)
Packet c1.large.arm	ARMv8	Ubuntu 18.04	4.15.0-36 -generic	2 x Cavium ThunderX (48)
Packet c2.large.arm	ARMv8	Ubuntu 18.04	4.15.0-46 -generic	Ampere eMAG 8180 (32)
AWS a1.4xlarge	ARMv8	Ubuntu 18.04	4.15.0-1028-aws	AL73400 AWS Graviton (16)
QEMU 4.0.0 virtual	RV64 IMAFDCSU	Fedora-Minimal -Rawhide-20190326	5.1.0-0.rc1. git0.1.1 .riscv64. fc31.riscv64	(4)
HiFive Unleashed	RV64 IMAFDC		4.15.0 -00044 -g2b0aa1	SiFive Freedom U540 (4)

Table 1: Test system summaries.

Machine emulators, such as QEMU, allow one system to emulate the behaviour of another. Using such an emulator a system could run software and operating systems built for a different ISA. Among other tasks it translates the instructions of the target machine that is being emulated, to instructions that are defined within the hosting systems ISA. The translation code of QEMU is altered to verify of the memcage method. A normally undefined instruction is mapped to a legal instruction. The scanner should see this instruction as a hidden instruction. The undefined instruction value `0b00000000000000000000000000001011` was mapped to an add instruction. The scanner program was able to find this artificial hidden instruction.

7.2.1 Instruction Length detection test

While emulating on the RISC-V virtual machine of QEMU 4.0.0, the instruction pointer in the register state returned along with a signal is incorrect. The instruction pointer does not seem to increase when executing valid instructions in the compressed instruction set extension. Emulation on QEMU also seems to wrongfully generate a page fault exception when a valid compressed instruction is aligned to the page boundary followed by a non executable page. Presumably the virtual machine always fetched 32 bits of which the latter half lies on the non executable page. This makes it impossible to test the method described in Section 5.7.2 on QEMU.

Using the instruction length detection method described in Section 5.7.2, the instruction length associated with all possible 16-bit instruction prefixes were detected on the HiFive Unleashed system. The results perfectly matched the RISC-V ISA specification. The traversal method described in Section 5.7.1 can only operate correctly if instruction lengths are correctly detected.

7.3 Performance

In this section we will investigate the performance of the implementations of the scan methods. This evaluation includes multi-core scaling behaviour. The performance of the memcage and ptrace methods on the various test systems described in Table 1 are compared. A Performance profile is made to find future areas of optimization.

7.3.1 Multicore Scaling

The task of the scanner program seems inherently parallel. There is no reason why two scanner units cannot concurrently scan instructions on different physical cores. To test if the performance of our implementations show good performance scaling an experiment was conducted on all systems described in Table 1. The experiment is conducted for both a scanner employing the memcage method and one using the ptrace method when available.

The scanner is run multiple times with different numbers of scanner units. The scanner program is compiled into a static binary. This binary was used to scan the first 50 million instructions in the A64 instruction space on the ARM systems. On RISC-V systems 4000 instruction prefixes were scanned, resulting in approximately 65.5 million scanned instruction. The scanned instruction

ranges contain no instructions that cause a hang. The normal blacklist described in Section 6.2.1 is used. The number of scanned instructions is logged by the manager every ten seconds. These performance results are processed into an average number of scanned instructions per second, or IPS. One scanner unit may end earlier than another, to ensure consistency any performance log that does not contain results for all units is discarded. The results are plotted together with an extrapolation of the performance of a single scanner unit to represent perfect performance scaling. The standard deviation of each average is shown with a vertical error bar.

Memcage In Figure 10 the scaling seems near perfect when using only a few cores but there is a slight drop of in performance gain when nearing the number off physical cores. At 16 scanner units the IPS is 93% that of perfect scaling. This drop off suggests that the scanner units do not maintain fully concurrent operation. There is no performance gain when increasing the number of scanner units beyond the number of physical cores. The same behaviour was seen on both the RPI3B+, OPC2 and quad-core systems, also reaching 93% of perfect scaling performance when using the same number of scanner units as available physical cores. The RISC-V HiFive Unleashed system showed similar behaviour but with better overall scaling, reaching 97% of perfect scaling.

The behaviour shown in Figure 10 does not translate to higher core count systems, as shown by Figure 11. In this figure near perfect scaling is shown for lower scanner unit counts, followed by a sharp regression in performance. This regression happens before reaching the number of physical cores. Similar behaviour is seen in Figure 12 although the regression is less sharp and near perfect scaling is only seen at very low scanner unit counts relative to the number of physical cores. There is also an increase in variance between measurements.

