Breaking the x86 ISA

Christopher Domas
xoreaxeaxeax@gmail.com

July 27, 2017

***Abstract*— A processor is not a trusted black box for running code; on the contrary, modern x86 chips are packed full of secret instructions and hardware bugs. In this paper, we demonstrate how page fault analysis and some creative processor fuzzing can be used to exhaustively search the x86 instruction set and uncover the secrets buried in a chipset. The approach has revealed critical x86 hardware glitches, previously unknown machine instructions, ubiquitous software bugs, and flaws in enterprise hypervisors.**

I. Overview

While the x86 architecture has been around for over 40 years, there exist no public tools for auditing and validating the processor's instruction set. With a history of processor errata, security flaws, and secret instructions, such introspection tools are necessary for establishing trust in a computing system built on an x86 platform. Here, we introduce the first effective technique for auditing the x86 instruction set, through guided fuzzing. The approach uses a depth-first instruction search algorithm in conjunction with page fault analysis to exhaustively enumerate the distinct x86 instructions, while requiring no pre-existing knowledge of the instruction format. The generated instructions are executed directly on an x86 platform, and the results of the execution – including observed instruction length and exceptions produced – are compared against the expected results from a disassembler. The technique reveals a multitude of undocumented instructions in a variety of x86 chips, shared software bugs in nearly every major assembler and disassembler, flaws in enterprise hypervisors, and both benign and security-critical bugs in x86 hardware. In this paper, we explore these issues, as well as the larger implications and risks of running software on closed-source hardware like the x86. Our work is released as a new open-source tool (*sandsifter*), allowing users to audit their processors for bugs, backdoors, and hidden functionality. The release of this toolset provides the first major step towards effective introspection of the black box x86 processor.

II. History

x86 is one of the longest continuously evolving instruction set architectures (ISAs) in history. With a design that began in early 1976 as the 8086, the ISA has undergone continuous revisions and updates, while still maintaining backwards compatibility and support for the original specification. In the 40 years since, the architecture has evolved with a multitude of new operating modes (figure 1), each adding an entirely new layer to the already complex design. Along with new modes came instruction set extensions, adding entirely new classes of instructions from a range of vendors (figure 2). In maintaining backwards compatibility, the processor has kept even those instructions and modes that are no longer used today. The result of these continuous modifications and evolutions is a processor that is a complex labyrinth of new and ancient technologies. Within this shifting maze, there are instructions and features that have been largely forgotten and lost over time.

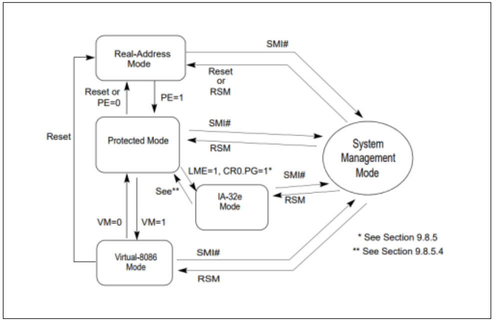
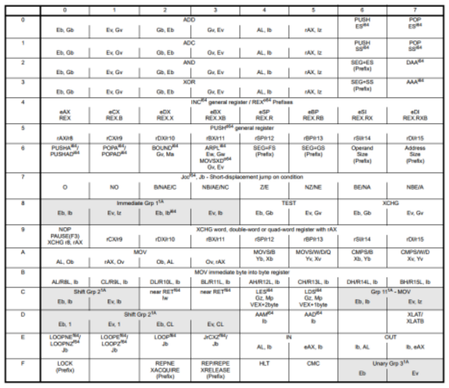


Figure 1. Evolution of x86 execution modes.

```
x87, IA-32, x86-64, MMX, 3DNow!,
SSE, SSE2, SSE3, SSSE3, SSE4,
SSE4.2, SSE5, AES-NI, CLMUL,
RdRand, SHA, MPX, SGX, XOP, F16C,
ADX, BMI, FMA, AVX, AVX2, AVX512,
VT-x, AMD-V, TSX, ASF
```

Figure 2. Evolution of x86 instruction set extensions.

With an immensely complex architecture, the security implications of lost or hidden features are a significant concern. Despite this, myriad undocumented instructions have crept into the architecture over the years (figure 3).



