



**HAL**  
open science

## At the bottom of binary analysis

Guillaume Bonfante, Alexandre Talon

► **To cite this version:**

Guillaume Bonfante, Alexandre Talon. At the bottom of binary analysis: instructions. 14th International Symposium on Foundations & Practice of Security, Dec 2021, Paris, France. 10.1007/978-3-031-08147-7\_21 . hal-03557004

**HAL Id: hal-03557004**

**<https://hal.univ-lorraine.fr/hal-03557004>**

Submitted on 4 Feb 2022

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# 1 At the bottom of binary analysis: instructions

2 Guillaume Bonfante<sup>1\*</sup> and Alexandre Talon<sup>2\*\*\*</sup>

3 <sup>1</sup> Université de Lorraine - LORIA

4 `guillaume.bonfante@loria.fr`

5 <sup>2</sup> G-SCOP, Univ Grenoble Alpes

6 `alexandre.talon@grenoble-inp.fr`

7 **Abstract.** Here we present a careful exploration of the set of instruc-  
8 tions for the x86 processor architecture. This is a preliminary step to-  
9 wards a systematic comparison of SMT-based retro-engineering tools.  
10 The latter arose in the context of binary code retro-engineering. All these  
11 tools rely themselves on more elementary disassembly tool. In this contri-  
12 bution, we attack the problem at its most atomic level: the instructions.  
13 We prepare, trading off between the size of the list and the correctness  
14 of the future comparison, a good list of instructions.

15 **Keywords:** Retro-engineering · Disassembly tools · x86.

## 16 1 Introduction

17 Deep analysis of binaries, and especially malwares, lead to many difficult ques-  
18 tions that are quite often undecidable. For instance, what is the value of `eax`  
19 when the program's control reaches instruction `jmp eax`? The reconstruction of  
20 a control flow graph (CFG), dead code identification, searching for buffer over-  
21 flow are other examples of the phenomenon. All these issues are not computable  
22 due to Rice's Theorem.

23 Nevertheless, these questions remaining open, researchers explored some par-  
24 tial solutions. There are many possibilities, exemplified by their respective un-  
25 derlying tools.

26 First, we must mention disassembly tools such as IDA<sup>3</sup>, Ghidra<sup>4</sup>, but also  
27 their more atomic versions: `capstone`<sup>5</sup>, `zydis`<sup>6</sup> or even `xed`<sup>7</sup>. All these tools  
28 are capable of extracting assembly code out of a binary executable or a piece  
29 of memory. Actually, for the first two tools of the list, they will perform some  
30 further analysis of the binary: they will rebuild – at least partially – the con-  
31 trol flow graph, function structures, virtual table, imports, export table, and so

\* Experiments have been conducted at LHS - LORIA

\*\* Supported by DGA - Direction Générale de l'Armement.

<sup>3</sup> <https://hex-rays.com/IDA-pro/>

<sup>4</sup> <https://ghidra-sre.org>

<sup>5</sup> <http://www.capstone-engine.org>

<sup>6</sup> <https://zydis.re>

<sup>7</sup> <https://intelxed.github.io>

32 on. All these tools start with the same step: they must extract and recognize  
 33 instructions from a sequence of bytes.

34

35 In another branch of retro-engineering tools, the idea is to provide a logical  
 36 model of the behavior of the processor so that the above-mentioned questions  
 37 can be reformulated in terms of logical formulae and solved using SMT solvers.  
 38 These are the original targets of the current study: the tools involving a logical  
 39 model of (the behavior of) the processor that is compatible with SMT solvers.  
 40 Let us call them SMT/symbolic tools.

41 It would be hard to mention all the contributions on solving retro-engineering  
 42 issues via SMT solvers, but let us mention a few of them and their associated  
 43 tools. Take for instance "Capture The Flag" style of issues, in [8], Springer and  
 44 Feng show how to use `angr` to find the user's input that will lead to the "wrong  
 45 path". To compute user data that will reach some particular point, `angr` (see [6])  
 46 performs some symbolic computation and solves the obtained constraints with  
 47 the help of an SMT solver engine. `Triton` (see [5]) is another example of such an  
 48 SMT/symbolic tool. In [4], Salwan, Bardin and Potet propose another typical  
 49 application. They show how to deobfuscate some virtualized code.