The scanner program performance is bound by how fast signals are delivered and processed, because each delivered signal represents a scan of an instruction. One would expect that if there is a sequential part of execution within the kernel interrupt service routine or signal delivery mechanism that the system would have an upper limit on the amount of signals that can be processed by the scanner units. This limit would be dependent on single thread performance. But this does not explain the sharp regression seen in Figure 11. Analysis reveals that the sharp regression is caused by a large increase in the number of cycles spent in _raw_spin_unlock_irqrestore called by the force_sig_info kernel function. Since both methods are dependent on signal delivery the regression cannot be mitigated by changes to their implementation.

Ptrace method The multi-core scaling behaviour of the ptrace method is much more erratic than that of the memcage method. On the lower core count systems there is a big spike in performance when the number of scanner units is equal to the amount of physical cores. At this number of scanner units performance even exceeds projected perfect scaling performance of a single unit. This behaviour is shown in Figure 13. Scaling behaviour is better when processor affinity is set for each scanner unit. When the affinity is set for each scanner unit the scaling behaviour becomes similar to that of the memcage method, as shown in Figure 14. Performance of the ptrace method is sensitive to scheduling of the individual scanner units. The effects of setting processor affinity are described in Section 7.3.2.

Similar to the memcage method higher core count systems do not show good scaling for the

ptrace method, as shown in Figure 15. Scaling is observed at lower scanner unit counts followed by a performance regression at higher numbers of scanner units, both the c2.large.arm and the c1.large.arm systems showed this behaviour.

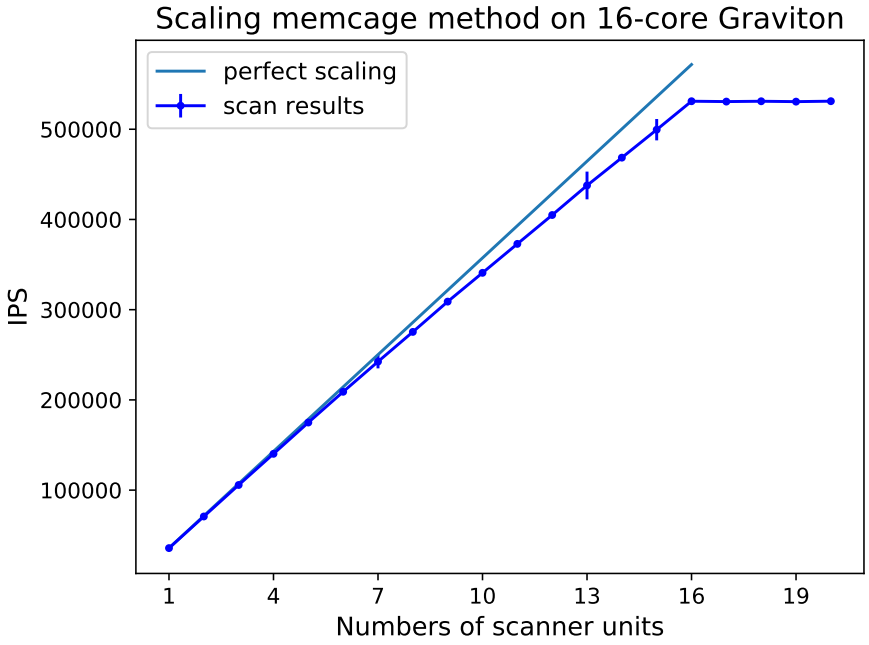

Figure 10: Performance scaling behaviour of memcage method on dedicated AWS a1.4xlarge 16 core system

