Security Analysis of x86 Processor Microcode

Daming D. Chen
Arizona State University
ddchen@asu.edu

Gail-Joon Ahn
Arizona State University
gahn@asu.edu

December 11, 2014

Abstract

Modern computer processors contain an embedded firmware known as microcode that controls decode and execution of x86 instructions. Despite being proprietary and relatively obscure, this microcode can be updated using binaries released by hardware manufacturers to correct processor logic flaws (errata). In this paper, we show that a malicious microcode update can potentially implement a new malicious instructions or alter the functionality of existing instructions, including processor-accelerated virtualization or cryptographic primitives. Not only is this attack vector capable of subverting all software-enforced security policies and access controls, but it also leaves behind no postmortem forensic evidence due to the volatile nature of write-only patch memory embedded within the processor. Although supervisor privileges (ring zero) are required to update processor microcode, this attack cannot be easily mitigated due to the implementation of microcode update functionality within processor silicon. Additionally, we reveal the microarchitecture and mechanism of microcode updates, present a security analysis of this attack vector, and provide some mitigation suggestions. A tool for parsing microcode updates has been made open source, in conjunction with a listing of our dataset[1].

1 Introduction

Since the 1970's, processor manufacturers have decoded the x86 instruction set architecture by internally decomposing x86 complex instruction set architecture (CISC) instructions into a sequence of simplified reduced instruction set computing (RISC) micro-operations (uops), in order to achieve greater performance and efficiency [1]. In doing so, microcode was introduced to help translate variable-length x86 instructions into a sequence of fixed-length micro-operations suitable for parallel execution by internal RISC execution units. By separating instruction decode and execution from the physical layout of processing logic, this new approach allowed for better implementation of multi-step CISC instructions, including optimization of the instruction execution sequence through techniques such as micro/macro-op fusion. Although this microcode was initially implemented on read-only memory, processor manufacturers soon introduced writable patch memory to provide an update mechanism for implementing dynamic debugging capabilities and correcting processor errata, especially after the infamous Pentium FDIV bug of 1994. The first known implementations of these microcode update mechanisms was with Intel's P6 (Pentium Pro) microarchitecture in 1995 [27], Advanced Micro Devices's (AMD's) K7 microarchitecture in 1999, and VIA's Nano in 2008 [2]. Perhaps ironically, AMD's K7 processors fails to properly validate the microcode patch RAM during built-in self-test (BIST), causing the microcode update mechanism itself to be listed as a processor errata [2]. Due to the volatile nature of this patch RAM, microcode updates do not persist after processor reset, although they are untouched by processor `INIT` [31]. As a result, microcode updates are typically integrated into the motherboard basic input/output system (BIOS), which is responsible for selecting the appropriate update and applying it during system power-on self-test (POST). However, since the motherboard BIOS is rarely updated by end-users or system administrators, most contemporary operating systems (e.g. Linux, Solaris,

[1] `https://www.github.com/ddcc/microparse`

Windows) also include update drivers to perform microcode updates during system startup using the same update mechanism. This mechanism is also accessible from within a virtualized environment, but should be filtered out by a well-designed hypervisor. On contemporary systems with symmetric multiprocessing (SMP), this mechanism should be executed synchronously on each logical processor (with the exception of Intel Hyper-Threading) to ensure that execution behavior is uniform.

2 Related Work

The basic principles behind this attack vector can be traced back to Ken Thompson's classic 1984 work, which proposed the concept of a malicious compiler undetectable even by source code analysis [42]. However, by utilizing malicious microcode updates, this attack vector can be extended to compromise processor hardware, severely impacting the security of existing computer systems.

Although hardware solutions have been developed to enforce trusted computing, such as trusted platform modules (TPM) and unified extensible firmware interface (UEFI) secure boot, the established chain of trust fails to account for security vulnerabilities within embedded firmware. In fact, these embedded vulnerabilities are much more common than one might think, as recent research has demonstrated the potential for malicious software to be embedded within network controllers [19], storage devices [17], and other peripherals [39]. At the same time, many of these trusted computing solutions have been shown to be themselves flawed, with demonstrated vulnerabilities within bootloaders, trusted platform modules, BIOS's [33], and even hardware-assisted trust solutions such as Intel Trusted Execution Technology [43].

To the best of our knowledge, no other published work has comprehensively analyzed processor microcode from a security perspective, likely due to the proprietary nature of processor microarchitecture and microcode. Although very little information is publicly available about the instruction encoding format of microcode and its operational mechanisms, implementation information is available within the *Intel® 64 and IA-32 Architectures Software Developer's Manual*, the *AMD® AMD64 Architecture Programmer's Manual*, and the *AMD® BIOS and Kernel Developer's Guide (BKDG)*. Production code implementing microcode update functionality is also provided by the open-source Linux kernel and Coreboot projects. In addition, certain architectural details are available in industry patent filings.

Nevertheless, there has been some high-level analysis of processor microcode. A basic analysis of the metadata accompanying Intel microcode updates was published by Molina and Arbaugh in 2000, determining the purpose of certain fields within the microcode update header [35]. Likewise, an anonymous report published in 2004 provided similar information about AMD microcode updates [5]. More recently, a technical report published by Hawkes in 2013 discovered the presence of additional metadata within the Intel microcode update binary, suggesting that recent Intel microcode updates are cryptographically verified using a RSA signature with a non-standard SHA hash algorithm [26].

3 Microarchitecture

Individual instructions within the x86 instruction set architecture can range from anywhere between one to fifteen bytes, although the general encoding format remains constant. Instructions consist of a one or two byte operation code (opcode), a register or memory operand byte (modR/M), a scale-index-byte addressing (SIB) byte, and multiple displacement and/or immediate bytes. In addition, instructions can also be prepended by prefix bytes that denote special repetition or memory addressing behavior, such as that performed by the `REP` or `LOCK` instruction prefixes.

During each instruction cycle, the processor fetches blocks of instructions from system memory, which are then segmented and stored within L1 instruction cache (trace cache). This step identifies and tags instruction boundaries, and also provides additional hints for branch prediction and instruction execution. Next, instructions are decoded from the cache and placed into dedicated issue positions at reservation stations for register renaming, then finally dispatched to functional units before retiring. On modern superscalar

Listing 1: Implementation for MOVS in AMD processors

```
    LDDF ;load direction flag to latch in functional unit
    OR ecx, ecx ;test if ECX is zero
    JZ end ;terminate string move if ECX is zero
loop:
    MOVFM- tmp0, [esi] ;move to tmp data from source and inc/dec ESI
    MOVTM- [edi], tmp0 ;move the data to destination and inc/dec EDI
    DECXJNZ loop ;dec ECX and repeat until zero
end:
    EXIT
```

processors, these steps do not necessarily occur sequentially, as concurrent dispatch and out of order execution allow for pipeline optimizations to maximize throughput.

Simple instructions are directly decoded into a short sequence of fixed-length RISC-like operations (also known as ROPs, uops, or Cops) by hardware, whereas complex instructions are decoded by microcode ROM. Examples of the former include `ADD`, `XOR`, and `JMP`, while the latter include `MOVS`, `REP`, and `CPUID`. For a more complete list, see [21].

As can be seen from the published `MOVS` microcode implementation for AMD processors (listing 1), microcode instructions (microinstructions) are the basic arithmetic, data, and control operations that compose regular x86 instructions [34]. A published multiway branch implementation of the `RDMSR` and `RDTSC` instructions shows the same to be true for Intel processors [25].

3.1 Capabilities

Over time, microcode has become responsible for handling more and more internal processor operations. Originally, it was primarily used to handle illegal opcodes or complex x86 instructions, such as floating-point operations, MMX primitives [32], and string move using the `REP` prefix [28]. More recently, it has been used to implement instruction set extensions such as AVX [30] and VT-d by handling the special virtual machine primitives `VMREAD`, `VMWRITE`, and `VMPTRLD` [4]. In addition, it is also responsible for saving processor state, managing cache operation, and handling interrupts with respect to C-states (power saving) and P-states (voltage/frequency operating point), e.g. flushing the L2 cache upon entry into or exit from C4 state [3] [37].

On newer Intel processors, including the Nehalem microarchitecture, processor microcode has been further enhanced to incorporate breakpoint functionality, allowing the microcode to intercept and modify requests made to platform devices. Not only is this triggering functionality capable of capturing short amounts of data or pulsing an externally-visible pin, but it can also send information about internal control flows and transactions during execution, such as exposing accesses to machine state registers, results of I/O operations, and page fault information (including address) [20].

