Uncovering Hidden Instructions in Armv8-A Implementations

Fredrik Strupe[*]
Silicon Labs, Norway
fredrik.strupe@silabs.com

Rakesh Kumar
NTNU, Norway
rakesh.kumar@ntnu.no

ABSTRACT

Though system and application level security has received and continue to receive significant attention, interest in hardware security has spiked only in the last few years. The majority of recently disclosed hardware security attacks exploit well known and documented hardware behaviours such as speculation, cache and memory timings, etc. We observe that security exploits in undocumented hardware behaviour can have even more severe consequences as such behaviour is rarely verified and protected against.

This paper introduces *armshaker*, a tool to uncover one such undocumented behaviour in the Armv8 architecture, namely hidden instructions. These are the instructions that are not documented in the ISA reference manual, but still execute successfully. We tested five different Armv8-A hardware platforms from four different vendors, as well as two Armv8-A emulators, and uncovered multiple hidden instructions. An interesting finding is that, though we did not discover any hidden instruction in the hardware itself, bugs in the system software can induce hidden instructions in the system that, from a user's perspective, are indistinguishable from hidden instructions in hardware.

Though *armshaker* did not find any hidden instruction in the hardware of the tested platforms, their existence cannot be ruled out, given the diversity of available Arm processors. Consequently, we make *armshaker* publicly available as open-source software to enable users to audit their own systems for hidden instructions.

1 INTRODUCTION

Computers and digital systems have become an integral part of modern life, and something both individuals and societies as a whole rely on to a great extent. This augments the importance of security in these systems, as a security breach can have potentially disastrous consequences. Therefore, system and application level security has received and continue to receive a lot of attention. Security at architecture and hardware level, in contrast, has only recently started to gain momentum after Kocher et. al. [10] showed that the security of the whole system can be compromised by exploiting hardware vulnerabilities.

Most of the existing research on architecture/hardware security focuses on exploiting well-known and documented hardware behaviours for devising offensive and defensive mechanisms. For example, Spectre [10] and its successors [3, 4, 11] alter a victim program's control flow to leak its memory contents by leveraging a well documented optimization called speculative execution. Similarly, observing the cache/memory response time can also be used to mount an attack.

We observe that undocumented or hidden behavior of the hardware can create new security vulnerabilities. A great amount of trust is put into the hardware designers and manufacturers, in that the hardware functions exactly as documented with no hidden or undocumented behavior. However, even without any malicious intent, security vulnerabilities can still be present in hardware as a result of design bugs [5, 10, 12] or manufacturing faults [14]. For instance, such hidden behaviour was uncovered by Domas [7], when he revealed the presence of a secret coprocessor in a particular x86 processor model that could be accessed by executing a certain combination of machine instructions. This discovery was enabled by earlier work of his on processor fuzzing of the x86 ISA [6]. Such hidden behavior needs to be identified and protected against any possible exploits. The fact that processor verification primarily targets verifying documented behaviour makes vulnerabilities due to undocumented behaviour even more likely.

This paper presents a mechanism to uncover one such undocumented behaviour, called *hidden instructions*. We define a hidden instruction as a particular instruction encoding that successfully executes on a processor without raising an expected exception, while at the same time not being officially documented or documented as unallocated or non-functional. Note that a hidden instructions does not necessarily mean that the processor does not raise any exception; rather, the processor raising a wrong exception or the system software handling the exception incorrectly also indicates presence of hidden instructions.

To uncover hidden instructions, we introduce a tool, called *armshaker* [15], that targets the latest version of the Arm ISA, namely Armv8-A. Arm is a particularly interesting target because of its widespread use – ranging from low-powered embedded devices to state-of-the-art super computers – with a particular prevalence in smart phones and a corresponding security impact potential. *armshaker* works by exhaustively searching through the whole instruction space of the three instruction sets in Armv8-A (A64, A32 and T32), executing instructions that are undefined in the ISA specification. If an undefined instruction executes without faults, it is marked as a hidden instruction and logged for further analysis. As Armv8-A itself is an architectural specification, *armshaker*, strictly speaking, tests implementations of the ISA. It is therefore not limited to testing only physical processors, but can also probe virtual (emulated) implementations such as ISA emulators.

We used *armshaker* to examine three different Armv8-A microarchitectures (Cortex-A53, Cortex-A72, and Cortex-A73) from four different vendors (Broadcom, Qualcomm, HiSilicon, and Allwinner)

[*]This work was done while the author was at NTNU.