	0	1	2	3	4	5	6	7
0	ADD						PUSH ES^{i64}	POP ES^{i64}
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
1	ADC						PUSH SS^{i64}	POP SS^{i64}
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
2	AND						SEG=ES (Prefix)	DAA^{i64}
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
3	XOR						SEG=SS (Prefix)	AAA^{i64}
	Eb, Gb	Ev, Gv	Gb, Eb	Gv, Ev	AL, Ib	rAX, Iz		
4	INC^{i64} general register / REX^{o64} Prefixes							
	eAX REX	eCX REX.B	eDX REX.X	eBX REX.XB	eSP REX.R	eBP REX.RB	eSI REX.RX	eDI REX.RXB
5	PUSH^{d64} general register							
	rAX/r8	rCX/r9	rDX/r10	rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15
6	PUSHA^{i64}/ PUSHAD^{i64}	POPA^{i64}/ POPAD^{i64}	BOUND^{i64} Gv, Ma	ARPL^{i64} Ew, Gw MOVSXD^{o64} Gv, Ev	SEG=FS (Prefix)	SEG=GS (Prefix)	Operand Size (Prefix)	Address Size (Prefix)
7	Jcc^{f64}, Jb - Short-displacement jump on condition							
	O	NO	B/NAE/C	NB/AE/NC	Z/E	NZ/NE	BE/NA	NBE/A
8	Immediate Grp 1^{1A}				TEST		XCHG	
	Eb, Ib	Ev, Iz	Eb, Ib^{i64}	Ev, Ib	Eb, Gb	Ev, Gv	Eb, Gb	Ev, Gv
9	NOP PAUSE(F3) XCHG r8, rAX	XCHG word, double-word or quad-word register with rAX						
		rCX/r9	rDX/r10	rBX/r11	rSP/r12	rBP/r13	rSI/r14	rDI/r15
A	MOV				MOVS/B Yb, Xb	MOVS/W/D/Q Yv, Xv	CMPS/B Xb, Yb	CMPS/W/D Xv, Yv
	AL, Ob	rAX, Ov	Ob, AL	Ov, rAX				
B	MOV immediate byte into byte register							
	AL/R8L, Ib	CL/R9L, Ib	DL/R10L, Ib	BL/R11L, Ib	AH/R12L, Ib	CH/R13L, Ib	DH/R14L, Ib	BH/R15L, Ib
C	Shift Grp 2^{1A}		near RET^{f64} Iw	near RET^{f64}	LES^{i64} Gz, Mp VEX+2byte	LDS^{i64} Gz, Mp VEX+1byte	Grp 11^{1A} - MOV	
	Eb, Ib	Ev, Ib					Eb, Ib	Ev, Iz
D	Shift Grp 2^{1A}				AAM^{i64} Ib	AAD^{i64} Ib		XLAT/ XLATB
	Eb, 1	Ev, 1	Eb, CL	Ev, CL				
E	LOOPNE^{f64}/ LOOPNZ^{f64} Jb	LOOPE^{f64}/ LOOPZ^{f64} Jb	LOOP^{f64} Jb	JrCXZ^{f64}/ Jb	IN		OUT	
					AL, Ib	eAX, Ib	Ib, AL	Ib, eAX
F	LOCK (Prefix)		REPNE XACQUIRE (Prefix)	REP/REPE XRELEASE (Prefix)	HLT	CMC	Unary Grp 3^{1A}	
							Eb	Ev

Figure 3. Blanks in the x86 opcode maps indicate a possible hidden instruction.

Whereas the techniques for finding bugs, secrets, backdoors in software are well studied and established, similar techniques for hardware are non-existent. This is troubling, in that it is the processor that enforces the security of the system, and is ultimately the system's most trusted component. It seems necessary to stop treating a processor as a trusted black box for running software, and instead develop systematic tools and approaches for *auditing* processors, in much the same way that we can audit software. This is the motivation behind our research – an approach to discovering the secrets and flaws built into the processors we blindly trust.

III. Prior Work

Prior work on x86 fuzzing focuses on the correct functioning of emulators and hypervisors. These techniques take the approach of random instruction generation, or generation based on pre-existing knowledge of the x86 instruction format. Random instruction generation produces poor instruction coverage, and cannot find arbitrarily complex instructions, such as those with long combinations of prefixes and opcodes. Generation based on knowledge of the x86 instruction set can produce better instruction coverage, but fails to find undocumented and mis-documented instructions, and is still unable to find arbitrarily complex instructions. In addition to these limitations, no known approach focuses on the processor hardware itself. Our proposed technique is the first x86 fuzzing work targeting the actual processor, and uses an effective search approach requiring no prior knowledge of the x86 instruction format.

