CSIT Conference 2021, Yerevan, Armenia, September 27 - October 1

Overcoming Challenges of Firmware Analysis: Fuzzing and Symbolic Execution Based on Partial Emulation

Fedor Niskov
MSU, ISP RAS
Moscow, Russia
e-mail: fedor.niskov@ispras.ru

Maxim Mishechkin
ISP RAS
Moscow, Russia
e-mail: mish.max@ispras.ru

Shamil Kurmangaleev
ISP RAS
Moscow, Russia
e-mail: kursh@ispras.ru

Abstract—**Software security is very important in the modern world. Due to the complexity of modern software, many automated tools and methods are developed. A famous and efficient approach is the combination of fuzzing and symbolic execution. However, while a large toolset is available for general-purpose computers, the situation with firmware analysis is much more difficult. Lack of information, mechanisms, tools as well as physical restrictions raises serious problems for automated scalable testing. A possible solution in this situation is partial emulation – execution of an interesting code fragment from the initial state in an emulator, based on user scripts. This paper presents a new dynamic symbolic execution (DSE) module based on partial emulation. The paper also describes a combination of fuzzing and DSE – the developed module has been integrated into Crusher (Fuzzer by ISP RAS). This technology has been tested on various model and real cases.**

Keywords—**Dynamic binary code analysis, fuzzing, dynamic symbolic execution, concolic, firmware, embedded devices, partial emulation.**

I. INTRODUCTION

Code analysis is an important part of information security. Nowadays, there are many tools and methods for general-purpose computers.

For example, fuzzing [1] is a popular dynamic analysis method for discovering inputs, which cause software crashes. Its basic idea is quite simple: to send randomized input data to the target program and check its exit status repeatedly, until a crash is detected. Advanced fuzzers apply various sophisticated techniques, use additional information about program execution and input data format [8][9][10].

However, it's often difficult for fuzzers to generate input data to pass some checks in the target program and get deeper into the tested code. In such cases, the so-called dynamic symbolic execution (DSE, or "concolic") can help the fuzzer. DSE means symbolic execution along concrete path: the emulator follows the path of ordinary concrete execution, but marks input bytes as symbolic variables, builds symbolic formulas and the path predicate. By inverting conditions in the path predicate and solving the resulting equation system, it's possible to find new inputs, which will cause new paths and increase the code coverage.

Combination of fuzzing and DSE is a famous approach [9][10][11][12], but the arsenal of existing tools is not available in case of firmware analysis. There are many difficulties in this area [2]:

- many tools have not been adapted for devices;
- due to their simplicity, many devices don't have mechanisms for testing;
- lack of information and specifications;
- necessary presence of physical device can cause problems for scaled testing.

A possible approach in this situation is partial emulation [13][14]. It means execution of an interesting code fragment in an emulator from initial state. Initial state means values of registers and memory locations ("dump"). Emulation is based on scripts written by the user. Such script loads the initial state, specifies final points of execution, sets input data, sets handlers for non-emulatable fragments if they are present (some access to hardware or software environment, which should be replaced with simplified implementation). In case of DSE emulation, the script also specifies which input bytes should be marked as symbolic.

This paper presents the developed DSE module based on the described partial emulation idea. The module uses the Angr [3][4] framework for symbolic execution. The paper also describes the combination of the developed DSE module and fuzzing.

One more task is considered – to provide the following feature: to load elements of the initial state during emulation (on demand) directly from the device via the debugging interface. In this case, the device is stopped at a breakpoint, and the emulator uses the debugger to load necessary data. Such runtime on-demand loading can be better than a full dump, because making such dump can be time-consuming, and sometimes it's uncertain which memory regions should be dumped. This task has been completed using the Avatar orchestration framework [5][6] – its purpose is combination of several tools (in this case, a symbolic emulator and a debugger); during this research, some technical drawbacks of Avatar have been revealed and fixed.

