A Case Study of the Application of Dynamic Symbolic Execution to Real-World Binary Programs

Duc Bui Hoang, Yunho Kim and Moonzoo Kim
Computer Science Department, KAIST
291 Daehak-ro(373-1 Guseong-dong), Yuseong-gu, Daejeon
ducbuihoang@kaist.ac.kr kimyunho@kaist.ac.kr moonzoo@cs.kaist.ac.kr

Abstract:
Analyzing binary programs is necessary in many situations when we do not have the programs source code. In recent years, many binary analysis tools have been developed such as CodeSurfer/x86, BitBlaze and S2E. In this paper we developed a binary symbolic execution engine based on BitBlaze. We applied the engine to generate hundreds of test cases for a real-world application on Windows: Acrobat Reader. Besides, we also discussed lessons learned from applying dynamic symbolic execution on real-world programs.

Keywords: Binary analysis, software testing, formal verification, concolic testing

1. Introduction

Analyzing binary programs is always in great demand. Software companies need to test the actual shipped software because compilation tools and post-processing tools such as basic block transformers and code obfuscators may introduce subtle bugs. Users also want to check bugs in binary libraries because it might not be feasible to obtain source code of third party components, even components developed by different groups in the same organization.

Dynamic symbolic execution (DSE) is a popular automated testing technique for generating test cases that explore execution paths of a program systematically [1]. Given an initial input, the dynamic symbolic execution technique executes the program both concretely and symbolically. The path constraint from conditional statements along the executed path is then negated and solved to generate a new input that lead the target program to exercise new path. This process is repeated so that execution paths of the target program are explored executed automatically and systematically.

In an effort to detect bugs in real-world software programs at the operating system level, we used dynamic symbolic execution on top of BitBlaze to generate test cases for Notepad and Adobe Acrobat Reader on Windows XP. In this paper, we will present our symbolic execution engine, experimental results and lessons we learned from our attempts.

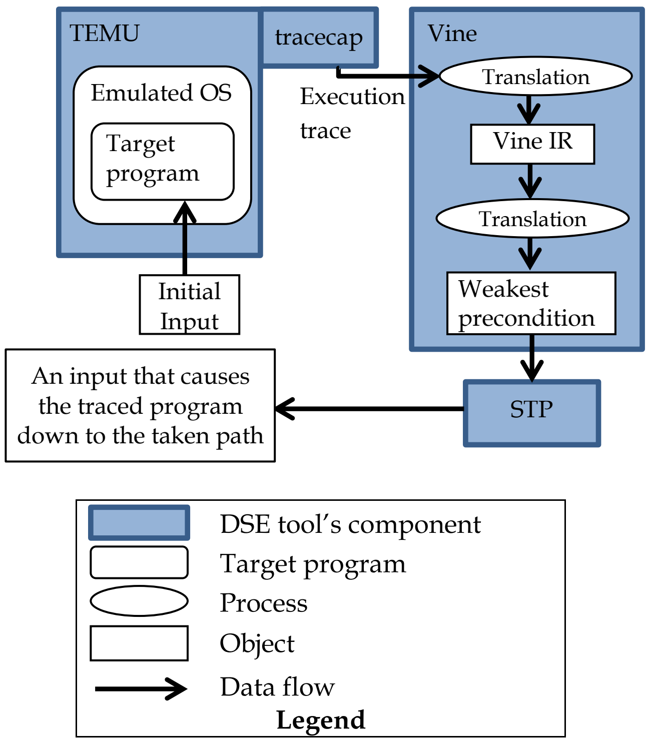

Figure 1 BitBlaze architecture

2. BitBlaze: A binary analysis platform for computer security

BitBlaze [2] is a platform for analyzing binary code for security applications developed by Song et al. at University of California at Berkeley. BitBlaze has three main components: TEMU, Vine and Rudder. TEMU, the dynamic analysis component, and Vine, the static analysis component, are open source while Rudder, the online dynamic symbolic execution component, is not publicly available. The architecture of BitBlaze is shown in Figure 1.

TEMU allows users to obtain an execution trace of a run of the target program and information related to its input. TEMU is built on top of QEMU virtual machine inside which the analyzed program runs. Therefore, users can gain information of all executed instructions of the analyzed program and its environment, i.e. all of its external libraries and operating system. Users can use **tracecap,** a plugin of TEMU that can mark all bytes in the input bytes as tainted to record all instructions related to the tainted bytes such as the instructions that have tainted source or destination operand. In the context of dynamic symbolic execution, these tainted bytes are equivalent to symbolic bytes.

Vine provides users with a functionality to perform analysis on the execution trace. Vine can lift the assembly code in the execution trace to Vine intermediate representation (Vine IR). Vine supports generating weakest precondition (wp) from the intermediate representation. The weakest precondition wp(P,Q) for a program P and post-condition Q is a Boolean predicate such that when wp(P,Q) holds, executing P is guaranteed to terminate in a state satisfying Q. The solution of the weakest precondition is an input that makes the target program to exercise the same path it takes in the execution trace. Vine interacts with STP, a SMT solver, to solve the satisfiability of the weakest precondition formula.

In Vine IR, the path constraint of the program is represented in a conjunction of post conditions. This conjunction is represented in a sequence of assertions. Every time the execution reaches a branching statement, Vine generates a post condition variable that records the path condition and an assertion to indicate that the program follows a specific path from inputs. The sequence of assertions represents a path taken in the execution trace of the program.