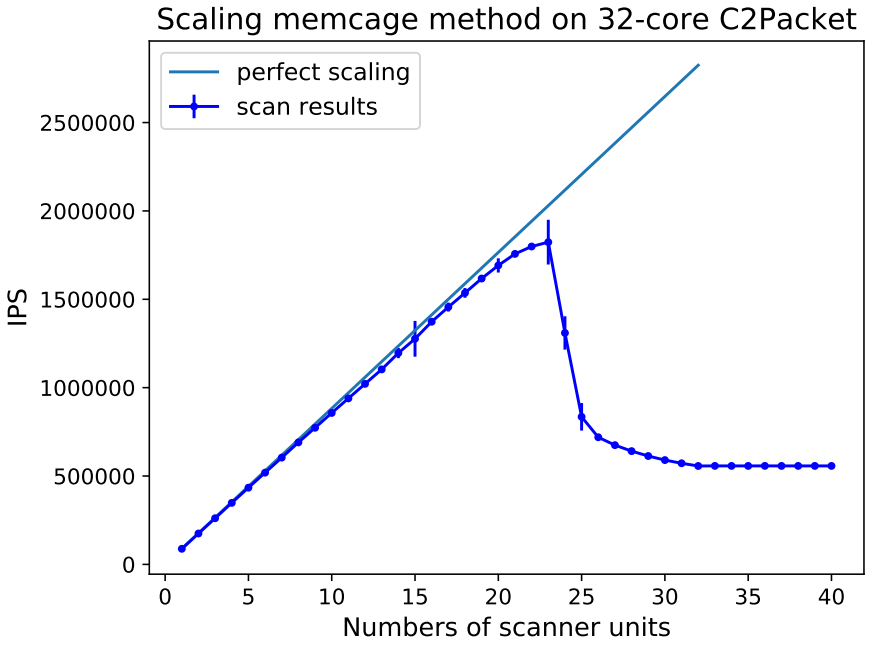

Figure 11: Performance scaling of memcage method on dedicated Packet c2.large.arm 32 core system

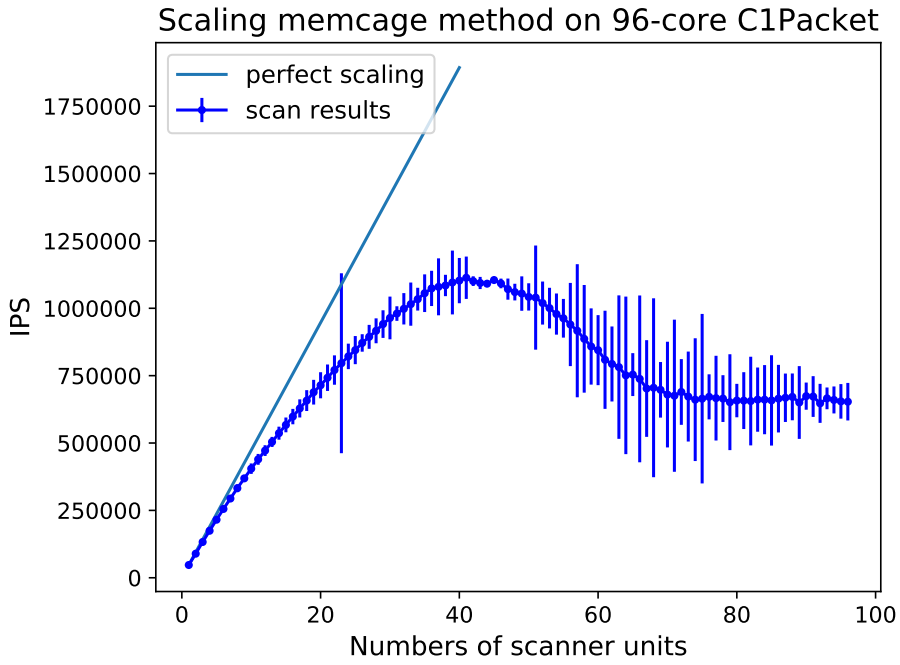

Figure 12: Performance scaling of memcage method on dedicated Packet c1.large.arm 96 core system