HASP '20, October 17, 2020,
© 2020 Association for Computing Machinery.
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *HASP '20: Workshop on Hardware and Architectural Support for Security and Privacy, October 17, 2020*, https://doi.org/10.1145/1122445.1122456.

for hidden instructions. In addition to hardware implementations, we also probed two Armv8-A emulators (QEMU and Arm Base Fixed Virtual Platform). *armshaker* uncovered multiple hidden instructions in all of the tested targets. Further analysis showed that the hidden instructions in the tested hardware platforms could be attributed to bugs and backward compatibility measures in the Linux kernel, rather than actual hardware bugs. For QEMU, the hidden instructions were the result of bugs in its underlying Arm instruction decoder. In addition to hidden instructions, *armshaker* also discovered bugs in commonly used disassemblers: the *libopcodes* disassembler used by *objdump* and Capstone[13].

We did further analysis to identify the root causes of the uncovered bugs, fixed them, and submitted patches to Linux, QEMU and GNU binutils – enabled by all of them being open-source. Our patches have been accepted by the respective projects.

Though *armshaker* did not find any hidden instruction in physical hardware, the results do reveal an interesting aspect of the relation between the underlying ISA implementation and the operating system or system software. Specifically, the operating system can induce hidden instructions in the system that, from a user's perspective, are indistinguishable from hidden instructions based in hardware.

This paper was derived from the first author's master's thesis – *Probing the Armv8-A ISA for Hidden Instructions through Processor Fuzzing* – done in 2020 at NTNU under the direction of the second author [16]. The key contributions of this work, summarized in this paper, include:

- **armshaker**: We design and implement a portable and open-source tool that automatically identifies divergent behavior of undefined instructions in hardware and software implementations of Armv8-A ISA. *armshaker* is publicly available at [15].
- **Hidden Instructions**: We use *armshaker* to uncover software-induced hidden instructions resulting from bugs in the Linux operating system kernel and the QEMU processor emulator. We also find bugs in two commonly used disassemblers: Capstone and the libopcodes disassembler.
- **Software Improvements**: We identify the root causes of the software bugs and submit patches to Linux, QEMU and GNU binutils which have been subsequently accepted for inclusion in the respective projects.

2 BACKGROUND AND MOTIVATION

armshaker targets Armv8 which is the latest version of the Arm architecture. To cater to the needs of diverse market segments ranging from supercomputers to low-power embedded devices, Armv8 comes in three different architecture profiles: *A*, *R*, and *M*. The *M* (microcontroller) profile targets low-power embedded systems; the *R* (real-time) profile targets real-time systems; and the *A* (application) profile targets mostly everything else where the performance is a primary concern. This work focuses on *A* profile and we therefore refer to the ISA as Armv8-A.

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

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

2.1 Instruction Sets

There are three instruction sets available in Armv8-A: A64, A32, and T32. However, only one of these can be used at any given time depending on the current execution state. The most relevant part of the instruction sets to our work is the instruction encoding, rather than the actual functionality available and how to use the instructions to get that functionality. Therefore, we next discuss the instruction encodings in these three instruction sets.

2.1.1 A64: The A64 instruction set was introduced in Armv8-A and is used in the AArch64 execution state. It uses a significantly different instruction encoding compared to the instruction sets in earlier versions of the Arm architecture. Nevertheless, most of the available functionality is the same as before. Furthermore, the instruction width is also fixed at 32 bits like previous instruction sets.

Consider Figure 1 as an example of an instruction encoding in A64, namely for the `ADD (immediate)` instruction. The extended mnemonic would be `ADD Rd, Rn, #imm12`, where `Rd` and `Rn` represent registers and `#imm12` represents an immediate value, with the functionality being that `#imm12` is added to `Rn` and stored in `Rd`.

As the figure shows, although `ADD (immediate)` represents a unique operation, it will have many different instruction encodings depending on the values of its operands and options. The non-changing part of the encoding that uniquely identifies the particular operation is called the opcode (operation code) part, which in the case of Figure 1 are the bits marked with binary values instead of placeholder labels, namely bit 30 to 23.

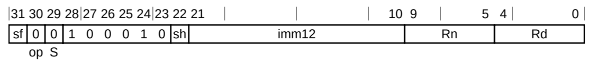

Figure 1: A64 encoding for `ADD (immediate)` instruction [1].

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

Comparing the A32 encoding of the `ADD (immediate)` instruction in Figure 2 with the A64 encoding of the same instruction in Figure 1, we observe that the two differs in the opcode value, register indicator lengths (resulting from A32 having fewer available registers), operand ordering and the presence of the condition code.

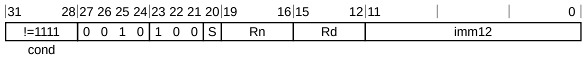

Figure 2: A32 encoding for `ADD (immediate)` instruction [1].

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

The Thumb instruction set originally used a 16-bit fixed-length encoding. However, because of the limited instruction space, an extension was later introduced to support certain 32-bit long instructions, making the instruction set variable-width. This was implemented by having the upper five bits of the encoding indicate whether an instruction is 16-bit or 32-bit. In the case of 32-bit instructions, a second half-word is fetched, making up the lower half of the 32-bit instruction. Specifically, a 32-bit T32 instruction is indicated by the upper five bits having a value greater than or equal to `0b11101` (where the `0b` prefix denotes a binary number).

