N-version Disassembly: Differential Testing of x86 Disassemblers

Roberto Paleari[†] Lorenzo Martignoni[‡] Giampaolo Fresi Roglia[†] Danilo Bruschi[†]

Dipartimento di Informatica e Comunicazione[†]
Università degli Studi di Milano
Milano, Italy
{roberto,gianz,bruschi}@security.dico.unimi.it

Dipartimento di Fisica[‡]
Università degli Studi di Udine
Udine, Italy
lorenzo.martignoni@uniud.it

ABSTRACT

The output of a disassembler is used for many different purposes (e.g., debugging and reverse engineering). Therefore, disassemblers represent the first link of a long chain of stages on which any high-level analysis of machine code depends upon. In this paper we demonstrate that many disassemblers fail to decode certain instructions and thus that the first link of the chain is very weak. We present a methodology, called *N-version disassembly*, to verify the correctness of disassemblers, based on differential analysis. Given a set of $n-1$ disassemblers, we use them to decode fragments of machine code and we compare their output against each other. To further corroborate the output of these disassemblers, we developed a special instruction decoder, the n^{th}, that delegates the decoding to the CPU, the ideal decoder. We tested eight of the most popular disassemblers for Intel x86, and found bugs in each of them.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Reliability, Verification

Keywords

Software testing, differential testing, automatic test generation

1. INTRODUCTION

Disassemblers translate a stream of machine code into a sequence of assembly instructions. This kind of tools is used in a wide range of different applications. For example, disassemblers are used in debuggers, to closely inspect the execution of a program and they represent the most important class of tools for reverse engineers. Disassemblers are also used in CPU emulators, bug finding tools, binary rewriting systems, sandboxes, and many other types of applications. CPU emulators rely on disassemblers to interpret short sequences of instructions and to emulate each instruction appropriately. Binary rewriting tools rely on disassemblers to transform the program into an intermediate form suitable for rewriting [2]. Finally, sandboxes for the execution of untrusted code, like the Google Native Client [31], instead rely on disassemblers to analyze an untrusted piece of code, and to tell if it adheres to a certain set of security policies and if it is safe to execute.

To disassemble a piece of machine code means to translate it into a sequence of assembly instructions. Therefore, the disassembly process is an iterative process that consists in two phases: (I) instruction decoding and (II) selection of the next instruction of the program to decode. In the first phase a piece of code is translated into a single assembly instruction. In the second phase the disassembler selects the next piece of code to decode according to the format (or the semantics) of the instruction previously decoded. This process is repeated until all the code is disassembled. At first sight, the iterative disassembly process just described appears trivial. Unfortunately, theoretically speaking, disassembling is equivalent to the halting problem [15]. The main reasons for such complexity are the impossibility of separating data from code, self-modifying code (i.e., code that modifies itself at run-time), and indirect control transfers (i.e., control transfers whose target is computed dynamically). Practically speaking instead, disassemblers assume that certain compilers conventions are respected, adopt several heuristics to tackle the aforementioned problems [4, 12, 13, 28], and work reasonably well. Nevertheless, when the code being disassembled violates the assumed conventions, when the heuristics fail, or even worse when code is obfuscated, disassemblers produce completely unreliable results.

The problems mentioned in the previous paragraph are all specific to the second phase of the disassembly process (i.e., the selection of the next instruction to disassemble). The first phase, the decoding of an instruction, has always been assumed to be a trivial translation to perform. Nowadays, CISC CPUs have a very large instruction set, where instructions can often be encoded in multiple ways and typically support multiple operand types. Therefore, although instructions decoding still remains a mechanical translation from one form to another, the complexity this kind of translation conceals is not negligible. A complex translation process implies code that is tedious to write, and such code is much more prone to bugs. Since, in any high-level analysis of machine code, disassemblers represent the first link of a long chain of interconnected phases, a bug in the disassembly module would produce *dangerous cascade effects* on all the subsequent phases.

In this paper we present a fully automated methodology to evaluate the correctness of the *instruction decoder*, the component of disassemblers that is responsible for the decoding of machine instructions. Our methodology, we call *N-version disassembly*, is based on differential analysis [21] and is currently specific for Intel x86 architecture. Given an arbitrary string of bytes, we use multiple

$(n-1)$ disassemblers to decode the instruction potentially starting with the first byte of the string, and then we compare the output of the various disassemblers to detect discrepancies. Any discrepancy is a symptom that the instruction decoder of at least one of the tested disassemblers is buggy. Since outputs can be discordant, we assign to each disassembler a coefficient that describes the confidence we have in its output; the coefficient is proportional to the number of disassemblers that agree on that particular result. To further corroborate the output of the tested disassemblers, we developed ourselves the n^{th} instruction decoder used for the differential analysis. Our instruction decoder does not perform itself the decoding, but instead delegates the duty to the *perfect instruction decoder* implemented in the CPU. For each input string, we use this decoder to infer some information about the instruction the string encodes (e.g., whether the instruction is valid, the length of the instruction, and the type of operands) and then we check whether the output produced by the tested disassemblers is compliant with the format of the instruction. Since, the output of the perfect decoder is correct by definition, any incompatibility between the format of the instruction inferred with the help of the CPU and the format of the instruction reported by a tested disassembler denotes that the disassembler is not working properly.

We demonstrate experimentally that the decoding of instructions of CISC architectures, which apparently looks like a trivial task, is in practice complex and error prone. Indeed, we used the proposed methodology to evaluate several off-the-shelf and popular disassemblers and we found that all of them contained bugs that could compromise the correctness of any high-level analysis that depends on their output.

In summary the paper makes the following contributions:

1. a fully automated methodology to evaluate the correctness of a disassembler;

2. an instruction decoder that leverages the physical CPU (i.e., the perfect instruction decoder) to infer the format of instructions;

3. an experimental evaluation of 8 off-the-shelf Intel x86 disassemblers, demonstrating the effectiveness of our methodology and the existence of multiple defects in all tested disassemblers.

The paper is organized as follows. Section 2 motivates our work. Section 3 presents the details of our testing methodology. Section 4 presents the results of our experimental evaluation. Section 5 describes the related work and Section 6 concludes the paper.

2. MOTIVATIONS

This section briefly presents the format used in the Intel x86 architecture to encode instructions; that should let the reader understand why developing a disassembler for a CISC architecture is a tedious and error prone task. Furthermore, the section discusses the potential implications of bugs in a disassembler.

2.1 Challenges in Instruction Decoding

Today, Intel x86 is the most widely adopted computer architecture. It is a complex CISC architecture, with an incredible number of different instructions, of variable length, and a myriad of subtle and complex details. All that makes the development of an instruction decoder for this architecture a very tedious and error prone task.

Figure 1 depicts the format of an Intel x86 instruction. An instruction is composed of different fields: it starts with up to 4 prefixes, followed by an opcode, an addressing specifier (i.e., `ModR/M` and `SIB` fields), a displacement and an immediate data field [16]. Opcodes are encoded with one, two, or three bytes, but three extra bits of the `ModR/M` field can be used to denote certain opcodes. In total, the instruction set is composed by more than 700 possible values of the opcode field. The `ModR/M` field is used in many instructions to specify non-implicit operands: the `Mod` and `R/M` subfields are used in combination to specify either registry operands or to encode addressing modes, while the `Reg/Opcode` sub-field can either specify a register number or, as mentioned before, additional bits of opcode information. The `SIB` byte is used with certain configurations of the `ModR/M` field, to specify base-plus-index or scale-plus-index addressing forms. The `SIB` field is in turn partitioned in three sub-fields: `Scale`, `Index`, and `Base`, specifying respectively the scale factor, the index register, and the base register. Finally, the optional addressing displacement and immediate operands are encoded in the `Displacement` and `Immediate` fields respectively. Since the encoding of the `ModR/M` and `SIB` bytes is not trivial at all, the Intel x86 specification provides tables describing the semantics of the 256 possible values each of these two bytes might assume. In conclusion, it is easy to see that elementary decoding operations, such as determining the length of an instruction, require to decode the entire instruction format and to interpret the various fields correctly.

