Symbolic Execution using Angr

Bala Murli Krishnan Muthu[1,2][20936500], Falgun Meshram[1,3][20958246], and Vijiithaa Sasidharan[1,4][20969795]

[1] Masters of Engineering, University of Waterloo
[2] `bmkmuthu@uwaterloo.ca`
[3] `fdmeshra@uwaterloo.ca`
[4] `vsasidha@uwaterloo.ca`

Abstract. Software nowadays has become more complicated, and security flaws and vulnerabilities are more likely to occur. Therefore, there is a need for more effective and superior methods of analyzing binaries and locating the faults associated with them. Though there is plenty of binary analysis tool available, testing and applying the specific tool that will satisfy the requirements has become a difficult job. Binary code vulnerability discovery and exploitation is also a challenging process. Scalable analysis of program attributes is also difficult due to the lack of high-level and semantically free information on the code structure. In this project, binary analysis of one of the widely known Linux tool "pwd" library has been implemented with the help of a binary analysis tool called "angr"[1] as it combines both symbolic[2] and concolic execution analysis.

Keywords: Symbolic Execution · CFG · Angr.

1 Introduction

Software companies currently invest a significant amount of time and effort in testing software to ascertain if it behaves appropriately and to identify and remove defects. There are considerable challenges in developing a set of test cases that spans the complete source code and in detecting inputs that lead to hard-to-trigger corner case failures. Symbolic Execution comes to the rescue over there and is a technique for analyzing a program to identify the inputs that trigger the execution of each component.[3] It is frequently used in software testing to explore as many possible program paths in a fixed amount of time, generate a set of concrete input values to analyze and identify each path, and detect a variety of errors, such as assertion violations, uncaught exceptions, security flaws, and memory corruption.

Symbolic execution is capable of exploring multiple paths simultaneously with different test input values. The main component of symbolic execution is for the program to consider symbolic values and verify the program. It follows two methodologies for each explored control flow path: (i) a first-order Boolean formula that describes the conditions met by the branches taken along that path, and (ii) a symbolic memory that converts variables to symbolic expressions or values. To determine whether there are any violations of the property along each explored path and whether the path itself is realistic, whether its formula can be satisfied by some assignment of concrete values to the program's symbolic arguments—a model checker, typically based on an SMT solver (satisfiability modulo theories) solver is used.

On the other hand, Concolic testing or DART(Directed Automated Random Testing)[4] is a hybrid approach to software verification that carries out symbolic execution in a way that interprets program variables as symbolic variables along with a concrete execution route. The technique is to detect and resolve vulnerabilities and locate and take advantage of them. Several tools from both academic institutions and researchers have used dynamic symbolic execution. One of the prominent tools discussed in this project is "angr"[1]. Angr is an open-source binary analysis tool for python and it is flexible, simple to use, multi-architecture, and cross-platform. This tool is developed by researchers at the University Of California, Santa Barbara from the Computer Security Lab.

2 Design Decisions

2.1 Control Flow Graph:

The analysis that can be carried out on a binary is a Control Flow Graph. In angr, CFG can be performed using 2 different angr methods: CFGFast and CFGAccurate. The functionality of both the methods are the same but it is used under different scenarios. In order to construct a graph of the binary more rapidly, CFGFast will make assumptions about the binary and is more powerful for well-formed binaries. On the other hand, CFG Accurate is used for binaries that are not well-formed and it will explore the whole binary in order to build a graph without making any assumptions. Compared to CFGFast, this approach is slower but more precise. Both the function manager developed by creating the CFG and the visual analysis of the graphs can be done using the CFGs produced by these two approaches.

2.2 Backward Slicing:

A backward slice indicates the statements can have an impact on how one statement is computed. It is a compilable and executable version of the original software with certain components missing. The computation of backward slicing can be easily verified with CFG. It reveals how the statements can have an impact on the attribute's value. The program builds a backward slice from a target, and all data flows in this slice arrive at the target. BackwardSlice, a built-in analysis in angr, creates a backward program slice.

2.3 Function Identifier:

The identifier locates common library functions in CGC binaries by using test cases. It prefilters by gathering some fundamental data about stack arguments and variables.

The following design decision have been made to explore the PWD's binary file and they are discussed briefly. CFG generated from the main function of pwd binary helped in the understanding of the paths and the available functions. For an instance, consider getopt_long() from the code, which is used to fetch the value of the argument and is stored in the local variable "c". Since the pwd binary was fully tested and not many of the bugs are exposed, we tried to check whether the value stored in the variable c is changed in the middle before the switch statement. The CFG gave an idea about the flow of execution and the path that program could take if the value of "c" is changed.