The 16-bit T32 encoding of the `ADD (immediate)` instruction can be seen in Figure 3, with the corresponding 32-bit T32 version in Figure 4. Note that some T32 instructions like `ADD (immediate)` have both 16-bit and 32-bit encodings, while others are exclusively 16-bit or 32-bit.

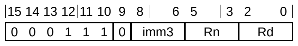

Figure 3: 16-bit T32 encoding for `ADD (immediate)` [1].

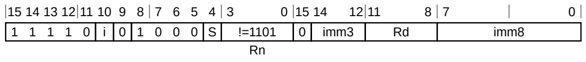

Figure 4: 32-bit T32 encoding for `ADD (immediate)` [1].

2.2 Unallocated Instructions

The maximum possible number of instructions in an instruction set is determined by the number of available bits in the instruction encoding. Since both A64 and A32 use a 32-bit encoding, they can represent about 4 billion unique instruction encodings. T32 can represent comparatively fewer unique encodings as some of the instructions need to fit in only 16-bits. However, none of the instruction sets utilize *all* of the available instruction encodings. Rather, there are chunks of unallocated encodings in each instruction set, resulting from the encodings not being assigned any instruction. Our analysis revealed that 64.3% of the encodings in A64 are unallocated, with the proportions being 12.2% in A32 and 31.6% in T32[1]. The reason for the big difference between A64 and A32 is primarily that the condition code in the A32 encoding effectively duplicates most of the defined instructions, reducing the amount of unallocated instructions. T32 has a higher percentage than A32 due to its lack of condition codes, but lower than A64 as it has a smaller instruction space. *armshaker* aims to uncover hidden instructions corresponding to these unallocated instruction encodings.

[1]Instead of manually searching the Architecture Reference Manual [1] to find unallocated encodings, we used the *libopcodes* disassembler as a proxy to get these numbers.

3 *ARMSHAKER*

armshaker is written in C and intended to run on the Linux operating system kernel [17], both of which are widely supported on most Armv8-A-based systems. Targeting Linux not only eases *armshaker* development, but also enables a wide range of systems to be probed without requiring any special hardware setup or software configuration. Also, we have taken care to use as few external dependencies as possible and generally avoid operating system features not universally available, to increase operability across platforms.

3.1 Overview

Conceptually, the execution flow of *armshaker* can be divided into a set of recurring stages, as shown in Figure 5. First, some initialization code is run before entering the main loop. Then, on every successive iteration of the loop, a new instruction to be tested is generated. The generated instruction is then disassembled using an external disassembler, and the resulting disassembly is checked. If the disassembly was successful – implying that the instruction is defined – the loop continues with the next instruction. Otherwise – in the case of it being undefined according to the disassembler – the instruction is executed, and its result is checked. If an illegal instruction signal (`SIGILL`) was received, then the loop is repeated as this is the expected behavior. However, if another signal or no signal at all was received, the instruction is marked as a hidden instruction and subsequently logged, before moving on to the next instruction.

3.2 Design Details

3.2.1 Initialization: The initialization stage does primarily two things. First, it processes the options passed to *armshaker* on the command line and sets the respective internal configuration variables. Then, it sets up the infrastructure needed for the following stages. This involves configuring the disassembler, clearing log files, initializing the instruction execution environment and so on.

3.2.2 Instruction Generation: A straightforward approach to instruction generation is to generate all possible bit combinations for the instruction encoding. For the 32-bit instruction encoding of Armv8-A, this would generate all bit combinations between `0x00000000` and `0xFFFFFFFF` (where `0x` denotes a hexadecimal number), i.e. 4 billion instructions. Such an approach generates each instruction/opcode multiple times with different source and destination operands. For example, an *add* instruction would be generated with all possible source and destination register combinations. In contrast, to uncover hidden instructions, we need to test an opcode only once to check whether the hardware can execute it or not. Though this redundant instruction generation slows down testing, we found that such an exhaustive search in the whole instruction space finishes in a reasonable time on modern processors.

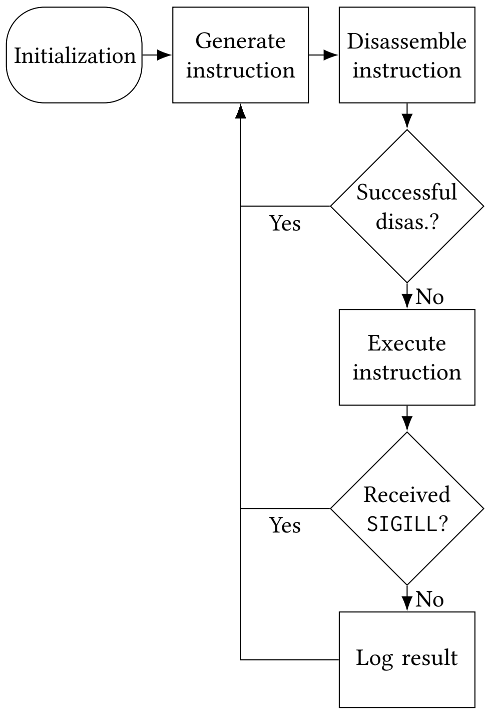

Figure 5: Execution flow of the *armshaker* loop.

