Report of *CHEx86*

Yiwei Yang

2020.9.25

From a processor design perspective, the paper considers how to introduce effective hardware mechanisms to defend against memory corruption and proposes an interpolation scheme at the microcode level of the processor to counteract vulnerabilities in the code.) by comparison, this processor-based defense solution is a 59% performance improvement which is not simple.

The paper presents the CHEx86 processor design, which contains three key components: microcode customization unit, shadow capabilities table, and the speculative pointer tracker in actual implementation. when the microcode customization unit is responsible for stubs, the speculative pointer tracker is responsible for tracking possible problems in the code, and the shadow capabilities table provides a query mechanism to help determine if there is a problem which is not simple.

Report of *Reverse Engineering x86 Processor Microcode*

Yiwei Yang

2020.9.15

1 Introduction

Microcode is a very typical Microarchitecture mapping in Complex instruction set computer(CISC). It represent the basic abstraction of the physical components of a CPU compared with the hardwired implementation introduced in Reduced instruction set computer(RISC) In April 7, 1964, IBM first introduced the IBM 360 that first introduce microcode to implement instruction set. Among 1970s, with the emergence of faster ROM than DRAMs, microcode programing gains its boom for sake of computer architecture. To get more complex instructions, datapath and controller, e.g. floating point support, User-Writable Control Store(WCS) is introduced to extend the instruction by clients themselves. This turned to be a flot for clients' naiveneess to generate good microcode and may result in incompatability. The AMD's K8 and K10 Microarchitecture is the last consumer-available CPU to support User-WCS by reverse enginering the mircrocode semantics. Those results can be used to generate Trojan attack over web browser remote execution.

1.1 Goal

The paper claims to solve problems of how the mircrocode update work and explore how reverse engineering can be carried out in a semi-automatic way. The program has to be prooved by the PoC Mircro programs. The x86 instruction set is notorious for its complexity. The K framework to formally define it is done in [1].

1.2 The overview of the Mircrocode

1.2.1 Hardwired Decode Unit

Hardwired Decode Unit(HDU) utilize the logic gate to perform hardcoding of the semantics of how Microarchitecture works. It utilize the Final State Autometa(FSM) [2] that is generally faster than microcode. But if the Hardwired Hardware that consists of multiple gates takes more than 1 cycle, it should wait until the next cycle to checkout the result. Therefore, HDU is only applicable in simple logic that takes short amount of time.

1.2.2 Microcoded Decode Unit

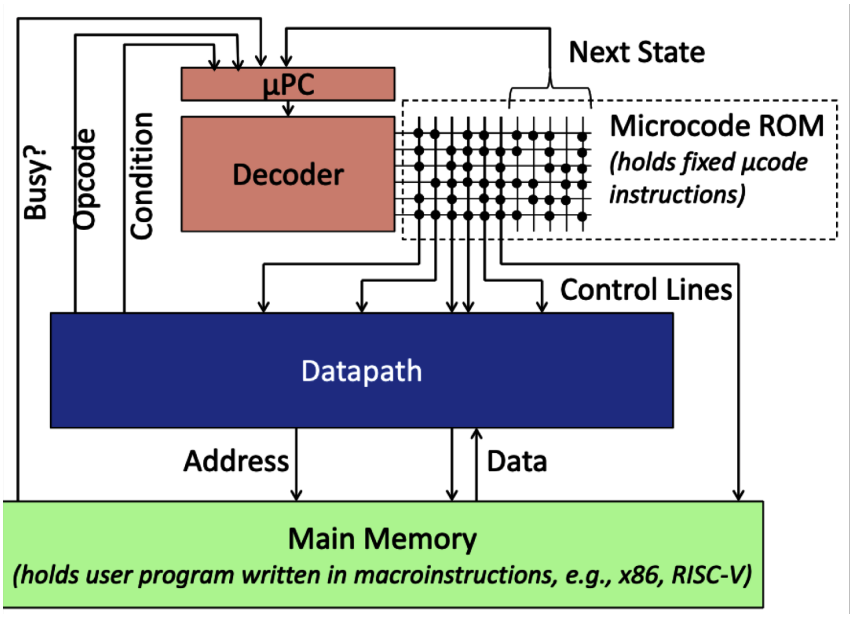

Figure R-1: The main logic of how MDU works [3]

The Microcode is hardcoded in the rom that is composed of the DRAMs nowadays. They use the on and offs register to store the logic of the mircroarchitecture.
Another consideration is the size of the microcode. The microcode is stored in a section of the CPU called the control store (since it is controlling the operation of the target architecture). The faster models in the family may need larger control stores, which requires more chip space. Larger chips are more expensive, but the larger control store will net better performance. The size of ROM is $2^{|\mu adress|} * |data|$.