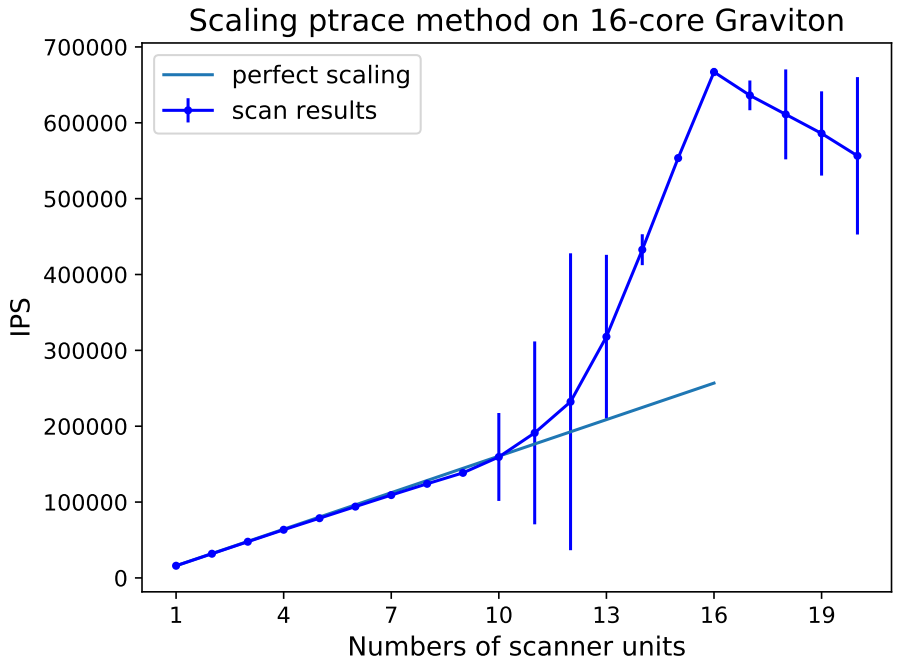

Figure 13: Performance scaling of memcage method on dedicated AWS a1.4xlarge 16 core system

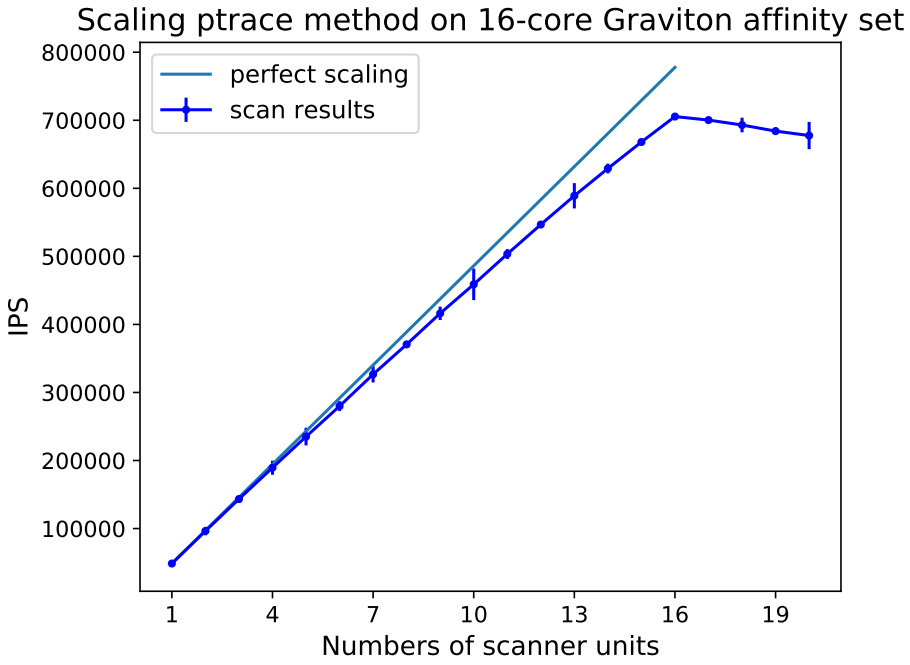

Figure 14: Performance scaling of the ptrace method on dedicated AWS a1.4xlarge 16 core system with processor affinity set

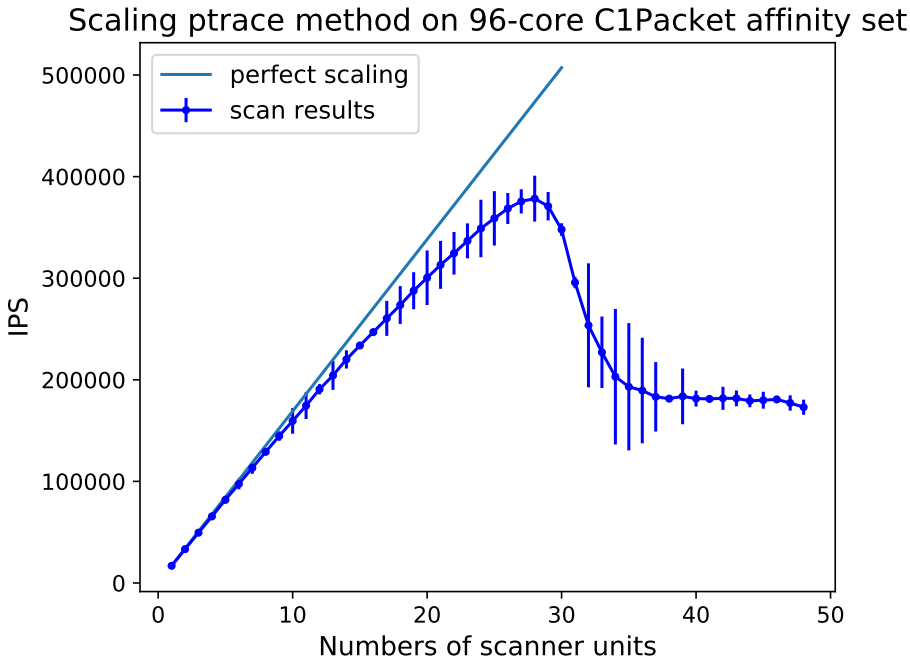

Figure 15: Performance scaling of memcage method on dedicated packet c1.large.arm system with processor affinity set