3.2 AMD

On AMD processors, instructions are categorized into one of two decode pathways: fastpath and microcode ROM (MROM) [23] [10].[2] In order to determine the address of the microcode entries in MROM that correspond to an MROM instruction, the MROM entry point unit utilizes the opcode, mod-R/M, and prefix bytes of the x86 instruction to generate the appropriate entry point microaddress. Note, that this microaddress does not necessarily correspond to a single microcode instruction, but rather to a line of ROPs in MROM (also known as microcode instructions), where the number of microcode instructions is equal to the number of functional units within the processor. Because certain microarchitectures have three functional

[2] There also exists a third pathway consisting of mixed fastpath and MROM instructions known as double dispatch that is used to maximize dispatch bandwidth.

units per logical processor, a line of microcode instructions in MROM can also be referred to as a triad. Below is a diagram showing the layout of a microcode triad (table 1).

Field	Size
Operation 1	8
Operation 2	8
Operation 3	8
Sequence Control	4

Table 1: Structure of microcode triad

Not all MROM instructions can be implemented by a single line of microcode instructions, so an additional sequence control field is appended to each triad in order to determine the microaddress of the next triad. Usually, this corresponds to the microaddress of the next triad, but is not necessarily true for microcode instructions that alter the microcode control flow, such as branching or jumping instructions. In addition, if the next triad is the last line of microcode for a respective MROM instruction, then sequence control is responsible for encoding an early exit signal that notifies selection control to pack additional fastpath instructions into vacant issue positions for execution during the current clock cycle. As a result, this sequence control field is used to store information related to branch prediction, control flow operations, and early exit signals.

Since the MROM is read-only and utilizes a fixed mapping from MROM instructions to microcode instructions, microcode ROM cannot be directly modified after manufacture to address unintentional bugs in microcode or implement new functionality for debugging. As a result, a microcode patch RAM is attached to the MROM to allow for modifications to existing microcode instructions. The memory space of this combined microcode storage is typically from `000h` to `C3Fh`, with the lower 3072 triads from `000h` to `BFFh` mapped to microcode ROM, and the upper 64 triads from `C00h` to `C3Fh` mapped to patch RAM. Internally, this may be implemented using two pairs of flash memory [16]. In addition, eight match registers with functionality similar to breakpoints are added to the processor, and can be set by a microcode update (also known as a microcode patch).

During execution of microcode instructions, if the current microaddress matches that of an address stored in a match register, execution jumps to a fixed offset in microcode patch RAM to execute the patch. These fixed offsets are shown in the jump table for each match register (table 2) [34]. To disable a match register, it is simply set to an address outside of the microcode memory space, e.g. `FFFFFFFFh` (-1), which will never match.

Match Register	RAM Offset
Match 1	00h
Match 2	02h
Match 3	04h
Match 4	06h
Match 5	08h
Match 6	0ah
Match 7	0ch
Match 8	0eh

Table 2: Microcode entry point jump table

3.3 Intel

Although the microarchitecture of Intel processors is not as well publicly documented, overall it appears to be quite similar. Regular x86 instructions (also known as macroinstructions) can be decoded either by

hardware or by the MROM, which issues a sequence of preprogrammed uops to complete the operation [28]. Hardware instructions are generally of three uops or shorter, whereas MROM instructions are either longer than four uops or not encodable within the trace cache [8].

Internally, there also exists a small patch RAM in addition to the MROM, which may be implemented by attaching a separate memory to the microcode ROM [15]. We believe that this memory space is also contiguous or otherwise cross-addressable in order to facilitate jumps from patch RAM to MROM. On the P6 microarchitecture, the patch RAM is capable of holding up to 60 microinstructions, with patching implemented by pairs of match and destination registers. When the current microaddress matches the contents of a match register, execution continues at the associated destination register, instead of the fixed offsets used in AMD microprocessors [29].

4 Microcode Updates

4.1 Update Structure

Since microcode updates are specific to the microarchitecture of a processor, an identifying processor signature value is used to determine compatibility. This signature is a 32-bit integer that encodes the stepping, model, family, type, extended model, and extended family information of the processor, and can be obtained in software by setting the `EAX` register to 1, executing the `CPUID` instruction, and then reading back the contents of the `EAX` register. As such, this value is also sometimes known as simply the CPUID.

Due to the challenge of distributing microcode updates individually by CPUID, processor manufacturers instead distribute update packages, from which the microcode update driver is responsible for selecting and loading the correct update. These update packages can be found on the websites of each processor manufacturer[3][4]. Since this format differs for each, they will be treated separately.

4.1.1 AMD

There exist three varieties of AMD microcode update packages, with one targeted for the Solaris operating system[5], and two targeted for the Linux operating system on 15h and non-15h microarchitectures, respectively. All of these packages are in little-endian binary format with a short header (table 3), followed by a table mapping from processor signatures to processor revision ID's (table 4), which eliminates duplicate microcode updates used by multiple processors from the same microarchitecture but with different processor signatures. Then, each individual microcode update is prepended by a short header (table 5) that specifies the size of the following microcode update, allowing the update driver to easily iterate through the microcode update package.

Field	Size	Value (Typical)
Magic Number	4	"AMD\0"
Table Type	4	0h
Table Size	4	Varies

Table 3: Structure of microcode update package header

Each microcode update consists of a header (table 6) followed by patch data. On newer microarchitectures, the patch data and certain metadata fields is observed to be encrypted. In addition to the match register fields discussed earlier, the header also contains an initialization flag field that specifies whether microcode instructions located at the fixed offset `10h` should be immediately executed after a microcode update applied. This is used to correct processor errata not directly caused by a miscoded instruction, possibly by

[3] AMD: `http://www.amd64.org/microcode.html`

[4] Intel: `https://downloadcenter.intel.com/Detail_Desc.aspx?agr=Y&ProdId=3425&DwnldID=22508`

[5] These were not analyzed.

Field	Size	Value (Typical)
Processor Signature	4	Varies
Errata Mask	4	0h
Errata Compare	4	0h
Processor Revision ID	2	Varies
Unknown	2	0h

Table 4: Structure of microcode update package table entry

Field	Size	Value (Typical)
Type	4	1h
Update Size	4	Varies

Table 5: Structure of header prepended to each microcode update

modifying an internal configuration register to disable optimizations that result in incorrect cache handling or power management behavior.

Field	Size
Date	4
Patch ID	4
Patch Data ID	2
Patch Data Length	1
Initialization Flag	1
Patch Data Checksum	4
Northbridge Device ID	4
Southbridge Device ID	4
Processor Revision ID	2
Northbridge Revision	1
Southbridge Revision	1
BIOS API Revision	1
Reserved	3
Match Register 1	4
Match Register 2	4
Match Register 3	4
Match Register 4	4
Match Register 5	4
Match Register 6	4
Match Register 7	4
Match Register 8	4

Table 6: Structure of AMD microcode update header

Other headers fields include the northbridge, southbridge, and BIOS API revision and/or ID fields, which appear to provide a mechanism for restricting microcode updates to specific combinations of processors and platform hardware, but have not been observed to actually be used (or implemented within the Linux update driver). In addition, there exists a patch length field that specifies the number of lines of patch data, and a checksum field that is calculated by taking the sum of the patch data as a sequence of 32-bit integers. Furthermore, a processor revision ID field mapped from the processor signature is used to ensure that a microcode update is being loaded onto the correct processor microarchitecture, a patch ID field is used to specify the microcode update revision, and a patch data ID field is used to verify compatibility of the

microcode patch format with the internal microcode patch mechanism.

4.1.2 Intel

In contrast to the binary format of AMD microcode update packages, Intel microcode update packages are distributed in plain-text form, with each microcode update is represented as rows of 32-bit big-endian integers in four hexadecimal columns separated by commas. C-style comments are used to denote non-microcode content, such as original filenames or dates. However, there does exist some older microcode updates that are distributed individually as binary files in little-endian format. It is believed that this reflects the format in which Intel distributes individual microcode updates directly to industry partners.

Each individual microcode update consists of a header (table 7) followed by patch data. Although official documentations makes reference to an extended update format with an optional extended signature table section [31], to the best of our knowledge this extended format has never been publicly used. In addition, the patch data has always been observed to be encrypted or otherwise obfuscated.