50 Again, on the problem of deobfuscation of virtualized code, we mention the  
 51 work of Souchet and Girault [7] based on `miasm`, another example of SMT/symbolic  
 52 tools. In their blog post, they describe how to cope with nanomites, an anti-  
 53 analysis trick that is used by the famous packer "Armadillo". Finally, the last  
 54 but not the least of the tools under our scope is `binsec`. In [2], Girol, Farinier  
 55 and Bardin consider another application of symbolic computations: bug tracking.  
 56 The tool `binsec` is at the core of their "robust reachability" exploration.

57 To sum up, all these tools need to describe symbolically the behavior of the  
 58 processor. To do that, they describe the semantics of each processor instruction.  
 59 However, the documentation by Intel [1] is quite huge and complex, thus prone  
 60 to errors which can be propagated to disassembly tools. In the long term, we  
 61 propose to build a platform to evaluate properly retro-engineering tools involv-  
 62 ing instruction semantics or syntax.

63

64 In principle, one has to run the tool on each instruction and then to verify the  
 65 correction of the tool on this instruction. However, that would mean we already  
 66 have access to the ground truth, that is to the actual semantics or syntax of the  
 67 instruction. But we do not. Thus, the idea of the platform is to perform a relative  
 68 verification rather than an absolute one. Given two tools with same purpose, we  
 69 propose a comparison between them. Our idea is that if they disagree on some  
 70 instruction, for sure *one* of them is wrong. If they agree, we may hope that  
 71 both are right. Such a hope should be stronger when the tools are developed  
 72 independently.

73 However, such an approach is unfeasible in practice if done naively. Indeed,  
 74 we are immediately overwhelmed by the number of processor instructions. The  
 75 first issue is that instructions may involve immediates, that is integers stored  
 76 within up to 32 bits. But, the number of registers (say around 20 for 64 bits x86

77 architectures) cannot be neglected since instructions may involve combinations  
 78 with three of them. To give an idea, extracting the SMT formulae for 10 000  
 79 instructions for two of the above mentioned SMT/symbolic tools, and then com-  
 80 paring the formulae with an SMT solver takes around 30 minutes (with obviously  
 81 some differences between the tools, but the order of magnitude is correct).

82 Thus, we have first turned our attention to an intermediate goal: grouping  
 83 instructions into broader classes. This is the topic of the present contribution.  
 84 Suppose that tools X and Y agree on `add eax, 0x1234`, they probably agree  
 85 on `add eax, 0x2345` too. That corresponds to the abstraction `add eax, imm`  
 86 where `imm` denotes some integer stored in two bytes. That forms a first class of  
 87 instructions. But we could go one step further in the abstraction by grouping all  
 88 instructions of the shape `add reg, imm` where `reg` represent a register and that  
 89 leads to a second (wider) class of instructions.

90 The more abstraction we perform, the more chances that the class is too  
 91 wide for our goal: two tools could agree on some instructions of the class and  
 92 disagree on others from the same class. In other words, there is a balance between  
 93 the level of abstraction and correction of the verification. We must choose a  
 94 compromise according to the situation: how long do the tests last. For instance,  
 95 disassembling 10 000 instructions with `capstone` takes 10ms. To conclude, the  
 96 level of abstraction is a parameter we can choose for our platform. The trade-off  
 97 we can obtain at apparently no cost on the precision is what we discuss here.