Therefore, *armshaker* takes this simple approach for instruction generation.

In addition to exhaustively searching the whole instruction space, *armshaker* also supports searching only a specific range of the instruction space through *search masks*. When a search mask is applied, only the bits in the instruction encoding matching the particular mask are incremented, with the remaining bits left unchanged. This option is particularly useful when analyzing the results from an exhaustive search. Another use of masked increment is for the variable-width T32 instruction set. Recall that T32 instructions with the upper five bits being lower than `0b11101` are 16-bit, while the remaining ones are 32-bit. To capture both widths in a single search, all instructions are stored in a 32-bit variable internally, with 16-bit instructions having the lower two bytes set to 0 – equivalent to the lower half-word being zero. When incrementing an encoding, the upper five bits are first checked, and if they indicate a 16-bit instruction, a masked increment is performed on the upper half-word, leaving the lower half-word unchanged. For 32-bit instructions on the other hand, no mask is applied. This ensures a smooth transition from 16-bit to 32-bit instructions.

3.2.3 Disassembly: Since *armshaker* aims to uncover hidden instructions, it does not need to execute all the instruction encodings generated by the previous stage, rather only the ones that are undocumented/undefined in Armv8-A. Even if it were to execute all the instruction encodings, we would still need a mechanism to verify whether an illegal instruction signal (`SIGILL`) was expected or not after executing a particular instruction. Therefore, it is necessary to know which of the instruction encodings correspond to undocumented/undefined instructions which requires looking up Arm Architecture Reference Manual [1].

Instead of manually searching the reference manual for undefined instructions, we use a disassembler as an automated reference manual lookup mechanism. For that, *armshaker* passes each generated encoding to a disassembler and checks the result. If the disassembly is successful – indicating that the particular instruction is in fact defined – the remainder of the loop is skipped and we move on to the next encoding. Otherwise, in the case of the disassembler failing to disassemble the instruction – indicating an undefined instruction – the instruction is executed. Since the disassembler only passes undefined instructions to the next stage, we expect the hardware to generate an undefined instruction exception for all the instructions.

By using a disassembler as a proxy to the Arm Architecture Reference Manual, we make a fundamental assumption that the disassembler faithfully represents the reference manual. However, we discovered that this is not always the case even for the most commonly used Arm disassemblers. There are several reasons for this, with the most common ones being bugs in the disassemblers, unsupported ISA extensions and inaccuracies in the interpretation of the ISA specification.

Our initial implementation used the *Capstone* disassembler [13], which is a framework using disassemblers from the LLVM project in the back-end. Unfortunately, a lack of full Armv8-A support combined with large amounts of instructions incorrectly being marked as undefined made it unsuitable for our needs. We did not investigate whether this behaviour was caused by Capstone itself or the use of an older LLVM back-end. Instead, we opted for the Arm disassembler in *libopcodes* – a part of the GNU binutils project [8] – which is the same disassembler employed by the widely used *objdump* utility. The disassembler in libopcodes had a much higher accuracy compared to Capstone, possibly because Arm actively contributes to its development. However, it was still not without a few bugs which lead to some instruction disassemblies being incorrect. We fixed these bugs and submitted patches, which are already accepted, to the repository.

3.2.4 Execution: In the execution stage, a given instruction is executed to check whether its execution results in an undefined instruction exception, i.e. an illegal instruction signal (`SIGILL`) from the kernel. If a `SIGILL` signal is received, *armshaker* continues to execute the next undefined instruction as receiving `SIGILL` is the expected behavior for undefined instructions. On the other hand, if a different or no signal is received – indicating a hidden instruction – the instruction is sent to the next stage for logging.

Executing an undefined instruction can have a wide range of different side-effects, potentially corrupting *armshaker* itself. Therefore, it is essential to isolate the instruction execution from the rest of the *armshaker* process to the extent where the probability of an instruction corrupting the *armshaker* execution environment is sufficiently low. For this we support two execution isolation techniques, namely *page-based* and *ptrace-based* isolation, each with their own isolation degree and corresponding pros and cons.

