VU VRIJE UNIVERSITEIT AMSTERDAM

VU Research Portal

IScanU: A Portable Scanner for Undocumented Instructions on RISC Processors

Dofferhoff, Rens; Göebel, Michael; Rietveld, Kristian; Van Der Kouwe, Erik

published in
2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)
2020

DOI (link to publisher)
10.1109/DSN48063.2020.00047

document version
Publisher's PDF, also known as Version of record

document license
Article 25fa Dutch Copyright Act

Link to publication in VU Research Portal

citation for published version (APA)
Dofferhoff, R., Göebel, M., Rietveld, K., & Van Der Kouwe, E. (2020). IScanU: A Portable Scanner for Undocumented Instructions on RISC Processors. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN): [Proceedings]* (pp. 306-317). Institute of Electrical and Electronics Engineers Inc.. https://doi.org/10.1109/DSN48063.2020.00047

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

E-mail address:
vuresearchportal.ub@vu.nl

Download date: 16. Jul. 2024

iScanU: a portable scanner for undocumented instructions on RISC processors

Dofferhoff, R.; Göebel, M.; Rietveld, K.F.D.; Kouwe, E. van der

Citation

Dofferhoff, R., Göebel, M., Rietveld, K. F. D., & Kouwe, E. van der. (2020). iScanU: a portable scanner for undocumented instructions on RISC processors. *Proceedings 50Th Annual Ieee/ifip International Conference On Dependable Systems And Networks*, 306-317. doi:10.1109/DSN48063.2020.00047

Version: Publisher's Version
License: Licensed under Article 25fa Copyright Act/Law (Amendment Taverne)
Downloaded from: https://hdl.handle.net/1887/3620481

Note: To cite this publication please use the final published version (if applicable).

2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)

iScanU: A Portable Scanner for Undocumented Instructions on RISC Processors

Rens Dofferhoff*, Michael Göebel†, Kristian F. D. Rietveld‡
LIACS, Leiden University
*rdofferhoff@live.nl, †michaelconnor1996@gmail.com, ‡krietvel@liacs.nl

Erik van der Kouwe
Vrije Universiteit Amsterdam
vdkouwe@cs.vu.nl

***Abstract*—Undocumented and faulty CPU instructions can cause undefined behavior and system instability, impairing software efforts such as OS crash recovery and resilience, and system security. Although often not considered, the identification of such undocumented instructions is critical. We present a portable RISC instruction scanner that is able to search for undocumented instructions on a wide range of RISC architectures, empowering users to verify the reliable and secure operation of their systems. We propose two methods to look for undocumented instructions. Both attempt to execute a single instruction word in a controlled manner, regaining control afterwards. Subsequently, we determine if the instruction word is considered valid by the processor, comparing this result to the processor's ISA specification. Our prototype scanner can scan multiple ARMv8 and RISC-V systems. Various inconsistencies were discovered in the QEMU emulator and disassemblers used as ground truth. Furthermore, we found an undocumented instruction on a RISC-V chip.**

***Index Terms*—Hardware verification, computer system reliability, hardware security**

I. Introduction

Correct and secure operation of computer systems depends on both software and hardware. Many efforts to improve the operation of computer systems focus mainly on the software stack, such as for instance development of operating system crash recovery methods [1] and hardening of the software system's security [2]. Hardware, and in particular processors, are treated as trusted black boxes. This unconditional trust is highly disputable. Hardware implementations do contain bugs and may contain instructions that are undocumented, undefined or faulty. A well-known example is the Pentium f00f-bug [3], which caused the CPU to stop working until reboot (known as "halt and catch fire instructions"). OS crash recovery methods cannot protect against such undefined behavior and will be unable to recover a system after execution of such a faulty instruction. More recently, we have seen some serious hardware security vulnerabilities that affect many modern processors, such as Meltdown [4] and Spectre [5]. These vulnerabilities cause a CPU to perform operations that allow code to bypass security boundaries, breaking system security but also hampering system reliability. We argue that processors should not be unconditionally trusted black boxes, but instead should be scrutinized by the end user to the same extent as they would the software they run.

For a long time, Intel's x86 architecture has been dominant in both the consumer and server markets. However, rising demand for low-power chips in mobile applications, Internet-of-Things (IoT) devices, and increasingly large server farms limited by cooling capability has made the CPU landscape much more diverse in recent years. Because of this, RISC processor designs, including the widely used ARM architectures, have become more popular. Due to the diversity and widespread use of RISC devices, there is a large chance some designs include undocumented or faulty instructions (whether intentional or not). This brings new challenges for processor verification. While large manufacturers employ state-of-the-art verification techniques, the Pentium f00f-bug demonstrates faults still do occur. On the other hand, the rise of open ISAs, such as RISC-V, creates a niche for smaller processor manufacturers. Smaller manufacturers will have fewer resources to spend on, and have less experience with, an extensive verification process, potentially increasing the chance of faults in a final product. Such a fault has been discovered by this research. Furthermore, RISC CPUs are commonly incorporated on System on Chips (SoCs) together with cores from other sources. SoC manufacturers have a need to verify that the different cores adhere to the specification before continuing integration. Application of state-of-the-art verification techniques may not be possible if the behavioral models or underlying design are not available to the SoC manufacturer.

When faults are not caught by the manufacturer, these faults end up residing in finished products, with ramifications for reliability and security that cannot be countered by software alone. Therefore, we see a more and more urgent need for third-party verification of processors. As a first step towards this important goal, we present a scanner program that empowers end users of a processor to check for undocumented instructions. Because underlying implementation details are commonly not available to end users, we must rely only on the provided ISA specification and the finished product itself.

We define an instruction word as undocumented when the processor recognizes the instruction word as valid but it does not encode for a valid instruction according to its instruction set architecture (ISA). Such undocumented instructions impact reliability. For example, crash recovery methods [1] often rely on fail-stop semantics in case an incorrect instruction is executed, as arbitrary operations performed by undocumented instructions are far harder to detect. Undocumented instructions can even affect security if those operations allow CPU-based isolation boundaries to be crossed.

978-1-7281-5809-9/20/$31.00 ©2020 IEEE
DOI 10.1109/DSN48063.2020.00047

Authorized licensed use limited to: Universiteit Leiden. Downloaded on June 07,2023 at 08:29:37 UTC from IEEE Xplore. Restrictions apply.

While previous work [6] has managed to automatically find undocumented instructions on Intel's x86 architecture, the solution was specific to the characteristics of that particular architecture and is not suited for others. For example, the approach is based specifically on the x86 trap flag and, because the x86 ISA uses variable-length instructions, implements an intricate detection mechanism to find the instruction length. RISC architectures typically have fixed-length instructions that, unlike on x86, must be aligned in memory. Moreover, to support multiple RISC architectures a generic replacement for the x86 trap flag must be found. This shows the need for new approaches to be able to efficiently scan instruction words in a portable way on multiple RISC architectures.