System	Ptrace	Memcage
RPI3B+	1.07x	1.01x
OPC2	1.07x	1.03x
AWS a1.4xlarge	1.06x	1.01x
Packet c1.large.arm	2.61x	1.19x
Packet c2.large.arm	2.14x	0.86x

Table 2: Performance changes when processor affinity is set

7.3.2 Effects of processor affinity

When using the -a command line option the number of available physical cores can be specified. The processor affinity of each scanner unit is set to a single physical core when using this option. Setting processor affinity influences the performance of the scanner. The changes in total performance are shown in Table 2.

Setting processor affinity has a large effect on the scaling behaviour of the ptrace method as shown in Section 7.3.1. Large performance increases are seen for the ptrace method on high core count systems. A smaller performance uplift can be seen on the lower core count systems.

Setting processor affinity does not change the scaling behaviour of the memcage method. Processor affinity does not have a significant impact on the total performance of the memcage method on the lower core count systems. The c1.large.arm is a dual socket system, setting affinity on such a system does increase performance. The only performance regression can be seen on the c2.large.arm system. Setting affinity is recommended when using the ptrace method or while scanning NUMA systems.

7.3.3 Comparison memcage and ptrace method

Table 3 shows the maximum achieved performance on all systems when running the scanner with the optimal number of scanner units found in Section 7.3.1. On all but one system the memcage method is over three times faster than the ptrace method. The AWS a1.4xlarge system is the only outlier, on this system the ptrace method is faster than the memcage method.

Something to keep in consideration is that when using the ptrace method there is no need to resolve the hangs described in Section 5.6.2. There are 2123 instruction in the A64 instruction set that cause such hangs. While a hang is unresolved the scanner units makes no progress. The existence of these hangs increases the overall run time. The size of this increase is dependent on the amount of time the manager takes to resolve them. Instructions that are known to cause hangs can also be blacklisted beforehand to negate this performance impact. To not skew the performance the results shown in Table 3 do not include any scans of instructions that would cause a hang.

System	Settings	Memcage (IPS)	Settings	Ptrace (IPS)	Difference
RPI3B+	4A	233,159	4A	61,931	3.76x
OPC2	4A	416,554	4A	125,085	3.33x
AWS a1.4xlarge	16A	533,871	16A	705,460	0.76x
Packet c1.large.arm	48A	1,326,964	28A	378,351	3.51x
Packet c2.large.arm	23	1,823,334	10A	500,326	3.64x

Table 3: Comparison of the maximum performance achieved for both methods over the first 50 million A64 instructions. Settings give the number of scanner units used followed by A if processor affinity was set.

7.3.4 System comparisons

Performance varies greatly even between similar systems, as shown in Table 4. Both the OPC2 and RPI3B+ are low power quad-core cortex A53 based systems running at 1.1Ghz and 1.4GHz respectively. They have the same cache setup and run kernels of the same version and major revision. These systems should therefore have similar performance, yet the the OPC2 system is over 2x faster when using the ptrace method and 1.80x faster when using the memcage method. Lmbench [12] was run on the systems in an attempt to find some performance differentiator. The difference in performance between the OPC2 and RPI3B+ systems seemed related to the difference in memory latency, but on inspection of lmbench results of the server systems this hypothesis was disproven. Table 4 shows that low power development boards may have better single core performance in the scanning task than server systems running at higher frequencies and with lower memory latency and bigger caches. Performance seems largely unrelated to frequency, cache size, memory bandwidth and memory latency.

7.3.5 Performance profile

To bring to light areas of future optimization a profile was done using perf. A single scanner unit was profiled for both the memcage and ptrace method. These profiles were created on the OPC2 system. The static binary compiled for the core scaling experiments in Section 7.3.1 was used.