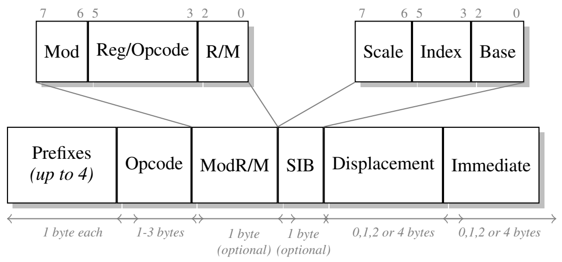


Figure 1: Intel x86 instruction format

The advent of several instruction extensions (e.g., Multiple Math eXtension (MMX) and Streaming SIMD Extensions (SSE)) made the instruction decoding process even more complicated. As an example, consider the byte strings `f3 ae` and `f3 0f e6`, encoding respectively the instructions `rep scasb` and `cvtdq2pd`. The byte `f3` is a prefix that is found in both instructions, but it serves two different purposes: it represents a `rep` prefix (to repeat the execution of an instruction) in the first case and a mandatory prefix for SSE instructions in the second case. Therefore, an instruction decoder has to consider that the prefix `f3` has to be interpreted differently, according to the subsequent sequence of bytes. The decoder must treat the prefix as a `rep` only when it is followed by a sequence of bytes that encodes a string or I/O operation (e.g., `scasb`), and as a preamble for SSE instructions otherwise (e.g., `cvtdq2pd`).

Finally, Intel x86 reference documentation sometimes lacks proper specifications for certain instructions, and states that others may have undefined effects in certain corner-cases. As an example, the use of a `rep` prefix with an instruction that is neither a string nor an I/O operation (e.g., `rep dec`) has undefined effects. In these situations, different instruction decoders often give completely different interpretations of the same byte sequence.

The aforementioned issues make instruction decoding a complex and tedious task. This complexity is testified by the size, in term of lines of code, of disassemblers for this architecture. Indeed, on average, the open-source disassemblers we evaluated in our experiments included 9000 lines of code. On the other side, we measured that the size of a disassembler for a RISC architecture (e.g., a SPARC) is less than a third.

2.2 Implications of Incorrect Disassembly

Disassemblers represent the front end of all high-level analyses and applications operating on machine code. Even the simplest bug in the instruction decoder might have serious implications on the final results provided by these tools. In the following paragraphs, we speculate on the potential implications of incorrect disassembly on debuggers, reverse engineering tools, CPU emulators, and sandboxes.

Debuggers and other reverse engineering tools are used in a wide variety of different contexts, ranging from software testing to malware analysis. The efficacy of these tools heavily depends on the accuracy of the module responsible for instruction decoding. Imagine a developer hunting for a defect in an application only available in binary form. If the faulty instruction were not properly decoded, then the defect could never be found. Moreover, even if the faulty instruction were decoded correctly but another one were not, there would be cascading effects that could prevent the developer from understanding the true behavior of the code fragment. Even worse, the developer would have no chance to realize that the disassembly he is trying to understand is wrong.

To emulate a physical CPU and to execute a program, CPU emulators mimic the fetch-decode-execute cycle of the CPU. Therefore, they have to correctly decode the instructions to execute. If the emulator did not decode an instruction exactly as the physical CPU would do, the instruction would not be properly executed, and consequently the emulated program could behave unexpectedly or could even fail to execute.

Sandboxes based on proof carrying code [24] or on simplified versions of this model (e.g., Google Native Client [31]) use disassemblers to analyze the code and to verify if it conforms to the selected security policies. To address the impossibiliy of analyzing any arbitrary piece of CISC machine code, sandboxes typically impose very strict constraints on the structure of the programs they execute, to guarantee that the entire code can be analyzed. For example, Google Native Client requires that all valid instruction addresses are reachable by a fallthrough disassembly that starts at the base address. However, such constraints are not sufficient to guarantee safety. Indeed, a bug in the disassembler could allow the sandbox to erroneously consider a malicious code as safe. For example, if the disassembler failed to recognize the length of an instruction, the sandbox would analyze a piece of code that differs from the code that would be executed.

3. TESTING DISASSEMBLERS

The perfect disassembler is the one that decodes instructions exactly as the CPU does. Obviously, we are assuming that the CPU always decodes correctly instructions: after all, the CPU is the hardware component that determines the semantics of a sequence of bytes. Therefore, ideally we would like to compare the output of the decoding performed by the CPU with the output of the disassembler under testing to verify if they correspond. Unfortunately, the fetch-decode-execute cycle is executed atomically by the CPU and thus, from software, we cannot look at the intermediate output of the three-step hardware execution cycle, but we can only tell whether an instruction is correctly executed or it generates an exception. However, the partial information about an instruction provided by the instruction decoder of the CPU plays an important role for validating the correctness of software-based instruction decoders, and it can be combined with the information obtained from other software-based disassemblers.

Our testing methodology is based on *differential testing* [21] and consists in comparing the output of multiple disassemblers. The disassemblers used for the testing are off-the-shelf disassemblers plus a special instruction decoder we have developed, which uses the CPU to infer part of the format of instructions. We call our methodology *N-version disassembly*, as we consider the output of n distinct disassemblers.

