At the bottom of binary analysis

Guillaume Bonfante, Alexandre Talon

At the bottom of binary analysis: instructions

Guillaume Bonfante[1] ⋆ and Alexandre Talon[2] ⋆ ⋆⋆

[1] Université de Lorraine - LORIA
guillaume.bonfante@loria.fr
[2] G-SCOP, Univ Grenoble Alpes
alexandre.talon@grenoble-inp.fr

Abstract. Here we present a careful exploration of the set of instructions for the x86 processor architecture. This is a preliminary step towards a systematic comparison of SMT-based retro-engineering tools. The latter arose in the context of binary code retro-engineering. All these tools rely themselves on more elementary disassembly tool. In this contribution, we attack the problem at its most atomic level: the instructions. We prepare, trading off between the size of the list and the correctness of the future comparison, a good list of instructions.

Keywords: Retro-engineering · Disassembly tools · x86.

1 Introduction

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

Nevertheless, these questions remaining open, researchers explored some partial solutions. There are many possibilities, exemplified by their respective underlying tools.

First, we must mention disassembly tools such as `IDA`[3], `Ghidra`[4], but also their more atomic versions: `capstone`[5], `zydis`[6] or even `xed`[7]. All these tools are capable of extracting assembly code out of a binary executable or a piece of memory. Actually, for the first two tools of the list, they will perform some further analysis of the binary: they will rebuild – at least partially – the control 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.

[3] https://hex-rays.com/IDA-pro/
[4] https://ghidra-sre.org
[5] http://www.capstone-engine.org
[6] https://zydis.re
[7] https://intelxed.github.io

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

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

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

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

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

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

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

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

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

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

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

2 Listing the instructions

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

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

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

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

2.1 Structure of assembly instructions

An *instruction* can be decomposed in the following format[8]:

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

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

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

2.2 Instructions enumeration

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

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

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

[8] More technical details can be found in `http://ref.x86asm.net/coder32.html`.

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

Actually, there are on average fewer than 112 prefixes combinations per instruction because some prefixes can be illegal with some opcodes. For instance, the lock prefix 0xF0 is illegal on many instructions such as `F0 8B C0` = `mov eax, eax`.

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

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

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

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

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

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

We could take the whole ModR/M+SIB/disp combination as parameter for our abstraction, hence taking one representative out of $9 \cdot 10^{12}$ for almost all instructions using the ModR/M encoding. The ModR/M+SIB is an encoding, hence not that straightforward. Moreover, some ModR/M values are illegal with some opcodes but not others. Therefore, to keep a more simple code, and to avoid accepting illegal values of ModR/M, we decided not to abstract the ModR/M byte, nor the SIB one. However, we take the displacement as a parameter in our abstraction: `add eax, [ebx+0x28]` and `add eax, [ebx+0x37]` are simply `add eax, [ebx+`*imm*`8]` to us. The same applies respectively for the class of 16-bit and 32-bit displacements. Using this abstraction we obtain 6376 different classes (instead of $9 \cdot 10^{12}$) of ModR/M+SIB+disp combinations of an instruction.

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

Illustration of the list We give a small extract of our list of instructions in Figure 1. We can see that on lines (2) to (5) the instruction includes some immediates on 32 bits, and in (4) also a disp on 8 bits. Thanks to our abstractions, 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.

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

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

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

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

3 Representing the abstractions: the automata

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

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

3.1 General automaton representation

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

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

3.2 Examples of specific automata

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

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

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

We mention here another level of abstraction we can consider, which may or may not be used depending if we want a full correctness or if we allow to answer something when reading an invalid instruction. This level of abstraction corresponds to considering that two nodes of the automaton are equivalent not only if the have the same outgoing arcs, but also if they simply have the same set of out-neighbors. For instance, if node u has three outgoing arcs towards nodes u_1, u_2 and u_3 and if node v has two outgoing arcs towards u_1 and u_3, we consider u and v to be equivalent. Doing this, we created some new paths from v to u_2, but any existing path is preserved, so that we answer correctly to the 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.

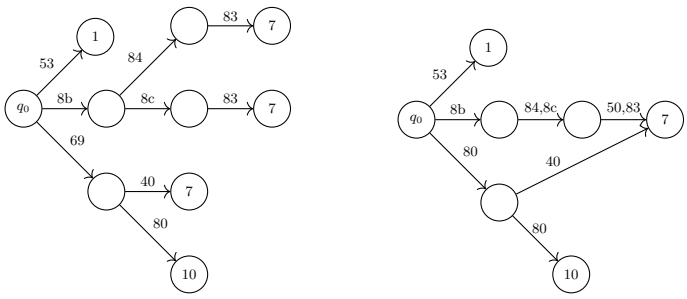

Fig. 2. The automaton built from Figure 1 (left) and its optimized version (right)

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

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

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

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

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

4 Concluding remarks

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

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

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

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

References

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