Consider another method "xgetcwd()" which is used to fetch the current working directory. Here, the output value from the function is stored in the variable "wd". The main functionality of this function is to ensure that the returned path is not NULL. Since there are multiple sub-function being invoked such as "puts(wd)" and "free(wd)" inside the body of the function, where the value can be set to NULL by pointer "buf". Hence to ensure that the value is not set to NULL in between the execution, an "IF" statement has been added to ensure its flow. This helped in ensuring that the function behaves as appropriate and CFG paved its pathway for a better understanding.

Let's look at a different method called "opendir("..")," which takes a path to the directory that has to be opened. The value of this function is stored in a pointer "check". This variable is evaluated against the NULL value, to make sure that the function behaves as per the expectation. Since "pwd" library is designed in a way that there are minuscule ways of discovering faults in the code, the execution of the function has been validated with the help of "IF" statement for symbolic verification.

3 Implementation

Angr tool is supported for Python3. It is highly recommended to install "angr" libraries in a python virtual environment as some of the libraries such as "z3, pyvex" require native library code which is forked from the original source library code. Suppose in case, these libraries are already installed in the system. In that case, there might be scenarios of over-writing the original libraries and there is no support from the official "angr" team for the problems arisen out of installing "angr" outside of the virtual environment.

Even though angr has cross compatibility, it has full support only on X86 architecture machines. A significant drawback faced while experimenting with this project in MAC M1 - ARM architecture was since this binary is compiled on the MAC system, angr wasn't able to map the MAC binaries to the project created for symbolic execution. Hence, to navigate from this scenario the binaries used in this project were compiled on Linux machines to eliminate any cross-compatibility concerns.

3.1 Installation in Mac OS:

Angr in Mac OS can be installed using the command "pip install angr" and there are some shortcomings to this installation. Despite binary distributions (”wheels”) existing on other platforms, angr needs the unicorn library, which pip needs to build from source on macOS. Because Python 2 is required for building unicorns from the source, installation in a virtual environment would fail as it tries to fetch python3. Hence an approach would be to Install unicorn independently and point it to your Python 2 installation if "pip install angr" fails for you.

As discussed above we were having trouble compiling source code in ARM architecture and using it for symbolic execution in angr. We had decided to use one of the GNU coreutils, namely pwd, to carry out symbolic execution. To get the source code for the same we cloned the git repo : git@github.com: coreutils/coreutils.git. This contained the source code for all GNU coreutils including pwd. Each of these source codes had a lot of dependencies contained inside it and therefore compiling these c files was not an easy task.

We followed the following steps to compile the required binary:

1] $ git submodule foreach git pull origin master
This pulled the latest files from upstream

2] $ export GNULIB_SRCDIR=/path/to/gnulib
This is an optional step to reduce download time if data is already in the cache.

3] $ ./bootstrap
The next step is to get and check other files needed to build, which are extracted from other source packages.

4] $ ./configure –quiet #[–disable-gcc-warnings] [*]
$ make
make command results in the generation of the binary required.

Once these steps were followed whenever any changes are made to pwd, we simply ran the ”make” command to get the updated binary. In this project, we have carried out three types of verification in the pwd source code which are elaborated as follows and the respective outcomes are described in a later section:

1] The source code consists of this snippet:

```
DIR *dirp, *check;
dirp = opendir ("..");
if (dirp == NULL)
  die (EXIT_FAILURE, errno, _("cannot open directory %s"),
       quote (nth_parent (parent_height)));
```

Our aim was to verify that if at the beginning of the execution the code opendir returned a non NULL value then during the consequent run it would not arbitrarily change to NULL. To accomplish this using angr we added two constraints, the first of which is checking that the return value of opendir at the beginning of the program is not NULL and then questioning angr if it is possible for opendir to return NULL later in the same run.

2] We then verified this snippet:

```
if (wd != NULL)
    {
      puts (wd);
      free (wd);
    }
```

Here our aim was to verify that the function free does not get called with a NULL pointer as that would be a memory error. A similar approach discussed above was used to verify the same.

3] This is the final snippet that we verified:

```
switch (c)
    {
    case 'L':
      logical = true;
      break;
    case 'P':
      logical = false;
      break;

    case_GETOPT_HELP_CHAR;

    case_GETOPT_VERSION_CHAR (PROGRAM_NAME, AUTHORS);

    default:
      usage (EXIT_FAILURE);
    }
```

Here the aim is to verify that there are no inputs that can make the passed command line arguments get modified in the course of their execution. A similar approach as discussed above was utilized to verify this snippet of code.