In the memcage method only 25% of cycles is spent within the main handler described in Section 5.6. The rest of the cycles are spent within kernel service routines, signal delivery and signal

System	Memcage (IPS)	Performance	Ptrace (IPS)	Performance
RPI3B+	62,094	1.0x	15,610	1.0x
OPC2	111,753	1.80x	32,215	2.06x
AWS a1.4xlarge	35,733	0.58x	48,611	3.11x
Packet c1.large.arm	47,328	0.76x	16,909	1.08x
Packet c2.large.arm	90,927	1.46x	66,137	4.24x
HiFive Unleashed	152,473	2.46x	-	-

Table 4: Highest average single core performance measured

Task	Percentage of cycles
Analysis stage	26%
Fetch stage	25%
Cache sync	35%
Miscellaneous	14%

Table 5: Distribution of cycles within main handler of the memcage method

handler return code provided by the operating system. Any optimization in the memcage method implementation would have limited impact on overall performance due to the method being depended on exception handling and signal delivery. The division of cycles within the main handler is shown in Table 5.

In the ptrace method less than 10% of cycles is spent in analysis and fetch stages. Around 36% of cycles are spent in the waitpid() function, waiting on the status changes in the tracee caused by an exception. The necessary ptrace functions make up 53%. Any optimizations to analysis and the fetch stage would not have significant performance impact.

7.4 Scan results

We will now present the results of the scans of multiple ARMv8 and RISC-V systems.

7.4.1 ARMv8 A64

In this thesis, the scan results of A64 instruction space on the RPI3B+ and OPC2 systems are described. We refer to the thesis of Michael Göebel [7], for a discussion on the scan results of the packet c1.large.arm, c2.large.arm and AWS a1.4xlarge systems. To reiterate Section 5.2, results are marked as hidden by the scanner if an instruction is not valid according to the disassembler but is seen as valid by the hardware. A result is marked as a disassembler fault if it is a valid instruction according to the disassembler but is not seen as valid by the hardware.

The scanner program produced the same results when using either method on both systems. The scanner produced 5,446,386 results on the OPC2 system, 2,957,312 out of the total were marked as hidden and 2,489,074 as disassembler faults. The constrained unpredictable filter program was applied to the instructions marked as hidden, leaving 195,584 possible hidden instructions. On further analysis all marked encoding values that are left belong to two separate instructions: INS(element), DUP(general). Both these instructions do exist in the ARMv8 manual as advanced SIMD copy instructions. There seems to have been a mistake by the capstone developers interpreting a constraint related to these instructions. A filter was built that removes the wrongfully reported values. After applying this filter there are no values left, meaning that our scanner was unable to find any hidden instructions on the OPC2 systems Allwinner H5 processor.

Running the scanner on the RPI3B+ systems results in the same instructions being marked as hidden. Therefore like the OPC2 systems there are no hidden instructions found by the scanner

on the RPI3B+ systems Broadcom BCM2837B0 processor. There were however 302,080 extra instructions marked as disassembler faults. This is due to the Broadcom CPU not implementing some optional instruction set extensions that the the Allwinner H5 does.

7.4.2 RISC-V

QEMU A scan of the RISC-V Compressed instruction space was done on the QEMU 4.0.0 virtual machine. The scanner program marked 2272 16-bit instruction values as hidden instructions. There were zero instruction values marked as disassembler faults. The compressed instruction extension contained 9 instructions that caused a hang. Analysis of the 2272 values lead to the following findings:

- **0b0000000000000000** is seen as valid instruction by the virtual machine but is defined as an illegal instruction in the RISC-V ISA specification.
- **0b100xxxxxxxxxx00** is reserved within the RISC-V ISA specification but is seen as a valid instruction by the virtual machine.
- **0b100111xxx1xxxx01** is reserved within the RISC-V ISA specification but is seen as a valid instruction by the virtual machine.
- **C.ADDI16S, C.LUI** should be reserved if the nzimm field is zero.
- **C.ADDI4SPN** should be reserved when the nzuimm field is zero.
- **C.ANDI** instructions with immediate equal to zero are wrongfully marked as hidden due to a disassembler fault.
- **C.SRLI64, C.SRAI64, C.SLLI64** instructions with immediate equal to zero are wrongfully marked as hidden due to a disassembler fault. These instructions are legal HINT instructions when executed on a RV64 system.

All the instructions in reserved encoding spaces are executed as NOPs. These hidden instructions have no effect on processor state.