98 In practice, how to enumerate the instructions? We could follow Intel's doc-  
 99 umentation. But even a simple enumeration of instructions is not easy. This has  
 100 been already observed by Mahoney and McDonald, see [3] for a good presen-  
 101 tation of the problem (with a completely other purpose: steganography, that is  
 102 creating a valid executable used for hiding some information inside its instruc-  
 103 tion bytes). Furthermore, this work has actually already been done several times  
 104 before by disassembly tool such as `capstone` or `zydis`. They both extract in-  
 105 structions out of some buffer of bytes. Moreover, they both give informations  
 106 about the structure of instructions. So, we may think of using such tools to  
 107 enumerate instructions. As a by-product of instruction enumeration, we could  
 108 observe differences between two disassembly tools: `capstone` and `zydis`. Ensur-  
 109 ing that they recognize generally the same instructions is a ground to, in the  
 110 future, compare the semantics of the instructions.

## 111 2 Listing the instructions

112 In this section we first describe the structure of an instruction, then the differ-  
 113 ent types or arguments and the number of possibilities for each type. Since some  
 114 arguments can take a huge number of values, we present a way to abstract them  
 115 and give the reduced number of combinations, once we apply our optimizations.  
 116

117 An instruction is two-fold: it can be seen as a machine code (a sequence of  
 118 bytes), and in a more abstract way as an assembly instruction, which we define  
 119 below. Let us first note that different assembly instructions can correspond to the

120 same sequence of bytes: even if they are morally identical, `mov eax, [2*eax]`  
 121 and `mov eax, [2*eax+0]` correspond to the unique sequence "8B 04 45 00 00  
 122 00 00". The reverse is also true: "F2 90" and "90" both correspond to the `nop`  
 123 instruction. We will refer indifferently to an instruction by its byte sequence or  
 124 its assembly instruction.

125 At first sight, one may say that there are around five hundred different oper-  
 126 ations or so. This is the number of different opcodes. But we have to go further  
 127 in details: an instruction contains more than its opcode alone. much more details  
 128 than its opcode alone.

## 129 2.1 Structure of assembly instructions

130 An *instruction* can be decomposed in the following format<sup>8</sup>:

$$\begin{aligned} \text{instruction} &::= \text{prefix}_1 \dots \text{prefix}_k \text{ opcode } \text{arg}_1 \dots \text{arg}_l \\ \text{arg} &::= \text{imm} | \text{reg} | \text{mem} \\ \text{mem} &::= (\text{seg}[\text{reg}_1 + \text{scale} * \text{reg}_2 + \text{disp}], \text{size}) \\ \text{prefix} &::= \text{rep}, \text{lock}, \dots \\ \text{opcode} &::= \text{add}, \text{jmp}, \text{call}, \dots \\ \text{seg} &::= \text{cs}, \text{ds}, \text{es}, \dots \\ \text{reg} &::= \text{eax}, \text{ebx}, \dots \end{aligned}$$

131 where *imm* denotes some immediate (an integer on up to 32 bits), the same  
 132 goes for the displacement *disp*, and  $\text{scale} \in \{1, 2, 4, 8\}$ . In the clause defining the  
 133 memory argument above, the segment register *seg* and  $\text{scale} * \text{reg}_2$ , are optional.  
 134 At least one among the base register and the displacement must be present.

135 The number of different instructions is pretty huge: the set of  $2^{32}$  immediates  
 136 is already so large that calling an SMT solver for each instruction is infeasible.

## 137 2.2 Instructions enumeration

138 **Prefixes enumeration** Let us consider the leftmost part of instructions: the  
 139 prefix combinations. Prefixes usually change the semantics of the instruction: for  
 140 instance making it a loop (like the `rep` prefix, 0xF2 and 0xF3).

141 There are in total 13 prefixes, sorted into 5 categories. A priori, there is  
 142 no bound on the number of prefixes of an instruction, a prefix can be even  
 143 duplicated. The only limitation is actually the size of an instruction, that is 15.  
 144 Naively, leaving one byte for the opcode, that makes around  $13^{14} \approx 4 \cdot 10^{15} \approx 2^{52}$   
 145 possibilities. It is obvious that this number needs to be reduced.

146 However, according to the documentation, for each category, only the right-  
 147 most one (or the leftmost one, depending on the processor) will be effective. To  
 148 reduce the number of prefix combinations, we take advantage of the fact that in  
 149 practice, 1) only the last read prefix from each group is used and 2) the order of

<sup>8</sup> More technical details can be found in <http://ref.x86asm.net/coder32.html>.