IV. Approach

Our goal is to find a way to programmatically exhaustively search the x86 instruction set, in order to find hidden or undocumented instructions, as well as instruction-level flaws like the Pentium f00f bug. To do this, we would generate a potential x86 instruction, execute it, and observe its results. The challenge with this is in the complexity of the x86 instruction set: x86 instructions can be between 1 and 15 bytes long (figure 3).

```
inc eax
40
lock add qword cs:[eax+4*eax+07e06df23h], 0efcdab89h
2e 67 f0 48 818480 23df067e 89abcdef
```

Figure 3. A 1 byte vs. a 15 byte x86 instruction.

With instructions up to 15 bytes long, the worst-case search space for the x86 ISA is 1.3×10^{36} instructions – a simple iterative search is infeasible, and randomly selecting possible instructions will only cover a tiny fraction of the potential search space. The search space can be reduced by only generating instructions that follow the formats described in x86 reference manuals, but this approach will fail to find undocumented instructions, and will miss hardware errors that are the result of invalid instructions. To effectively reduce the instruction search space, we propose a search algorithm based on observing changes in instruction lengths.

Searching the Instruction Set

The instruction search process, which we call *tunneling*, runs as follows. A 15 byte buffer is generated as a potential starting instruction; for example, for searching the complete instruction space, we use a buffer of 15 0 bytes as the starting candidate. The instruction is executed, and its length (in bytes) is observed. The byte at the end of the instruction is then incremented. For example, in the case of the 15 byte zero buffer, the instruction will be observed to be two bytes long; thus, the second byte is incremented, so that the buffer is now {0x00, 0x01, 0x00, 0x00, 0x00, …}. The process is then repeated with the new instruction. If this incrementation results in a change in the observed instruction length or exception generated, the resulting instruction is incremented from its new end. When the end of an instruction has been incremented 256 times (exhausting all possibilities for the last byte of that instruction), the increment process moves to the previous byte in the instruction (figure 4).

```
000000000000000000000000000000
000100000000000000000000000000
000200000000000000000000000000
000300000000000000000000000000
000400000000000000000000000000
000401000000000000000000000000
000402000000000000000000000000
000403000000000000000000000000
000404000000000000000000000000
000405000000000000000000000000
000405000000010000000000000000
000405000000020000000000000000
000405000000030000000000000000
000405000000040000000000000000
```

Figure 4. The instruction search starting at 0.

This technique allows effectively exploring the meaningful search space of the x86 ISA. The less significant portions of an instruction (such as immediate values and displacements) are quickly skipped in the search, since they do not change the instruction length or exceptions. This allows the fuzzer process to focus on only meaningful parts of the instruction, such as prefixes, opcodes, and operand selection bytes. This approach reduces the instruction search space from a worst-case 1.3×10^{36} instructions, down to a very manageable 100,000,000 instructions (as observed in lab scans, described in section IV). We implement the instruction generation and execution logic in a process called the *injector.*

Resolving Instruction Lengths

However, the instruction tunneling approach only works if there is a reliable way to determine the length of an arbitrary (potentially undocumented) x86 instruction. Since the instruction may be undocumented, disassembling the instruction is not an option. An alternate naïve approach to determining instruction length is to set the x86 trap flag, execute the instruction, and observe the difference between the original and new instruction pointers. However, this approach fails on instructions that throw faults – since a faulting instruction does not execute, there is no change in the instruction pointer when the instruction is stepped with the trap flag. We wish to find *all* potentially undocumented or flawed instructions – including those normally restricted to kernel, hypervisor, or system management code – so exploring even faulting instructions is critical to the approach. For example, an instruction such as “inc eax” can execute in ring 3 and below; an instruction such as “mov eax, cr0” can execute in ring 0 and below; and an instruction such as “rsm” can execute only in ring -2 (System Management Mode). For effective results, the injector should be able to identify instructions in more privileged rings, even if it cannot actually execute those instructions.