We present a design to efficiently scan instruction words on arbitrary RISC CPUs, capable of searching a full 32-bit instruction set within a day on all the hardware we tested. The scanner runs atop a POSIX-compliant OS. Two approaches are proposed to retain control when executing arbitrary instruction words, namely the ptrace method relying on the CPU's single step facilities and the memcage method relying on the memory management unit (MMU). Together, these approaches cover all hardware we tested without requiring any architecture-specific features, combining maximum portability with good performance. Our prototype scanner supports scanning multiple ARMv8 and RISC-V systems, and discovered various inconsistencies in the disassemblers we used, as well as flaws in the QEMU emulator. Furthermore, we found an undocumented instruction on a RISC-V chip.

Contributions: We make the following contributions:

- Two approaches to efficiently find undocumented instructions, with good portability among architectures.
- A prototype scanner supporting multiple ARMv8 and RISC-V systems, which has been made available as open source software [7].
- An evaluation showing the effectiveness and efficiency of our approach.
- A number of bugs identified in the Capstone disassembler [8] and QEMU system emulator [9], as well as an undocumented instruction in the SiFive Freedom U540 RISC-V CPU.

II. Related work

There has been little previous research on the subject of searching for undocumented instructions. To the best of our knowledge, Domas [6] was the first to design a scanner to search for undocumented instructions. This scanner specifically targeted the Intel x86 architecture, which is a variable-length CISC ISA with a maximum instruction size of 15 bytes, making the search space too large for an exhaustive search. To shrink the total search space, candidate instructions are straddled over page boundaries, using the MMU to make part of the candidate instruction inaccessible and determining which interrupt results when attempting to execute the candidate instruction. The x86 Trap Flag feature is used to retain control when executing arbitrary instructions. In this paper, we manage to achieve similar results on RISC systems without the need to use architecture-specific interfaces to set the trap flag (see Section IV-C), and provide the fallback memcage option in cases where no such hardware feature is available (see Section IV-D). All modern general-purpose processors with MMU provide the needed memory protection. Therefore, the memcage method provides a more general way of scanning for undocumented instructions than provided by Domas. Xi et al. [10] extend the work of Domas by further reducing the search space and associated efficiency gains. Like Domas' work, this extension only applies to x86 and an alternative to the trap flag is not proposed, which is one of the main contributions of our work.

In the development stages of processors, manufacturers verify their processor designs. Dessouky et al. reviews state-of-the-art approaches used to detect hardware vulnerabilities at design time, showing that a protection gap currently exists, leaving chip designs vulnerable to software-based attacks that can exploit these hardware vulnerabilities [11]. The discussed verification approaches are only viable with access to the CPU design and assumes that hardware is a faithful reproduction of that design, while our system allows independent third parties to test their CPUs hardware for undocumented instructions.

To use a verification technique such as 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. [12] describes how the ARMv8 ISA is converted to machine readable format, used to create Verilog models for formal verification. Our system uses disassemblers as a ground truth for correct processor behavior, analog to previous work done for the x86 architecture [10], [13]. It should be noted that the disassembler may be inconsistent with the official ISA specification, although in practice the number of false positives due to disassembler bugs was small enough to make manual verification of the results viable. In the future, we could extend our system to use Verilog models where available. On the contrary, in emulator testing and identification physical CPUs are used as the ground truth [14], [15]. Note that in our research a physical CPU cannot be used as ground truth, because it is these CPUs that are the object of scrutiny.

III. Overview

We present iScanU, a design to efficiently search the instruction space on RISC CPUs and identify undocumented instructions in a portable way. Our scanner supports RISC ISAs with fixed-length and hybrid-length encodings. It exhaustively checks all instruction words in the search space, which commonly has a size of 2^{32} (e.g. ARMv8 A64 and RISC-V). The exact range to be searched can be specified along with the scan method. The scan is performed in parallel by a specified amount of worker processes, controlled by a manager process. To find undocumented instructions we compare processor behavior when executing an instruction word to the ISA-specified behavior. We use OS signals to determine the validity of instruction words as seen by the processor while running the scanner in userland. When executing an invalid instruction

Authorized licensed use limited to: Universiteit Leiden. Downloaded on June 07,2023 at 08:29:37 UTC from IEEE Xplore. Restrictions apply.

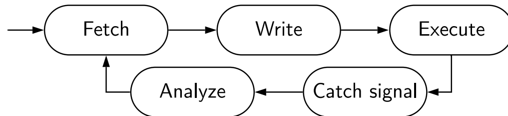

Fig. 1. Stages in the scan loop.

word, the OS delivers a SIGILL signal to the offending process. We ensure that in other cases a different signal is delivered in order to regain control. We use a disassembler to represent the ISA. We consider instruction words undocumented when they are not recognized by the disassembler as valid and do not yield SIGILL when executed.

Each worker process follows the loop shown in Figure 1. First an instruction word to check is generated (fetch) and written to executable memory (write). The worker process then tries to execute the instruction word. This results in a signal, which is caught and analyzed.

Three requirements must be satisfied to use our scan loop: (1) a signal must be guaranteed after every scanned instruction word no matter its validity; (2) the execution of valid and invalid instruction words should result in different signals; and (3) execution of an instruction word must not result in a crash or corruption of the scanner. All requirements must be met in an architecture-independent way to ensure portability between different ISAs. In this paper, we present two approaches that satisfy these requirements: the ptrace and memcage methods.

IV. Design & Implementation

In this section, we discuss the design of the scanner program. We first describe the analysis stage which determines whether an instruction word executes according to the ISA, and then we propose two approaches to ensure a single instruction word safely executes within the scan loop. Finally, state protection and instruction fetch are described.

A. Analysis stage

In the analysis stage, the input is the response of the CPU to the attempted execution of the instruction word and the output is whether this particular instruction word represents an undocumented instruction. The fetch stage selects the instruction word to be executed beforehand, see Section IV-F. We assume here the scanner program has already written this instruction word to memory and attempted to execute it.

To determine whether we found an undocumented instruction, we need to know whether the CPU actually executes the instruction word and whether it encodes for a valid instruction. Both steps occur in the analysis stage.

Attempting to execute an invalid instruction word causes the CPU to raise an interrupt. The OS handles this interrupt and, in response, informs the process that triggered it. On UNIX-based operating systems, it sends the SIGILL POSIX signal to the process. In many cases where the instruction word is valid, the operation it specifies will fail and result in a different kind of signal. For example, SIGSEGV or SIGBUS is received if the instruction references an incorrect memory address. In other cases, if the instruction word executes without errors, no signal results and the CPU starts executing the next instruction in memory. However, using the methods described in Sections IV-C and IV-D, our scanner ensures that after a successful execution a signal is *always* generated before any other code executes. As such, to determine whether the CPU considers the instruction word valid, we can simply check whether the signal is SIGILL (indicating an invalid instruction) or some other signal (indicating a valid instruction).

To decide whether a particular instruction word represents a valid instruction according to the documentation, we use a disassembler as a ground truth (see Section IV-B). The analysis stage passes the current instruction word to the disassembler. If the disassembler is unable to decode it, we assume it does not encode for a valid instruction according to the ISA.