150 prefixes between two groups is not important. So, we only allow instructions for  
 151 which there is at most one prefix per group and when the groups of the prefixes  
 152 respect some arbitrary fixed order. This parameter of our abstraction reduces  
 153 the number of prefixes combinations from  $10^{15}$  to 112.

154 Actually, there are on average fewer than 112 prefixes combinations per in-  
 155 struction because some prefixes can be illegal with some opcodes. For instance,  
 156 the lock prefix 0xF0 is illegal on many instructions such as F0 8B C0 = mov  
 157 **eax, eax**.

158 To reduce more the number of instructions, we could ignore useless combi-  
 159 nations of prefixes. For instance, the segment prefixes (0x2E, 0x3E, 0x26, 0x36,  
 160 0x64 and 0x65) have no effect when the function has no memory argument. 26  
 161 40 and 40 both mean **inc eax**. The same goes for the prefixes changing the  
 162 address or operand size (0x66 and 0x67) when non applicable. The REP prefixes  
 163 (0xF2 and 0xF3) have no effect on most instructions. We could skip "useless"  
 164 combination of prefixes: reject any sequence of bytes if its assembly version is  
 165 already in our list.

166 **Argument enumeration** Some opcodes have no argument, like **nop**, **ret** for  
 167 instance. Other generally have a source and a destination arguments. Some take  
 168 two registers, like **mov eax, ebx** or a register and a memory address like **mov**  
 169 **eax, [4\*ebx+ecx+1]**. Others may also take a constant like **add eax, 0x1742**.

170  
 171 Let us begin with the registers. An x86 processor has 8 general-purpose  
 172 registers plus others used in floating point arithmetic, and other specific registers  
 173 like **cr1**, **dr1**, **tr3**...

174 Many instructions expect their arguments in the so-called ModR/M format.  
 175 For better comprehension, we describe it here. The ModR/M encoding is stored  
 176 on one byte: three bits for the "reg" part which encodes which register is con-  
 177 cerned (as source or destination, depending on the opcode), two bits for the  
 178 "mod" and the remaining three bits for the "R/M" part. The "reg" part can  
 179 also designate 8-bit parts of the general registers according to the opcode: 83 C3  
 180 03 encodes **add ebx, 0x3** while 80 C3 03 means **add b1, 0x3** and the value of  
 181 the ModR/M byte is 0xC3 for both instructions. Also, the prefix 0x66 changes  
 182 the given register 32-bit register to its lower 16 bits: 89 D8 is **mov eax, ebx** and  
 183 66 89 D8 is **mov ax, bx**.

184 The "mod" + "R/M" parts combined designate either a register or a memory  
 185 address such that the corresponding memory cell is read or written to. While  
 186 some ModR/M gives addresses like [**edx**], others allow for other parameters: a  
 187 displacement (**disp**) and/or a SIB (scale base index). The **disp** is simply a con-  
 188 stant on 1, 2 or 4 bytes the value of which shifts the address, like [**edx + 0x1742**]  
 189 for a **disp** on two bytes. We can also have a **disp** alone like in [**0x17421742**],  
 190 with no register. Let us now describe the SIB. It encodes addresses of the shape  
 191 [**base + scale \* index**] where **base** and **index**, encoded on three bits, can be any  
 192 of the eight general-purpose registers (except for **esp** for the index). The scale  
 193 is encoded on two bits and can be equal to 1, 2, 4 or 8.

194 To sum up, the source and destination operands of an instruction are gener-  
 195 ally expressed using the ModR/M system, including sometimes a disp values,  
 196 and/or a SIB value. The ModR/M takes one byte, the SIB one byte, and the disp  
 197 can have a size of 1, 2 or 4 bytes. Assuming all ModR/M values expect a SIB  
 198 byte and a 32-bit displacement, we obtain  $256^{1+1+4} = 2^{48} \approx 2.8 \cdot 10^{14}$  possibili-  
 199 ties. With a careful analysis, taking into account forbidden patterns, we obtain  
 200 that there are around  $9 \cdot 10^{12}$  valid possibilities for the ModR/M+SIB+disp  
 201 combinations.