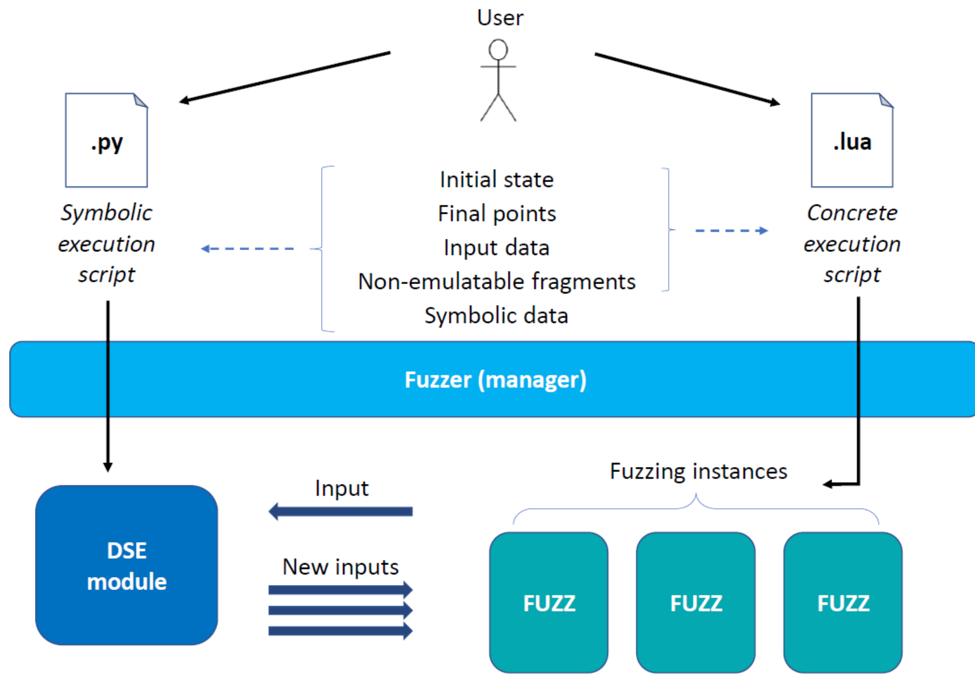

Fig. 1: Architecture of the developed system

II. Technology Description

For partial emulation with dynamic symbolic execution, the DSE module has been developed. The module takes an input and the user's script in Python, and generates new inputs, which lead to new paths.

The additional feature of the initial state runtime loading has also been implemented. For this purpose, some technical flaws of the Avatar framework have been fixed – these improvements have been suggested to the framework's official repository and approved by its developers [7]. The improvements are as follows:

- support for MIPS Big Endian;
- performance improvement with more efficient plan of requests to the debugger;
- ability of execution without a memory map.

The developed DSE module has been integrated with Crusher – fuzzer by ISP RAS [15][16]. The module is used to improve efficiency of partial emulation fuzzing. The fuzzer already supported this type of fuzzing, using Lua-QEMU (concrete execution based on the modified QEMU emulator and user scripts in the Lua language). Fig. 1 shows the general architecture of fuzzing with DSE: the user writes two scripts – for concrete and symbolic execution; the DSE module helps to generate new inputs.

The user script performs the following tasks:

- loading the initial state (values of registers and memory locations);
- specifying final execution points;
- setting input data (storing input bytes to some registers or memory locations);
- describing how to handle non-emulatable fragments (some access to software or hardware environment);
- specifying which input bytes should be marked as symbolic (in the DSE script).

The dynamic symbolic execution algorithm, implemented by the DSE module, is shown in Fig. 2.

General structure of a typical DSE script is shown in Fig. 3.

```
// 𝒥 – concrete input value
State = InitialState
while State.PC ∉ StopAddresses:
    NewStates = EmulBlock(State)
    for State_i in NewStates:
        if State_i.PathPredicate|Input=𝒥:
            // concrete path
            NextState = State_i
        else:
            // inverted condition
            if Solver.Solvable(State_i.PathPredicate):
                AddNewInput(Solver.Solution)
    State = NextState
```

Fig. 2: The DSE algorithm

```
import FuzzerDSE as dse
dse.FuzzerConfig = sys.argv[1]
...
dse.Dump = [{ 'addr': 0x12340000, 'data': 'abcdef...' }, ... ]
dse.Reg = { 'pc': 0xABCD0000, 'reg1': 123, ... }

def InitHandler(state0, input_data):
 ...
 dse.MarkSymbolic(range(len(input_data)))
 dse.WriteData(state0, 0xFFFF0000, range(len(input_data)))
 dse.StopAddrs = [0xABCD1234]
 ...
dse.InitHandler = InitHandler

dse.run()
```