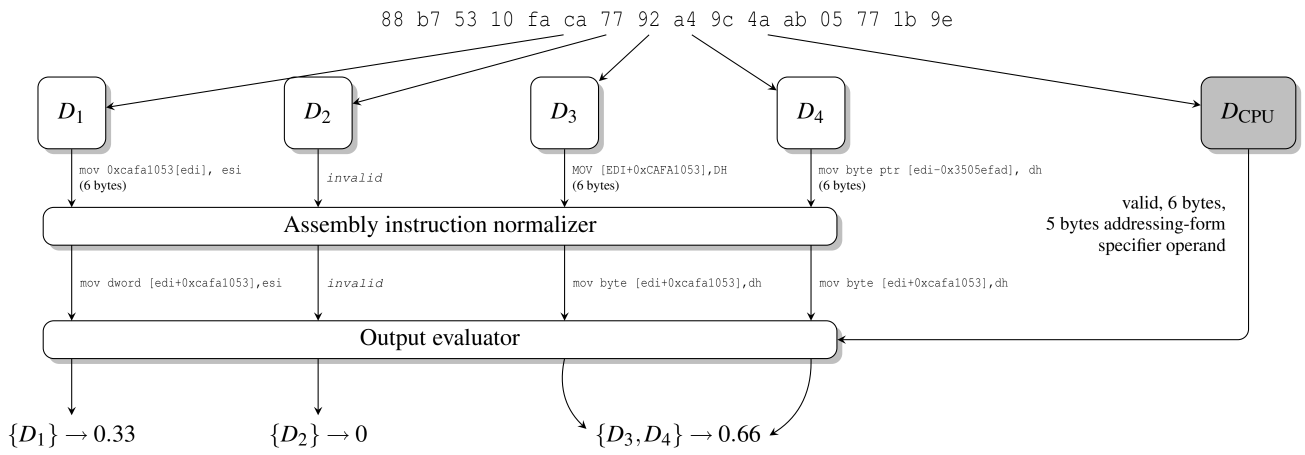
Figure 2 illustrates our testing methodology in action. Given an arbitrary sequence of bytes, we feed this sequence to a set of off-the-shelf disassemblers (D_1, ..., D_4 in figure) and to our custom disassembler (D_{CPU} in figure) and then we compare their output. Recall that our goal is to test only the instruction decoder component; therefore, in the case the input string encodes a sequence of more than one instruction, we consider only the output relative to the first one. Since not all the off-the-shelf disassemblers offer an API which allows to analyze in detail the format of an instruction, the output we consider is the assembly representation of the instruction. Although all disassemblers are supposed to produce the exact same output, in practice their output might differ slightly. For example, certain disassemblers make explicit the size of the various operands, while others do not. Therefore, we process the output of the various off-the-shelf disassemblers through a normalization routine, to remove the aforementioned differences and to make their output easily comparable. The normalized assembly instructions are then compared to each other and grouped when they correspond. The output produced by our custom disassembler, D_{CPU}, is not comparable as is with the output produced by the other disassemblers. Indeed, D_{CPU} produces in output only partial information about the format of the instruction (i.e., the length, the opcode, and the format of non-implicit operands) while $D_1, \ldots, D_4$ output an assembly representation of the instruction. Therefore, we can only check if the format inferred by D_{CPU} is compatible with the format reported by $D_1, \ldots, D_4$. Since D_{CPU} decodes an instruction through the decoder of the physical CPU, its output is considered correct by definition. Therefore, all the instructions (decoded by the other disassemblers) whose format is not compatible with the format reported by D_{CPU} are considered wrong and discarded. In summary, we end up with one or more groups of disassemblers that agree on their output and whose output is compatible with that of D_{CPU}. The rationale is that the probability that the output of a group of disassemblers is correct is proportional to the number of agreeing disassemblers. Clearly, the higher the number of off-the-shelf disassemblers considered in the testing, the higher the confidence in the result obtained. The corner-case situation is when $n = 2$: a disassembler can be tested only against D_{CPU}. In such a situation we could validate only partially the correctness of the output because D_{CPU} provides only partial information about the instruction (i.e., whether it is valid, the length, and the format of non-implicit the operands). Thus, we would not be able to validate the correctness of the mnemonic of the instruction nor of the format of implicit operands. However, we are not concerned with this problem since our approach is specific for the Intel x86 architecture and since there are plenty of disassemblers available.

We generate the sequences of bytes used to test disassemblers automatically, by leveraging two different techniques: (I) completely random input generation, and (II) an algorithmic technique, that provides high opcode coverage.

In the remaining of this section we discuss the proposed testing methodology in detail. To ease the presentation, the technique we have developed to generate inputs is described at the end of the section, since it is a variation of our custom instruction decoder.

3.1 Decoding with Off-the-shelf Disassemblers

Given a sequence of bytes, we invoke each of the off-the-shelf disassemblers in turn to try to decode this sequence and then we exam-



88 b7 53 10 fa ca 77 92 a4 9c 4a ab 05 77 1b 9e
D_1
D_2
D_3
D_4
D_{CPU}
mov 0xcafa1053[edi], esi (6 bytes)
invalid
MOV [EDI+0xCAFA1053],DH (6 bytes)
mov byte ptr [edi-0x3505efad], dh (6 bytes)
valid, 6 bytes, 5 bytes addressing-form specifier operand
Assembly instruction normalizer
mov dword [edi+0xcafa1053],esi
invalid
mov byte [edi+0xcafa1053],dh
mov byte [edi+0xcafa1053],dh
Output evaluator
$\{D_1\} \to 0.33$
$\{D_2\} \to 0$
$\{D_3, D_4\} \to 0.66$

Figure 2: Overview of *N-version disassembly* in action

ine the output produced. As our test inputs encode both valid and invalid instructions, a disassembler can interpret the input either as a valid or an invalid instruction. When a disassembler decodes a valid instruction, it also generates an assembly representation of the instruction, and the list of the bytes that encode the instruction. From this list we infer the length of the instruction. We use the assembly representation because not all disassemblers offer an API to inspect the internal structure of instructions.

We configure the disassemblers to decode instructions using the same assembly syntax (i.e., the Intel syntax). Nevertheless, even if a valid instruction is properly decoded by two or more disassemblers, the provided assembly representations very often differ in many subtle details. Indeed, we have not yet found two disassemblers that use the same exact assembly syntax. Consider the sequence of bytes shown in Figure 2. Both D_3 and D_4 consider the sequence `88 b7 53 10 fa ca` a valid instruction, but the assembly representations provided by the two disassemblers differ. As an example, D_4 explicitly states that the first operand refers to a 8-bit memory location, while in the output provided by D_3 the size of the first operand has to be inferred from the size of the second one. Similarly, the memory address displacement provided by D_4 is negative, while D_3 translated it into a positive value. To tackle such kind of differences in the syntax we normalize the assembly outputs using a set of manually developed normalization rules. For example, both instructions resulting from the normalization of the output of D_3 and D_4 make explicit the size of the memory locations and use positive displacements. Currently our prototype implementation of *N-version disassembly* supports 8 different disassemblers and adopts about a hundred different normalization rules. It is worth noting that to support new disassemblers, new normalization rules might be required and that the effort to write new rules depends on the disassembler.

3.2 CPU-assisted Instruction Decoder

We have developed a custom instruction decoder, we called *CPU-assisted instruction decoder*, that performs the decoding leveraging directly the physical CPU. Using this decoder, given an arbitrary sequence of bytes, we can detect if (I) it encodes a valid instruction, (II) the length of the instruction encoded by the sequence, and (III) the format of non-implicit operands of the instruction. Since the decoder delegates the decoding to the CPU, it is very simple (less than 500 lines of C code) and well tested.

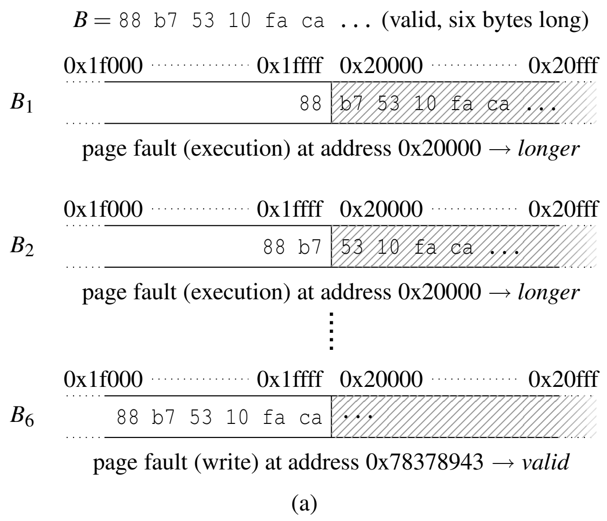
Our algorithm makes specific assumptions about the behavior of the Intel x86 CPU, especially about how the CPU fetches the code from memory. We do not know if the algorithm would work with other CPUs. However, our feeling is that RISC CPUs do not satisfy our assumptions.