Field	Size
Header Version	4
Update Revision	4
Date	4
Processor Signature	4
Checksum	4
Loader Revision	4
Processor Flags	4
Data Size	4
Total Size	4
Reserved	12

Table 7: Structure of Intel microcode update header

Instead of using something similar to AMD's processor revision ID field, a processor signature is directly stored in the microcode update header to determine compatibility between an individual microcode update and the current processor. In addition, a processor flags field is used to further differentiate between multiple processors with the same processor signature. Compatibility is verified by left shifting the value `1h` by the 3-bit platform ID stored in bits 50 - 52 of MSR `00000017h`, then computing the bitwise AND of this value with the processor flags field of the update header, and checking if the result is nonzero. There also exists an update revision field that specifies the revision of the microcode update.

In addition, microcode updates for newer processors belonging to the Atom, Nehalem, and Sandy Bridge microarchitectures contain an additional undocumented header within the update data block (table 8). Previous reverse engineering has determined that this header includes additional date, update revision, update length, and processor signature fields, as well as a 520 byte block containing a 2048-bit RSA modulus that appears to be constant within each processor family. This is followed by a four byte RSA exponent with the fixed value `11h`, as well as a RSA signature computed using SHA-1 or SHA-2 hash algorithm [26]. This information corresponds with that published in other sources, which indicate that a SHA-1 hash digest may be generated after the patch data is encrypted using a symmetric block cipher such as AES or DES [40].

4.2 Update Mechanism

The microcode update mechanism is very similar across all x86 processor manufacturers, primarily by using processor model-specific register (MSR) registers to read the current microcode revision and write the new microcode update. Below are the appropriate MSR registers for each (table 9).

Following is the general microcode update process, with integrity and verification checks omitted.

Field	Size	Value (Typical)
Unknown 1	4	0h
Magic Number	4	a1h
Unknown 3	4	20001h
Update Revision	4	Varies
Unknown 4	4	Varies
Unknown 5	4	Varies
Date	4	Varies
Update Length	4	Varies
Unknown 6	4	1h
Processor Signature	4	0h
Unknown 7	56	0h
Unknown 8	16	Varies
RSA Modulus	256	Varies across processor family
RSA Exponent	4	11h
RSA Signature	256	Varies

Table 8: Structure of undocumented additional Intel microcode update header

Manufacturer	Revision	Update	Status
AMD	8bh	c0010020h	N/A
Intel	8bh	79h	N/A
VIA	8bh	79h	1205h

Table 9: Microcode update MSR registers

1. Clear `EAX`,read the current processor signature using `CPUID`, and load the matching microcode update into kernel memory. On Intel processors, also check that the processor flags field matches.
2. Clear `EAX` and `EBX`, and read the current microcode revision using `RDMSR` from the revision MSR.
3. Write the memory address of the microcode update using `WRMSR` to the update MSR. On Intel processors, also perform `CPUID` with `EAX` = 1 to synchronize each logical processor.
4. Read the new microcode revision, and return success if it matches that of the update. Otherwise, return failure.

4.3 Update Driver

4.3.1 Linux

Microcode updates on Linux are performed by the in-tree `microcode` kernel module, which supports both AMD and Intel processors. For AMD processors, note that only microarchitectures 10h and later are supported, even though the update mechanism is the same on the earlier K8 and 0fh microarchitectures.

When the module is first loaded, microcode updates for the system are automatically loaded from either the `amd-ucode` or `intel-ucode` directories within the local Linux firmware repository, e.g. `/lib/firmware/`. Typically, this process occurs during kernel initialization, since the update module and microcode update files can be integrated into the boot *initramfs* image.

While AMD microcode packages can be automatically parsed and loaded by this module, Intel microcode packages need to first be processed by the usermode `iucode-tool` (previously `microcode.ctl`), which extracts the appropriate microcode updates for the installed processor(s) from the update package, converts them into binary format, and places them in the firmware directory with the correct naming convention.

Once the module has been loaded, updates can also be triggered via *sysfs* at `/sys/devices/system/cpu/microcode/reload`. The current microcode revision and processor flags are also exported to `processor_flags` and `flags` for each logical processor at `/sys/devices/system/cpu<number>/microcode/`.

4.3.2 Windows

Although less well documented, microcode updates are performed by bundled device drivers on Windows XP and later. Unlike the Linux update module, these Windows drivers have binary microcode updates integrated within the `.data` or `PAGELK` segments, and cannot load microcode from manufacturer-supplied update packages. After a microcode update is successfully loaded, these drivers update processor configuration values stored in the registry[6].

Below are the microcode update driver versions bundled within recent versions of the Windows operating system (listing 10).

OS	Filename	Version	Date
Windows XP (SP3)	`update.sys` (Intel only)	5.1.2600.5512	2008-04-14
Windows 7 (SP1)	`mcupdate_AuthenticAMD.dll`	6.1.7600.16385	2009-07-13
Windows 7 (SP1)	`mcupdate_GenuineIntel.dll`	6.1.7601.17514	2010-11-20
Windows 8.1	`mcupdate_AuthenticAMD.dll`	6.3.9600.16384	2013-08-22
Windows 8.1	`mcupdate_GenuineIntel.dll`	6.3.9600.16384	2013-08-22

Table 10: Windows microcode update driver versions

Specifically, we note that very few AMD microcode updates are bundled within recent versions of the Windows operating system, and none at all with Windows XP. For example, Windows 7 (SP1) includes only three AMD microcode updates (table 11). In addition, the bundled microcode updates do not appear to have been regularly updated to align with new update packages released by AMD. However, significantly more Intel microcode updates are included (table 14).

Date	Processor Revision	Patch ID	Checksum
2008-03-06	00002031h	02000032h	8ff3faeah
2008-04-30	00001022h	01000083h	074388a8h
2008-05-01	00001020h	01000084h	1fcc8590h

Table 11: Microcode bundled within `mcupdate_AuthenticAMD.dll`

5 Methodology

An overview of our methodology is shown in figure 1.

As a preliminary step, we begun by conducting a literature review of published research analyzing processor microcode. However, since this subject is proprietary and relatively undocumented, we were forced to expand the scope of our search to include alternative sources of documentation. This encompassed industry patents assigned to Advanced Micro Devices and Intel Corporation published both domestically and abroad, and technical documentation published by both vendors, as well as source code from the Linux kernel and Coreboot projects that implement support for microcode updates.

Due to the presence of microcode encryption and integrity verification mechanisms, we began by searching the web for preexisting microcode updates to analyze. Targeted search engine queries allowed us to locate microcode updates publicly available on the websites of processor manufacturers, as well as those hosted by third party developers or firmware modders. Since many of these sites are no longer online, historical

[6] `HKEY_LOCAL_MACHINE\HARDWARE\DESCRIPTION\System\CentralProcessor`

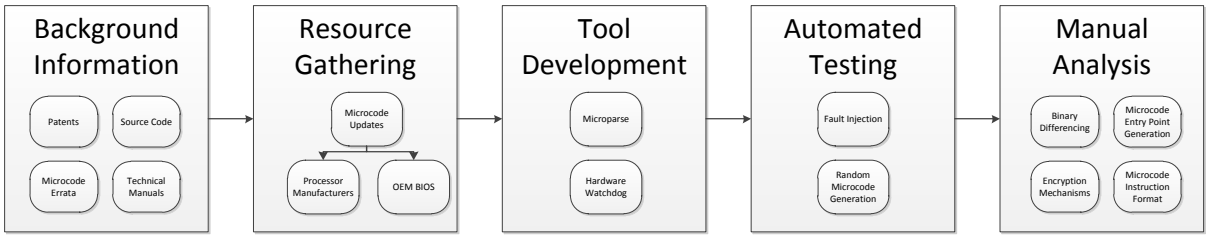

Figure 1: Overall analysis methodology

databases such as the Internet Archive were helpful in locating microcode no longer directly available. Binary microcode updates were also committed into the source code repositories for various open source projects such as Linux and Coreboot, which were useful for extraction and analysis.

Next, we developed a custom tool written in Python, named microparse, to interpret and modify the structure of individual microcode updates and microcode update packages, which we used to extract and catalog the published updates that had been gathered. This tool has been made open source, and is available online together with catalog data[7]. After determining particular microarchitectures of interest, we looked for documentation on processor errata to determine the changes made in each microcode update revision, which were then compared against each other to determine the structure of the binary microcode patch.