Fig. 3: DSE script general format

Case	Crash discovery time (sec)	
	Without DSE	With DSE
Model #1 (hit 2 letters)	26,4	13,1
Model #2 (hit 3 letters)	221,9	13,0
Model #3 (hit 4 letters)	∞	13,1
Model #4 (hit 3 letters sequentially)	36,9	20,7
Model #5 (hit 4 letters + symbolic addressing)	∞	18,1
Model #6 (checksum)	∞	13,2
Real #1 (OS OpenWrt, IP packet processing)	∞	94,0
Real #2 (OS DD-WRT, CVE-2021-27137)	∞	830,0

Fig. 6: Testing results

This technology has broad prospects. Besides helping the fuzzer to pass checks and reach new paths, symbolic execution has other possible applications – it can be used in various code analysis algorithms. For example, it's possible to track data elements, marking them symbolically (e.g., taint analysis, uninitialized variables analysis); to check some constraints during execution and detect security violation; to traverse code paths and prove some properties of the target program. This great variety of algorithms can also be adapted to firmware analysis in the style of partial emulation. Furthermore, integration of symbolic execution algorithms, fuzzing and static analysis is a promising research direction.

```
1   void function(int *input)
2   {
3     int *ptr = NULL;
4     int a = *input;
5     if (0 < a && a < 10000) {
6       if (a * a == 25) {
7         *ptr = 123; // crash
8       }
9     }
10  }
```

Fig. 4: Listing of the example function

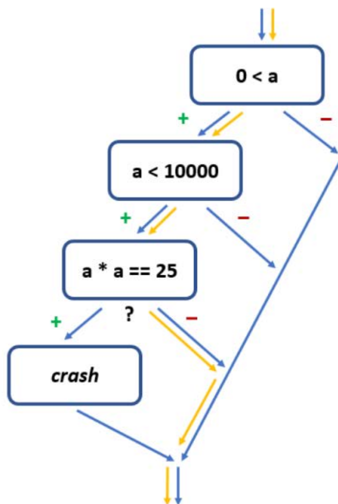

Fig. 5: CFG of the example function and the concrete path (a=1)

III. Example

Let's consider an example, which demonstrates how DSE can help fuzzing. Fig. 4 is a listing of function with a bug, which we want to find. The function takes a number as input. To get to the crash point, the following condition must be fulfilled: the number squared is equal to 25. If simple fuzzing is used (just giving random numbers to the function until the crash is detected), then it will take too long time, since it's difficult to hit the specific number value.

DSE can help fuzzer to pass such problematic condition. Let's consider some concrete input – for instance, number 1. With such input, execution has a path which is shown with orange arrows in the CFG (control flow graph – Fig. 5): it reaches the problematic condition but doesn't pass it. Let's perform DSE with this input: the input number is marked symbolically – it will be a symbolic variable α. Symbolic execution is performed along this path; the path predicate – the set of met conditions – is as follows:

$$\begin{cases} 0 < \alpha \\ \alpha < 10000 \\ \alpha^2 \neq 25 \end{cases}$$

During execution, we try to invert the conditions (to make them opposite), it leads to a new equation system:

$$\begin{cases} 0 < \alpha \\ \alpha < 10000 \\ \alpha^2 = 25 \end{cases}$$

This system can be solved automatically using a solver, and the solution is a new input, which will change the condition and cause a new path. So, in this example, it's possible to get number 5 and reach the crash point.

IV. Testing

The developed technology has been tested on various cases. In each case, the described system of fuzzing and DSE was used to find the bug in the tested code. The time of crash discovery was measured – without and with DSE help. The results of the testing are given in Fig. 6. Symbol ∞ means that the experiment's time exceeded the maximal limit, while the bug still had not been found, and it was pointless to continue the experiment (because, anyway, the resulting time value would be much greater than the value of the adjacent column). The testing has demonstrated that DSE helps the fuzzer to

achieve results much faster in many cases. It also proves correctness and vitality of the implementation.

The test suite contains several model cases. They reflect typical problematic situations, which researchers face during fuzzing, such as long constants and checksums. For example, if the target program checks that some input bytes are equal to a constant, it's difficult for the fuzzer to hit the constant; and the wider this constant is, the longer time it takes for the fuzzer to guess it. It's notable that coverage-guided fuzzers can have more success when the target program checks input bytes sequentially, one by one. Test 5 was designed to check an important property – the symbolic engine's capability to handle symbolic addressing (when memory access address is not concrete but symbolic; for example, using input number as index in a table). Checksum computation is another example where DSE is helpful: using input bytes, the target program calculates the checksum, which must be equal to the right value to continue data processing, and DSE can help to pass this check in case of simple checksum algorithms.