3.2.1 Decoding Instruction Length

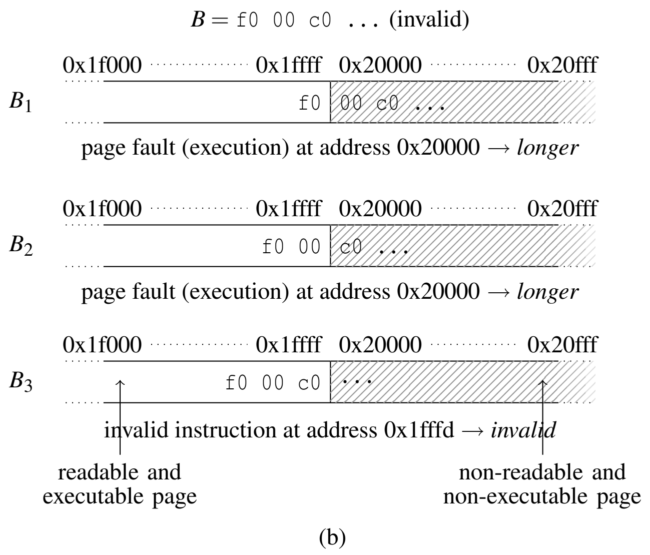
Given an arbitrary sequence of bytes $B = b_1 \dots b_n$, the first goal is to detect if the bytes represent a valid instruction. Then, for valid instructions, we have to infer their length. Our decoder exploits the fact that the CPU fetches from the memory the bytes of the instruction and decodes them incrementally. The decoder executes the input string B in a specially crafted execution environment, such that every fetch of the bytes composing the instruction can be observed.

The decoder initially partitions B into subsequences of incremental length ($B_1 = b_1$, $B_2 = b_1 b_2$, $\dots$, $B_n = b_1 \dots b_n$) and then executes one subsequence after another, using single-stepping. Since the goal is to intercept the fetch of the various bytes of the instruction, the i^{th} subsequence B_i (with $i = 1 \dots n$) is placed in memory such that it overlaps two adjacent memory pages, m and m'. The first i bytes are located at the end of m, and the remaining bytes at the beginning of m'. The two pages have special permissions: m allows read and execute accesses, while m' prohibits any access. When the instruction is executed, the i bytes in the first page are fetched incrementally by the CPU. If the instruction is *longer* than i bytes, when the $(i+1)^{th}$ byte is fetched the CPU raises a page fault exception (where the faulty address corresponds to the base address of m' and the cause of the fault is an instruction fetch) because the page containing the byte being read, m', is not accessible. If the instruction is i bytes long instead, the CPU executes the instruction without accessing the bytes in m'. In such a situation the instruction can be both valid and invalid. The instruction is *valid*, and i bytes long, if it is executed without causing any exception; it is also valid if the CPU raises a page fault or a general protection fault exception. A page fault exception occurs if the instruction tries to read or write data from the memory (in this case the faulty address does not correspond to the base address of m'); a general protection fault exception is raised if the instruction has improper operands (e.g., it expects aligned operands but alignment is not respected). The instruction is *invalid* instead, if the CPU raises an illegal instruction exception. If the instruction is either valid or invalid the decoder returns, otherwise, it repeats the process with the next subsequence, B_{i+1}.

Figure 3 shows our CPU-assisted decoder in action on two different sequences of bytes, one valid and one invalid. The first sequence is $B =$ `88 b7 53 10 fa ca ...`, corresponding to the



$B =$ 88 b7 53 10 fa ca ... (valid, six bytes long)
0x1f000 0x1ffff 0x20000 0x20fff
B_1
88 b7 53 10 fa ca ...
page fault (execution) at address 0x20000 → *longer*
B_2
88 b7 53 10 fa ca ...
page fault (execution) at address 0x20000 → *longer*
B_6
88 b7 53 10 fa ca ...
page fault (write) at address 0x78378943 → *valid*
(a)



$B =$ f0 00 c0 ... (invalid)
0x1f000 0x1ffff 0x20000 0x20fff
B_1
f0 00 c0 ...
page fault (execution) at address 0x20000 → *longer*
B_2
f0 00 c0 ...
page fault (execution) at address 0x20000 → *longer*
B_3
f0 00 c0 ...
invalid instruction at address 0x1fffd → *invalid*
readable and executable page
non-readable and non-executable page
(b)

Figure 3: Computation of the length of instructions using our CPU-assisted instruction decoder: (a) valid and (b) invalid instructions.

instruction `mov byte [edi+0xcafa1053],dh`. The decoder allocates two adjacent memory pages and removes any permission from the second one. Then, it starts with the first subsequence $B_1 =$ `88`. The byte is positioned at the end of the page and then executed through single stepping. The CPU fetches and tries to decode the instruction but, since the instruction is longer than one byte, it tries to fetch the next bytes from the protected page, raising a page fault. The decoder detects the fault and concludes that the instruction is longer than one byte (in our example the faulty address is 0x20000, the base address of the second page). It repeats the procedure with $B_2 =$ `88 b7` and gets the same result. It tries again with B_3, B_4, B_5, and finally tries with six bytes. Since the instruction is six bytes long, the CPU executes the instruction without accessing the protected memory page. However, the instruction writes into the memory and thus causes a page fault. As in this case the faulty address (0x78378943) differs from the address of the protected page, our decoder can tell the instruction is valid and that it is six bytes long. It is worth noting that a sequence of bytes cannot encode, at the same time, a valid instruction and a prefix of a longer instruction. Indeed, such a situation would be ambigous for the CPU. The second byte sequence in the example of Figure 3(b) is $B =$ `f0 00 c0` ... and represents an invalid instruction. Exactly as before, our decoder executes the first two subsequences B_1 and B_2 and detects that the instruction is potentially longer because the CPU fetches a third byte from the protected page. When B_3 is executed, the CPU does not fetch more bytes but instead raises an illegal instruction exception, testifying that B_3 is neither a valid instruction, nor a valid prefix for longer instructions.

It is worth noting that, although the CPU-assisted decoder is implemented as a user-mode application, it can also successfully decode privileged instructions (i.e., instructions that can be executed only in kernel-mode). That is possible because the CPU raises a special exception when a valid privileged instruction is executed without appropriate privileges, and this exception differs from the one raised to notify an attempt to execute an invalid instruction.

3.2.2 Decoding Non-implicit Operands

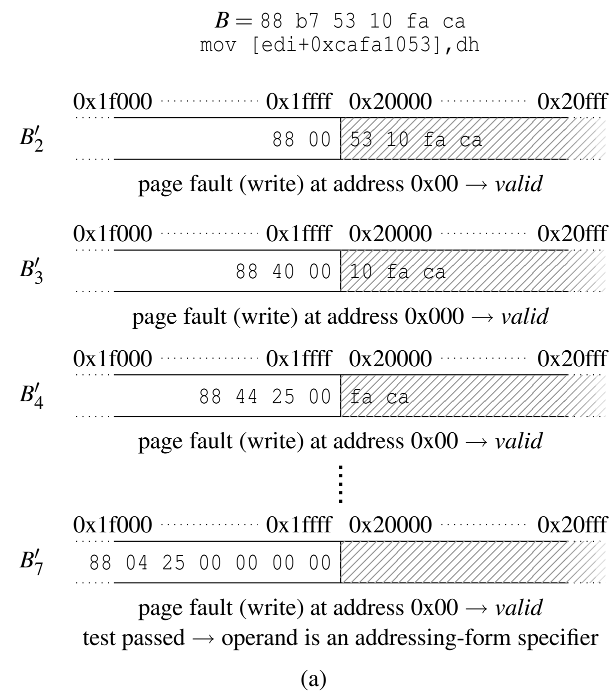
Once the decoder has found the length of an instruction, it tries to infer the type and the value of the non-implicit operands of the instruction (i.e., the operands that are not implicitly encoded in the opcode of the instruction). The technique used by our decoder to achieve this goal is an extension of the technique described in the previous paragraphs. Currently, our CPU-assisted decoder is capable of decoding addressing-form specifier operands and immediate operands.