In many cases, these updates appeared to be encrypted due to significant differences in the binary patch data between revisions, with the few similarities that occurred a statistical result of the birthday paradox. At this point, we were able to apply fault injection techniques and analyze the differences in timing to characterize the encryption mechanism being utilized. This was achieved by modifying the Linux microcode update module to time the update using the `RDTSC` instruction, which returns the value of the time stamp counter (TSC), a 64-bit register that records the number of processor cycles performed since reset. In order to ensure that this time is accurate, we execute and store the values of `RDTSC` twice before performing the update, then once after the update, allowing us to subtract the time required by `RDTSC` from the final result [25]. A usermode script configured to run at system startup was used to permute the microcode update, perform the update, record the result, and then restart the system. This sequence of steps is shown below:

1. Boot system with initial microcode revision.
2. Automatically load modified `microcode` kernel module.
3. Start script `microcode_fault.py`.
   (a) Reset hardware watchdog timer.
   (b) Check if finished, otherwise increment the try counter for current bit offset. If at or beyond the counter threshold, reset the try counter and increment the current bit offset.
   (c) Send current status to remote webserver.
   (d) Create a modified microcode update file.
   (e) Read current microcode revision.
   (f) Trigger kernel microcode update.
4. Read modified microcode update file from kernel module.
   (a) Record time stamp counter.

[7] `https://www.github.com/ddcc/microparse`

(b) Record time stamp counter.

(c) Perform microcode update.

(d) Record time stamp counter.

(e) Compute difference between first two and last two readings, then output the difference of these two computations.

5. Parse system log for output difference from script.

(a) If successful, write the result (previous microcode revision, current microcode revision, and cycle difference) to file, and set the current try counter to max. Otherwise, store the kernel log.

(b) Delete the modified microcode update file.

(c) Restart the system.

For the older AMD updates that were not encrypted, we were able to apply frequency analysis to obtain an overall idea of the structure of the microcode updates, which was then combined with background knowledge to determine specific information about the structure of each microcode line. Further fault injection testing was also performed to determine the function of other structures that were not documented. This was achieved by performing bruteforce testing on certain fields of the microcode update; for example, filling the update with invalid data, setting the match registers to a microaddress, and then executing a specific instruction to determine if the system crashes. Reverse engineering of processor microarchitecture and microcode updates was necessary in order to determine how the microcode update mechanism within the processor functioned. Due to a lack of public documentation on the operation of this mechanism, careful analysis of technical documentation and industry patent filings was required to determine the location and capabilities of microcode in the instruction decode step of processor code execution. Fault injection testing was used to determine the scope and mechanism of microcode integrity verification mechanisms, including encryption and/or obfuscation.

Since this process could result in system instability, we designed a hardware watchdog timer communicating over USB using a FTDI FT232R and an Atmel ATtiny2313a. If the system failed to reset the watchdog timer within a preset time interval, then the watchdog would cause a solid state relay to trigger the motherboard power or reset switch input. Below is a schematic of the design (figure 2).

We also attempted to develop our own microcode updates in order to identify the format of the internal microcode instruction set, with the overall goal of developing a proof of concept malicious microcode update. This work is still ongoing.

Testing was performed on the following systems (table 12), which include processors manufactured by both AMD and Intel. The software used for testing was a standard distribution of Ubuntu 12.10, albeit with a modified Linux 3.8.13 kernel.

Manufacturer	**Architecture**	**Processor**	**CPUID**
AMD	K8	Athlon 64 X2 4800+	60fb2h
AMD	10h	Phenom X3 8650	100f22h
AMD	10h	Phenom II X6 1045T	100fa0h
AMD	12h	A8-3850 X4	300f10h
AMD	15h	FX-4100	600f12h
Intel	P6	Pentium II 233 (80522)	634h
Intel	Sandy Bridge	Core i3-330M	20652h
Intel	Sandy Bridge	Core i5-2500k	206a7h

Table 12: Listing of microprocessors that were tested

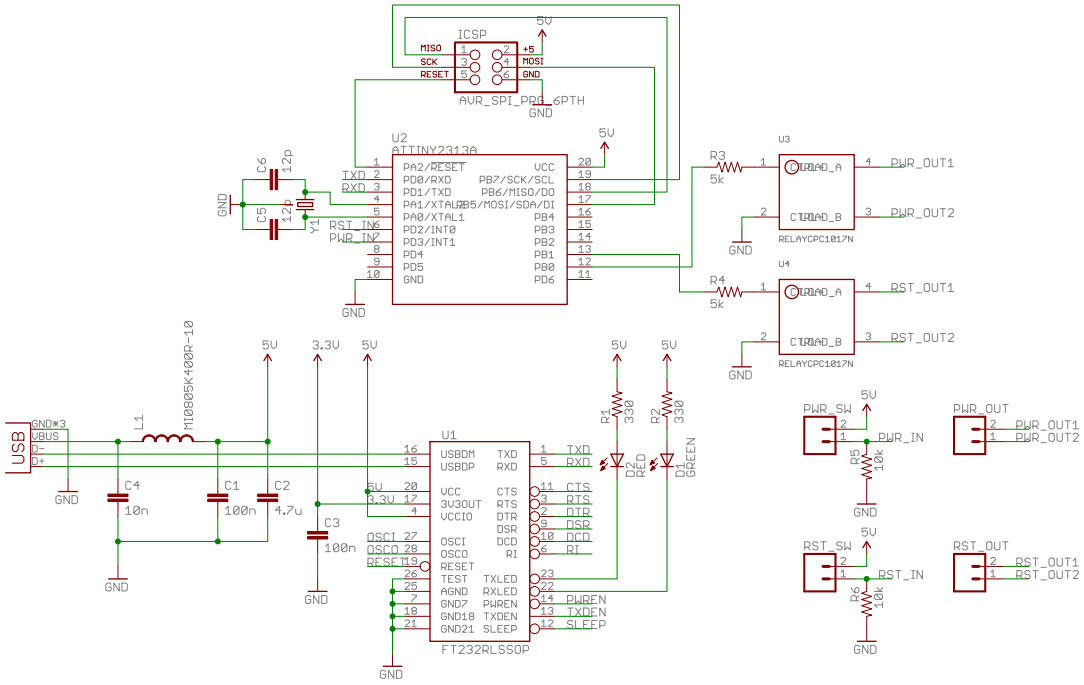

Figure 2: Schematic of watchdog timer

6 Results

6.1 AMD

Due to the lack of publicly released AMD microcode updates, we were able to only gather a dataset of 44 unique microcode updates. These updates date from February 6, 2004 through March 3, 2013, and span the K8 (2003) through 15h (2013) microarchitectures. Note that no microcode updates appear to be publicly available for the 16h (2014) microarchitecture. Using the processor signature to processor revision ID mapping tables, we observed a number of processor signatures that do not exist in any public CPU databases[8], indicating that they likely correspond to internal testing or engineering sample processors. Strangely enough, we also observe one microcode update with an invalid date within the date metadata field, likely caused by a manual error during the release process.

In particular, we observe four distinct categories of microcode updates, based on the value of the patch data ID metadata field. The corresponding characteristics are summarized in table 13.

Based on our fault injection testing results, only the checksum, patch data ID, patch data length, update revision, and processor revision ID metadata fields are parsed for microcode updates with patch data ID `8000h` and `8003h`. Updates with patch data ID `8003h` appear to support wildcard matching using the latter

[8] e.g. `http://www.cpu-world.com`, `http://www.etallen.com/cpuid.html`

Patch Data ID	Microarchitecture	Encryption	Memory Space
8000h	K8, 0fh, 10h, 11h	N	000h - C3Fh
8001h	14h	Y	?
8002h	15h	Y	?
8003h	12h	N	00000000h - FFFFFFFFh

Table 13: Listing of AMD patch data id's and microarchitectures

16-bits of the match register, possibly due to the presence of an on-die graphics engine. In contrast, all fields are ignored for microcode updates with patch data ID 8001h, except for the three unknown fields that remain set to aah, and the encrypted match register fields. The same is true for microcode updates with patch data ID 8002h, except that the three unknown fields are now set to zero.

Fault injection tests on the 15h microarchitecture clearly demonstrate the presence of encryption (figure 3), as the updates must first be decrypted before metadata fields can be verified. These results show that the microcode update mechanism takes on average 753913 cycles, with a sample standard deviation of 114841 of cycles. Note that bit offsets 480 through 495 correspond to the patch data ID field, with modifications to this field resulting in early termination with an average of 628 cycles, likely due to the modified update no longer appearing compatible with the internal microcode update mechanism. Bit offsets 672 through 2719 correspond to the match registers at the end of the header through the binary patch data. Changes to this segment also result in early termination, but with a longer average of 428864 cycles. Finally, changes to bit 3600 increased the length of time taken by the microcode update mechanism to an average of 4022609 cycles, suggesting that this unidentified value could correspond to some sort of length field or additional operation flag. In comparison, performing the update with an unmodified microcode update takes on average 20916597 cycles.