The most interesting instruction words have a discrepancy between validity as considered by the CPU (as found in the signal analysis) and validity according to the documentation (as found using the disassembler). There are two possible cases: (1) valid according to disassembler but not valid on the CPU, and (2) invalid according to disassembler but valid according to the CPU. We will discuss these here in turn.

The first case is identified whenever the assembler (seemingly) correctly disassembles the instruction word but the OS sends a SIGILL signal on attempts to execute it. If we assume the scanner program is correct, one of the following causes must hold: (1) the disassembler does not properly represent the ISA specification, (2) the disassembler assumes an instruction set extension not implemented by the processor, or (3) the processor violates the ISA specification by not implementing a mandatory instruction. In our testing, the number of instruction words for which this happened was small enough to allow these causes to be distinguished through semi-automated inspection. This semi-automated processing consists of a simple script that groups similar instructions words. Subsequently, it is determined through manual inspection using the ISA manual whether these groups truly represent undocumented instructions or are mistakes in the ground truth. A filter script is used to remove the false positives. Some of these groups are quite large and result in millions of false positive results to be pruned at a time.

The second case is identified when the CPU correctly executes an instruction word (that is, the program does not receive SIGILL) but the disassembler fails to disassemble it. Again assuming the correctness of the scanner program, one of following causes must hold: (1) the instruction word represent an undocumented instruction, (2) the disassembler does not properly reflect the ISA specification, or (3) the disassembler is unable to decode an instruction set extension that the processor implements. Again, we distinguished these causes through semi-automated inspection.

It should be noted that which instructions are valid may depend on the current operating mode. In particular, attempting the execution of a privileged instruction in an unprivileged mode will result in a signal with signal number SIGILL, just like the executions of invalid instruction words. The

Authorized licensed use limited to: Universiteit Leiden. Downloaded on June 07,2023 at 08:29:37 UTC from IEEE Xplore. Restrictions apply.

scanner program is executed from within user space, which runs in unprivileged mode. The program can however still detect the existence of instructions on a higher privilege level. In the analysis stage we can differentiate invalid and privileged instructions by means of the `si_code` that is delivered along with the signal number of the signal. The attempted execution of invalid instruction words will result in `si_code ILL_ILLOPC` while privileged instruction will result in `ILL_PRVOPC`. This means we do not need to operate in kernel mode, which avoids considerable complications because unexpected instructions can do far more harm in kernel mode. Possible caveats are described in Section VI.

B. Ground truth

In this work the ground truth is provided by disassemblers, analog to previous work done for the x86 architecture [10], [13]. For ARMv8 we use Capstone 4.0.1 [8]. Some instructions within the ARMv8 specification are marked as 'constrained unpredictable' for certain field values. The results of these particular instructions can vary between implementations, but are constrained to a list of options such as undefined, no-operation, and load unknown values. 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 instruction words being marked as undocumented instructions. To remove these from the output, we filter these cases from the results. For RISC-V we use Capstone for all 4-byte wide instruction words. For compressed instructions we found Capstone to be insufficient. Therefore, we disassemble 2-byte instruction words using a different disassembler, namely the riscv-disassembler by M.J. Clark [16]. This disassembler seems to better reflect the ISA specification for the compressed instruction extension.

C. Ptrace method

To be able to perform the analysis described in the previous section, the worker processes must ensure that the attempted execution of every instruction word is immediately followed by a signal, transferring control back to the analysis code and preventing additional instructions from being executed. There are multiple approaches to achieve this, and we found that no single approach allows us to reach our goals in a portable way. The ptrace method, described in this section, can guarantee immediate signal delivery in all cases, but requires specific hardware support. The memcage approach presented in Section IV-D only requires MMU support, making it more widely applicable, but additional caution must be taken to ensure a signal is always generated. Moreover, our evaluation shows that performance characteristics differ between methods and platforms, making both worthwhile in specific cases.

The ptrace method guarantees signal generation 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 can 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.

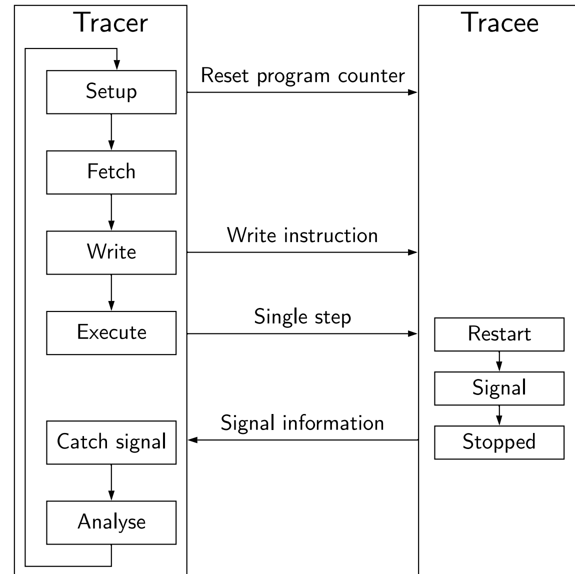

Fig. 2. Overview of the ptrace method worker process.

Figure 2 shows the design of a worker process under the ptrace method. The scanner consists of a tracer/tracee process pair, with the worker process acting as tracer. Using the `ptrace` single step function, the tracer allows the tracee to execute a single instruction word, followed immediately by a `SIGTRAP` signal if it executes correctly. If the instruction word is invalid, `SIGILL` will still be delivered. Regardless of the instruction type or validity, a signal returns control back to the tracer, which starts the analysis stage.

The ptrace method cannot be used on all RISC systems. Ptrace single stepping is dependent on on-chip debug capabilities. Some ISAs make no demands when it comes to these capabilities, such as RISC-V. This results in some processors having no support for ptrace single stepping within the kernel. The ptrace method has been shown to work on the ARMv8 ISA. In cases where the ptrace method cannot be used, the memcage method is still available as a fallback.

D. Memcage method

The ptrace method guarantees to trigger a signal after each execution of an instruction word, but can only be used if the hardware supports single step execution of code. To provide a more general alternative, we additionally developed the memcage method. The memcage method only requires memory protection capabilities, which are available on all general-purpose processors equipped with a MMU. Given that the scanner program was designed to be run from userland (which implies the need for memory protection), the memcage method

Authorized licensed use limited to: Universiteit Leiden. Downloaded on June 07,2023 at 08:29:37 UTC from IEEE Xplore. Restrictions apply.

does not introduce any additional architectural requirements in practice.

The memcage method ensures a signal for every instruction word by placing it in memory in a particular way. We allocate a single executable page and write the instruction word to be executed at its end. This page will be referred to as the instruction page. The instruction page is followed by a non-executable page. An overview of this setup is depicted in Figure 3. Now, if the instruction word is invalid, it will generate the usual SIGILL signal. If it encodes for a legal instruction, its execution does not cause a signal on its own, and the instruction does not alter control flow, the next instruction lays in a memory page that is not executable. Attempts to execute this instruction will result in a segmentation fault. Accordingly, this setup guarantees a signal for all instructions that do not alter control flow.