Any Intel x86 instruction (Figure 1) is composed by an optional prefix, an opcode, and optional operands. To ease the presentation we assume instructions have no prefix; in practice, prefixes are detected using a white-list and considered part of the opcode. Given an instruction, encoded by the sequence of bytes $B = b_1 \dots b_n$, the format of operands is detected by performing a series of tests on some instructions derived by changing the bytes of B that follow the opcode and represent the operands of the instruction. If the opcode is j bytes long, the remaining $n-j$ bytes represent the operands, and the sequence of bytes encoding the instruction can be written as $B = b_1 \dots b_j b_{j+1} \dots b_n$.

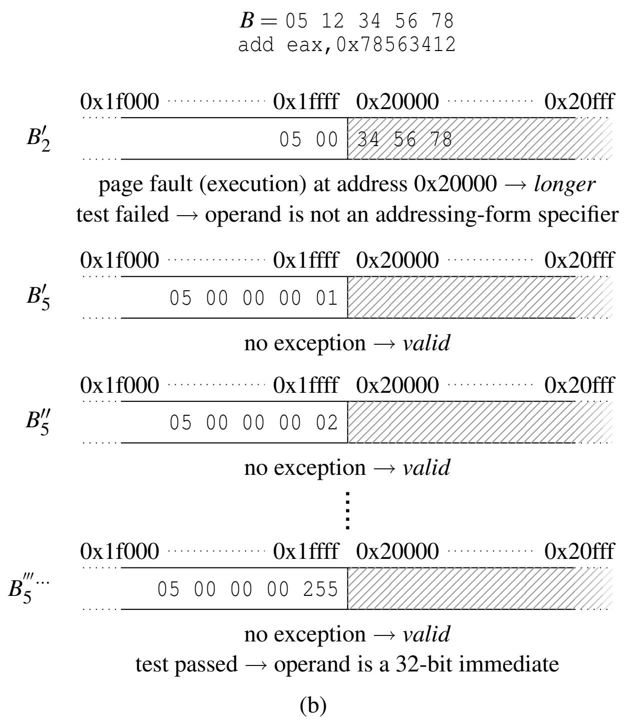
The assumption at the root of our algorithm is that each type of operand is encoded using a different encoding: immediate operands (*Imm*) are encoded as is (in little-endian), addressing-form specifier operands (*Addr*) are encoded using `ModR/M` and `SIB` encoding, and $Imm \cup Addr \neq Imm \cap Addr$. In other words, an immediate operand does not necessarily represent a valid addressing-form specifier operand, and vice versa. Therefore, given an instruction encoded as $B = b_1 \dots b_j b_{j+1} \dots b_n$, we expect a new sequence $B' = b_1 \dots b_j b'_{j+1} \dots b'_m$ to be valid if $b'_{j+1} \dots b'_m$ represents a new operand of the same type of $b_{j+1} \dots b_n$. Note that m and n can differ since operands of the same type can differ in length. Contrarily, we expect another sequence of bytes $\overline{B} = b_1 \dots b_j \overline{b_{j+1}} \dots \overline{b_m}$ to be invalid if $\overline{b_{j+1}} \dots \overline{b_m}$ represent an operand of a different type. Therefore, if an instruction with a j bytes long opcode has an immediate operand, the following holds:

$$\forall b'_{j+1} \dots b'_m \in Imm, B' = b_1 \dots b_j b'_{j+1} \dots b'_m \text{ is valid.}$$

In other words, the bytes following the opcode encode an immediate operand if the combination of the opcode with all the possible immediate operands gives always valid instructions. Fortunately, with few tests it is possible to estimate if the previous equation holds. In fact, it is sufficient to verify if it holds for a small number of operands in $Imm \setminus Addr$. The same applies for an instruction with an addressing-form specifier operand. Our current prototype of the decoder uses only five tests to detect addressing-form specifier operands and four tests to detect 32-bit immediate operands. Since



$B =$ 88 b7 53 10 fa ca
mov [edi+0xcafa1053],dh
0x1f000 0x1ffff 0x20000 0x20fff
B'_2 88 00 53 10 fa ca
page fault (write) at address 0x00 → *valid*
B'_3 88 40 00 10 fa ca
page fault (write) at address 0x000 → *valid*
B'_4 88 44 25 00 fa ca
page fault (write) at address 0x00 → *valid*
B'_7 88 04 25 00 00 00 00
page fault (write) at address 0x00 → *valid*
test passed → operand is an addressing-form specifier
(a)



$B =$ 05 12 34 56 78
add eax,0x78563412
0x1f000 0x1ffff 0x20000 0x20fff
B'_2 05 00 34 56 78
page fault (execution) at address 0x20000 → *longer*
test failed → operand is not an addressing-form specifier
B'_5 05 00 00 00 01
no exception → *valid*
B''_5 05 00 00 00 02
no exception → *valid*
$B'''^{\cdots}_5$ 05 00 00 00 255
no exception → *valid*
test passed → operand is a 32-bit immediate
(b)

Figure 4: Decoding of non-implicit operands using our CPU-assisted instruction decoder: instructions with (a) addressing-form specifier operand and (b) immediate operand.

the opcode can have a variable length (from one to three bytes), our CPU-assisted decoder performs the aforementioned tests with opcodes of incremental length (i.e., $j = 1 \ldots 3$).

Figure 4 shows some of the tests performed by our CPU-assisted instruction decoder to infer the format of the operands of two instructions: the first instruction has an addressing-form specifier operand and the second one a 32-bit immediate operand. For the first instruction, the decoder initially assumes that the opcode is one byte long, and performs the analysis of the remaining bytes to detect if they encode an addressing-form specifier operand. To do that it combines the opcode `88` with other valid addressing-form specifier operands of variable length, some of which cannot be interpreted as immediate operands. The first test consists in replacing the alleged operand with a single byte operand and in executing the resulting string. The CPU successfully executes the instruction. The same procedure is repeated with operands of different length (two, three, and seven bytes). All the sequences of bytes are found to encode valid instructions. Therefore, the input instruction is composed by a single byte opcode followed by an addressing-form specifier operand (`b7 53 10 fa ca` in the figure). The same procedure is applied also to the second instruction. The addressing-form specifier operand decoding fails, so the decoder attempts to verify whether the last four bytes of the instruction encode a 32-bit immediate. All tests performed are passed.