The test suite also includes cases with real firmware for routers. The first real case is about OS OpenWrt: an artificial bug was inserted into the code, which process input IP packet. In order to get to the crash point, it's necessary to set proper values for packet fields as well as set the correct checksum. The second real case is about OS DD-WRT and its known bug – CVE-2021-27137. The bug is in the code, which handles a SSDP request (the UPNP subsystem): there is an unsafe copy operation, which causes stack buffer overflow, and it's necessary to set proper values of packet string fields to get to the point of this operation. The DSE module can help greatly to generate inputs to satisfy complex constraints in real firmware.

V. CONCLUSION

Thus, the idea of fuzzing and symbolic execution based on partial emulation has been successfully implemented and tested. The partial emulation approach can help to overcome many difficulties of firmware analysis – physical restrictions, lack of information, mechanisms and tools. This is a promising approach, allowing automated scalable testing. The developed system has been tested on various cases, including real firmware. The testing has shown that DSE can greatly help the fuzzer to increase code coverage, pass complex checks in the code, and achieve results faster in many cases. So, this efficient combination of fuzzing and symbolic execution has been successfully adapted to firmware analysis.

The further research directions include the following:

- application of the developed technology to new cases of real firmware;
- optimization and performance augmentation;
- adding new code analysis algorithms to the DSE module.

REFERENCES

[1] H. Liang et al., "Fuzzing: State of the art", *IEEE Transactions on Reliability*, vol. 67, pp. 1199--1218, 2018.

[2] M. Muench et al., "What You Corrupt Is Not What You Crash: Challenges in Fuzzing Embedded Devices", *NDSS*, 2018.

[3] Y. Shoshitaishvili et al., "SOK: (State Of) The Art Of War: Offensive techniques in binary analysis", *2016 IEEE Symposium on Security and Privacy (SP)*, pp. 138--157, 2016.

[4] The Angr framework. [Online]. Available: https://angr.io

[5] J. Zaddach et al., "Avatar: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares", *NDSS*, vol. 14, pp. 1--16, 2014.

[6] M. Muench et al., "Avatar 2: A multi-target orchestration platform", *Proc. Workshop Binary Anal. Res. (Colocated NDSS Symp.)*, vol. 18, pp. 1--11, 2018.

[7] F. Niskov, (2020) Patch for Avatar. [Online]. Available: https://github.com/avatartwo/avatar2/pull/65

[8] The AFL fuzzer. [Online]. Available: https://lcamtuf.coredump.cx/afl

[9] N. Stephens et al., "Driller: Augmenting Fuzzing Through Selective Symbolic Execution", *NDSS*, vol. 16, pp. 1--16, 2016.

[10] P. Godefroid, M.Y. Levin, D. Molnar, "SAGE: whitebox fuzzing for security testing", *Communications of the ACM*, vol. 55, no. 3, pp. 40--44, 2012.

[11] L. Zhang, V.L.L. Thing, "A hybrid symbolic execution assisted fuzzing method", *TENCON 2017 IEEE Region 10 Conference*, pp. 822--825, 2017.

[12] E. Bounimova, P. Godefroid, D. Molnar, "Billions and billions of constraints: Whitebox fuzz testing in production", *35th International Conference on Software Engineering (ICSE), IEEE*, pp. 122--131, 2013.

[13] D. Straghkov, (2020) Description of methods and software tools allowing to emulate UEFI modules. [Online]. Available: https://www.ispras.ru/conf/2020/video/compiler-technology-11-december.mp4#t=2225

[14] Yoav Alon, Netanel Ben-Simon, (2018) 50 CVEs in 50 Days: Fuzzing Adobe Reader. [Online]. Available: https://research.checkpoint.com/2018/50-adobe-cves-in-50-days/

[15] Crusher (Fuzzer by ISP RAS). [Online]. Available: https://www.ispras.ru/technologies/crusher

[16] Crusher (materials on GitHub). [Online]. Available: https://github.com/ispras/crusher