Instructions that *do* alter control flow, such as branch and jump instructions, may not continue with the next instruction in memory. To ensure signal generation in all cases, we must consider the memory areas that can be reached with the different addressing modes and take precautions where required. Table I considers the different addressing modes found in RISC architectures and for ARMv8 and RISC-V in particular.

When the registers are all set to zero, register-relative and absolute addressing can only reach a small region of memory at the bottom and top of the memory space. The executable pages belonging to the text section of the scanner are never located here. Therefore, these instructions always cause a signal to be generated.

Program counter (PC)-relative branches and jumps do require precautions, as they can reach addresses surrounding the instruction word to be tested. The maximum offset from the PC that can be reached is typically small. We mitigate this by surrounding the instruction page with two non-executable blocks of memory (referred to as guards) sized after this maximum offset, see Figure 3. Jumps or branches that land within these guards will immediately result in a signal. To handle backward jumps that land within the instruction page, we fill this page with filler instructions that trigger a signal other than SIGILL. In cases where instructions have offsets that are too large to make these approaches viable, they can simply be blacklisted. We have encountered no such instructions on ARMv8 and RISC-V. Together, these precautions ensure any instruction within these bounds is correctly classified.

While our approach ensures a signal for all documented instructions, it is fundamentally impossible to cover every possible effect an undocumented instruction may have. If an undocumented instruction with extended capabilities were to exist it could cause the scanner to crash, confirming the instruction word that caused the crash represents an undocumented instruction or has capabilities not conforming to ISA specification. To avoid detection entirely an instruction must cause SIGILL for all possible values of its immediate fields, which is highly unlikely as most pages either contain valid instructions or are non-executable, and any pages not specifically guarded by our approach would be very hard to reach.

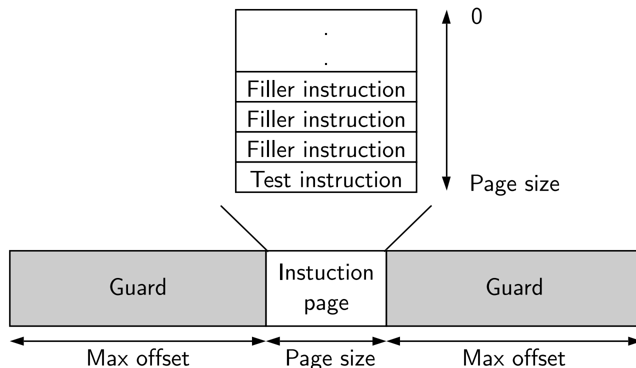

Fig. 3. Memory construct to protect program state and guarantee a signal. Grey area designates non-executable memory.

Figure 4 shows the design of the scanner program using the memcage method. The main signal handler is set for all signals that may result from instruction execution. Each attempted execution of an instruction word will cause a signal. This signal ensures that the main handler will be called, returning control so the scanner can start analysis and prepare the next iteration. An architecture-specific 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 signal. This context will be restored when the signal handler returns. After the steps shown in Figure 1 are taken, the context structure is altered such that upon return of the signal handler all general-purpose registers will be zero and the program counter is reset. In addition to the main signal handler, there is an entry handler to store a good known state and start the scan loop. The entry handler is started by the manager process by sending an SIGUSR signal to the worker process.

Some instructions may form self-contained infinite loops. For example, PC-relative jump and branch instructions add a (scaled) offset to the program counter. When this offset is zero, this instruction will be repeated indefinitely. This causes the worker process to hang because it cannot progress to the next instruction word. The manager resolves such hangs by periodically checking the progress of all worker processes. If it detects that a worker is hanging, it sends the worker process a user-defined signal. Each worker process has a special signal handler installed for this specific signal. Within this handler the instruction word causing the hang is logged and the analysis stage is started, after which scanning resumes with the next instruction word. The hang handler is shown in Figure 4. To make hang detection possible, worker processes write progress updates to memory shared with the manager process. How these pages are protected against modification by tested instructions is described in Section IV-E. The ptrace method does not suffer from these self-contained loops as signal generation in this case does not depend on the resulting

Authorized licensed use limited to: Universiteit Leiden. Downloaded on June 07,2023 at 08:29:37 UTC from IEEE Xplore. Restrictions apply.

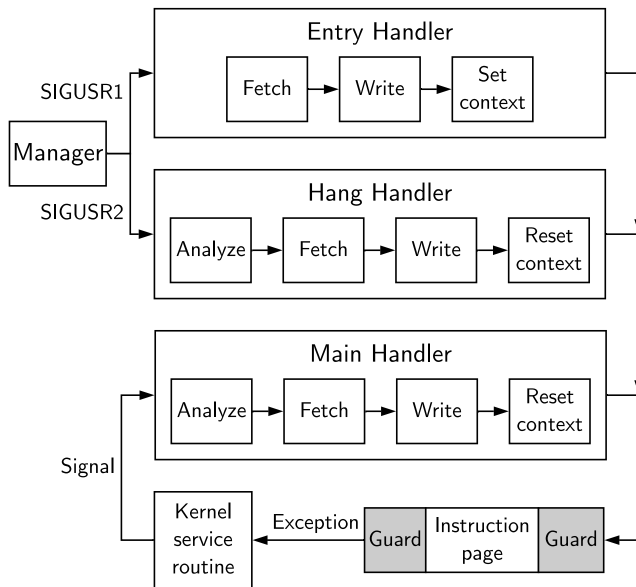

Fig. 4. Overview of the memcage worker process.

program counter after instruction execution.

Note that the memcage method relies on memory protection and does not inherently require memory virtualization. This method could therefore work with just a memory protection unit (MPU) instead of an MMU, assuming enough physical memory is available for the guard pages to be sufficiently large to guarantee signals for control flow altering instructions.

E. Program state protection

An exhaustive scan of the instruction space will trigger the execution of store instructions that will write data to memory. The scanner program must guarantee that store instructions cannot write to parts of memory where the variables of the scanner program reside. By ensuring the tested instructions cannot reach these parts of the address space, we guarantee the generation of a signal, namely a harmless `SIGSEGV` signal that is caught by the scanner program and handled later.

Because virtually all RISC architectures are load-store architectures, only store instructions can overwrite program variables. Like control flow instructions, store instructions can employ different addressing modes which must all be considered to ensure the state of the scanner program is protected in all cases. Table I shows the effective reach of addressing modes for all types of instructions. Again, register-relative and absolute addressing are dealt with by setting the general-purpose registers to zero before the instruction words are executed. The maximum offsets are rather limited due to the limited instruction sizes. As no writeable pages of the scanner program are located in the lowest and highest regions of the memory space, a segmentation fault is generated.