HiFive Unleashed The scanner program was run on the HiFive Unleashed system using the traversal technique described in section 5.7.1 in order to scan the entire RV64GC instruction space. A total of 8,405,814 instruction values were marked as hidden instructions and 25,144,389 values as disassembler faults. Analysis on the hidden values resulted in the following findings:

- **0b100xxxxxxxxxx00** is reserved within the RISC-V ISA specification but executes on the HiFive Unleashed board. In contrast to the QEMU results this range of instructions do not execute as NOPs. On this system the instructions in this range result in a SIGSEGV with a si_code of SEGV_MAPERR when scanned. This suggest that the instructions do a memory access in unmapped memory resulting in the segmentation fault. The associated fault address

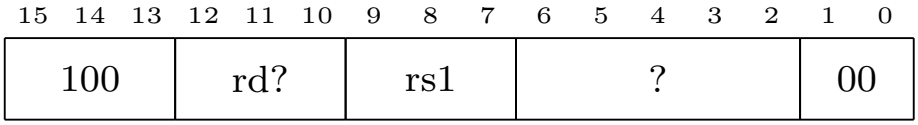

Figure 16: Fields within the encoding of the undocumented instruction. Format of destination register field rd is unknown. The rs1 field specifies the register number of the source register containing the address.

is 0 which is indeed unmapped. On deeper inspection the instructions turned out to be register relative load instructions. The instructions load four bytes from memory and perform sign extension. A register source field was identified within the encoding. This field grants control of the address where the value is loaded from. The register source field uses the 3-bits wide standard register number scheme of the C-extension. A destination register field was also identified, however the format is unknown limiting the number of addressable destination registers. The encoding fields are shown in Figure 16. When executed on the Spike RISC-V ISA simulator the instruction leads to an illegal instruction exception. This simulator is often seen as the golden reference for the behaviour of RISC-V systems. The instructions can only be part of an non-standard extension since they are in conflict with the existing C extension. No such extension is documented in the SiFive U540 manual [9]. Correspondence with the manufacturer confirmed that this is a known bug in the processor that was fixed from Oct 2017 onward. The bug leads to an unintentional non-standard extension. The undocumented instruction is an atomic memory operation, all permission checks are performed and therefore this undocumented instruction does not pose a security risk.

- **FENCE** instructions are wrongfully seen as illegal by the capstone disassembler when the rd and rs1 fields are not zero or the fm field is something other than 0b0000 or 0b1000. The rd and rs1 field along with the other possible values of the fm field are reserved for future extension. To ensure forward compatibility it is explicitly stated in the ISA specification that current implementations must treat any FENCE with values other than zero in these fields as a normal FENCE instruction. The behaviour of the HiFive Unleashed is correct. The fault lies with the capstone disassembler not recognizing the values as valid FENCE instructions.

- **FENCE.I** instructions are wrongfully seen as illegal by the capstone disassembler when the rd, rs1 or immediate fields are not zero. Similar to the FENCE instruction usage of these fields are reserved for future extension and for forward compatibility current implementations must ignore these fields and execute the instruction as a normal FENCE.I.

- **FCVT.D.S, FCVT.D.W, FCVT.D.WU** instructions with a value in the rounding mode field other than 0 are wrongfully seen as illegal by the capstone disassembler. Although different values in the rounding mode field have no influence on the instruction behaviour it is not illegal according to the ISA specification.

- **C.ANDI** instructions with immediate equal to zero are wrongfully marked as hidden due to a disassembler fault.

- **C.SRLI64, C.SRAI64, C.SLLI64** instructions with immediate equal to zero are wrongfully

marked as hidden due to a disassembler fault. These instructions are legal HINT instructions when executed on a RV64 system.

8 Limitations

As uncovered by Domas [5] instructions can be hidden by the manufacturer by only making them visible when certain model specific or system register bits are set. Any attempt to execute such an instruction will result in an illegal instruction exception until the specific bits are set. The instruction encoding is altered depending on system state. The scanner in this thesis would be unable to find instructions hidden in such a manner without adaptations.

Another way that a manufacturer could hide an instruction is by making it illegal for lower privilege levels only. The scanner program in this research runs in user space, commonly the lowest privilege level. Normally all instructions can be resolved from user space as described in Section 5.2.1. The ARMv8 specification defines a possible constraint named "Constraint_UNDEFEL0". It is mentioned no where else in the specification but its existence is telling for the capability of ARMv8 implementations to hide instructions in such a way. It is unclear what kind of signal would result from an instruction with such a constrained and if it could still be detected using the method described in Section 5.2.1.

Any hidden instruction residing in a blacklisted range of instruction encoding, will not be found by the scanner program. In this thesis the MSR instruction range is blacklisted on ARM systems. Hidden instructions could reside within this range.

Our scanner only checks for the existence of hidden instructions, it does not test if the instruction actually does what the ISA describes. So hidden functionality can still be hidden within a legal instruction encoding. Some analysis can be done on the instruction type based of delivered signal information but this is left to future work.