In contrast, fault injection tests on the 12h microarchitecture indicates that encryption is not present (figure 4). On average, the microcode update mechanism takes 1116 cycles, with a sample standard deviation of 132 cycles. Although modifications to certain microcode metadata fields (e.g. patch ID, patch data ID) result in early termination, this is likely due to the modified update no longer appearing compatible with the internal microcode update mechanism. However, modifications to the patch data length field indicate that the value of this field directly correlates to the number of cycles, indicating that the microcode update mechanism utilizes this field to determine the amount of patch data to read from system memory.

6.1.1 Encryption

On the 15h microarchitecture, binary comparison of multiple microcode revisions for a single processor produces no useful similarities, indicating that a chained block or stream cipher is likely being used for encryption.

Since changes to offsets 672 through 2719, measuring exactly 2048 bits in size, result in early termination, it is possible that this segment contains a hash that is used to verify the integrity of the encrypted update data, in an encrypt-then-MAC approach. Although it is possible to use the reverse, MAC-then-encrypt, this is rather unlikely, as not only is it less secure, but 2048 bits is also rather long for the block size of a block cipher.

6.1.2 Linux Update Driver

During our fault injection testing, we encountered a number of segmentation faults caused by the Linux microcode loader that could potentially be problematic. In particular, the microcode loader utilizes the length fields within the microcode update package to iterate through data or dynamically allocate memory.

Due to the fact that individual updates within update packages are prepended by a short header that specifies its total size, and that the microcode module uses this value as a pointer offset to search through a update package, an erroneous length can trigger invalid kernel paging requests by iterating beyond the

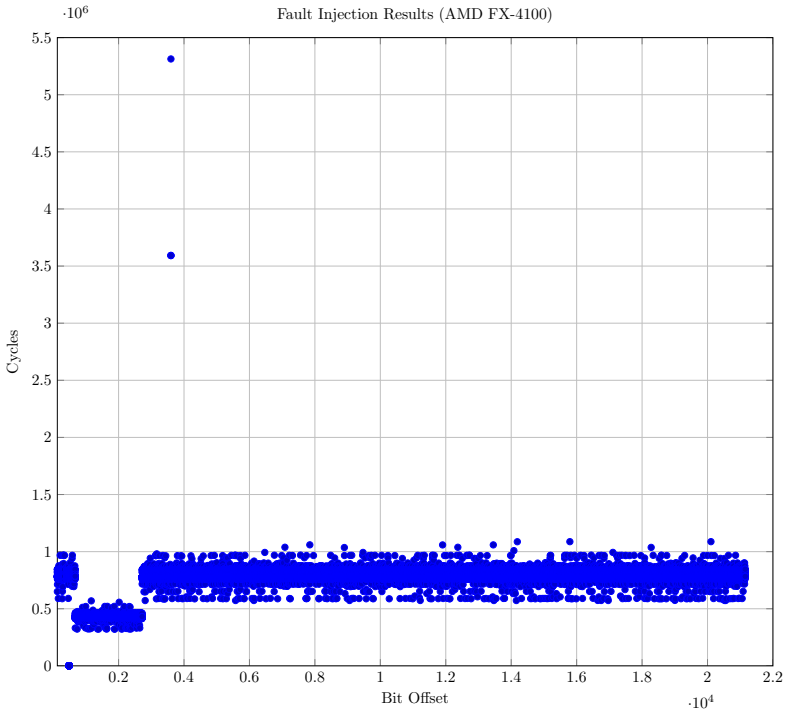

Figure 3: AMD 15h microarchitecture test results

allocated buffer. Similar problems also exist with the the size field within the initial header of the mapping table. Adding a check to ensure that these offsets do not exceed the buffer size would resolve this problem.

More bizarre, however, is the fact that the microcode module caches updates internally, with eviction occurring only when a newer revision patch is applied. As a result, if a user attempts to apply an invalid update, whether deliberately modified or accidentally corrupted, it will be internally cached by the microcode module, regardless of whether the update applies successfully. Later attempts to perform a microcode update with the correct update but of the same revision will then fail to apply, since the version from the internal cache is preferred. Also note that the microcode loader will refuse to load an update with the same revision as the currently installed version. As a workaround, the internal cache can be cleared by unloading and reloading the microcode update module, or restarting the system. This issue could be fixed by clearing the cache if an update fails to apply, or always overwriting the internal cache if the revisions are equal.

Furthermore, only the microcode update attempt for the first logical processor will actually read the microcode update package from the filesystem, and overwrite the internal microcode cache, if appropriate. Microcode update attempts performed on other logical processors will always read from the internal cache and fail if the cache is empty. This leads to unexpected behavior; in a scenario where a microcode update is triggered, the update package is removed from the filesystem, and then the update is triggered again, all processors but the first will be updated from the internal cache, since the first will attempt to read from

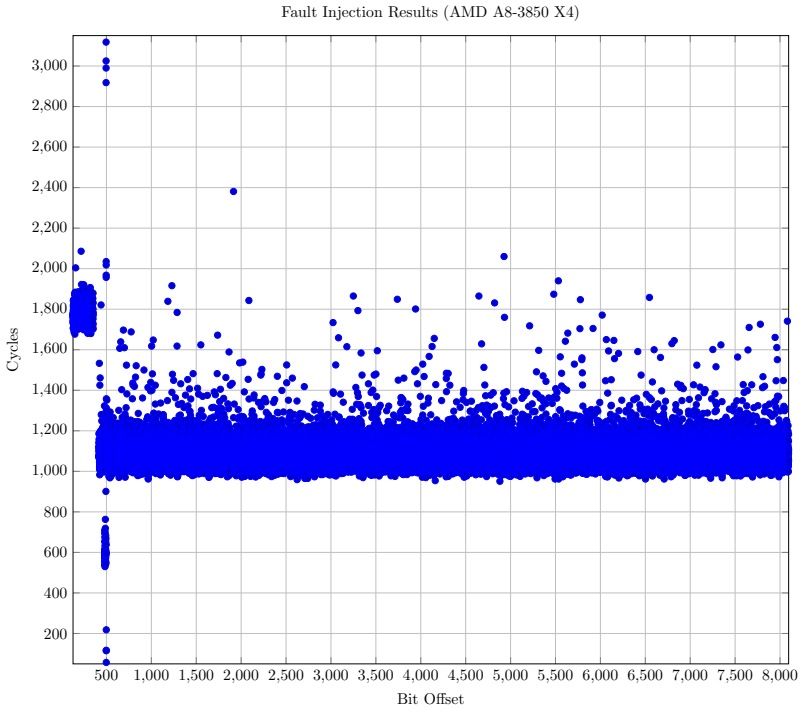

Figure 4: AMD 12h microarchitecture test results

the filesystem and terminate, while the others will use the internal cache. This could be fixed by aborting a microcode update if any update attempt for a logical processor fails, instead of continuing regardless.

6.1.3 Microcode

Although we are still reverse engineering the format of the microcode instructions, we are able to observe that a large number of sequence control and instructions are shared between individual microcode updates, indicating that some blocks of microcode are used in multiple processors. All observed microcode updates also ended with the `8001EFFFh` or `8003DFFFh` sequence control values.

Nevertheless, we were able to develop garbage microcode updates for a K7 microarchitecture processor, which lacks published microcode due to the functionality being listed as an errata. By setting the initialization flag and filling the data block of the microcode update with values such as `00000000h` or `FFFFFFFFh`, performing a microcode update will cause nondeterministic hangs, likely due to the lack of an exit signal sequence control value causing the processor to continue executing microcode beyond the patch RAM. This unpredictable behavior can manifest as hanging of the usermode shell and kernel thread responsible for performing the microcode update, or even a complete lockup of the entire system requiring a hard reboot.

6.2 Intel

We were able to gather a dataset of 498 unique microcode updates for Intel processors, dating from January 25th, 1995 through September 14, 2013. These span the P6 (1995) through Haswell (2013) microarchitectures, and can be categorized into three broad categories based on the target microarchitecture, roughly corresponding to the P6/Core (Pentium through Pentium III, Core 2), Netburst (Pentium 4, Pentium D), and Atom/Nehalem/Sandy Bridge (Core i3/i5/i7) processor microarchitectures. All of these updates are encrypted or otherwise obfuscated, although the mechanism has clearly changed over time.