Although rare, PC-relative addressing may be supported for store instructions. In such a case, a store may be performed that ends up either within the guard pages or the instruction page. The guard pages do not permit writes, so a segmentation fault is raised. Modifications to the instruction page are harmless, as the instruction page can be fully rewritten before the execution of the next instruction word is attempted. By limiting ourselves to RISC, we do not expect instructions that perform both a store as well as a branch/jump operation.

An undocumented instruction could theoretically use a new addressing mode that is able to reach the scanner variables. However, this is unlikely to occur due to the shared memory pages being a small target within the virtual memory space and all general-purpose registers being set to zero. Moreover, if any of the encoded immediate values for this instruction does cause a segmentation fault, the instruction will still stand out when analyzing the results.

Like the other registers, the stack pointer is set to zero to prevent rogue memory accesses. However, when starting a signal handler, the kernel assumes the stack pointer points to writable memory because it needs to write the signal handler parameters on stack. As a consequence, 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, we use the `sigaltstack` call to set an alternative stack address for signal handling. This way, the kernel restores the stack pointer register before calling signal handlers.

The ABI associated with the ISA will often designate certain functions to general-purpose registers, including not just the stack pointer, but also a pointer to thread local storage or a global pointer. For all these registers, valid values need to be restored to be able to execute the scanner program. These values or references to them cannot be stored in global variables because the global pointer is invalid. We let the entry handler store known good values of these registers at 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 an architecture-specific function using some inline assembly, good values for any important registers can be recovered.

F. Instruction fetch stage

The instruction fetch stage generates the next instruction word to be analyzed. Within this paper, we only consider exhaustive search. However, there are some instructions which have side effects that cannot be contained by our system. An example is the `BLX` instruction on ARM, which switches between the regular and Thumb instruction sets. For this purpose we check each instruction word against a manually determined blacklist. As an extension to exhaustive search, our system also considers instruction length to prune the search space where possible. Some RISC ISAs have instruction sets with a hybrid-length encoding meaning that an encoded instruction can have a few different fixed lengths. Examples of such instruction sets are RISC-V C or the ARM THUMB2 extension. In both of these an instruction word can be 16 or 32 bits wide. The first 16 bits fetched by the processor determines the length of the

Authorized licensed use limited to: Universiteit Leiden. Downloaded on June 07,2023 at 08:29:37 UTC from IEEE Xplore. Restrictions apply.

TABLE I
OVERVIEW OF DIFFERENT ADDRESSING MODES COMMON ON RISC ARCHITECTURES AND THE MEMORY REGION THAT CAN EFFECTIVELY BE REACHED ON ARMV8 AND RISC-V ARCHITECTURES.

Addressing mode	ARMv8	RISC-V	Effective Reach
Register-relative loads/stores/jump	$reg + 64\ KiB$ / $reg - 512\ B$	$reg \pm 2\ KiB$	*reg* is set to zero, so only the lowest or highest few pages of the memory space can be reached. Only RISC-V supports register-relative jumps.
PC-relative branch	$PC \pm 1\ MiB$	$PC \pm 4\ KiB$	Branches to within the instruction page are handled with filler instructions. The instruction page is surrounded by two blocks of non-executable memory (referred to as guards) sized after the maximum offset. Figure 3 illustrates this setup.
PC-relative jump	$PC \pm 128\ MiB$	$PC \pm 1\ MiB$	Handled similarly to branches, but note the larger offsets requiring larger guards.
PC-relative loads	$PC \pm 1\ MiB$	N/A	Loads are harmless, no precaution necessary.
Absolute addressing	N/A	N/A	Many RISC architectures handle absolute addressing through register-relative addressing with the base register set to zero. The range of the memory space that can be reached with a single instruction is small. To achieve a larger reach, two instructions are required where the first instruction is used to initialize the base register to a large value.

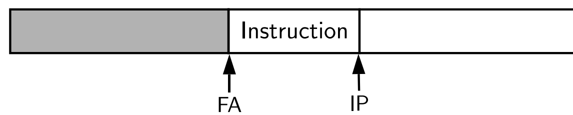

Fig. 5. An incorrect size is identified by the fault address (FA) returned in the signal info struct pointing to the page boundary, while the instruction pointer (IP) has not moved.

instruction. To support the RISC-V C extension new analyzers and fetch stages were developed for the memcage method.

In order to scan hybrid-length instruction sets efficiently, we traverse the instruction space by first executing all prefixes as instructions with the smallest possible length. Instruction words are aligned to the end of the instruction page. If the length is incorrect, an attempt to execute the instruction word will cause a signal, since part of the instruction word lies in memory that is not executable. The resulting register state and signal information can help us identify this situation, as shown in Figure 5. For this method to work properly, the neighboring page must be readable and writable but not executable. If the length is determined to be incorrect, the short instruction word is used as a prefix to full-length instruction words and we enumerate all possible instruction words with that prefix. Using this traversal method, we prune the search space considerably if the instruction set contains many shortened instructions. For example, in case of the RISC-V C extension, only a quarter of the normal 32-bit instruction space remains.

G. Portability

The scanner program has been purposely designed to be portable to different RISC architectures. Porting the scanner only involves the configuration of a small number of architecture-specific properties and functions, such as instruction length and register file configuration, and implementation of a common interface to query the ground truth and alter context structures for this particular architecture. One of the advantages of the ptrace method over the memcage method is the ease of portability, as there are fewer architecture-specific functions to overload, and the fact that less initial configuration is required. Although the memcage method does not rely on a common kernel interface, its implementation is generic in nature and only requires a number of additional architecture-dependent variables and functions to be provided, such as a function to set and recover important ABI registers from the altstack address, as descibed in Section IV-E.

V. EVALUATION

We conducted a number of experiments using our prototype scanner implemented in C++ to test its effectiveness. Firstly, we verify whether our design is able to find undocumented instructions by performing a scan against QEMU [9] emulated processors which have been modified on purpose to include undocumented instructions. Secondly, we perform a full scan of a variety of ARMv8 systems and a RISC-V development board. Finally, we investigate the performance characteristics of the scanning program. The surveyed systems are listed in Table II. In the remainder of this section Orange PI PC2 will be shortened to OPC2 and Raspberry Pi 3B+ to RPI3B+.

A. Verification with QEMU

To verify the program's ability to find undocumented instructions, the program needs to be tested on a platform on which such instructions are known to exist. We did not have such a platform available as real hardware. Therefore, we chose to modify QEMU [9] to emulate a system with purposely inserted undocumented instructions. We achieved this by remapping invalid instruction words to handlers of legitimate instructions. We inserted a total of eight artificial undocumented instructions and chose the instruction words such that the various paths through the main handler are tested.

The scanner was found to be able to find all eight artificial undocumented instructions when using the memcage method, verifying its correct operation. The ptrace method could not be verified using QEMU because of inaccurate emulation: every scanned instruction word except the first yields a `SIGSEGV` signal with a fault address at the start of the instruction page, this is also the case for a breakpoint instruction that should raise `SIGTRAP`. This must be caused by inaccurate emulation

Authorized licensed use limited to: Universiteit Leiden. Downloaded on June 07,2023 at 08:29:37 UTC from IEEE Xplore. Restrictions apply.