To effectively determine the length of even faulting instructions, we introduce a 'page fault analysis' technique, wherein instructions are incrementally moved across page boundaries to induce page faults. A candidate instruction is generated (a 15 byte value, generated by the incrementation process described earlier), and placed in memory so that the first byte of the instruction is on the last byte of an executable page, and the rest of the instruction lies in a non-executable page. The instruction is then executed. If a page fault occurs *during* the instruction fetch, the processor triggers the #PF exception, and the address of the page boundary is reported in the CR2 register. This indicates to the injector process that part of the instruction lies in the non-executable page; any other result indicates that the entire instruction was fetched from memory. If the injector determines that the instruction does not yet reside entirely in executable memory, the instruction is moved back a byte, so that the first two bytes are on an executable page, and the rest are on the non-executable page. The process is repeated until no #PF exception occurs, or until a #PF exception is received with an address other than the page boundary. At this point, the number of bytes lying in the executable page indicate the length of the instruction (figure 5).

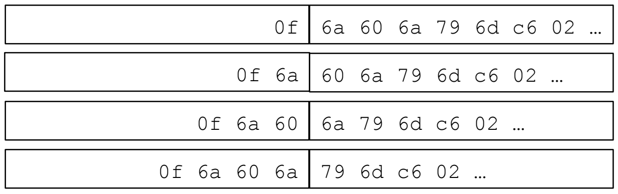


Figure 5. A candidate instruction is moved across a page boundary to determine its length. The first page is executable, while the second page is non-executable. When the instruction does not throw a #PF exception with CR2 set to the page boundary address, the entire instruction must lie within the executable page. In the example, the check is complete when the executable page contains 0f 6a 60 6a (punpckhdq mm4,[rax+0x6a]).

Once the non-#PF/CR2 combination is observed, the instruction fetch is known to be complete. However, it is still not clear whether the fetched instruction exists or not. For this, the injector observes any exceptions thrown by the instruction. Non-existing instructions will generate a #UD (undefined opcode) exception, existing instructions will either successfully execute or throw a different exception.

Interestingly, the approach allows resolving the length even of non-existing instructions. For example, 9a13065b8000d7 is an illegal instruction, but its length is known to be 7 bytes, because this is when the processor stops decoding the instruction. This provides some small insight into the pipelining architecture and the format of potential future instructions.

This approach allows the injector to detect even privileged instructions: whereas a non-existing instruction will throw a #UD exception, a privileged instruction will throw a #GP exception if the executing process does not have the necessary permissions for the instruction. By observing the type of exception thrown, the injector can differentiate between instructions that don't exist, versus those that exist but are restricted to more privileged rings. Thus, even from ring 3, the injector can effectively explore the instruction space of ring 0, the hypervisor, and system management mode.

Persistence

The tunneling algorithm combined with fault analysis to resolve instruction lengths brings us close to an effective x86 instruction fuzzing approach, but other problems arise. Foremost, the injector process is fuzzing the very processor it is running on. As such, it is important to avoid accidentally corrupting the system or process state. As a basic protection against this, we restrict the injector to ring 3 – this avoids the possibility of catastrophic system failures, except in the case of serious hardware bugs. Although the injector is limited to ring 3, it is still able to resolve the existence of instructions in more privileged rings through the page fault analysis.

While the operating system should not crash from the injector's ring 3 fuzzing, it is still possible for the injector to corrupt itself with one of the generated instructions.

The first situation to guard against is faulting instructions. For this, we hook every exception that a generated instruction might trigger (in Linux, sigsegv, sigill, sigfpe, sigbus, sigtrap). The injector's exception handler receives these signals, restores the system registers to a 'known good' state, and resumes the fuzzing process where it left off.

We should also guard against state corruption; specifically, the process state is corrupted if a generated instruction writes into the injector's address space. This is overcome by initializing all registers to 0 and mapping the NULL pointer into the injector process's memory. This ensures that computed memory addresses such as [eax + 4 * ecx] resolve to 0, rather than an address within the process's normal memory space. Mapping the page at address 0 into memory allows more detailed fault analysis for some types of instructions. For example, without address 0 mapped, "mov eax, [ecx + 8 * edx]" will generate a #GP exception, as will "mov cr0, eax". Since both instructions generate the same exceptions, the injector cannot determine that one is privileged and one is not. By mapping 0 into the process's address space, the unprivileged instruction can successfully execute, allowing the injector to differentiate it from the privileged instruction. However, this mapping is not strictly necessary for the search. With the registers initialized to zero prior to instruction execution, memory accesses with a displacement may still cause a process state corruption; for example, "inc [0x0804a10c]" may hit the .data segment of a 32 bit process, regardless of the register initialization values. However, as the tunneling approach for instruction searching only manipulates a single byte of the instruction at a time, it will explore "inc [0x0000000c]", "inc [0x0000a100]", "inc [0x00040000]", and "inc [0x08000000]", but will never search "inc [0x0804a10c]". In practice, this prevents the tunneling process from ever corrupting its own state.