3.3 Evaluating Disassemblers Output

Starting from the normalized output of a set of off-the-shelf disassemblers and the information provided by our CPU-assisted instruction decoder, we have to infer which disassemblers gave correct results and which did not. Let $S = \{s_1, s_2, \ldots, s_{n-1}\}$ be the set of normalized disassembler outputs, where s_i is the output of the disassembler D_i. First, we compare the output of a disassembler, s_i, with the results of our CPU-assisted decoder and discard the former if it does not agree with the latter. The two outputs agree if: (I) both indicate that the sequence of byte does not encode a valid instruction or (II) both detect a valid instruction, the decoded instructions have the same length, and the types of non-implicit operands match. We denote with $\overline{S} \subseteq S$ the set of normalized disassemblers outputs compatible with the output of the CPU-assisted decoder.

When two off-the-shelf disassemblers agree with the CPU-assisted decoder and recognize a valid instruction, contradictions between their output might still be possible. In fact, there might exist two normalized assembly instructions $s_j, s_k \in \overline{S}$ such that $s_j \neq s_k$. To deal with disagreeing outputs, we associate to each output a coefficient of agreement. The rationale is to be more confident in the output with the highest agreement among disassemblers. We group disassemblers with equivalent output into equivalence classes. Given $s_i \in S$, its equivalence class is $[s_i] = \{s_j \in \overline{S} \mid s_i = s_j\}$. Then we can define the *coefficient of agreement* for s_i as $c(s_i) = |[s_i]|/|\overline{S}|$, where $c(s_i) \in [0,1]$ is directly proportional to the probability that s_i is correct. Note that if $s_i \in S \setminus \overline{S}$, then $c(s_i) = 0$.

As an example, consider the assembly instructions provided by the disassemblers in Figure 2. D_2 reports that the input sequence does not encode any valid instruction, but this result contradicts the CPU-assisted decoder; hence, $s_2 \notin \overline{S}$, and $c(s_2) = 0$. The other disassemblers instead correctly decoded the sequence of bytes as a valid six-byte instruction, thus $\overline{S} = \{s_1, s_3, s_4\}$. However, the normalized assembly representations s_3 and s_4 match, while s_2 differs. Their respective confidence indexes are $c(s_3) = c(s_4) = 0.66$, and $c(s_1) = 0.33$.

3.4 Exhaustive Input Generation

The completeness of the testing depends on the completeness of the inputs that are fed to the various disassemblers. To generate the strings of bytes, representing valid and invalid instructions, that we give in input to the disassemblers under analysis we use two strategies: *random input generation* and *CPU-assisted input generation*. Details about the two strategies are given in the next paragraphs.

3.4.1 Random Input Generation

The Intel x86 instruction set is very dense: a randomly generated string of bytes represents a valid instruction with a high-probability. In our experiments we have measured that about 75% of randomly generated strings can be executed by the CPU, and thus represent valid instructions. The advantage of using random input generation is that we can verify the ability of a disassembler at detecting invalid instructions and at decoding instructions with exotic combination of prefixes and operands (i.e., combinations that are very unlikely to be found in compiled programs).

3.4.2 CPU-assisted Input Generation

The format of Intel x86 instructions is very complex and it is not likely that a random input generator can cover the entire opcode space. For this reason we adopt a second strategy to generate inputs, based on our CPU-assisted decoder. With this approach we generate only valid instructions, avoid redundancy, and cover the entire instruction set of the architecture. The CPU-assisted decoder is used to enumerate the opcodes of the various instructions supported by the CPU.

We generate opcodes incrementally, by iterating over all possible strings up to three bytes. We assume each string is a valid opcode and we use the technique described in Section 3.2.2 to verify if the alleged opcode, when combined with a proper operand, encodes a valid instruction. In other words, we combine each alleged opcode with different operands to see if all combinations of the opcode with a particular type of operands yields always a valid instruction. When we find a valid instruction (and its format), we instantiate a small set of strings representing combinations of the current opcode with valid operands, and we try to prepend some possible combinations of prefixes. Operands are selected according to some heuristics developed to generate corner-case encodings. As an example, for an addressing-form specifier operand, our heuristics select operands obtained from multiple combinations of the `ModR/M`, `SIB`, and `Displacement` fields.

4. EVALUATION

In this section we present the results of the evaluation of our testing methodology with eight off-the-shelf disassemblers. The tested disassemblers are reported in Table 1, together with the release of the software used in our experiments, and the size of the disassembler in lines of code[1]. The release we tested correspond to the latest available at the time of writing the paper. Our results witness the effectiveness of the proposed testing methodology, and testify that the development of a perfect instruction decoder is a very challenging task. Indeed, we found bugs in each disassembler. Some of these bugs are very serious since they can be triggered even by code produced by standard compilers.

4.1 Evaluation of the CPU-assisted Decoder

The output of the CPU-assisted instruction decoder is assumed to be correct since it leverages the CPU to perform the decoding. Therefore, when the output of a disassembler is incompatible with the output of our decoder, we attribute this incompatibility to a bug in the former. Clearly we are making two strong assumptions: first, the output of our decoder is independent from the specific model of the CPU used during testing and, second, our decoder contains no bugs. We confirmed that both assumptions are correct through an extensive evaluation of our CPU-assisted decoder.

[1]Some of the products under testing (i.e., ndisasm, OllyDBG, and Google Native Client) include other functionalities besides instruction decoding. The numbers in the table are relative only to the size of the instruction decoding component.

Disassembler	Release	Lines of code
diStorm64 [7]	1.7.30	5,647
Ida Pro [14]	5.2	*n/a*
libopcode [11]	2008-11-27	13,559
Native Client [31]	2009-06-18	14,497
ndisasm [23]	2.05.01	10,221
OllyDBG [32]	2001	3,442
Udis86 [22]	1.7	6,560
XED2 [17]	2009-03-04	*n/a*

Table 1: Disassemblers evaluated

CPU	Supported features				
	MMX	SSE	SSE2	SSE3	SSE4
Intel P4 (3.0GHz)	✓	✓	✓		
Intel P3 (1.2GHz)	✓	✓			
Intel Core2 (2.0GHz)	✓	✓	✓	✓	
Intel Xeon (2.8GHz)	✓	✓	✓	✓	

Table 2: Intel x86 CPUs used to evaluate the CPU-assisted instruction decoder

To demonstrate that the output of our CPU-assisted instruction decoder does not change if the CPU on which it is executed changes, we compared the output of our decoder produced using four different CPUs. The CPUs we used for the evaluation are reported in Table 2. All CPUs were all used in 32-bit mode. We generated randomly 40,000 different 16 bytes long input strings and compared the output our CPU-assisted decoder produced on the four machines. With few exceptions, the output corresponded. We manually analyzed all the exceptions and verified that all involved instructions belonged to extended instruction sets not supported by all CPUs. For example, the string `0F 5A 5C BA 00` represents one of the aforementioned exceptions. The string encodes an SSE2 instruction which is not supported by one of the CPUs we used for the evaluation. In conclusion, if two CPUs support the same set of features, our CPU-assisted instruction decoder produces the same exact output.

To test the correctness of our CPU-assisted instruction decoder we converted each of the 40,000 input strings used for the previous experiment into a program that executed the first instruction encoded by the string through single stepping and returned zero, if the execution of the instruction resulted in an illegal instruction exception, or an integer greater then zero otherwise. The return value represented the number of successfully executed bytes, computed as the difference between the instruction pointer before and after the execution of the first instruction of the string. We executed all the generated programs and compared the output with the output produced by our CPU-assisted instruction decoder. We manually analyzed all the cases in which we observed disagreeing outputs and verified that all cases involved control transfer instructions (e.g., jumps, function calls, and returns). Indeed, control transfer instructions break the calculation of the instruction length because the instruction pointer after the control transfer does not necessarily point to the instruction following the previous one.