202 We could take the whole ModR/M+SIB/disp combination as parameter for  
 203 our abstraction, hence taking one representative out of  $9 \cdot 10^{12}$  for almost all  
 204 instructions using the ModR/M encoding. The ModR/M+SIB is an encoding,  
 205 hence not that straightforward. Moreover, some ModR/M values are illegal with  
 206 some opcodes but not others. Therefore, to keep a more simple code, and to avoid  
 207 accepting illegal values of ModR/M, we decided not to abstract the ModR/M  
 208 byte, nor the SIB one. However, we take the displacement as a parameter in our  
 209 abstraction: `add eax, [ebx+0x28]` and `add eax, [ebx+0x37]` are simply `add`  
 210 `eax, [ebx+imm8]` to us. The same applies respectively for the class of 16-bit  
 211 and 32-bit displacements. Using this abstraction we obtain 6376 different classes  
 212 (instead of  $9 \cdot 10^{12}$ ) of ModR/M+SIB+disp combinations of an instruction.

213  
 214 There remains a set of arguments to consider: the immediates. They appear  
 215 for instance in `mov ecx, 0x1234` and `add edx, 0xabcd1234`. We treat them as  
 216 for the case of displacements: we consider `add edx, 0xabcd1234` as `add edx,`  
 217 `imm32`. For a given instruction, each immediate on  $k$  bits can have  $2^k$  values,  
 218 so if a prefix+opcode combination expects a 32-bit immediate, thanks to this  
 219 abstraction we divide by  $2^{32} \approx 4 \cdot 10^9$  the number of instructions using this  
 220 combination.

221 **Illustration of the list** We give a small extract of our list of instructions in  
 222 Figure 1. We can see that on lines (2) to (5) the instruction includes some im-  
 223 mediates on 32 bits, and in (4) also a disp on 8 bits. Thanks to our abstractions,  
 224 we do not list the values for these constants. This allows us to save  $256^4$  lines  
 for (2) and (3),  $256^5$  for (4) and  $256^8$  for (5).

```
(1) 53                                     (push ebx)
(2) 8b 84 83 .. .. .. ..                 (mov eax,[ebx+eax*4+imm32])
(3) 8b 8c 85 .. .. .. ..                 (mov ecx,[ebp+eax*4+imm32])
(4) 69 40 .. .. .. .. ..                 (imul eax,[eax+disp8],imm32)
(5) 69 80 .. .. .. .. .. .. .. ..       (imul eax,[eax + imm32_1], imm32_2)
```

**Fig. 1.** A few instructions as they appear on our list. ”..” denotes any byte value.

225

226 **Final number of instructions** What we described above consists in finding  
 227 a compromise between enumerating all possible instructions to be tested on  
 228 disassembly tools, and reducing their number by keeping only one instruction by

229 equivalence class according to some parameter of abstraction. This is a trade-  
 230 off between precision and the final number of instructions we enumerate. We  
 231 believe, as we explained, that we kept the precision and obtained a reasonable  
 232 number of instructions with that in mind.

233 In the end, enumerating the opcodes along with ModR/M and SIB bytes,  
 234 and with valid prefixes in order, one per group, we obtain a total of 72 million  
 235 instructions (one representative per class), 72 174 844 to be precise. In this list,  
 236 we kept useless prefixes as long as they are in order and at most one per group.  
 237 We must specify that this list includes the deprecated AMD 3DNow! extension,  
 238 the VEX instructions but not the (too many) EVEX ones.