Examination of the metadata yields similar results to that of AMD microcode updates, including a number of updates that correspond to processors with CPUIDs that do not exist in any online processor databases. These are likely microcode updates for internal testing or engineering sample processors for which the hardware has never been publicly released, but the microcode has somehow leaked. One microcode update also contains an invalid date, likely occurring manually during the process of packaging microcode updates for public release.

The first and earliest category of microcode updates target the P6 and early Core microarchitectures, which include the Pentium and Core 2 families except the Pentium 4 and Pentium D. These updates contain a data block of 2000 or 4048 bytes respectively, with the former typically differing by a 864 or 944 byte block and the latter differing by a 3096 byte block, both at a constant offset when compared against other revisions with the same processor signature. A significant number of updates also share common patch data blocks of sizes 1056 at offset 3e0h (less commonly at 390h or 398h for the early Pentium Pros), or 952 at offset 3e0h and 104 at offset 798h, and 236 at c68h. This may indicate the presence of shared loader or exit handler code, or alternatively common patches for the same processor errata.

The next category of Intel microcode updates target the the Netburst microarchitecture (Pentium 4, Pentium D, Celeron), and contain a data block size of 2000 through 7120 bytes, in increments of 1024. This data block appears to be unique and shows no significant similarities when multiple revisions for the same processor signature are compared against each other.

Lastly, the final category of Intel microcode updates consists primarily of the Atom, Nehalem and Sandy Bridge microarchitectures (Core i3/i5/i7), in addition to some newer Core 2 or Pentium 4 processors. The data block size of these updates ranges from 976 through 16336 bytes in increments of 1024, with the exception of 3024 and 14288, plus 5120 bytes. These updates are unique in that they contain an additional undocumented metadata header (table 8) of 96 bytes in size within the binary patch block, as discussed in [26].

6.2.1 Encryption

Based on our results, it appears possible that the first category of microcode updates is encrypted using a block cipher with a block size of $8 * 8 = 64$ bits or less, where $gcd(864, 944, 3096) = 8$. In addition, this block cipher does not appear to be chained, as changes to individual blocks can be distinguished.

These results also support the assertion by some sources that the patch data block is not entirely filled with microcode, but instead consists of patch microcode followed by a block of randomly generated garbage data to deter reverse engineering [24]. In particular, patch RAM only has capacity for 60 microcode instructions on the P6, and at minimum only the first 864 bytes of each microcode update differ, with the latter oftentimes remaining constant. Although it is possible that the latter could be some sort of shared binary microcode section, an early microcode update released in 1995-09-05 for processor signature `00000611h` (Pentium Pro 150) with revision 000b0026h has the latter portion completely zeroed out. In addition, our fault injection results show that modifications to this latter block do not affect acceptance of microcode by the processor (figure 5). ...

In addition, the same source also claims that the patch data is split into blocks of varying lengths that are encoded differently, and that it is comprised of a short initialization section followed by actual patch data [24], but this cannot be confirmed without decryption of the patch block. Nevertheless, it is a reasonable conclusion, as a similar feature exists in AMD microcode updates.

Not much is known about the second category of microcode updates, although the final category of

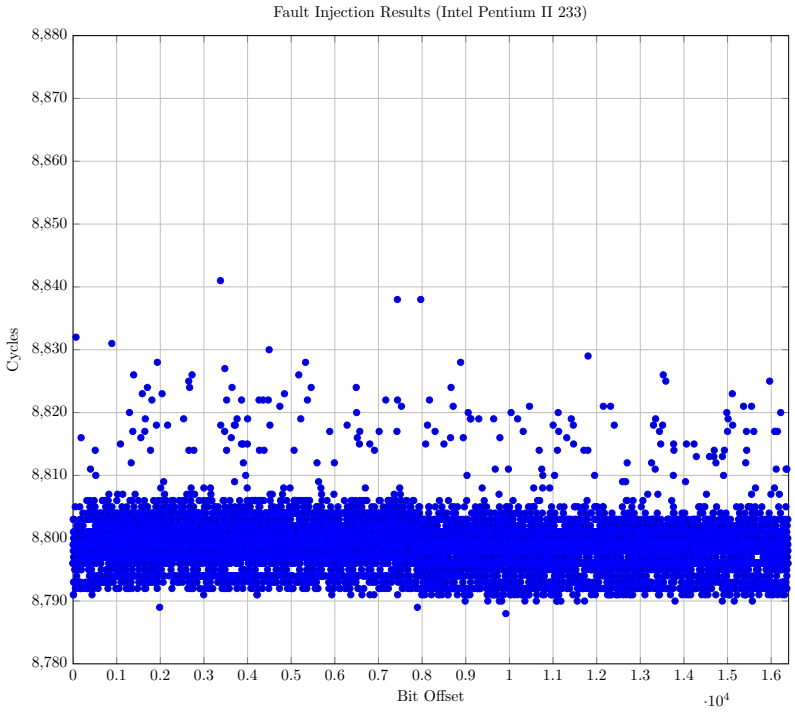

Figure 5: Intel P2 microarchitecture test results

updates has been revealed to contain a shared 256 byte RSA public key and 4 byte exponent, which accounts for the 260 byte noted similarity. This is followed by a RSA signature containing a SHA-1 or SHA-2 hash digest, and the actual patch microcode encrypted using the AES or DES block cipher [26]. It is also believed that encrypt-then-MAC is used.

7 Discussion

With the trend towards integrating multiple logical processors within a single physical processor package, ensuring synchronization of microcode updates between processors is a major issue. In fact, manufacturer documentation for Intel processors [31] notes that microcode update facilities are not shared between each logical processor, and must be performed for each processor. However, note that this behavior differs for processors incorporating "Hyper-Threading Technology", in which a single processor core appears as two independent logical processors to the system. In this case, only one logical processor per core needs to load the update [31].

Breaking these synchronization assumptions will lead to nondeterministic execution of program code, with operation behavior depending on the specific logical processor that a program executes on. For example, if a microcode update is only loaded on three out of four cores on a processor, then there roughly exists a

1/4 chance of program code executing on an unpatched processor and causing unexpected behavior. This may be particularly problematic if the program (i.e. operating system) attempts to workaround processor errata by checking the current processor revision, which will differ between individual processor cores. In particular, consider the implications if it were not one processor core left unpatched, but infected by a malicious microcode update. Work is currently ongoing to examine this attack vector.

There exists similar implications for instruction primitives handled directly by the microprocessor such as encryption or cryptographic operations, which accentuates the risk of microcode attacks due to their importance in ensuring the security and privacy of modern computer systems. In particular, although microcode updates cannot be loaded in VMX non-root operation with a proper hypervisor, attempted updates performed in VMX mode can result in unpredictable system behavior, destabilizing the entire machine. Alternatively, a VMM can also drop attempts by a virtual machine guest to write to the microcode update MSR, while emulating the microcode update signature during read of the microcode revision MSR.

We plan to look into manufacturer-specific performance information in order to determine characteristics about the number of microcode instructions per x86 instruction, as benchmarking microcode instructions may provide additional information about the execution pathway of x86 instructions. On more recent Intel processors, data about the number of uops decoded by the microcode sequencer and floating-point microcode assists are recorded by PEBS functionality, and is accessible from user mode. Additional fuzzing with modified microcode can be tracked by examining PEBS functionality recording uops decoded by the microcode sequencer and floating-point microcode assists. Modifications to the match registers and known-bad microcode update values would be a slow method to determine MROM entry point addresses for common x86 instructions.

From a hardware perspective, there exist a number of possible additional analysis techniques. Hardware-based epoxy decapping and analysis under a microscope with fuming sulfuric acid has been successful in revealing the contents of "secret" memories within microcontrollers, albeit for a much larger scale semiconductor manufacturing process. Other techniques such as differential power analysis have also been shown effective in disclosing secret encryption keys stored within commercial FPGAs, but may be difficult to apply to microprocessors due to increased silicon complexity. There also exist proprietary physical debugging interfaces for microprocessors via exposed surface mount pads or land grid array pins, e.g. Intel's XDP or other JTAG-style connectors, which may be useful for obtaining more information about processor internals.

It is interesting to note that at least for earlier Intel microprocessors, such as the Pentium Pro, processor microcode was the single largest source of bugs identified during the development process, accounting for over 30% of the total, whereas on the Pentium 4 processor, microcode accounted for less than 14% of the bugs [7]. Nevertheless, these statistics indicate that the likelihood of discovering microcode bugs is relatively high, which could result in significant security vulnerabilities.