4 Theoretical Foundation

The angr is very versatile with the symbolic execution and the stages of the symbolic execution[5] can be described with the simple stages as shown in the figure 1, loading the binary file to the angr project with the required parameters, presetting the program state variable to begin the symbolic execution, starting to create the paths and exploring the paths based on the program state. While

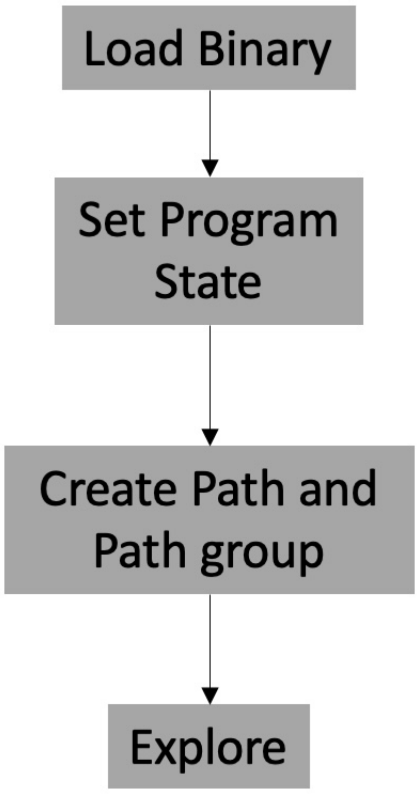


Fig. 1. Stages of Symbolic execution in Angr.

loading the binary, angr interprets the binary file and makes it used with the angr's project class. The project class has much useful information like the entry state in which the angr needs to process, the architecture information, and similar. Also, the binary file can be loaded with explicit rules on the library files, some external libraries can be added or a few libraries can be excluded while loading. The entry base address of the binary can also be passed to the project class so that the angr would start the exploring stage from the base address. The project class would create a symbolic state tracker, which is used by angr to keep track of the state values in the different paths.

Every different path on the binary would have its own angr's symbolic state. The symbolic state tracker keeps hold of the information like the variable, values, registers, memory layout, symbolic values, and symbolic constraints. When the path is created, for each step the symbolic state tracker is updated for the specific path constraints, which are later evaluated for the feasibility of the overall path. Angr would categorize each path into the following three types dead end, active, and error. Whenever the path is causing a violation on the path along with the constraints, when the angr is still processing the path and not yet finished, the path is placed in the active stage. When the execution is fully completed without any errors, then the path is stored in the dead end. Also if any path satisfies the given initial constraints and angr finds a path satisfying the constraints, then the path is stored in the found category, also angr stops the exploring phase when one path is found that satisfies the constraints, after which it stops exploring the other paths.

Angr is most used in the field of finding the bugs and vulnerabilities of the software code[6], one such field is CTF(Capture The Flag) competitions. In [7], a step-by-step approach to decoding the flags from the binary files is given.

5 Gap Analysis

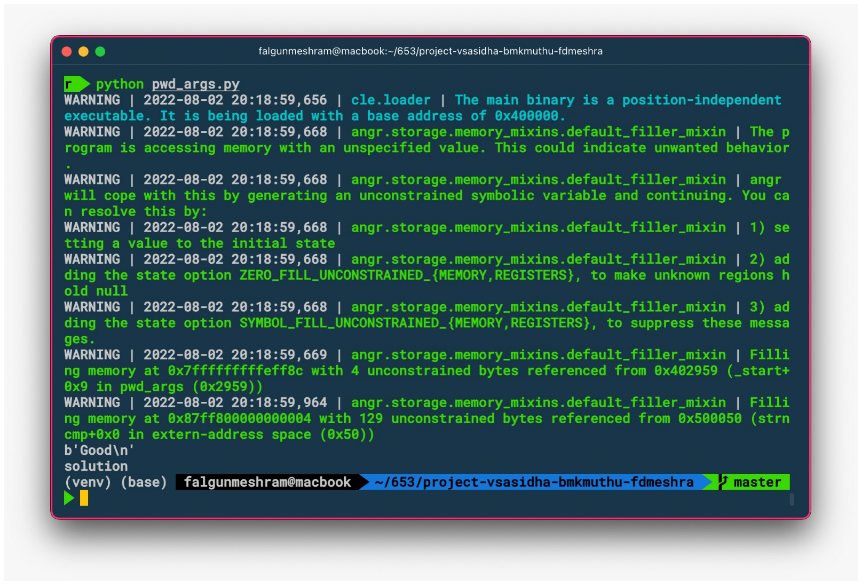
The Angr was designed to symbolically find the path based on the input state variables and change the state of the input variables to find a new path and mark it as a dead end or active or error path. This gave the users some control on adding constraints on the path and finding bugs[8] or finding different possible paths. Initially, angr was used to analyze the code, a basic code was used to analyze the explore class of angr. Here is the initial code snippet that was used to find the path which would lead the angr to find “Good” on the stdout, the output is shown in Figure 2.

```
int main(int argc, char* argv[]) {
printf("Good Job");
exit(0);
}
```