239 As we said at the beginning, our number of instructions depends on which  
 240 parameters we used for the abstraction. For instance, if we did not set a limit of  
 241 one prefix per group, we would have had a really huge number of instructions.  
 242 Then if we only limit at one prefix per group, we obtain  $6.8 \cdot 10^{24}$  instructions. If  
 243 we also forbid the prefix combinations which are the same up to a permutation,  
 244 this number goes down to  $3.0 \cdot 10^{24}$ . By keeping only one representative for all  
 245 disp values, we list  $2.6 \cdot 10^{16}$  instructions: a big improvement, but not enough.  
 246 The size of the list decreases to 72 million if we also abstract the immediates (our  
 247 final choice). We could then abstract the SIB values to obtain a smaller list of  
 248 262 000 instructions, or 23 700 if we also abstract the ModR/M, that is keeping  
 249 only the opcode and the one per group ordered prefixes combination. To sum  
 250 up, according to the needs and constraints, any parameter of the abstraction can  
 251 be switched on and off, resulting in a different number of instructions.

### 252 3 Representing the abstractions: the automata

253 Each time we want to compare two tools (e.g. `capstone` versus `zydis` or `binsec`  
 254 versus `triton`), we need to perform some specific instruction abstraction, that  
 255 is an abstraction for which the tools will answer in a coherent way for all the  
 256 elements of each class of instructions.

257 We may consider several abstractions. If we want to verify that `capstone` and  
 258 `zydis` agree on the size of the instructions, we may take one abstraction while  
 259 we may use another one if we want to verify that they agree on the registers  
 260 read or written. For the first case, we have no need to process the SIB of the  
 261 instruction whereas for the second case we need this information, resulting in  
 262 smaller equivalence classes.

#### 263 3.1 General automaton representation

264 We could represent our list of instructions as a list, but there is a much more  
 265 compact way of representing our abstractions: an automaton. Such an automaton  
 266 consists of a directed acyclic graph whose edges are labelled by bytes: it reads  
 267 a flow of bytes which corresponds to a path in the graph starting from the  
 268 root, until reaching a leaf with a value corresponding to the information of the  
 269 instruction. The automaton is more compact than the list. At least (but not



270 exclusively) for a same opcode we store once all the common prefixes when  
 271 building the automaton, hence saving some space. Section 3.2 illustrates this:  
 272 the **8b** first byte is shared between two instructions.

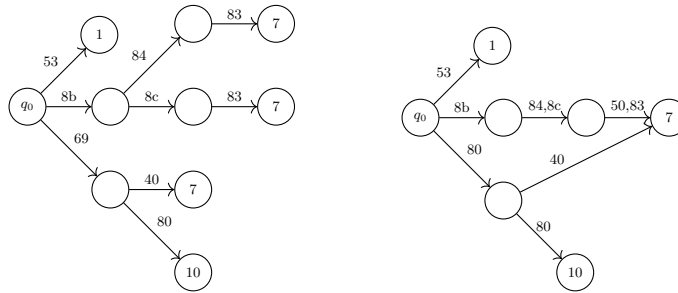
### 273 3.2 Examples of specific automata

274 We can also build more specific automata to obtain some disassembly infor-  
 275 mation like the size, the mnemonic, or the target information of a jump for  
 276 instance. Here we describe another way to perform the abstraction: set the goal  
 277 and automatize the creation of the classes.

278 Let us assume we want to know the size of any instruction. We propose  
 279 to first build, from any level of abstraction (so that the list of instruction is  
 280 reasonable), an automaton where each leave stores a integer between 0 (illegal  
 281 instruction) and 15: the size of the instruction read. We can imagine that from  
 282 72M instructions to only 16 different leaves, many subpaths will be shared by a  
 283 lot of instructions. For instance, **add** and **sub**, among many others, will have the  
 284 same size if given the same arguments.

285 Once the first version of the automaton is built, we automatically optimize it  
 286 by merging equivalent subarborescence, using classical algorithmic techniques.