Page-Based Isolation: Similar to [6], this technique executes an instruction in the same process environment as *armshaker*, however in a dynamically allocated memory page with some boilerplate code. First, in the initialization stage of *armshaker*, an executable memory page is allocated with the `mmap()` system call and some boilerplate code is loaded onto it for storing/restoring register values on entry/exit, resetting register values before execution, and a placeholder for the instruction to be tested. Also, a signal handler is set up which is triggered every time a signal is received from the kernel and stores the signal number in a variable. Later, in every iteration of the *armshaker* loop, the instruction to be tested is written to the placeholder in the executable page and subsequently executed before jumping back to the main loop. Whether the execution triggered the signal handler or not can then be determined by reading the variable set by the signal handler. This variable stores the signal number in the case of a trigger and zero otherwise. If the signal stored in the variable is anything other than `SIGILL`, it indicates a hidden instruction.

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

A somewhat obscure but important detail for this method to work is that the first-level instruction and data caches in the Arm architecture are not *coherent*, meaning that an update to either of them does not necessarily propagate to the other immediately. This means that self-modifying code like described above – that is, repeatedly writing to and executing a page – might lead to only the data cache being updated, with the instruction cache containing an older instruction. As a result, instead of executing the most recent instruction, *armshaker* would execute whatever instruction was in the page when the instruction cache was last updated, effectively skipping over some instructions. To solve this problem, every time a new instruction is written to the placeholder, *armshaker* writes the data cache contents back to main memory and reloads them into the instruction cache, such that the newest instruction is fetched. This can be done with the `__clear_cache()` function in GCC.

While this technique works in most cases and performs well, the isolation it provides is relatively weak. For example, any hidden instruction changing the program counter or certain system registers can corrupt *armshaker*. While this rarely happened in practice in our tests, to make *armshaker* more robust, we propose a second technique based on process tracing functionality available in Linux.

Ptrace-Based Isolation: This execution isolation technique, as the name suggests, is based on the `ptrace()` system call in Linux. With ptrace, a process (the tracer) can trace another process (the tracee), enabling the tracer to alter the tracee's register content, memory content and execution flow. Concretely, the technique works by letting a separate child process execute the instructions to be tested, with the main *armshaker* process using ptrace to modify the child's state on every iteration of the loop. This means that if an instruction corrupts the child process, its state can simply be reset or the process restarted without interrupting *armshaker*.

In ptrace-based isolation, *armshaker*, in the initialization stage, uses the `fork()` system call to create an identical copy of itself, which becomes the child process to be traced. The child process then jumps to a function with an infinite loop. The loop contains three instructions: first, a breakpoint instruction that stops execution until resumed by the tracer; second, a placeholder instruction that will be replaced with the instruction to be tested; and finally, a branch instruction that jumps back to the breakpoint instruction.

After initialization and at the beginning of every iteration of the main loop, the child will be in a stopped state and open for modification by the main process. The main process then resets all of the child's registers, writes the instruction to be tested in the placeholder, and resumes the child process. In response, the child will execute the instruction. If executing the instruction generates a signal (like `SIGILL`), the child will stop and notify the main process. Otherwise, it will continue to the branch instruction, jump back to the breakpoint instruction and stop again (also notifying the main process). The main process can then retrieve the generated signal to determine whether a `SIGILL` signal was generated or not and take appropriate action, before continuing with the main loop.

The ptrace-based isolation method provides great isolation between the main *armshaker* process and the instruction execution environment, owing to the inter-process isolation implicitly provided by Linux. It also has the added benefit of making it easier to set and retrieve the state of the system before and after executing an instruction, which can help in analyzing the behavior of hidden instructions. However, the frequent use of `ptrace()` system calls adds significant overhead to the total execution time. Therefore, we recommend to use ptrace-based isolation primarily when page-based isolation has failed for a given system or when analyzing a subset of the instruction space.

3.2.5 Logging: If the execution of an instruction resulted in anything other than a `SIGILL` signal, the instruction is marked as hidden and subsequently logged. This is done in the final logging stage and is intended to convey both the presence of hidden instructions and to give an overview of the state changes caused by the execution of the hidden instructions.

The log uses a plain comma-separated values (CSV) format, with each line corresponding to a single instruction. The first entry in each line indicates the instruction, followed by the signal generated by executing the instruction. After that a variable number of entries follows, each representing a register value change in the state before and after executing the instruction.

4 METHODOLOGY

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

The full list of target systems is presented in Table 1. The first column lists the particular target systems, followed by the respective chipset manufacturers in the second column. The chipset manufacturer is relevant because it usually indicates the manufacturer of the processor, with potentially differing implementation details.

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

Table 1: Target systems.

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

Table 2: Hidden instructions detected in the target systems.

Finally, the last column presents the microarchitectures used in these systems.

