Breaking the x86 ISA

Christopher Domas
xoreaxeaxeax@gmail.com

July 17, 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

Here, we introduce the first effective approach for fuzzing the x86 instruction set. Using a page fault analysis, we've uncovered critical x86 hardware glitches, hidden processor instructions, ubiquitous software bugs, and flaws in enterprise hypervisors. We explore these issues, as well as the larger implications and risks of running software on black-box 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. This provides the first major step towards introspecting the black box x86 processor.

II. History

We must face the unpleasant truth that our processors are treated as trusted black boxes on which to run our software. Yet in reality, x86's lesser known history is full of secrets and failures: hardware flaws, from the Pentium f00f to the Cyrix comma bugs; corporate secrets, from Intel's mysterious "Appendix H", to the undocumented ICE execution mode on earlier x86 designs; all the way to restricted backdoors, as with AMD and VIA's password protected registers. This is the motivation behind our research - an approach to discovering the secrets and flaws built into the processors we blindly trust.

III. 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 should generate a potential x86 instruction, execute it, and observe its results. The most significant challenge with this is in the complexity of the x86 instruction set: x86 instructions can be 15 bytes long - 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.

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 an increase in the observed instruction length, 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. 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. This allows the fuzzing process to focus on only meaningful parts of the instruction, such as prefixes, opcodes, and operand selection bytes.

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, so exploring even faulting instructions is critical to the approach. Additionally, if, for practical reasons, the approach is run in one privilege ring, we may wish to explore instructions that can only execute in more privileged rings. 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 Managemnet Mode). For effective results, a fuzzer 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 place it 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 general protection exception occurs *during* the instruction fetch, the processor triggers the #GP interrupt, and the address of the page boundary is reported as the exception argument. This indicates to the fuzzing 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 fuzzer 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 #GP fault occurs, or until a #GP fault 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.

The approach allows resolving the length even of illegal (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. Analyzing illegal instructions opens the door to analyzing privileged instruction: whereas an illegal instruction will throw a #UD exception, a privileged instruction will throw a #GP exception. By observing the type of exception thrown, the fuzzer can differentiate between instructions that don't exist, versus those that exist but are restricted to more privileged rings. Thus, even from ring 3, we can effectively explore the instruction space of ring 0 and ring -2.

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, in fuzzing hardware instructions, it is important to avoid permanently corrupting the system or process state. As a basic protection against this, we restrict the fuzzer to ring 3 – with this, we only have to worry about the process state being corrupted, rather than the entire system state. Analyzing the fault type and operand still allows the fuzzer to explore instructions in more privileged rings.

Although restricting the fuzzer to ring 3 prevents the fuzzer from crashing the system, it is still possible for the fuzzer to crash itself. Specifically, the process state is corrupted if a generated instruction writes into the fuzzer's address space. This is overcome by initializing all registers to 0 and mapping the NULL pointer into the fuzzing 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 as well allows more detailed instruction 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 fuzzer 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 fuzzer to differentiate it from the privileged instruction. 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 fuzzing 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.

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, and catching the single step interrupt. This allows regaining control after both errant jump instructions and non-branching instructions.

With this, we are now able to effectively explore the x86 instruction set, reducing 10^36 conceivable 15 byte combinations down to a few million candidate instructions. 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.

IV. Results

We ran the instruction fuzzer on dozens of x86 processors. 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 bugs.

On an Intel Core i7 processor running in 64 bit mode, the following undocumented instructions were found. 0f0dxx: this is currently documented as prefetchw for /1 (ie reg field = 1), other reg fields aren't documented, but still execute. 0f18xx: until the -061 (June 2016) version of the reference manuals, about half of these instructions were undocumented, but would still run (the Device Under Test 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). 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.

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. 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 following instructions. As a demonstration, we created a program that executes a malicious function when run on baremetal, but runs as a benign process in QEMU. The same program, analyzed in IDA, will appear to not execute any malicious code. 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, it appears tools would be better to follow Intel's implementation.

In terms of processor errata, the tool found issues on Intel, Transmeta, and XXXX [PENDING DISCLOSURE] processors. On Intel, the tool successfully found the original Pentium f00f bug. On Transmeta, errata were found on four byte versions of illegal instructions beginning with 0f71, 0f72, and 0f73. When executing these instructions in combination with randomly generated instructions, the processor will sporadically fetch only three bytes of the four byte instructions before generating the #UD signal. Lastly, 'killer poke' instructions were discovered on XXXX processors. These instructions, executed from an unprivileged process, appear to lock the processor entirely. The details of the instructions and the processors affected will be enumerated when responsible disclosure is complete, and an updated version of this whitepaper will be released.

V. Conclusion

Although we treat our processors as trusted black boxes, they are riddled with the same flaws and secrets we find in software. With the release of the sandsifter x86 fuzzing tool [1], the reader is encouraged to audit their own processors for defects and hidden instructions. This work provides a critical stepping stone towards introspecting x86 chips, and validating the processors we all blindly trust.

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