1.3 Microcode Structure

The encoding methods affects the exection efficiency and storage efficiency over ROM, thus affecting how instructions exectes. The current struction may be the combination of the following 2 encoding, named nano encoding.

1.3.1 Horizontal Encoding

Horizontal encoding represents multiple parallel operations per μinstruction, which brings fewer microcode steps per macroinstruction but sparser encoding.

1.3.2 Vertical Encoding

Vertical encoding is typically a single datapath operation per μinstruction, basically the reverse of horizontal encoding.

1.4 Microcode Update

The update is usually triggered by teh patches. The patches is uploaded by the BIOS, UEFI or the OS during the early stage. The stage can be interpreted as the following graph. that teh microcode patch intercept the ROM entry point, so taht during the instruction decode, the microcode sequencer can check the trigger and redirect control to the patch on RAM. The microcode patches is not persistent.

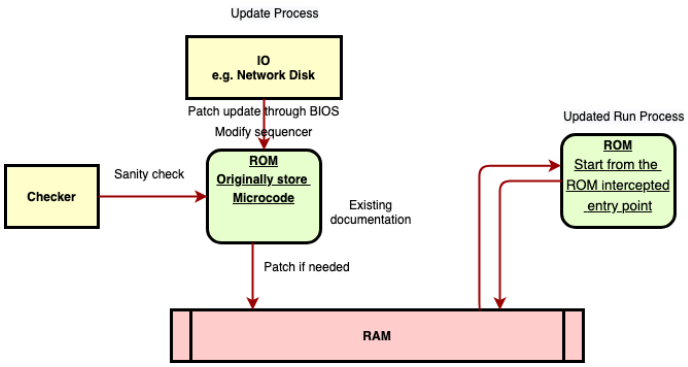

Figure R-2: The diagram of update process

2 Proposed Solution

AMD K8 and K10 processors are the only commercially available, modern x86 microarchitectures lacking strong cryptographic protection of microcode patches, so that it's easily cracked. AMD release 3 patcheds in 2004, and follows the update procedure of above. More specifically, the corresponding virtual address should be written to MSR **0xc0010020** which takes around 5000 cycles to initiate. They use the checksum to verify the file and return protection fault if failed. Then it copy the traids in the actural match register Patch RAM is mapped into the address space of the microcode ROM, whereby the patch triads directly follow the read-only triads. The match register are held by a mircrocode ROM address. Theh triad stored at the location will be intercepted at the location and redirect the control to the triad in the patch TAM at the offset match register index. The shared address space enables the microcode in the patch RAM to jump back to the microcode ROM. This procedure can be utilized.

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

Figure R-3: The format of uploading file format[4]

2.1 Reverse Engineer Method

They proposed a black-box attack process of steps from setting up a low-noise environment to heap map the ROM microcode to find some information and do microcode semantics and eventually hook the program to do the hack.

2.1.1 Low-Noise Environment

To record the CPU internal semantics, the upper level instructions or actions like multiprocessing or OS code execution are avoided to make the test easily meet the demand of get information from the heap

map. This is implemented by rawly connect the input and output of the pin to the Raspberry Pi via several bus. For hardware analysis, they choose good delayered ROM array to sample data on when they are connected to the GPIO serial port.

2.1.2 Heap Maps

The match Register hold microcode ROM addresses. Since the address map is not available, ~~so~~ the microinstruction inpretations are gotten by anaylizing heap map for each macroinstruction that we know their semantics and their distinct corresponding micro interpretations. Although they don't know the proproetary encoding, so with lack of the microcode update base, microinstructions control ALU and register file accesses, they formed various general assumptions about the instruction fields, which can be systematically tested using semiautomatic tests, for example, opcode, immediate value, source and destination register fields). Here's the semi-automated 2-tiered approach:

1. Identified fields by means of bits that cause similar behavior. For example, change of used registers, opcode, and immediate value. Three microinstructions in a triad to the same value are set to avoid side effects from other unknown microinstructions.

2. Exhaustively brute-forced each field to identify all addressable values, that consits of detailed information on why a specific microinsturction caused a crash.

3. Pad the triad with no-operation microinstructions and design tests to reuse microinstruction from existing microcode update to evaluate the new ones.