Each of the target systems can be briefly described as follows. The Raspberry Pi 4 Model B and Orange Pi Lite 2 are single-board computers, essentially concentrating a computer into a single credit card-sized board. The Huawei P8 Lite and Motorola Moto G5S are both Android-based smartphones. The Oculus Quest is a head-mounted display (virtual reality headset) that runs Android and is very similar to a smartphone, the user interface notwithstanding. QEMU [2] is a commonly used open-source emulator capable of emulating a wide range of systems, many of which are Arm-based. In our particular tests, we use version 5.0.0 and emulate a Raspberry Pi 3, but the underlying Armv8-A implementation is shared across all emulated systems. Finally, we have the Arm Base Fixed Virtual Platform (FVP) which is a proprietary emulator developed by Arm, intended to be used for software development. There are various FVPs available emulating different processor models, but the one we use is the Base FVP that emulates a generic processor as opposed to a particular model.

For all the target systems, we use *armshaker* to perform an exhaustive search of all three instruction sets (A64, A32 and T32), except for the Motorola Moto G5S as its Android distribution runs on a 32-bit kernel, which prevents us from testing the A64 instruction set on this particular device.

5 EVALUATION

This section presents and analyzes the results obtained by probing the target systems. We begin by listing the number of hidden instructions marked for each system, followed by an analysis of these instructions. Finally, we discuss the security implications of the uncovered hidden instructions.

5.1 Hidden Instructions

The number of hidden instructions uncovered in each target system are presented in Table 2. As the table shows, hidden instructions were uncovered in all of the tested systems. However, note that a hidden instruction in Table 2 does not corresponds to an unique opcode, rather an unique instruction encoding which includes the opcode, operands, and all other fields[2]. Consequently, an instruction with variable operands or options may be marked multiple times.

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

[2]As hidden instructions are not defined in the ISA reference manual, there is no simple way to divide instruction encoding bits into opcode, operands, and other fields.

5.2 Hidden Instruction Analysis

Our analysis reveals that all hidden instructions in the hardware target systems can be attributed to bugs and backward compatibility measures in the Linux kernel. For QEMU, the hidden instructions are a result of bugs in its underlying Arm instruction decoder. As such, we did not find any hidden instruction in the hardware implementations themselves. Nonetheless, our results do show that hidden instructions, as perceived by the system user, can also be induced by the software running on the system and not only by the hardware.

5.2.1 Linux Bugs: Most of the hidden instructions can be attributed to three bugs and one backward compatibility measure in Linux. All of these bugs are related to Linux kernel's exception handling mechanism. To better understand the root cause of the bugs, we first present the exception handling mechanism of Linux kernel before discussing the bugs themselves.

Exception handling in Linux: When an instruction causes the processor to generate an exception, the execution is transferred to the kernel. Then, the Linux kernel compares the instruction that caused the exception (i.e. the instruction encoding) to a set of predefined instruction hooks. Each hook contains, among other things, a *mask* and a *value*. The *mask* is applied to the instruction before comparing it against the *value* in the hook. In the case of a match, the matched hook indicates how to handle the exception. If none of the hooks match, Linux sends a `SIGILL` (undefined/illegal instruction) signal to the offending process.

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

T32 SETEND Emulation: Linux can emulate the `SETEND` instruction, as it is deprecated in Armv8-A. The instruction sets the endianness of the system – indicating the byte-order used when operating on multi-byte variables. For T32, the encoding for SETEND is exclusively 16-bit wide and equals 0xb650 for the little-endian option and 0xb658 for big-endian. Due to this 16-bit encoding, the corresponding Linux hook uses a 0x0000fff7 mask to get the lower 16 bits from the instruction encoding of the exception-causing instruction. However, recall that T32 uses variable length instruction encoding with some instructions being 16-bit and others 32-bit wide. As a result, if a 32-bit wide unallocated instruction causes an exception, the SETEND hook will mask out the upper 16-bits; and if the lower 16-bits match the SETEND encoding, the unallocated instruction will be treated as a SETEND instruction.

As an example, consider the T32 instruction encoding 0xeaa0b650 which is a 32-bit instruction. However, looking up this particular encoding in the Armv8 Architecture Reference Manual reveals that it is unallocated. Nevertheless, Linux will execute it as a SETEND instruction, since applying the instruction mask yields 0xeaa0b650 & 0x0000fff7 = 0x0000b650 – which matches the instruction *value* in the hook. Effectively, the bug causes these unallocated 32-bits instructions to act as hidden SETEND instructions.

This incorrect matching accounts for all of the 2184 hidden instructions found in T32 for the Orange Pi Lite 2 and Oculus Quest. The reason that only these systems are affected and not the others is that SETEND emulation has to be enabled when compiling the kernel – or optionally setting the /proc/sys/abi/setend file to 1 during runtime – which was the case only for these two systems. The other systems were not affected simply because they didn't have SETEND emulation enabled.

To fix the bug, we modified the mask to 0xfffffff7, which makes it include the upper half of the instruction value. We submitted a patch with this fix to the Linux kernel which has been included in both the official mainline kernel and older versions with long-term support.