4.2 Evaluation of Off-the-shelf Disassemblers

Having verified that our CPU-assisted instruction decoder produced correct results, we performed the testing of the eight software disassemblers reported in Table 1. For our experiments, we employed

Disass.	**Over supported**	**Not supported** *Opc.*	*Instr.*	**Incorrect** *Opc.*	*Instr.*
diStorm64	10	209	1084	1	1
Ida Pro	461	5	12	49	283
libopcode	331	22	376	105	815
Native Client	479	54	534	133	8232
ndisasm	282	26	388	70	642
OllyDBG	484	136	515	26	176
Udis86	289	4	6	3	4
XED2	44	0	0	12	122

Table 3: Summary of the experimental results

the CPU with the most complete set of features available: an Intel Xeon (2.8GHz), supporting the instruction set extensions MMX, SSE, SSE2, and SSE3.

We generated about 60,000 different input strings: 40,000 were generated randomly and the remaining 20,000 string were selected randomly from those generated using the CPU-assisted technique described in Section 3.4. While the input strings generated by our CPU-assisted decoder varied in length and included only a single instruction, the strings generated randomly were all 16 bytes long and thus could contain more than one instruction. The whole testing process took about 15 hours, at an average rate of 1 second to execute all disassemblers over a single input.

Table 3 summarizes the results of our experimental evaluation. Defects are separated into three categories: (*over supported*) sequences of bytes that are considered valid instructions by the disassembler, but that are invalid on the physical CPU; (*not supported*) sequences that encode valid instructions for the physical processor, but are recognized as invalid by the disassembler; (*incorrect*) strings of bytes that correspond to valid instructions, but for which there exists another disassembler that produces an output with an higher coefficient of agreement[2]. For sequences classified as *not supported* or *incorrect* we report the number of inputs and unique opcodes that proved the defect. For *over supported* instructions we report only the number of inputs that witnessed the defect, as these sequences of bytes do not have a corresponding valid opcode. Note that some *over supported* instructions might include some of the 54 instructions belonging to the SSE4 extended instruction set, not supported by the CPU we used for the evaluation.

We randomly chose some of the instruction sequences that were reported to be incorrectly decoded and we manually inspected them. The manual verification confirmed the results of our testing methodology, with very few false positives due to some inaccuracies in our normalization rules. We believe the high number of defects we found testifies the effectiveness of our approach. None of the disassemblers was found to be bug-free. Overall, the most accurate disassembler was XED2; that is not surprising since it is developed by Intel. The worst was the disassembler part of the Google Native Client sandbox: we found that, for a large number of instructions, operands are not decoded properly. Considering the way the sandbox operates, we speculate that developers did not put too much effort in the decoding of operands, since only opcodes are essential for the purposes of the application.

The most recurring problem we observed involves the decoding of instructions with multiple prefixes; the tested disassemblers tend to consider instructions with exotic combination of prefixes valid, when they are not. The general purpose instructions are typically decoded correctly by all disassemblers; instead, instructions belonging to less commonly used instruction set extensions (e.g., floating-point instructions, MMX instructions, and SSE instructions) often present more problems. We also noticed that often disassemblers perform a partial interpretation of the semantics of the instruction. As an example, the semantics of some arithmetic instructions defines that immediate byte operands are sign-extended to double word size. Some disassemblers automatically perform this sign-extension. Similarly, some disassemblers omit instruction prefixes that are supposed to be ignored by the hardware CPU. Since the goal of a disassembler is to decode instructions and its output is used for many different purposes, such partial interpretation of the semantics of instructions can easily confuse users and thus should not be performed.

Table 4 reports some of the defects found in the tested disassemblers. For example, Udis86 did not recognize the sequence of bytes `db e0` as valid. Indeed, the Intel specification does not mention this instruction at all. However, this sequence was considered valid by the physical CPU and by the other disassemblers, that associated the sequence with the mnemonic opcode `fneni`. Ida Pro recognized the string `f6 5c 34 ae`, but incorrectly decoded the displacement field of the instruction. XED2 decoded the string `8e 0b` as the instruction `mov cs, word [ebx]`. Nevertheless, a side-note in the Intel specification precises that the code segment register cannot be loaded with a `mov` instruction, and any attempt to do so should result in an illegal instruction. Finally, the sequence `d4 cd` corresponds to the `aam` instruction, that has a 8-bit immediate operand; libopcode recognized the opcode correctly, but decoded the argument as a 32-bit value.

All the aforementioned defects could potentially be used by a programmer to obfuscate his code, by preventing disassembly, and impede reverse engineering attempts. Unfortunately, we observed that even a stream of instructions generated by a well-behaved compiler can trigger some of the bugs we found. At this aim, we disassembled all the instructions executed by Windows Media Player 9. To correctly identify the beginning of valid instructions, we executed the program in single-step mode, and recorded the sequence of 20 bytes pointed by the instruction pointer after each step. For each of the tested disassemblers, we were able to find some sequences of bytes that were not decoded correctly. For example, OllyDBG does not recognize the string `f3 90` (i.e., the `pause` instruction), libudis86 treats the string `db e2` as a `fclex` instruction instead of `fnclex`, and libopcode treats the string `66 9d` as two separate instructions instead of one (`popfw`).

5. RELATED WORK

Several pieces of research work focused on developing new disassembly algorithms to address the limitation of existing ones. The various algorithms proposed typically aim at improving the coverage of the program and to construct more complete control flow graphs in the presence of indirect control transfers and other subtle situations [5, 26, 9, 27]. Other research instead investigated techniques to obfuscate the machine code to impede disassembly and reverse-engineering [19] and others again proposed algorithms to disassembly obfuscated program [8, 18, 29].

The previous works most closely related to ours are probably DERIVE and BinREEF [10, 30]. DERIVE uses assemblers to infer information about how the operands of assembly instructions are encoded. The goal is exactly the opposite of the goal of our CPU-assisted instruction decoder. Indeed, our decoder aims at inferring the assembly instruction encoded by a particular sequence

[2]Note that we discarded all the inputs for which the highest coefficient of agreement of the outputs produced by the various disassemblers was smaller than 0.75. The rationale was to ignore inputs for which it was not possible to assess with enough confidence what was the correct result.

Disassembler	Input	Decoded instruction	Correct result
diStorm64	`26 59`	invalid instruction	`es pop ecx`
Ida Pro	`f6 5c 34 ae`	`neg [esp+esi+0x52]`	`neg [esp+esi-0x52]`
libopcode	`d4 cd`	`aam 0xffffffcd`	`aam 0xcd`
Native Client	`0f 21 83`	`mov dr0,ebx` (*7 bytes*)	`mov ebx,dr0`
ndisasm	`82 76 e5 dc`	invalid instruction	`xor byte [esi-0x1b],0xdc`
OllyDBG	`d9 7f d2`	`fstcw [edi-0x2e]`	`fnstcw [edi-0x2e]`
Udis86	`db e0`	invalid instruction	`fneni`
XED2	`8e 0b`	`mov cs, word [ebx]`	invalid instruction

Table 4: Examples of incorrectly decoded sequences of bytes

of bytes. BinREEF is a framework for evaluating decompilers, disassemblers, and other code obfuscation tools. Some of the experimental results described in their paper are quite qualitative, as they are strongly connected with the experience of the reverse engineer. Instead, in our work we propose a specific methodology to evaluate the correctness of instruction decoders, an essential component of disassemblers. Our results do not depend on the user's expertise, and testify the existence of real defects in the source code of the instruction decoder.