A heapmap of a specific 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. One cycle of testing corresponds to the operting system's **ret** and **call** which makes all the heap map corresponds to one complete instruction logic that manifest microcode ROM locations to intercept it. And teh vector instructions are covered by substracting the reference heap map. Test cases are fueled to make the whole process automated.

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	-

Figure R-4: The Truncated microcode ROM heat map[4]

2.1.3 Microcode Hooks

After reverse engineering the microcode encoding, they arbitrarily change CPU behavior for any microcoded macroinstruction and intercept control for any microcode ROM address.

2.1.4 Microcode API

Basically the microcode API is the general batch of microinstructions that has meanings. The general information can be exploited from the patch headers, RTL language that used to transfer match register information or different x86 eflags register values. The control software is implemented in Python on Raspberry Pi which is easy to carry out the tests.

3 Experiments

3.1 Process

The framework runs through 190 test iterations perminute and node in case there are no faults. One fault adds a delay of 12 seconds due to the reboot.

3.2 Consult on the RISC86 Manual[5] to get more information

The inforamtion about operand fields, operations and encoding differ is hinted in the manual by AMD. Single microinstructions can be addressed can those preceding microinstructions can be ignored. **RegOp, LdOp, StOp**, and **SpecOp** is used for arithmetic and logic operations, memory reads, memory writes, and special operations such as write program counter, respectively. The overall semantics will be disassembled to microcode with 40% of the whole instruction code with some unknown operation discarded.

3.2.1 RegOp

The **mul** and **imul** should be the first bits operand. For example, **3o** allows the three form operand mode **2:= reg1 op reg3/imm**, the flags field decide whether the resulting flags of the curreent **RepOp** should be commited to.

3.2.2 SpecOp

The PC writer, so that conditinonal branching is enabled. If the condition is not satisfied he microcode sequencer will decode the givern address after it. This takes 4 high bits of the 5 bits **cc** field. and write PC should be the third placeholder.

3.2.3 LdOp & StOp

reg1, reg2 and **reg3** will encode the micro register. In the process, microcode can access other registers with temporary value. For example, The special **pcd** register is read-only and contains the address of the next macroinstruction to decode that can be used to checkout next stage. **regmd** and **regd** declare an item as loaded on the stack and in a local register respectively. This helps to check jump semantics.

3.3 Intercepting x86 instructions

The procedure is done by intercept vector instructions by writing related triad addresses. The vector path instruction is to replaced simpler than deduce whether hooking on direct ones exist. The add (shrd and imul) logic is to imitate the semantics ourselves and do the sequence complete at the last triad. Differently, the div is to intercepted by jump back to ROM afterwards with do a loop with compulsary match register redirect that we do not get feature of ignoring a match register temporarily. So we need to intercept a negligible triad and do the logic and jump to subsequent triad. The iterative process should make sure teh source and destination GPR should hold the intermeidate result to avoid identified by the CPU.

3.4 Microcode RTL

They compose a Microcode RTL compliance to the x86 ISA to assemble bit vector. 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.

4 Utility of the Discovery

4.1 Remote Microcode Attacks

Use the just in time (JIT) compilation technology, the microcode hook can dynamically injected to the CPU so that if a div ebx is execting, the attacker can simply change the alignment and skip certain code.

4.2 Cryptographic Microcode Trojans

The hook can be the fault injection relying on inherent computation bugs to the current Elliptic Curve Cryptography that widely uesd in the public key. Also, it can trigger side channel for timing attack.

5 Possible Improvements

Current AMD processors employ strong cryptographic algorithms to protect the microcode update mechanism. So basically there's no way to improve. But for the improvement for this paper. I think with the tool of fuzzing smartly like deploying ML or RL, more microinstructions will be found.

References

[1] Sandeep Dasgupta, Daejun Park, Theodoros Kasampalis, Vikram S Adve, and Grigore Roşu. A complete formal semantics of x86-64 user-level instruction set architecture. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1133–1148, 2019.

[2] SudhaSuresh. Difference between hardwired and micro-programmed control unit. 2020.

[3] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[4] Philipp Koppe, Benjamin Kollenda, Marc Fyrbiak, Christian Kison, Robert Gawlik, Christof Paar, and Thorsten Holz. Reverse engineering x86 processor microcode. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 1163–1180, 2017.

[5] AMD K6 design team. Risc86 instruction. In *AMD-K6 Processor Code OptimizationApplication Note*, 2000.