TABLE II
OVERVIEW OF THE SURVEYED SYSTEMS.

Full Name	ISA	OS	Kernel	CPU	#cores
QEMU 3.1.0 virtual	ARMv8	Debian 9.8	4.9.0-8-arm64	Virtual ARM Cortex-A53	8
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 (bare-metal, not shared)	2 x 48
Packet c2.large.arm	ARMv8	Ubuntu 18.04	4.15.0-46-generic	Ampere eMAG 8180 (bare-metal, not shared)	32
AWS a1.4xlarge	ARMv8	Ubuntu 18.04	4.15.0-1028-aws	AL73400 AWS Graviton (run on dedicated host)	16
QEMU 4.0.0 virtual	RV64 IMAFDCSU	Fedora-Minimal-Rawhide-20190326	5.1.0-0.rc1.git0.1.1 .riscv64.fc31.riscv64	Virtual RV64GC	4
HiFive Unleashed	RV64 IMAFDC	freedom-u-sdk image	4.15.0-00044-g2b0aa1	SiFive Freedom U540	4

by QEMU since this problem does not present itself when the ptrace method is run on real hardware, and both methods always yield the same results when run on hardware.

B. ARMv8 A64 scan results

We will now present the results of scans of the ARMv8 A64 instruction space for the ARM systems described in Table II.

a) QEMU: A full scan of the ARMv8 A64 instruction set space on QEMU 3.1.0 was not possible because several instruction words caused faults that could not be handled by the scanner. For example, the instruction word `11010101010110111101000001000000` caused the scanner to crash, even though the manual indicates that this is an unallocated encoding which should have led to an illegal instruction fault. The scanner crashes while executing an undocumented instruction with unknown side effects.

The majority of the search space was explored successfully, by iteratively adding instructions to the blacklist in case of a crash and restarting the scan. As can be seen in Table III about 78 million instruction words were marked as undocumented and approximately 72 million instruction words remain after filtering known false positives. Note that a single incorrect entry in the disassembler can cause millions of false positives. Many instruction words encode the same instruction with different immediates, registers, etc. Similarly, a single undocumented instruction or ISA implementation bug causes many discrepancies. For the hardware platforms that were surveyed, in the end zero undocumented instructions were detected. To further investigate the QEMU results, a number of ranges from the collection of 72 million detected undocumented instruction words were manually inspected. All were determined to be marked as unallocated in the ISA manual. In the interest of time not all 72 million could be manually investigated, but it is probable that most, if not all, of these undocumented instructions are unallocated according to the ISA manual and should not execute on the emulator.

Note that the number of disassembler faults is significantly lower compared to the hardware platforms. We found that, compared to other platforms, QEMU is marking more instructions as valid that are also considered as valid by the disassembler. This concerns optional instruction set extensions that are not implemented by other platforms[1]. Single unimplemented instructions with large immediate fields cause many thousands of values to be marked as disassembler faults, this causes the large difference of disassembler faults between QEMU and the other systems and also the large difference in undocumented instructions between the systems.

In conclusion, QEMU appears to accept many more instruction values that include instructions known by the disassembler, but also instructions that are not known by the disassembler and therefore in the majority of cases undocumented. As a consequence, the QEMU 3.1.0 aarch64 full system emulator does not properly emulate an aarch64 chip.

b) Hardware: We scanned the ARMv8 A64 instruction space on the ARM platforms shown in Table II, these scans took between 2-12 hours depending on the system and parallelization factor. The results of the scans of the ARMv8 A64 instruction space on various hardware platforms are shown in Table III. Most of the instruction words marked as undocumented are caused by constrained unpredictable instructions being presumed as undefined by the Capstone disassembler. The results that remain after filtering these instructions are also shown in the table. As was discussed in Section IV-B, constrained unpredictable instructions can be implemented in different ways while still conforming to the ISA specification. This may lead to either more or less instruction words being wrongfully marked as undocumented, as shown by the larger number of initially undocumented instruction words for the Packet c2.large.arm system that after filtering has the same number of remaining results as the other systems. On all but the Packet c1.large.arm system 195,584 instruction words remain after filtering the constrained unpredictable instructions. These remaining instruction words are false positives, namely they belong to DUP (general) and INS (element) instructions. Capstone appears to wrongly assume a constraint to be present for these instructions. After these instruction words are filtered out there are no more

[1]Although the fact that QEMU is not throwing an illegal instruction exception for these instructions does not mean these instructions are actually implemented in QEMU, they could simply be no-ops.

Authorized licensed use limited to: Universiteit Leiden. Downloaded on June 07,2023 at 08:29:37 UTC from IEEE Xplore. Restrictions apply.

TABLE III

NUMBER OF INSTRUCTION WORDS MARKED AS DISASSEMBLER FAULT (VALID ACCORDING TO DISASSEMBLER, INVALID TO CPU) OR AS UNDOCUMENTED INSTRUCTIONS (INVALID ACCORDING TO DISASSEMBLER, VALID TO CPU) ON THE ARMV8 TEST SYSTEMS. FILTERS WERE APPLIED IN ORDER TO EXCLUDE FALSE POSITIVES (INCLUDING THE INS AND DUP INSTRUCTIONS) FROM THE RESULTS.

System	Disassembler fault	Undocumented		
		Initial	Ex. constr. unpredict.	Ex. INS/DUP
QEMU 3.1.0	393,216	78,389,761	72,625,153	72,429,569
RPI3B+	2,791,154	2,957,312	195,584	0
OPC2	2,489,074	2,957,312	195,584	0
AWS a1.4xlarge	2,489,074	2,957,312	195,584	0
Packet c1.large.arm	2,489,074	12,600,320	6,888,448	6,692,864
Packet c2.large.arm	2,489,074	6,103,040	195,584	0

results left except on the Packet c1.large.arm system. This can be explained by the fact that the Packet c1.large.arm system implements ARMv8.1 ISA, while the disassembler used only supports ARMV8.0-A and as a consequence marked all new instructions in V8.1 as illegal, producing many false positives as a result. Indeed, after ARMv8.1 instructions were excluded, no undocumented instruction words remain.

Concluding, no undocumented instructions were found on any of the surveyed ARM systems. The disassembler faults are caused by the disassembler supporting optional instruction set extensions that the processors do not implement.

C. RISC-V scan results

a) QEMU: We also scanned the RV64C instruction space, containing just 16-bit compressed instructions, on the QEMU 4.0.0 virtual machine. A full scan of the entire RV64GC instruction space was not attempted due to the length detection mechanism not functioning properly because of problems in the emulation by QEMU[2]. The scanner program marked 2272 16-bit instruction words as undocumented. No disassembler faults were found. The compressed instruction space contained 9 instruction words that caused a hang. From an analysis of the 2272 undocumented instruction words follows that QEMU sees several instructions as valid that are in fact defined to be illegal or reserved according to the ISA specification. Other instructions were marked as undocumented due to disassembler faults. All the undocumented instructions in reserved encoding spaces are executed as no-operations and have no effect on processor state. None of the undocumented instructions we found pose a reliability or security risk.

b) Hardware: On the HiFive Unleashed, our scanner marked 8,405,814 instruction words as undocumented and 25,144,389 as disassembler faults. From an analysis of the undocumented instruction words follows that one true positive undocumented instruction has been found and the other instruction words are false positives caused by disassembler faults. For the latter, the details can be found in the online appendix [17].