8 Mitigation

Due to the fact that microcode update functionality is embedded within the silicon of existing processors, there are no significant mitigations that can be directly applied, as changes to the microcode update mechanism or encryption algorithms are impossible. However, since microcode updates require system privileges to be performed, users are advised to strengthen existing security protections, including user access controls, and ensure that access to processor MSRs are appropriately filtered by hypervisors, where applicable. Users are also advised to reset the processor to restore original processor microcode, although the integrity of the system BIOS/UEFI should be verified as well. Note that software reboots using kexec-like functionality are insufficient, as they only replace a running kernel in system memory without actually resetting the processor itself.

However, there are a number of changes that future development work into processor microarchitecture could incorporate in order to prevent these types microcode attacks. Elimination of side channel analysis vectors could occur by "pausing " the processor time-stamp counter (TSC) during microcode update operations. This could be implemented in hardware, or within the microcode itself by storing the value of the time-stamp counter before and after the microcode update, then calculating the difference and subtracting

it from the hardware time-stamp counter before returning.

In addition, the timing pathways of digital circuitry could be balanced to prevent side channel attacks by comparing the differences in gate delay between successful and unsuccessful operation pathways. This would entail the addition of clock cycle delays to certain microprocessor operation, which could have a slight impact on overall performance.

Furthermore, it may be advisable to consider implementing so-called "e-fuse" capability within processors, in which a fuse can be blown by the BIOS/UEFI or operating system to disable further microcode updates. However, a processor reset could also reset the value of this fuse, preserving microcode update functionality by trusted sources such as BIOS/UEFI while otherwise disabling the behavior.

9 Conclusion

Our results show that microcode attacks are a viable attack vector against the security of x86 microprocessors. Due to the importance of processor microcode in handling instruction decode and execution for contemporary processors, compromise of processor microcode can allow an attacker to modify any existing instruction for malicious purposes, including interfering with the operation of virtual machine primitives or exception handling for floating-point numbers. Possible attack scenarios include compromising a hypervisor to allow escape of malicious code into the host, or decreasing the precision of floating-point instructions on financial computer systems. Due to the write-only nature of microcode patch RAM, these attacks are extremely difficult to prevent against or detect, as it is impossible to read out the contents of microcode RAM or verify that a loaded microcode update is actually legitimate.

In fact, this class of attacks is not limited to just processor microcode, but also affects other devices connected to the system bus that can be updated by an end-user, such as network controllers, graphics cards (and their BIOS), storage drive firmware, or even optical drive firmware. Compromise of low-level integrated remote management functionality such as Intel vPro, Active Management Technology, or Management Engine could allow malicious attackers to maintain a long-term persistent infection while remaining invisible to system administrators. Since many of these devices have direct memory access (DMA), any compromise of these devices can lead to virtually unrestricted system control, much like with the FireWire DMA exploit.

More broadly, similar flaws have been demonstrated among a variety of embedded equipment such as automobiles[11], credit cards[9], GPS receivers[36], network devices[14], satellite phones[18], cell phones[6], smart meters[38], police radios[13], and even programmable logic controllers utilized by utility grids and nuclear facilities[12]. These results, in conjunction with our research on processor microcode, show that embedded firmware is highly vulnerable, as hardware manufacturers do not pay enough attention towards ensuring the authenticity and integrity of this embedded software, despite its significant level of control over platform hardware and higher-level security mechanisms. In fact, it is clear that many hardware manufacturers have relied on the principle of security through obscurity to keep their firmware secure, which is not enough. As more and more devices are computerized with embedded microprocessors and connected to larger networks, ensuring that firmware is secure and bug-free will become an increasingly important security challenge.

However, recent publications indicate that both AMD [41] and Intel [22] have begun applying formal verification techniques to prove the operational correctness of their respective microcode. Although the limitations and scope of these techniques is not externally apparent, comprehensive application of such techniques could significantly reduce the number of post-silicon bugs and improve overall reliability, eliminating the need for end-user microcode updates.

10 Acknowledgments

We would like to thank Dr. Michael Huth at Imperial College London and Dr. Michael Goryll at Arizona State University for their comments and suggestions.

References

[1] NEC Corp. v. Intel Corp, 1989.

[2] Advanced Micro Devices. *AMD Athlon™ Processor Model 10 Revision Guide*, October 2003.

[3] Alexander, B., Anderson, A., Huntley, B., Neiger, G., Rodgers, D., and Smith, L. Power and Thermal Management in the Intel® Core™ Duo Processor. *Intel Technology Journal 10* (2006).

[4] Alexander, B., Anderson, A., Huntley, B., Neiger, G., Rodgers, D., and Smith, L. Architected for Performance - Virtualization Support on Nehalem and Westmere Processors. *Intel Technology Journal 13* (2010).

[5] Anonymous. Opteron Exposed: Reverse Engineering AMD K8 Microcode Updates, July 2004.

[6] Arapinis, M., Mancini, L., Ritter, E., Ryan, M., Golde, N., Redon, K., and Borgaonkar, R. New privacy issues in mobile telephony: fix and verification. In *Proceedings of the 2012 ACM conference on Computer and communications security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 205–216.

[7] Bentley, B., and Gray, R. Validating the Intel® Pentium® 4 Processor. *Intel Technology Journal* (Q1 2001).

[8] Boggs, D., Baktha, A., Hawkins, J., Marr, D. T., Miller, J. A., Roussel, P., Singhal, R., Toll, B., and Venkatraman, K. The Microarchitecture of the Intel® Pentium® 4 Processor on 90nm Technology. *Intel Technology Journal 8* (2004).

[9] Bond, M., Choudary, O., Murdoch, S. J., Skorobogatov, S. P., and Anderson, R. J. Chip and skim: cloning emv cards with the pre-play attack. *CoRR abs/1209.2531* (2012).

[10] Butler, M., Barnes, L., Sarma, D., and Gelinas, B. Bulldozer: An approach to multithreaded compute performance. *Micro, IEEE 31*, 2 (March 2011), 6–15.

[11] Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., Savage, S., Koscher, K., Czeskis, A., Roesner, F., and Kohno, T. Comprehensive experimental analyses of automotive attack surfaces. In *Proceedings of the 20th USENIX conference on Security* (Berkeley, CA, USA, 2011), SEC'11, USENIX Association, pp. 6–6.

[12] Chen, T., and Abu-Nimeh, S. Lessons from stuxnet. *Computer 44*, 4 (2011), 91–93.

[13] Clark, S., Goodspeed, T., Metzger, P., Wasserman, Z., Xu, K., and Blaze, M. Why (special agent) johnny (still) can't encrypt: a security analysis of the apco project 25 two-way radio system. In *Proceedings of the 20th USENIX conference on Security* (Berkeley, CA, USA, 2011), SEC'11, USENIX Association, pp. 4–4.

[14] Cui, A., and Stolfo, S. J. A quantitative analysis of the insecurity of embedded network devices: results of a wide-area scan. In *Proceedings of the 26th Annual Computer Security Applications Conference* (New York, NY, USA, 2010), ACSAC '10, ACM, pp. 97–106.

[15] de Vries, H. Intel Pentium 4 Northwood, April 2003.

[16] de Vries, H. AMD Deerhound Core (K8L-Rev.H), June 2006.

[17] Domburg, J. Hard disk hacking, 2013.

[18] Driessen, B., Hund, R., Willems, C., Paar, C., and Holz, T. Don't trust satellite phones: A security analysis of two satphone standards. In *Security and Privacy (SP), 2012 IEEE Symposium on* (2012), pp. 128–142.

[19] Duflot, L., Perez, Y.-A., and Morin, B. What if you can't trust your network card? In *Recent Advances in Intrusion Detection*, R. Sommer, D. Balzarotti, and G. Maier, Eds., vol. 6961 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2011, pp. 378–397.

[20] Feltham, D., Looi, C., Tiruvallu, K., Gartler, H., Fleckenstein, C., Looi, L., St. Clair, M., Spry, B., Callahan, T., and Mauri, R. The Road to Production - Debugging and Testing the Nehalem Family of Processors. *Intel Technology Journal 14* (2010).

[21] Fog, A. *Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for Intel, AMD and VIA CPUs*, February 2012.

[22] Franzén, A., Cimatti, A., Nadel, A., Sebastiani, R., and Shalev, J. Applying smt in symbolic execution of microcode. In *Proceedings of the 2010 Conference on Formal Methods in Computer-Aided Design* (Austin, TX, 2010), FMCAD '10, FMCAD Inc, pp. 121–128.