The python code used to find the path that contains the statement ”Good” is shown below:

```
import angr

#The actual path to the binary executable
path_to_binary = "???"
project = angr.Project(path_to_binary)
main = project.loader.main_object.get_symbol("main")
initial_state = project.factory.
        blank_state(addr=main.rebased_addr)
simulation = project.factory.simgr(initial_state)
simulation.explore(find=lambda s:
        b"Good" in s.posix.dumps(1))
if simulation.found:
  s = simulation.found[0]
  print(s.posix.dumps(1))
  solution_state = simulation.found[0]
  print(solution_state.posix.dumps(
        sys.stdin.fileno()).decode())
else:
  raise Exception('Could not find the solution')
```

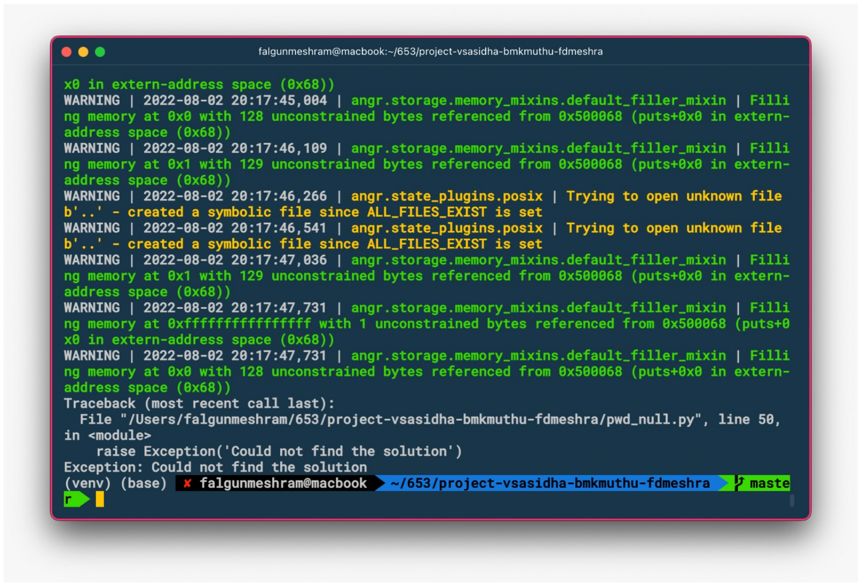

falgunmeshram@macbook:~/653/project-vsasidha-bmkmuthu-fdmeshra

```
r python pwd_args.py
WARNING | 2022-08-02 20:18:59,656 | cle.loader | The main binary is a position-independent
executable. It is being loaded with a base address of 0x400000.
WARNING | 2022-08-02 20:18:59,668 | angr.storage.memory_mixins.default_filler_mixin | The p
rogram is accessing memory with an unspecified value. This could indicate unwanted behavior
.
WARNING | 2022-08-02 20:18:59,668 | angr.storage.memory_mixins.default_filler_mixin | angr
will cope with this by generating an unconstrained symbolic variable and continuing. You ca
n resolve this by:
WARNING | 2022-08-02 20:18:59,668 | angr.storage.memory_mixins.default_filler_mixin | 1) se
tting a value to the initial state
WARNING | 2022-08-02 20:18:59,668 | angr.storage.memory_mixins.default_filler_mixin | 2) ad
ding the state option ZERO_FILL_UNCONSTRAINED_{MEMORY,REGISTERS}, to make unknown regions h
old null
WARNING | 2022-08-02 20:18:59,668 | angr.storage.memory_mixins.default_filler_mixin | 3) ad
ding the state option SYMBOL_FILL_UNCONSTRAINED_{MEMORY,REGISTERS}, to suppress these messa
ges.
WARNING | 2022-08-02 20:18:59,669 | angr.storage.memory_mixins.default_filler_mixin | Filli
ng memory at 0x7ffffffffeff8c with 4 unconstrained bytes referenced from 0x402959 (_start+
0x9 in pwd_args (0x2959))
WARNING | 2022-08-02 20:18:59,964 | angr.storage.memory_mixins.default_filler_mixin | Filli
ng memory at 0x87ff800000000004 with 129 unconstrained bytes referenced from 0x500050 (strn
cmp+0x0 in extern-address space (0x50))
b'Good\n'
solution
(venv) (base)  falgunmeshram@macbook  ~/653/project-vsasidha-bmkmuthu-fdmeshra  master
```