The undocumented instruction that we detected is encoded by instruction words from the `0b100xxxxxxxxxxx00` range. This range of words is marked as reserved within the RISC-V ISA specification but these instruction words do execute on the HiFive Unleashed board. In contrast to the QEMU results, this range of instruction words does not execute as no-operations. On this system the instruction words in this range result in a `SIGSEGV` with a `si_code` of `SEGV_MAPERR` when scanned. This suggests that they perform a memory access in unmapped memory, resulting in the segmentation fault. On deeper inspection the instruction words turned out to encode for an register-relative load instruction. This instruction loads four bytes from memory and performs sign extension. The fields identified in the encoding are shown in Figure 6. When executed on the Spike RISC-V ISA simulator[3], considered the golden reference for the behavior of RISC-V systems [18], the instruction leads to an illegal instruction signal. The instruction can only be part of a non-standard extension since they are in conflict with the existing C-extension. No such extension is documented in the SiFive U540 manual [19]. Correspondence with the manufacturer confirmed that this is an internally known bug in the processor that was fixed from October 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 reliability or security risk.

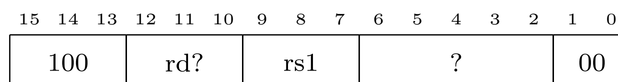

Fig. 6. Fields within the encoding of the undocumented instruction. Format of destination register field `rd` is unknown. The `rs1` field specifies the register number of the source register containing the address.

D. Performance

As the final experiment, we consider the performance characteristics of both proposed instruction scan approaches on different systems. Based on these results, the most suitable scan method and parallelization factor can be selected.

Given that the task of the scanner program is inherently parallel, as there are in theory no dependencies between two worker processes concurrently scanning instruction words on

[2]The instruction pointer returned in the signal handler does not increase when executing a sequence of 2-byte instructions in QEMU 4.0.0.

[3]https://github.com/riscv/riscv-isa-sim

Authorized licensed use limited to: Universiteit Leiden. Downloaded on June 07,2023 at 08:29:37 UTC from IEEE Xplore. Restrictions apply.

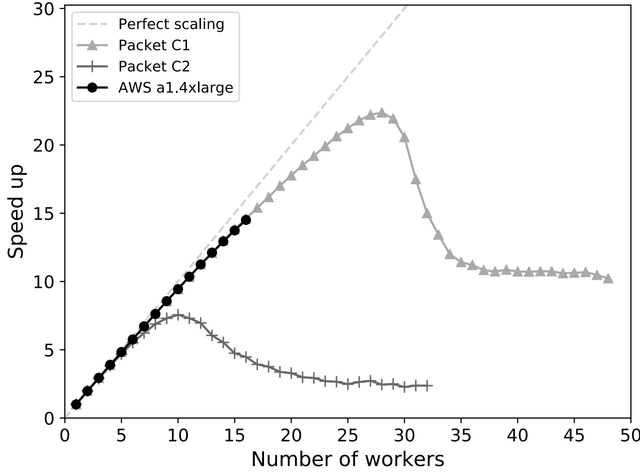

Fig. 7. Performance scaling behavior of ptrace method for cloud instances with large number of cores.

Fig. 8. Performance scaling behavior of memcage method for cloud instances with large number of cores.

different physical cores, we first investigate whether both scanning approaches show good performance scaling when the number of worker processes is increased. For each test system, we performed experiments by increasing the number of worker processes until the number of available cores on a single socket is reached. We set processor affinity to assign each worker process a specific core to improve performance. For each ARM system, the first 50 million instruction words in the A64 instruction space have been scanned. For the RISC-V system, 4000 16-bit instruction prefixes, in many cases leading to the fetch of a full 32-bit instruction word, were scanned, resulting in approximately 65.5 million scanned instruction words. The number of scanned instruction words is logged by the manager process every ten seconds. These performance results are processed into an average number of scanned instruction words per second (IPS). To plot the performance scaling behavior, we determined the attained IPS for increasing numbers of worker

TABLE IV
OPTIMAL NUMBER OF WORKER PROCESSES.

System	Optimal #workers	
	Ptrace	Memcage
HiFive Unleashed	-	4
RPI3B+	4	4
OPC2	4	4
AWS a1.4xlarge	16	16
Packet c1.large.arm	28	48
Packet c2.large.arm	10	20

processes and computed the speedup compared to the IPS attained by a single worker process. The manager statically divides equal parts of the search space among the workers at startup. The goal of the experiment is to estimate the scanning performance of multiple worker processes operating in parallel, so we stop the measurement as soon as the first worker process completes its job.

For the ARM single board computers with a small number of cores, the ptrace method shows near perfect scaling up to the number of available physical cores (recall that the ptrace method is not supported on the RISC-V architecture). There is no performance gain when the number of worker processes is increased beyond the number of physical cores, as the CPU utilization was observed to be maximized when the number of workers equals the number of physical cores. For the cloud platforms the results vary, as can be seen in Figure 7. On the AWS platform, with 16 physical cores, the near perfect performance scaling keeps up until 16 cores are reached. On the Packet c2, the ptrace method achieves a maximum speedup of a factor 7.6. When more workers are added, the achieved speedup slides below a factor 5 and stabilizes. The Packet c1 scales well until approximately 30 worker processes, after which the achieved speedup decreases.

With regard to the memcage method, similar near perfect scaling is seen for the ARM single board computers. The RISC-V development board also exhibits near perfect scaling. The results for the different cloud platforms are shown in Figure 8. Again, the AWS platform scales well until 16 cores and the speedup stabilizes there. The Packet c2 shows better scaling than the Packet c1, and scales nicely until approximately 20 worker processes, contrary to the ptrace method where the scaling topped out around 10 worker processes. On the Packet c1 platform we observe that the attained speedups increase at a slower pace compared to the Packet c2, however the performance continues to improve until the number of available cores, in contrast to the ptrace result, and a slightly higher overall speedup is achieved.

For both Packet systems, the performance starts to decrease as more workers are added after a certain point. Profiling reveals that this regression is caused by an 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.

Based on the previous results, we determined the optimal

Authorized licensed use limited to: Universiteit Leiden. Downloaded on June 07,2023 at 08:29:37 UTC from IEEE Xplore. Restrictions apply.