[23] Goddard, M. D., and Christie, D. S. Microcode Aatching Apparatus and Method, August 1998.

[24] Gwennap, L. P6 Microcode Can Be Patched. *Microprocessor Report* (September 1997).

[25] Haertel, M. Subject: bochs still no go, December 2001.

[26] Hawkes, B. Notes on Intel Microcode Updates, March 2013.

[27] Hennessy, J. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann/Elsevier, Waltham, MA, 2012.

[28] Hinton, G., Sager, D., Upton, M., Boggs, D., Carmean, D., Kyker, A., and Roussel, P. The Microarchitecture of the Pentium® 4 Processor. *Intel Technology Journal* (Q1 2001).

[29] Hong, Y. E., Leong, L. S., Choong, W. Y., Hou, L. C., and Adnan, M. An Overview of Advanced Failure Analysis Techniques for Pentium® and Pentium® Pro Microprocessors. *Intel Technology Journal* (Q2 1998).

[30] Intel Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, April 2012.

[31] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developer's Manual*, March 2013.

[32] Kagan, M., Gochman, S., Orenstien, D., and Lin, D. MMX™ Microarchitecture of Pentium® Processors With MMX Technology and Pentium® II Microprocessors. *Intel Technology Journal* (Q3 1997).

[33] Kauer, B. Oslo: Improving the security of trusted computing. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium* (Berkeley, CA, USA, 2007), SS'07, USENIX Association, pp. 16:1–16:9.

[34] McGrath, K. J., and Pickett, J. K. Microcode Patch Device and Method for Patching Microcode Using Match Registers and Patch Routines, August 2002.

[35] Molina, J., and Arbaugh, W. P6 Family Processor Microcode Update Feature Review, September 2000.

[36] Nighswander, T., Ledvina, B., Diamond, J., Brumley, R., and Brumley, D. Gps software attacks. In *Proceedings of the 2012 ACM conference on Computer and communications security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 450–461.

[37] Rogers, A., Kaplan, D., Quinnell, E., and Kwan, B. The core-c6 (cc6) sleep state of the amd bobcat x86 microprocessor. In *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design* (New York, NY, USA, 2012), ISLPED '12, ACM, pp. 367–372.

[38] Rouf, I., Mustafa, H., Xu, M., Xu, W., Miller, R., and Gruteser, M. Neighborhood watch: security and privacy analysis of automatic meter reading systems. In *Proceedings of the 2012 ACM conference on Computer and communications security* (New York, NY, USA, 2012), CCS '12, ACM, pp. 462–473.

[39] Stewin, P. A primitive for revealing stealthy peripheral-based attacks on the computing platform's main memory. In *Research in Attacks, Intrusions, and Defenses*, S. J. Stolfo, A. Stavrou, and C. V. Wright, Eds., vol. 8145 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2013, pp. 1–20.

[40] Sutton, J. A. Microcode Patch Authentication, October 2003.

[41] Tang, G., Bahal, R., Wakefield, A., and Ramachandran, P. Generating amd microcode stimuli using vcs constraint solver. Tech. rep., AMD, Inc and Synopsys, Inc, 2010.

[42] Thompson, K. Reflections on Trusting Trust. *Communications of the ACM 27*, 8 (1984), 761–763.

[43] Wojtczuk, R., and Rutkowska, J. Attacking intel® trusted execution technology. *Black Hat DC* (2009).

A Appendix

A.1 Bundled Intel Microcode on Windows

Date	Processor Signature	Update Revision	Processor Flags	Checksum
2007-09-26	000006f2h	0x0000005ah	00000001h	594ddba0h
2007-03-15	000006f2h	0x00000057h	00000002h	07e77759h
2007-09-16	000006f6h	0x000000cbh	00000001h	6f5dfa09h
2007-09-16	000006f6h	0x000000cdh	00000004h	a77fc94bh
2007-09-16	000006f6h	0x000000cch	00000020h	b5503da1h
2007-09-16	000006f7h	0x00000068h	00000010h	18729a7eh
2007-09-17	000006f7h	0x00000069h	00000040h	4e779cf4h
2007-09-24	000006fah	0x00000094h	00000080h	613bce61h
2007-07-13	000006fbh	0x000000b6h	00000001h	b3176c40h
2009-05-11	000006fbh	0x000000b9h	00000004h	b6a7f0c9h
2009-04-28	000006fbh	0x000000b8h	00000008h	7db01441h
2007-07-13	000006fbh	0x000000b6h	00000010h	5e5a71a7h
2009-05-11	000006fbh	0x000000b9h	00000040h	70fed5b1h
2007-07-13	000006fbh	0x000000b6h	00000080h	2831cee4h
2007-08-13	000006fdh	0x000000a3h	00000001h	89c0d09eh
2007-08-13	000006fdh	0x000000a3h	00000020h	89c0d07fh
2007-08-13	000006fdh	0x000000a3h	00000080h	89c0d01fh
2005-04-21	00000f34h	0x00000017h	0000001dh	2cbd6146h
2005-04-21	00000f41h	0x00000016h	00000002h	0a12a70ah
2005-04-22	00000f41h	0x00000017h	000000bdh	326135c1h
2005-04-21	00000f43h	0x00000005h	0000009dh	77812c17h
2005-04-21	00000f44h	0x00000006h	0000009dh	9f60db18h
2005-04-21	00000f47h	0x00000003h	0000009dh	af2cef0dh
2006-05-08	00000f48h	0x0000000ch	00000001h	5b9afec7h
2008-01-15	00000f48h	0x0000000eh	00000002h	0e158e10h
2005-06-30	00000f48h	0x00000007h	0000005fh	d0938263h

2005-04-21	00000f49h	0x00000003h	000000bdh	f85d53b8h
2005-12-14	00000f4ah	0x00000004h	0000005ch	5e7996d9h
2005-06-10	00000f4ah	0x00000002h	0000005dh	dfbc9997h
2005-12-15	00000f62h	0x0000000fh	00000004h	0976d137h
2005-12-15	00000f64h	0x00000002h	00000001h	680b0995h
2005-12-23	00000f64h	0x00000004h	00000034h	c66dbf02h
2006-04-26	00000f65h	0x00000008h	00000001h	5c58f575h
2006-07-14	00000f68h	0x00000009h	00000022h	0d8bb650h
2007-09-19	00010661h	0x00000038h	00000001h	8a2d6f19h
2007-03-16	00010661h	0x00000031h	00000002h	891e5cc8h
2007-03-16	00010661h	0x00000033h	00000080h	9e99cc48h
2008-01-19	00010676h	0x0000060ch	00000001h	fbac0f6ch
2008-01-19	00010676h	0x0000060ch	00000004h	fbac0f69h
2008-01-19	00010676h	0x0000060ch	00000010h	fbac0f5dh
2008-01-19	00010676h	0x0000060ch	00000040h	fbac0f2dh
2008-01-19	00010676h	0x0000060ch	00000080h	fbac0eedh
2008-04-28	00010677h	0x00000705h	00000010h	a6db99ddh
2008-04-09	0001067ah	0x00000a07h	00000011h	83067f5ah
2008-04-09	0001067ah	0x00000a07h	00000044h	83067f27h
2008-04-09	0001067ah	0x00000a07h	000000a0h	83067ecbh
2009-04-21	000106a4h	0x00000011h	00000003h	24e504ach
2009-04-14	000106a5h	0x00000011h	00000003h	c2d891c3h
2009-04-10	000106c2h	0x00000218h	00000004h	8fb7c1bah
2009-04-10	000106c2h	0x00000219h	00000008h	556338c1h
2009-08-25	000106cah	0x00000107h	00000001h	f851a3d9h
2009-08-25	000106cah	0x00000107h	00000004h	7deb58b2h
2009-08-25	000106cah	0x00000107h	00000008h	be667ca5h
2009-08-25	000106cah	0x00000107h	00000010h	482cae0eh
2009-04-06	000106d1h	0x00000026h	00000008h	deac5852h
2010-03-08	000106e4h	0x00000002h	00000009h	bdbb308ah
2010-04-05	000106e5h	0x00000004h	00000013h	f7762473h
2010-06-10	00020652h	0x0000000ch	00000012h	1e7bd02bh
2011-03-01	00020655h	0x00000002h	00000092h	267e87ffh
2010-06-18	000206c2h	0x0000000fh	00000003h	fecacce7h

Table 14: Microcode bundled within `mcupdate_GenuineIntel.dll`