We also provide an alternative fuzzing strategy via random instruction generation. In this approach, it is possible for the injector process to become corrupted, but we have observed that in practice, this is still extremely rare – a 32 bit process with 1 KB of writable critical program data has only a one in four million chance of being corrupted by an arbitrary memory access, and even then only for instructions that allow a 4 byte displacement in the memory calculation.

Despite these protections, the process state may still be corrupted by some specific instructions and be unable to recover its original state. To solve this, we are forced to blacklist a small subset of the instruction space. Specifically, we disallow execution of segment register loads (lds, les, lfs, lgs, lss, mov seg) and system call instructions (int 0x80, int 0xe, sysenter, syscall), which could corrupt the process state to the point that the exception handlers cannot recover it.

The last challenge in maintaining coherent execution state is resuming execution after an instruction is tested, and dealing with generated branch instructions. Both issues are solved by setting the x86 trap flag immediately prior to instruction execution,. The trap flag allows one instruction to execute, and then throws a single step exception. By catching the single step exception, the injector can catch execution after the instruction runs, and detect that an instruction successfully executed. This allows regaining control after both errant jump instructions and non-branching instructions.

Finding Anomalies

With the tunneling algorithm and page fault analysis, we are now able to effectively explore the x86 instruction set, reducing 10^{36} conceivable 15 byte combinations down to approximately 100,000,000 candidate instructions (as observed during the tests described in RESULTS). However, a means of identifying the unusual or interesting instructions is still necessary. For this, we wrap the injector with a *sifter* process. The sifter is responsible for recording anomalous results from one or more injectors. To do this, the sifter uses an existing disassembler to predict the length of an injected instruction. It then compares the observed length of the instruction with the expected length of the instruction. A difference in length generally indicates a software bug (the disassembler and processor disagree on the instruction). On the other hand, if the sifter sees that an instruction exists, but the disassembler does not recognize the instruction, it generally indicates an undocumented instruction on the processor (since, presumably, the disassembler is written based off of processor documentation). The tool has also produced the hypervisor and hardware bugs described in RESULTS; with the exception of a critical hardware bug, these have been due to incorrect exception generation by the hardware or hypervisor. This tends to cause incorrect instruction length resolution in the injector, which the sifter then flags as a software bug. At this point, manual analysis is necessary to correctly classify the bugs as software, hardware, or hypervisor.

For our research, we used the Capstone disassembler due to its ease of Python integration. However, code exists for swapping Capstone with objdump or ndisasm.

The Sandsifter Framework

These techniques form our "sandsifter" x86 fuzzing tool, which we release as open source. The tool calculates and executes each candidate instruction, and compares its observed length and fault behavior to the expected values provided by a disassembler and architecture documentation. Any deviations from the expected behavior are logged for analysis.

V. RESULTS

We ran the processor fuzzer against following x86 processors: Intel Core i7-4650U, Intel Quark SoC X1000, Intel Pentium, AMD Geode NX1500, AMD C-50, VIA Nano U3500, VIA C7-M, Transmeta TM5700, and another, currently unspecified x86 processor. The tool discovered undocumented instructions in all major processors, shared bugs in nearly every major assembler and disassembler, flaws in enterprise hypervisors, and critical x86 hardware errata.

Hidden Instructions

In this section, we use the notation xx to denote an arbitrary byte, {aa-bb} to indicate any byte between aa and bb, and, following Intel notation, /n to denote n as the reg field of the instruction's modr/m byte.