9 Conclusion

In this research two methods to search for hidden instructions are developed and implemented on both ARM AArch64 and RV64GC. The scanner is used to scan various systems, including both real silicon and emulators. Various mistakes were found in the used disassemblers. No hidden instructions were found on the ARM systems. Hidden instructions were discovered on the QEMU 4.0.0 virtual machine while emulating RV64GC. A hidden instruction was found on the SiFive Freedom U540 processor, this instruction loads 4 bytes from memory into a register. The instruction resides in a range of encodings explicitly defined as reserved in the RISC-V ISA specification.

While the ptrace method only functions on systems with single step capability and support in the linux kernel, the memcage method is more general and is implemented and shown to work on RISC-V as well as on ARM systems.

The implemented scanner is fast enough using either method to search common instruction spaces of size 2^{32} in less than a day on most systems. On most systems the memcage method delivers better performance. Most of the CPU cycles are spent in kernel interrupt service routines and signal delivery functions reducing the impact of future optimizations. Performance differences between systems seem not to have any meaningful relation with common performance characteristics such as clock speed, cache size or memory latency.

The memcage method showed predictable near perfect multi-core scaling on systems with less than 16 cores. On higher core count system scaling breaks down and performance regressions occur. Setting processor affinity for the scanner units is recommended when using the ptrace method or scanning NUMA systems. The performance of the ptrace method is very sensitive to scheduling of the scanner units. The ptrace method shows similar multi-core scaling behaviour when processor affinity is set

10 Future work

This section makes suggestions for future research.

- A different analyzer stage can be build that attempts to identify the instruction type from the signal information and register state at the time of the generated exception. This result can be compared to the disassembler.
- The existing implementation can be ported to many different RISC ISAs.
- More intelligent instruction space traversal methods could be implemented in order to speed up the scanning.
- Kernel modules can be written for chips that are capable of single stepping but that do not have support for this feature in the mainline kernel. The method that uses these kernel modules can be very similar to ptrace method. Such a method would be able to support more ISAs than the ptrace method and might achieve higher performance.
- Some instructions in the ARMv8 ISA are "constrained unpredictable" for certain encoding field values. These instructions may execute in a number of ways and our result show that they do. Similarly RISC-V has "HINT" instructions that may influence performance but not state. Research can be done to determine what the result of these instructions are and if these instructions break specification.
- Similar to Domas [5], scans can be done for different values of model specific and systems registers, to see if scan results change.

References

[1] Capstone. The Ultimate Disassembly Framework. URL: http://www.capstone-engine.org/.

[2] Michael J Clark. michaeljclark/riscv-disassembler, Sep 2018. URL: https://github.com/michaeljclark/riscv-disassembler.git.

[3] Robert R Collins. The intel pentium f00f bug description and workarounds. *Doctor Dobbs Journal*, 1997.

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

[5] Christopher Domas. Hardware Backdoors in x86 CPUs. *Black Hat, USA*, 2018.

[6] Loïc Duflot. CPU Bugs, CPU Backdoors and Consequences on Security. In Sushil Jajodia and Javier Lopez, editors, *Computer Security - ESORICS 2008*, pages 580–599, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[7] Michael Göebel. Developing and Verifying Methods That Search for Hidden Instructions on RISC Processors. *BSc Thesis, Leiden Universiteit, LIACS*, 2019.

[8] Andreas Hobein. Trouble with ptrace self-attach rule since kernel 2.6.14, Aug 2006. URL: https://lkml.org/lkml/2006/8/31/241.

[9] SiFive Inc. SiFive FU540-C000 Manual v1p0, Apr 2018. URL: https://sifive.cdn.prismic.io/sifive/834354f0-08e6-423c-bf1f-0cb58ef14061_fu540-c000-v1.0.pdf.

[10] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[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] Larry W McVoy, Carl Staelin, et al. lmbench: Portable Tools for Performance Analysis. In *USENIX annual technical conference*, pages 279–294. San Diego, CA, USA, 1996.

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

[14] Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi. End-to-end verification of processors with ISA-formal. In *International Conference on Computer Aided Verification*, pages 42–58. Springer, 2016.

[15] Jianping Zhu, Wei Song, Ziyuan Zhu, Jiameng Ying, Boya Li, Bibo Tu, Gang Shi, Rui Hou, and Dan Meng. Cpu security benchmark. In *Workshop on Security-Oriented Designs of Computer Architectures and Processors (SecArch 2018)*, SecArch'18, pages 8–14, New York, NY, USA, 2018. ACM. URL: http://doi.acm.org/10.1145/3267494.3267499, doi:10.1145/3267494.3267499.

Appendix A Labor division

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 6: Division of labor