287 We mention here another level of abstraction we can consider, which may  
 288 or may not be used depending if we want a full correctness or if we allow to  
 289 answer something when reading an invalid instruction. This level of abstraction  
 290 corresponds to considering that two nodes of the automaton are equivalent not  
 291 only if they have the same outgoing arcs, but also if they simply have the same  
 292 set of out-neighbors. For instance, if node  $u$  has three outgoing arcs towards  
 293 nodes  $u_1$ ,  $u_2$  and  $u_3$  and if node  $v$  has two outgoing arcs towards  $u_1$  and  $u_3$ , we  
 294 consider  $u$  and  $v$  to be equivalent. Doing this, we created some new paths from  
 295  $v$  to  $u_2$ , but any existing path is preserved, so that we answer correctly to the  
 296 instructions of our list. This allows us to obtain a smaller automaton, in terms  
 of number of nodes, with the drawback of not detecting invalid instructions.



**Fig. 2.** The automaton built from Figure 1 (left) and its optimized version (right)

297 We built, and optimized, the automata to provide the following sorts of  
 298 information: the size of the full instruction (and whether it is a valid instruction),  
 299

300 its mnemonic, its "type", the number of "operands", the type of each such  $i$ -  
 301 th "operand", and the information about jumps (including calls). For a jump  
 302 target, we give the position of its first byte in the instruction, its size, whether  
 303 the immediate is a signed or unsigned integer, and whether the jump is relative  
 304 or absolute. Here by operand we mean the operands with status explicit or  
 305 implicit in `zydis` (never or almost never the eflags for instance). For the  $i$ -th  
 306 operand automaton, we give the register, the size and position of first byte if it  
 307 is an immediate, or the memory address as some scale, index and base, plus some  
 308 displacement (size and position of first byte) if any. These automata enable us,  
 309 given a flow of bytes corresponding to instructions, to retrieve the useful data  
 310 for each instruction: we have the type, position and size of the operands so that  
 311 we can read the values of the immediates inside the instruction. So they can be  
 312 used together as a fast alternative to the disassembly tools.

313 The choice to have several automata and not a big one was made on purpose:  
 314 storing all the information in one automaton would be much bigger. Indeed, for  
 315 instance many instructions have `eax` as "first" operand, many have `ebx` as "sec-  
 316 ond" operand, but not so many have both `eax` and `ebx` in this order. A unique  
 317 automaton is a product of all the automata it combines, hence many more states.  
 318 As we explained earlier: the more different information we put in one automaton,  
 319 the smaller the equivalence class we obtain, hence a bigger automaton.

320

321 We obtain, using `Zydis` to build the list with the relevant data, the fol-  
 322 lowing information (number of different leaves values of each automata): 698  
 323 mnemonics, from 0 to 4 operands, a size from 0 (invalid instruction) to 14 for  
 324 the instructions, 27 different encodings for the target of a jump. We find also  
 325 respectively 32 523, 22 543, 35 and 5 different values for the first to the fourth  
 326 operands.

327 Apart from the number of leaves, *i.e.* the number of different values, the  
 328 number of internal nodes is also interesting. Compressing the automata as we  
 329 described earlier, we obtain 167 internal states for the mnemonic, 463 for the  
 330 instruction size, 99 for the number of operands, 39 for the jump information,  
 331 and respectively 701, 625, 95 and 22 for the information about the first to the  
 332 fourth operands. The level of compression of the automaton we can achieve also  
 333 gives us information about the complexity of the list of instructions according to  
 334 the parameter studied. It gives information about how some instructions share  
 335 a common subsequence as a suffix, with the same parameter value.

## 336 4 Concluding remarks

337 We introduced in this paper the notion of a list of instructions abstracted by  
 338 some parameters. We explain how to reach a sufficient level of abstraction to ob-  
 339 tain a list of 72 million (equivalence classes of) instructions. This list will be used  
 340 in a future paper to compare the semantics of instructions given by four tools:  
 341 `angr`, `binsec`, `miasm` and `triton`. In that paper, we will even go further, consid-  
 342 ering indirectly two instructions with the same assembly string to be equivalent.

343 We will also, in some way, consider `add eax, 0x17` and `add ebx, 0x17` to be  
 344 equivalent, by deducing the result of the second from the first one by replacing  
 345 all references to `eax` by references to `ebx`.