In [20] we presented EmuFuzzer, a testing methodology specific for CPU emulators, based on fuzzing. Using EmuFuzzer we have been able to find several defects in state-of-the-art IA-32 emulators. Some of these defects were concerning the instruction decoder embedded in the CPU emulator. As we already conjectured during Section 2, this fact testifies that any bug in the decoder might have cascading effects on the other components of the tool. Indeed, malware developers can equip their software with specific procedures, that exploit bugs in the instruction decoder of the emualtor, to detect when they are dynamically analyzed [25].

Finally, the idea of N-version disassembly is inspired by N-version programming [3] and N-variant systems [1, 6].

6. CONCLUSIONS

Disassemblers translate machine code into assembly instructions. Other than being used as stand-alone tools, they are often employed as the first link of a long chain of components that perform sophisticated analyses on machine code. In these situations, a bug in the disassembler would produce cascade effects on the subsequent modules. Disassembly is particularly challenging on a complex, CISC architecture such as Intel x86, where the huge number of instruction and many subtle details make instruction decoding a very complex task. In this paper, we described *N-version disassembly*, a fully automated methodology for testing the instruction decoder component of disassemblers. Our idea is to compare the output of *n* disassemblers against each other, where one of them is a special instruction decoder we developed, that leverages the physical CPU to provide accurate results, while the others are off-the-shelf disassemblers. We demonstrated the effectiveness of our methodology by evaluating it over 8 off-the-shelf disassemblers, finding multiple bugs in each of them. Malicious programmers can exploit these bugs to obfuscate their code and to thwart reverse engineering attempts. Moreover, some of the bugs we found can even be triggered by the code generated by standard compilers.

7. REFERENCES

[1] D. Bruschi, L. Cavallaro, and A. Lanzi. Diversified Process Replicae for Defeating Memory Error Exploits. In *3rd International Workshop on Information Assurance*. IEEE Computer Society, 2007.

[2] D. Chanet, B. De Sutter, B. De Bus, L. Van Put, and K. De Bosschere. Automated Reduction of the Memory Footprint of the Linux Kernel. *ACM Transactions on Embedded Computing Systems*, 6(4):23, 2007.

[3] L. Chen and A. Avizienis. N-Version Programming: A Fault-Tolerance Approach to Reliability of Software Operation. In *Proceedings of the Twenty-Fifth International Symposium on Fault-Tolerant Computing*, 1995.

[4] C. Cifuentes and M. V. Emmerik. Recovery of Jump Table Case Statements from Binary Code. *Science of Computer Programming*, 2001.

[5] C. Cifuentes and K. J. Gough. Decompilation of Binary Programs. *Software—Practice and Experience*, July 1995.

[6] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-tuong, and J. Hiser. N-Variant Systems: A Secretless Framework for Security through Diversity. In *Proceedings of the 15th USENIX Security Symposium*, 2006.

[7] G. Dabah. diStorm64. `http://ragestorm.net/distorm/`.

[8] M. Dalla Preda, M. Madou, K. De Bosschere, and R. Giacobazzi. Opaque Predicates Detection by Abstract Interpretation. In *Proceedings of the 1st International Workshop on Emerging Applications of Abstract Interpretation*, 2006.

[9] B. De Sutter, B. De Bus, K. De Bosschere, P. Keyngnaert, and B. Demoen. On the Static Analysis of Indirect Control Transfers in Binaries. In *Proceedings of the International Conference on Parallel and Distributed processing Techniques and Applications (PDPTA)*, 2000.

[10] D. R. Engler and W. C. Hsieh. DERIVE: A Tool That Automatically Reverse-Engineers Instruction Encodings. In *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Optimization (Dynamo)*, 2000.

[11] Free Software Foundation. GNU Binutils. `http://www.gnu.org/software/binutils/`.

[12] I. Guilfanov. Simplex method in IDA Pro, 2006. `http://www.hexblog.com/2006/06/simplex_method_in_ida_pro.html`.

[13] I. Guilfanov. Jump tables, 2008. `http://hexblog.com/2008/01/jump_tables.html`.

[14] Hex-Rays. IDA Pro. `http://www.hex-rays.com/idapro/`.

[15] R. N. Horspool and N. Marovac. An Approach to the Problem of Detranslation of Computer Programs. *The Computer Journal*, 23(3):223–229, 1980.

[16] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*, Nov. 2008. Instruction Set Reference.

[17] Intel Corporation. XED2. `http://www.pintool.org/`.

[18] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static Disassembly of Obfuscated Binaries. In *Proceedings of USENIX Security*, August 2004.

[19] C. Linn and S. Debray. Obfuscation of Executable Code to Improve Resistance to Static Disassembly. In *Proceedings of the 10th ACM conference on Computer and communications security (CCS)*, 2003.

[20] L. Martignoni, R. Paleari, G. Fresi Roglia, and D. Bruschi. Testing CPU emulators. In *Proceedings of the 2009 International Conference on Software Testing and Analysis (ISSTA), Chicago, Illinois, U.S.A.* ACM, July 2009. To appear.

[21] W. M. McKeeman. Differential Testing for Software. *Digital Technical Journal*, 10(1), 1998.

[22] V. Mohan. Udis86. `http://udis86.sourceforge.net/`.

[23] NASM Team. The netwide assembler. `http://www.nasm.us/`.

[24] G. C. Necula and P. Lee. Proof-carrying code. Technical Report CMU-CS-96-165, School of Computer Science, Carnegie Mellon University, Sept. 1996.

[25] R. Paleari, L. Martignoni, G. F. Roglia, and D. Bruschi. A fistful of red-pills: How to automatically generate procedures to detect CPU emulators. In *Proceedings of the 3rd USENIX Workshop on Offensive Technologies (WOOT), Montreal, Canada*. ACM, Aug. 2009.

[26] B. Schwarz, S. Debray, and G. Andrews. Disassembly of Executable Code Revisited. *Prooceedings of the Working Conference on Reverse Engineering (WCRE)*, 2002.

[27] H. Theiling and A. Angewandte. Extracting Safe and Precise Control Flow from Binaries. In *Proceedings of the 7th Conference on Real-Time Computing Systems and Applications*, 2000.

[28] J. Tröger and C. Cifuentes. Analysis of Virtual Method Invocation for Binary Translation. In *Proceeding of the 9th Working Conference on Reverse Engineering (WCRE)*, 2002.

[29] S. Udupa, S. Debray, and M. Madou. Deobfuscation: Reverse Engineering Obfuscated Code. In *Proceedings of the 12th Working Conference on Reverse Engineering*, 2005.

[30] L. Vinciguerra, L. Wills, N. Kejriwal, P. Martino, and R. Vinciguerra. An Experimentation Framework for Evaluating Disassembly and Decompilation Tools for C++ and Java. In *Proceedings of the 10th Working Conference on Reverse Engineering*, 2003.

[31] B. Yee, D. Sehr, G. Dardyk, B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. A sandbox for portable, untrusted x86 native code. In *IEEE Symposium on Security and Privacy*, 2009.

[32] O. Yuschuk. OllyDbg. `http://www.ollydbg.de/`.