On an Intel Core i7-4650U processor running in 64 bit mode, the following undocumented instructions were found. 0f0dxx /non-1: this is currently documented as prefetchw for /1; other reg fields are not documented, but still execute. 0f18xx: until the -061 (December 2016) version of the reference manuals, about half of these instructions were undocumented, but would still run (the tested processor was released in 2012); they're now documented as reserved nops (presumably in place of a future instruction). 0f{1a-1f}xx: similar to 0f18xx, this doesn't appear until the -061 references, but executed at least back to Ivy Bridge. 0fae{e9-ef, f1-f7, f9-ff}: these seem to have existed for a long time, but were undocumented until the -051 references (June 2014) (only the r/m field = 0 were documented prior to this). dbe0, dbe1: these execute but do not appear in the opcode maps. df{c0-c7}: these execute but do not appear in the opcode maps. f1: this executes but does not appear in the opcode maps; there is a note in SDM vol. 3 that it and d6 will not produce a #UD (interestingly, d6 *does* produce a #UD, at least in Ivy Bridge). {c0-c1, d0-d1, d2-d3}{30-37, 70-77, b0-b7, f0-f7}: these execute, but are not in the opcode maps; we believe they are SAL aliases. f6 /1, f7 /1: these execute, but aren't in the opcode maps; we suspect they are aliases for the /0 version.

On an AMD Geode NX1500, the following undocumented instructions were found. 0f0f{40-7f}{80-ff}{xx}: these are in the AMD 3DNow! Instruction range, but are not documented for a range of xx that still execute. dbe0, dbe1: these execute but do not appear in the AMD opcode maps. df{c0-c7}: these execute but do not appear in the AMD opcode maps.

VIA does not release programming manuals detailing the processor specifications the way that Intel and AMD do. Classifying an instruction as undocumented or not on VIA is done by comparing the instruction against Intel and AMD documentation, and any available VIA documentation on

specific instruction set extensions. On a VIA Nano U3500 and VIA C7-M, the following undocumented instructions were found. 0f0dxx: undocumented by Intel for non-/1 reg fields. 0f18xx, 0f{1a-1f}xx: undocumented by Intel until December 2016. 0fa7{c1-c7}: these fall into the VIA padlock instruction extensions range, but are not documented in the padlock reference. 0fae{e9-ef, f1-f7, f9-ff}: these are undocumented by Intel for non-0 r/m fields until June 2014. dbe0, dbe1: these do not appear in any Intel, AMD, or VIA documentation.
df{c0-c7}: dbe0, dbe1: these do not appear in any Intel, AMD, or VIA documentation.

Software Bugs

The tool discovered innumerable bugs in disassemblers, the most interesting of which is a bug shared by nearly all disassemblers. Most disassemblers will parse certain jmp (e9) and call (e8) instructions incorrectly if they are prefixed with an operand size override prefix (66) in a 64 bit executable. In particular, IDA, QEMU, gdb, objdump, valgrind, Visual Studio, and Capstone were all observed to parse this instruction differently than it actually executes. On Intel processors executing in 64 bit mode, the 66 override prefix appears to be ignored, and the instruction consumes a 4 byte operand, as it does without the prefix. Most disassemblers misinterpret the instruction to consume only a 2 byte operand instead (those that assume a 4 byte operand still miscalculate the jump target, assuming it is truncated to 16 bits). This difference in instruction lengths between the disassembled version and the version actually executed opens opportunities for malicious software. By embedding an opcode for a long instruction in the last two bytes of the physical instruction, the physical instruction stream can hide malicious code in the following instruction. Disassemblers and emulators, thrown off by the misparsing of the initial instruction, miss this malicious code in the subsequent instructions (figure 6).

```
66e90000              jmpw   4f5
0500000000            add    $0x0,%eax
0500000000            add    $0x0,%eax
48b8b811223344ffe090  movabs $0x90e0ff44332211b8,%rax
48b8b811223344ffe090  movabs $0x90e0ff44332211b8,%rax
48b8b811223344ffe090  movabs $0x90e0ff44332211b8,%rax
48b8b811223344ffe090  movabs $0x90e0ff44332211b8,%rax
```

Figure 6. Masking malicious code from objdump and GDB. The opening jmp is misparsed as a 4 byte instruction, throwing off the parsing of the subsequent instructions. A malicious "jmp payload" instruction (for the example, payload is 0x11223344) is embedded in the "movabs" instructions. While the disassembler sees "movabs", the processor will execute the embedded "jmp payload" instead.