346  
 347 Using the automata we describe in section 3.2 in a malware detection tool,  
 348 we could observe a few differences between `capstone` and `zydis` even they most  
 349 generally agree. We found some instructions accepted by `capstone` and not by  
 350 `zydis`. One example is `8e 0f 10 2c`: `capstone` says it is a `mov cs, word ptr`  
 351 `[edi]` while `zydis` returns the error "bad register", meaning that the opcode  
 352 does not accept this value of ModR/M. We tested this instruction on our com-  
 353 puters to verify whether or not they raise some "Illegal instruction". On the  
 354 example above, the instruction is not valid, agreeing with `zydis`. However, one  
 355 of the issues we must face is the evolution of processors. Indeed, the set of in-  
 356 structions varies across time. It is not easy to delineate the "right" set of (x86)  
 357 instructions. Some instructions appear like the AMD 3DNow! in 1998 and then  
 358 become unsupported in 2010. Some specific instructions can disappear, for in-  
 359 stance there is no `push ds` in 64-bit architectures.

360 *Acknowledgments* We thank Fabrice Sabatier who gave us some examples of  
 361 "nasty" instructions.

## 362 References

- 363 1. Intel 64 and IA-32 Architectures Software Developer Manuals, [https://](https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html)  
 364 [software.intel.com/content/www/us/en/develop/articles/intel-sdm.html](https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html)
- 365 2. Girol, G., Farinier, B., Bardin, S.: Not all bugs are created equal, but robust reach-  
 366 ability can tell the difference. In: Computer Aided Verification 2021. LNCS, vol.  
 367 12759, pp. 669–693 (2021)
- 368 3. Mahoney, W., McDonald, J.T.: Enumerating x86-64 – it’s not as easy  
 369 as counting, [https://www.unomaha.edu/college-of-information-science-and-](https://www.unomaha.edu/college-of-information-science-and-technology/research-labs/_files/enumerating-x86-64-instructions.pdf)  
 370 [technology/research-labs/\\_files/enumerating-x86-64-instructions.pdf](https://www.unomaha.edu/college-of-information-science-and-technology/research-labs/_files/enumerating-x86-64-instructions.pdf)
- 371 4. Salwan, J., Bardin, S., Potet, M.: Symbolic deobfuscation: From virtualized code  
 372 back to the original. In: DIMVA 2018. LNCS, vol. 10885, pp. 372–392 (2018)
- 373 5. Saudel, F., Salwan, J.: Triton: A dynamic symbolic execution framework. In: Sym-  
 374 posium sur la sécurité des technologies de l’information et des communications. pp.  
 375 31–54 (2015)
- 376 6. Shoshitaishvili, Y., Wang, R., Salls, C., Stephens, N., Polino, M., Dutcher, A.,  
 377 Grosen, J., Feng, S., Hauser, C., Kruegel, C., Vigna, G.: SoK: (State of) The Art  
 378 of War: Offensive Techniques in Binary Analysis. In: IEEE Symposium on Security  
 379 and Privacy (2016)
- 380 7. Souchet, A., Girault, É.: Taming a wild nanomite-protected mips binary with  
 381 symbolic execution: No such crackme, [https://doar-e.github.io/blog/2014/](https://doar-e.github.io/blog/2014/10/11/taming-a-wild-nanomite-protected-mips-binary-with-symbolic-execution-no-such-crackme)  
 382 [10/11/taming-a-wild-nanomite-protected-mips-binary-with-symbolic-](https://doar-e.github.io/blog/2014/10/11/taming-a-wild-nanomite-protected-mips-binary-with-symbolic-execution-no-such-crackme)  
 383 [execution-no-such-crackme](https://doar-e.github.io/blog/2014/10/11/taming-a-wild-nanomite-protected-mips-binary-with-symbolic-execution-no-such-crackme)
- 384 8. Springer, J., chang Feng, W.: Teaching with angr: A symbolic execution curriculum  
 385 and CTF. In: USENIX Workshop ASE 18 (2018)