Fig. 2. Output when the solver engine found the path containing "Good"

With the same approach, the angr was tested for the simple if statement with the following code. The code gets the input integer value and stores it in the variable a. And then if the value of the variable a is 10, then the intended output is printed. By using the same python code, as shown above, the angr needs to find the path which leads to ”Good”. This can only happen when the input value of a is equal to 10. Here the constraint (a==10) is forced to the symbolic execution to find the path.

```
int main(int argc, char* argv[]) {
int a = 0;
scanf("%d",&a);
if(a==10)
printf("Good Job");
exit(0);
}
```



```
falgunmeshram@macbook:~/653/project-vsasidha-bmkmuthu-fdmeshra
x0 in extern-address space (0x68))
WARNING | 2022-08-02 20:17:45,004 | angr.storage.memory_mixins.default_filler_mixin | Filli
ng memory at 0x0 with 128 unconstrained bytes referenced from 0x500068 (puts+0x0 in extern-
address space (0x68))
WARNING | 2022-08-02 20:17:46,109 | angr.storage.memory_mixins.default_filler_mixin | Filli
ng memory at 0x1 with 129 unconstrained bytes referenced from 0x500068 (puts+0x0 in extern-
address space (0x68))
WARNING | 2022-08-02 20:17:46,266 | angr.state_plugins.posix | Trying to open unknown file
b'..' - created a symbolic file since ALL_FILES_EXIST is set
WARNING | 2022-08-02 20:17:46,541 | angr.state_plugins.posix | Trying to open unknown file
b'..' - created a symbolic file since ALL_FILES_EXIST is set
WARNING | 2022-08-02 20:17:47,036 | angr.storage.memory_mixins.default_filler_mixin | Filli
ng memory at 0x1 with 129 unconstrained bytes referenced from 0x500068 (puts+0x0 in extern-
address space (0x68))
WARNING | 2022-08-02 20:17:47,731 | angr.storage.memory_mixins.default_filler_mixin | Filli
ng memory at 0xffffffffffffffff with 1 unconstrained bytes referenced from 0x500068 (puts+0
x0 in extern-address space (0x68))
WARNING | 2022-08-02 20:17:47,731 | angr.storage.memory_mixins.default_filler_mixin | Filli
ng memory at 0x0 with 128 unconstrained bytes referenced from 0x500068 (puts+0x0 in extern-
address space (0x68))
Traceback (most recent call last):
  File "/Users/falgunmeshram/653/project-vsasidha-bmkmuthu-fdmeshra/pwd_null.py", line 50,
in <module>
    raise Exception('Could not find the solution')
Exception: Could not find the solution
(venv) (base)  ✘ falgunmeshram@macbook  ~/653/project-vsasidha-bmkmuthu-fdmeshra   maste
r 
```

Fig. 3. Output when the solver engine cannot find the path containing ”Good”

By adding the constraints to the symbolic execution engine of angr, the following code was tested with a newer approach of the python script. The path would be successful when the ”Good” is printed to the stdout, as well the path will be aborted once the ”Try” is printed to the stdout. This was done using the two-parameter while calling the explore class of angr. The find parameter is used to find the path that prints ”Good” and the avoid parameter is used to abort that particular path found which prints ”Try” on the stdout. For the below example, the constraint enforced was $(b < 12) \wedge (b > 10 \wedge b < 20)$, so the value of a is independent of this example and the value of b would be 11. Angr was able to find the input value for b successfully by exploring the possible paths. When the successful path is not found, the solver engine throws a runtime error as shown in Figure 3

```
int main(int argc, char* argv[]) {
int a = 0;
int b = 0;
scanf("%d %d",&a,&b);
a=1;
if (b>=12)
printf("Try\n");
while(b>10 && b<20){
b++;
a=10;
}
if(a==10)
printf("Good Job\n");
exit(0);
}
```

The other way to set the input values for the variable is using the memory address. The angr can be fed with the actual address of all the constraint variables. But finding the address of the variable is difficult and changes across the platform. The executable code from the c file was generated on the Linux machine and the angr was used in a mac machine so the variable address found on the Linux machine would not match the address of the other machine. But when the input variable is defined in such a way it is a user input like using scanf, then the code does not require to be hard coded with the memory address of the variable, which might not work on another machine. The same was used with analyzing the pwd file.