As a demonstration of the impact on emulators, we created a program that runs as a benign process in QEMU, but executes a malicious function when run on baremetal (figure 7). The same program, analyzed in IDA, objdump, Capstone, or Visual Studio, will also appear to not execute the malicious code.

```
// trampoline
__asm__ ("\
    .globl trampoline_return       \n\
    mov $trampoline_return, %rax  \n\
    jmp *%rax                      \n\
    ");

// attack
__asm__ (".byte 0x66,0xe9,0x00,0x00,0x00,0x00");

if (1) {
    printf("malicious\n");
}
else {
    __asm__ __volatile__ ("trampoline_return:");
    printf("benign\n");
}
```

Figure 7. A malicious program that prints "benign" when run under QEMU, but "malicious" when run on baremetal. The assembly trampoline at the top is copied into low memory, as a target for the mis-emulated jmp instruction, while the jump on baremetal simply falls through to the next instruction.

These types of emulation failures (of which we found many) have important security consequences in terms of antivirus and sandboxing techniques. If an analysis engine cannot faithfully emulate the underlying architecture, it is easy for malicious softer to mask its true behavior.

The confusion in these instructions is likely caused by differences in AMD and Intel processors; AMD processors obey the override prefix, only fetching a two byte operand. However, due to AMD's small market share, tools would be better to follow Intel's implementation. QEMU misinterprets the instruction, even when emulating an Intel processor.

Hypervisor Flaws

To facilitate faster instruction enumeration, we rented a 20 core Azure instance to run some of our initial instruction scans. In this process, we accidentally discovered a bug in the Azure hypervisor – if the trap flag is set during a cpuid instruction, the hypervisor fails to emulate the trap correctly. Since cpuid will cause a vmexit, the hypervisor must emulate cpuid, *and* remember to check the trap flag state, to see if a single step exception should be triggered in the guest. Azure neglects this second step, so that a single step over a cpuid is missed (figure 8).

```
pushfq
orq %0, (%%rsp)
popfq
cpuid
/* trap should trigger here */
correct:
nop
/* a trap here is a hypervisor bug */
nop
```

Figure 8. A test program to reveal incorrect hypervisor emulation of a trap cpuid combination.

This bug does not present a security concern, but does highlight the troubling complexity of faithful x86 emulation.

Hardware Errata

In terms of processor errata, the tool found issues on Intel, AMD, Transmeta, and an as-yet unspecified processor.

On Intel, the tool successfully found the f00f bug on a

Pentium processor, wherein a “lock cmpxchg8b eax” instruction would cause a complete processor lock.

On AMD, the tool discovered that some processors generate a #UD (undefined opcode) exception prior to completing the instruction fetch. Per AMD specifications, a #PF (page fault) exception occurring during an instruction fetch should supersede a #UD exception, but in the instruction search, which places the last bytes of the instruction on a non-executable page, some processors generate the #UD before the final bytes are moved off of the read/write page. It appears that AMD discovered this at around the same time as this research; the newest AMD Architecture Programmer’s Manual (March 2017) was updated to allow this situation.

On the Transmeta TM5700, errata were found on four byte versions of instructions beginning with 0f71, 0f72, and 0f73. When a floating point exception is pending, these instructions receive an #MF (floating point) exception after the first three bytes of the instruction are fetched, even if the last byte of the instruction is on an unmapped page. A #PF exception is the correct behavior in this situation, since the instruction cannot be completely fetched without a page fault.

Lastly, a so-called ‘halt and catch fire’ instruction was discovered on an as-yet unnamed x86 processor. This instruction, executed in ring 3 from an unprivileged process, appears to lock the processor entirely. To rule out kernel bugs, the instruction was tested against three Linux kernels and two Windows kernels, yielding the same results. Kernel debugging with serial I/O and interrupt hooks appeared to corroborate the results. At the time of this paper’s publishing, the vendor has not been provided sufficient time to respond to the issue. The details of the instruction and the processors affected will be enumerated when responsible disclosure is complete, and an updated version of this whitepaper will be released. Such instructions pose a critical security risk, as they allow unprivileged users to mount denial of service attacks against shared systems.

VI. Conclusion

Although we treat our processors as trusted black boxes, they are riddled with the same flaws and secrets we find in software. Through guided instruction fuzzing based on a depth-first search and page fault analysis, the sandsifter toolset is able to exhaustively enumerate and test all reasonably distinct instructions in the x86 ISA. The process has revealed hidden instructions, software bugs, hypervisor flaws, and critical processor failures. With the release of the sandsifter tool [1], the reader is encouraged to audit their own processors for defects and hidden instructions. This work provides an important first step towards introspecting x86 chips, and validating the processors we all blindly trust.

References

[1] https://github.com/xoreaxeaxeax/sandsifter