T32 Breakpoint Traps: The same problem as with the SETEND hook also affects certain breakpoint hooks. A breakpoint instruction, such as UDF #1 (encoding 0xde01), generates a SIGTRAP signal upon execution. The corresponding Linux hook uses a mask 0xffff which masks out the upper 16-bits of instruction encoding. As a result, the undefined 32-bit T32 instructions where the lower 16-bits are same as the encoding of breakpoint instructions, regardless of their upper 16-bits, act as hidden breakpoint instructions.

This bug is present only in the 32-bit Linux version (the 64-bit version uses a direct comparison without masking). It therefore only appears on the Motorola Moto G5S and Arm Base FVP where it accounts for 736 of the hidden instructions.

Similar to the SETEND bug, this bug is fixed by extending the instruction mask of the hook to 0xffffffff. We also submitted a patch with this fix to the Linux kernel and it has been accepted by the maintainers of the Arm port of Linux and included into the mainline kernel.

A32 Breakpoint Traps: The two bugs discussed above manifest themselves only on T32, while A32 and A64 remain unaffected as their Linux hooks use correct masks due to their exclusive 32-bit instruction encoding. However, a similar bug appears in the breakpoint hook for the UDF #16 instruction (encoding 0xe7f001f0) in A32. Specifically, the hook uses a mask 0x0fffffff which masks out the first four bits of the instruction encoding. These bits correspond to the conditional code in A32 and define the conditions that need to be true for the instruction to execute. Masking out these bits leads to unconditional execution of the instruction. However, the Armv8 Architecture Reference Manual explicitly states that UDF should be executed only if the condition code bits in the encoding are 0xe. For all other cases, the encodings are unallocated. In essence, the wrong mask causes these unallocated instructions to execute as breakpoint instructions. This bug affects both 64-bit and 32-bit kernels, and accounts for 15 of the hidden instructions in A32 in each of the tested systems.

However, a deeper inspection of the related parts in the Linux source code revealed that this behavior/bug is intentional, which was confirmed through correspondence on the Linux Kernel Mailing List [9]. The behavior is in fact a backward compatibility measure for Armv6 and earlier versions where the instruction encodings corresponding to conditional UDF instructions were legal. As such, due to differences in the specification of Armv6 and Armv8, this behaviour leads to hidden instructions in Armv8.

Uprobe Traps: Finally, there are two bugs in Linux related to a set of Uprobes hooks. Uprobes is a user-level tracing feature offered by Linux, with two UDF instructions used for breaking and single-stepping. The first bug is that in A32, these instructions have the same behavior as the breakpoint instructions above in that the condition code is ignored and some of the unallocated instructions act as breakpoints. This accounts for an additional 30 hidden instructions.

The second bug affects T32. A bug in the status register mask of the Uprobes hook makes the hook applicable to both T32 and A32. There are two problems with this. First, the Uprobes source code mentions that Thumb (T32) is not supported, therefore, it shouldn't be triggered from T32. And second, wrongly triggering the hook causes two unallocated T32 instructions to be treated as hidden tracing instructions.

This bug accounts for the two hidden instructions in T32 for the Arm Base FVP. Out of all the tested systems, it is only present in the Arm Base FVP simply because it was the only system with a kernel compiled with Uprobes support. We have submitted a patch with the fix to the Linux kernel that has yet to be reviewed.

5.2.2 QEMU Bugs: After removing the hidden instructions induced by Linux bugs, we end up with 69,632 hidden instructions in QEMU for both A32 and T32. We used *armshaker* to analyze the side-effects of these hidden instructions (like changes in register values) and correlated them with the ISA reference manual. Our analysis revealed that these hidden instructions correspond to the VMUL (floating-point) and VQDMULL instructions – both performing vector (SIMD) multiplication. Further analysis of the QEMU source code revealed that the root cause of the hidden instructions is a pair of bugs in the Arm instruction decode logic in QEMU. These bugs make the decoder treat certain bits in the instruction encoding as instruction *options*, while they should be a part of the opcode. This essentially reduces the opcode bits while increasing the option bits, which in turn makes certain unallocated instructions map to VMUL or VQDMULL instructions.

For the VMUL (floating-point) bug, consider the encoding in Figure 6. It's apparent that only bit 20 indicates the operand size option (sz). However, QEMU also includes bit 21 as a part of the

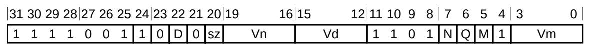

Figure 6: A32 encoding for floating-point `VMUL` [1].

size option, effectively executing the instruction even when bit 21 is 1, in which case it is unallocated.

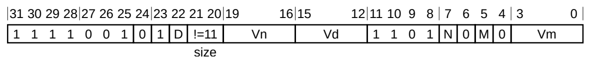

Figure 7: A32 encoding for `VQDMULL` instruction [1].