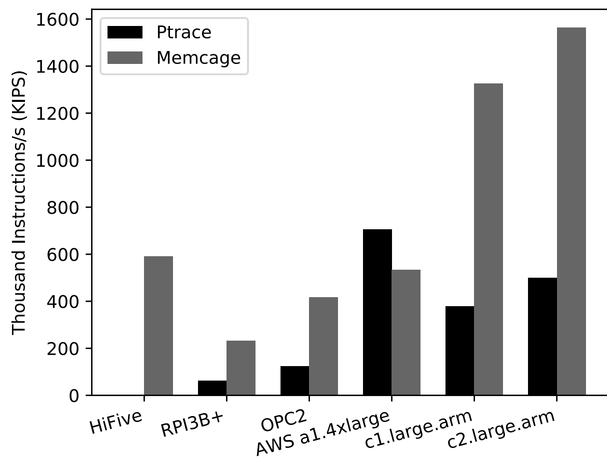

Fig. 9. Comparison of maximum performance achieved for both scan methods.

number of worker processes for all systems. These optimal settings are listed in Table IV and the maximum performance achieved using these settings is depicted in Figure 9. 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 exception, on this system the ptrace method is faster than the memcage method. Profiling shows that most time is spent within kernel service routines, signal delivery, and signal handler return code provided by the kernel. Next to kernel version used, the performance of the scanner depends on several other factors such as processor affinity.

VI. Limitations

While we have shown our tool can find undocumented instructions in an efficient and portable way, it is possible for our design to exhibit false negatives for several reasons.

As shown by Domas [13], 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 signal until the specific bits are set. Instruction legality is altered depending on system state. The memcage method sets all general purpose registers (GPR) to zero in order to guarantee signal delivery. The scanner in this paper would therefore be unable to find undocumented instructions with state requirements tied to the GPRs without adaptations. Iterating over all possible configurations of processor state and re-scanning in order to find such an undocumented instruction would be time intensive due to the number internal states a processor can have. This makes finding an undocumented instruction with a state requirement almost impossible.

Another way a manufacturer could hide an instruction is by making it illegal in lower privilege levels. Our scanner program runs in user space, commonly the lowest privilege level. Although typically all instructions can be resolved from user space as described in Section IV-A, it would be possible to create an ISA in which privileged instructions are fully hidden from user space processes. In this case, execution of a privileged instruction in user space would result in `ILL_ILLOPC` instead of `ILL_PRVOPC`, which implies that our method would be unable to detect this privileged instruction. Such an architecture would be out of the ordinary and be inefficient to virtualize. The ARMv8 specification defines a possible constraint named `Constraint_UNDEFEL0`. It is not mentioned elsewhere 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 constraint and if it could still be detected using the method described in Section IV-A. Exploitation of such privileged instructions would be difficult however.

Our scanner only checks for the existence of undocumented instructions, it does not test if the instruction actually does what the ISA describes. Undocumented functionality can still be hidden within a legal instruction encoding. Some analysis can be done on the instruction type based of delivered signal information and register state but this is left to future work. It should be noted that undocumented instructions hidden in this way would cause legitimate programs to fail if they trigger the conditions for undocumented behavior.

VII. Conclusion

Although undocumented and faulty instructions in processors pose serious reliability and security threats, processors have not been scrutinized to the same extent as software systems. To this end, we presented a scanner program which employs two different methods to allow third parties to efficiently scan the instruction space on RISC processors for undocumented instructions. We show that our design is effective, as our prototype found many undocumented instructions in the QEMU emulator as well as an undocumented instruction on a RISC-V processor. Both methods show good performance, allowing common 32-bit instruction spaces to be searched in less than a day on slower systems and mere hours on faster ones. Both methods show good performance scaling up to 16 cores, allowing effective use of modern multi-core processors.

To foster future research the source code of our scanner is freely available [7]. We made the used methods as platform independent as possible to encourage porting to other architectures. Aside from using the existing scanner on different processors and architectures, future work in this area includes: (1) expanding the analysis stage to detect basic instruction types, which can be compared with the ISA specification in an attempt to search for hidden functionality among valid instructions encodings and to identify the basic function of undocumented instructions that are found; (2) development of more intelligent traversal strategies to prune the search space and speed up scanning such as automatic instruction format recognition; (3) usage and creation of different ground truths; (4) performing repeated scans of the instruction space with different model-specific or system register bits set to detect undocumented instructions with state requirements.

Authorized licensed use limited to: Universiteit Leiden. Downloaded on June 07,2023 at 08:29:37 UTC from IEEE Xplore. Restrictions apply.

REFERENCES

[1] K. Bhat, D. Vogt, E. van der Kouwe, B. Gras, L. Sambuc, A. S. Tanenbaum, H. Bos, and C. Giuffrida, "OSIRIS: Efficient and Consistent Recovery of Compartmentalized Operating Systems," in *DSN*, Jun. 2016, best Paper Session.

[2] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013, pp. 48–62.

[3] R. R. Collins, "The intel pentium f00f bug description and workarounds," *Doctor Dobb's Journal*, 1997.

[4] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *USENIX Security Symposium 2018*, 2018, pp. 973–990.

[5] 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 *IEEE Security&Privacy 2019*, 2019.

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

[7] iScanU Project, "iScanU Source Code," https://github.com/RensDofferhoff/iScanU.

[8] C. Project, "Captone - The Ultimate Disassembly Framework," http://www.capstone-engine.org/, last accessed August 2019.

[9] F. Bellard, "QEMU, a fast and portable dynamic translator." in *USENIX Annual Technical Conference (ATC 2005)*, vol. 41, 2005, p. 46.

[10] X. Li, Z. Wu, Q. Wei, and H. Wu, "Uisfuzz: An efficient fuzzing method for cpu undocumented instruction searching," *IEEE Access*, vol. 7, pp. 149 224–149 236, 2019.

[11] G. Dessouky, D. Gens, P. Haney, G. Persyn, A. Kanuparthi, H. Khattri, J. M. Fung, A.-R. Sadeghi, and J. Rajendran, "Hardfails: Insights into software-exploitable hardware bugs," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 213–230. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/dessouky

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

[13] C. Domas, "Hardware Backdoors in x86 CPUs," *Black Hat, USA*, 2018.

[14] L. Martignoni, R. Paleari, G. F. Roglia, and D. Bruschi, "Testing cpu emulators," in *Proceedings of the eighteenth international symposium on Software testing and analysis*, 2009, pp. 261–272.

[15] O. Sahin, A. K. Coskun, and M. Egele, "Proteus: Detecting android emulators from instruction-level profiles," in *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2018, pp. 3–24.

[16] M. J. Clark, "RISC-V Disassembler," https://github.com/michaeljclark/riscv-disassembler.git, last accessed August 2019.

[17] iScanU Project, "Online Appendix," https://github.com/RensDofferhoff/iScanU/blob/master/appendix.pdf.

[18] A. Waterman and K. Asanović, *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20190608-Base-Ratified*. RISC-V Foundation, March 2019, p. 157.

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

Authorized licensed use limited to: Universiteit Leiden. Downloaded on June 07,2023 at 08:29:37 UTC from IEEE Xplore. Restrictions apply.