6 Outcome

In this project, the different modules and classes of the symbolic execution tool angr were explored with the help of pwd binary executable of Linux tools. During the initial stages of learning the angr, the tool was well designed to perform the basic operations of the binary. From viewing the top-level interfaces on the binary file, the angr has many options on the project class, which was handy to know the information about the binary executable, if the binary is an unknown file. Also loading the binary to the angr using the project class was straightforward, the project class did lots of analyses on the binary file and the interface was also kept clean.

To explore the binary file in an efficient way angr provides the solver engine which evaluates the constraints and outputs the solutions. The angr provides tools to search and execute the binary file using the simulation manager. The control flow graph analysis was a very helpful tool that angr comes with and in a few scenarios, the angr took too much time to solve the overall graph. It was noticed that the z3 engine was used in the background by angr to solve the paths. When the path was solved the angr approached it with the three states as mentioned before and the three states were evaluated by simply checking with all the constraints along the path lead to satisfiable assignments or not, if not the path is considered unsatisfiable. By using the control flow graph produced by the angr, a particular function could be targetted and constraints on that function invocation can be assumed, then let the simulation manager find the solution or any possible path to the function. The graph gave an idea to navigate within the lengthy code and focus on the area of the test and enable to hook the function with the address.

7 Conclusion

In this project, PWD's source code was used as a testing file to perform symbolic execution, and finding potentially fatal bug was very low as the GNU source codes have been heavily tested by the open source community with a very minimal bugs and vulnerability. Therefore, the focus was on verifying that certain bugs would not be present in the source code as discussed in the above sections using symbolic execution, rather than focusing on finding a new bug. In this project, the three code snippets from pwd were successfully verified by using Angr , and the output results are attached to the git repository. The overall Control flow graph of the pwd was also attached to the git repository, which happened to be one of the useful resource to explore the pwd source code by exploring the binary.

References

1. F. Wang and Y. Shoshitaishvili, "Angr - The Next Generation of Binary Analysis," 2017 IEEE Cybersecurity Development (SecDev), 2017, pp. 8-9, doi: 10.1109/SecDev.2017.14.
2. Baldoni, Roberto and Coppa, Emilio and D'Elia, Daniele Cono and Demetrescu, Camil and Finocchi, Irene, "A Survey of Symbolic Execution Techniques", ACM Comput. Surv., 2018
3. Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: three decades later. Commun. ACM 56, 2 (February 2013), 82–90. https://doi.org/10.1145/2408776.2408795
4. Ting Chen, Xiao-song Zhang, Shi-ze Guo, Hong-yuan Li, Yue Wu, "State of the art: Dynamic symbolic execution for automated test generation", Future Generation Computer Systems, Volume 29, Issue 7, 2013, Pages 1758-1773, ISSN 0167-739X,. DOI:https://doi.org/10.1016/j.future.2012.02.006
5. Stephens, Nick & Grosen, John & Salls, Christopher & Dutcher, Andrew & Wang, Ruoyu & Corbetta, Jacopo & Shoshitaishvili, Yan & Kruegel, Christopher & Vigna, Giovanni. (2016). Driller: Augmenting Fuzzing Through Selective Symbolic Execution. 10.14722/ndss.2016.23368.

6. Y. Shoshitaishvili et al., "SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis," 2016 IEEE Symposium on Security and Privacy (SP), 2016, pp. 138-157, doi: 10.1109/SP.2016.17.
7. Jacob Springer and Wu-chang Feng, "Teaching with angr: A Symbolic Execution Curriculum and CTF", 2018 USENIX Workshop on Advances in Security Education (ASE 18), https://www.usenix.org/system/files/conference/ase18/ase18-paper_springer.pdf
8. Shoshitaishvili, Yan & Wang, Ruoyu & Hauser, Christophe & Kruegel, Christopher & Vigna, Giovanni. (2015). Firmalice - Automatic Detection of Authentication Bypass Vulnerabilities in Binary Firmware. 10.14722/ndss.2015.23294.