Likewise, consider the encoding in Figure 7 for the `VQDMULL` bug. For certain vector instructions, bit 24 is used for the `U` option, indicating whether the instruction operates on signed or unsigned numbers. For `VQDMULL`, however, this bit should always be 0, with 1 being unallocated. In spite of this, QEMU still executes the instruction as normal when `U` is 1.

These bugs account for the remaining hidden instructions identified in QEMU, and are present in both A32 and T32 as a result of the two sharing much of the same vector decode logic, with the main encoding difference being in the uppermost opcode bits. For `VMUL` the location of the bit triggering the bug is the same for both A32 and T32. For `VQDMULL` on the other hand, the bit triggering the bug is bit 28 (alternatively, bit 12 of the upper half-word) in T32 as opposed to bit 24 in A32, but the effect is otherwise identical.

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

5.3 Security Implications

The bugs in QEMU cause certain undefined instructions to be executed as `VMUL` or `VQDMULL`. Such hidden instructions can be used by programs to detect whether they are being emulated with QEMU. This can be done by checking whether executing a particular hidden instruction results in undefined instruction exception or not. Such emulator detection can be used to avoid dynamic analysis and selectively execute malicious code. For example, if the application detects the presence of an emulator it can decide not to execute malicious code; while doing so in emulator absence. The risk is exacerbated by the fact that QEMU is by far the most commonly used Arm emulator and forms the basis for several Android malware analysis solutions. Although there are many other ways to detect emulation – ranging from reading system information to detecting performance traits particular to emulators – executing only a single test instruction is significantly faster and less noisy. For a proof of concept of this technique using one of the hidden instructions identified, see the C program in Listing 1.

Similarly, the hidden `SETEND` instructions could be used as part of an obfuscation scheme where the endianness of a program is changed at runtime, using seemingly undefined instructions. This can make static analysis of the program more difficult.

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

volatile sig_atomic_t handler_activated = 0;

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

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

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

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

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

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

    return 0;
}
```

Listing 1: QEMU detection code.

6 CONCLUSION

As hardware becomes the new venue for system security vulnerabilities, we are witnessing a growing number of attacks exploiting well-known and documented hardware behaviour such as speculation, cache and memory timings, etc. We observed that the security exploits in undocumented hardware behaviour, such as hidden instructions, can have even more severe consequences as such behaviour is rarely verified and protected against. To prevent attacks, we need to first discover and understand such behavior.

To enable research in this direction, we introduced *armshaker*: a tool that can exhaustively search the instruction space of the A64, A32 and T32 instruction sets in the Armv8-A ISA for hidden instructions. We used *armshaker* to uncover hidden instructions in five different Armv8-A hardware platforms from four different vendors, as well as two Armv8-A emulators. Interestingly, none of the hidden instructions were due to bugs in physical hardware, but rather bugs in the system software. However, for the system user, the hidden instructions induced by system software bugs are indistinguishable from those induced by hardware bugs themselves.

Though *armshaker* did not find any hidden instruction in the hardware of the tested platforms, their existence cannot be ruled out completely, given the diversity of available Arm processors. Consequently, we make *armshaker* publicly available as open-source software to enable users to audit their own systems for hidden instructions.

REFERENCES

[1] Arm Limited 2019. *Arm® Architecture Reference Manual - Armv8, for Armv8-A architecture profile* (ddi0487e ed.). Arm Limited. https://static.docs.arm.com/ddi0487/ea/DDI0487E_a_armv8_arm.pdf

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

[3] Atri Bhattacharyya et al. 2019. SMoTherSpectre: Exploiting Speculative Execution through Port Contention. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 785–800. https://doi.org/10.1145/3319535.3363194

[4] G. Chen et al. 2019. SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS P)*. 142–157.

[5] Robert R Collins. 1997. The Intel Pentium F00F Bug Description and Workarounds. *Doctor Dobb's Journal* (1997).

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

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

[8] The Free Software Foundation (FSF). [n.d.]. GNU Binutils. https://www.gnu.org/software/binutils/

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

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

[11] Esmaeil Mohammadian Koruyeh et al. 2018. Spectre Returns! Speculation Attacks Using the Return Stack Buffer. In *Proceedings of the 12th USENIX Conference on Offensive Technologies* (Baltimore, MD, USA) *(WOOT'18)*. USENIX Association, USA, 3.

[12] Moritz Lipp et al. 2018. Meltdown. *arXiv preprint arXiv:1801.01207* (2018).

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

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

[15] Fredrik Strupe. 2020. armshaker: Processor fuzzer targeting the Armv8-A ISA. https://github.com/frestr/armshaker

[16] Fredrik Strupe. 2020. Probing the Armv8-A ISA for Hidden Instructions through Processor Fuzzing.

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