3. Binary symbolic execution engine (BSEE)

We implemented a symbolic execution engine similar to Rudder in OCaml on top of Vine. While Rudder is an online symbolic execution engine that generates path constraint formula at the same time the program runs, BSEE is an offline symbolic execution engine which performs analysis on execution traces after the target program runs. Firstly, the execution trace of the target program is lifted to Vine IR. In order to generate the new input, we negate one condition in the path constraint in the Vine IR file. The negated IR is then translated to the weakest precondition formula. Finally, STP, a SMT solver, will solve this formula and give the input that leads the target program into a directed path, if any exists. Figure 2 shows the architecture of BSEE.

The main algorithm in BSEE is as the following:
* Input: an IR file; output: a set of new inputs that exercise different branches of the program.
1. Create a list of post condition variables.
2. Choose a post condition variable in the list.
3. Negate the condition in the corresponding assertion.
4. Remove all instructions in the IR after the negated assertion.

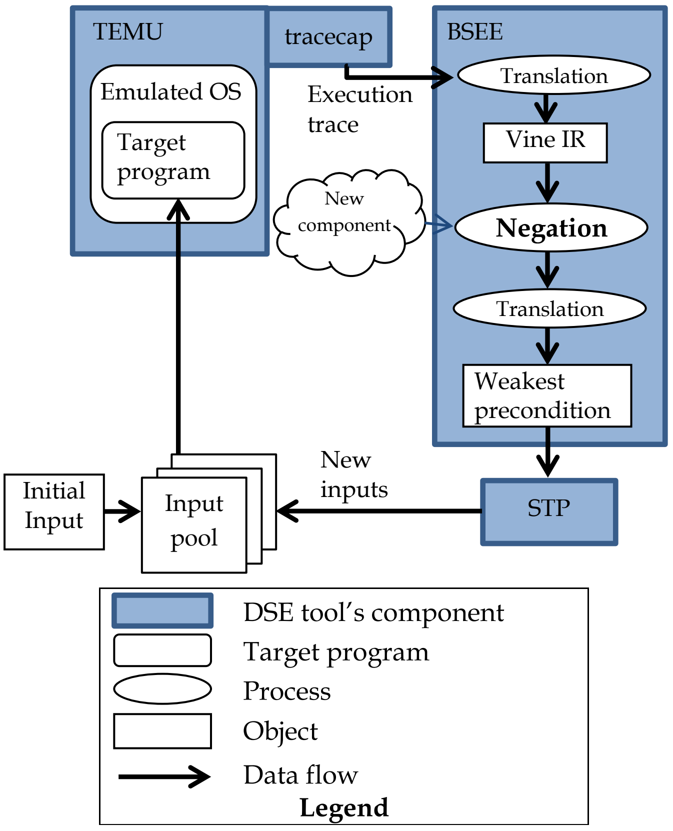

Figure 2 BSEE architecture

5. Generate the weakest precondition for the modified IR in the STP syntax.
6. If STP finds a set of values that satisfies the weakest precondition formula within a time bound, we have a new input file that comprises the sequence of the values.
7. Repeat step 2 to 6 for the next variable in the list until there is no remaining variable.

BSEE chooses and negates post conditions as the following. A post condition variable can be chosen in the list from the first to the last of the variable list. In step 3, the condition for the assertion corresponding to the chosen post-condition variable is negated by adding the negation operator to the condition inside assertions.

The following example illustrates how the algorithm works in order to generate a new test case that exercises the different branch for a branching statement.

Given an if statement and with the initial input x=1
```
  if(x!=5)
```
gcc may translate it into the following assembly code
```
  0x0804841b: cmp   $0x5, %eax
  0x0804841e: je    0x804842c <main+56>
```
A run of the instruction with x=1 results the following contiguous excerpt from the execution trace:

```
0804841b: cmp  $0x5,%eax
0804841e: je  0x0804842c
08048420:  … (not jump)
```

A summary of Vine IR corresponding to the above trace

```
/*cmp   $0x5,%eax*/
T_81_1520:reg32_t = R_EAX_5:reg32_t - 5:reg32_t;
R_ZF_13:reg1_t = T_81_1520:reg32_t == 0:reg32_t;
/*je    0x000000000804842c*/
cond_960:reg1_t = R_ZF_13:reg1_t == false;
assert(cond_960:reg1_t);
```

The application of the steps 2 and 3 to the example results the following assertion:

```
cond_960:reg1_t = R_ZF_13:reg1_t == false;
assert(!cond_960:reg1_t);
//End of file
```

4. Experimental results

4.1. Experimental settings

Machine settings for testing Notepad and Acrobat Reader are presented in Table 1. The input of the programs is a file. The initial input file consists of 743 bytes of ASCII character '0'. 743 bytes is necessary size for generating 737-byte-long new input while 737 bytes is the minimal file size that TEMU can record executed instructions. We found 737 bytes size by trying various file sizes following binary search strategy. The generated input is a sequence of bytes.

In the experiments, we traced the runs of Acrobat Reader and Notepad since they open the initial input file until a time bound of 10 minutes reaches or the target program completes opening the file. Time bound for STP is 30 seconds.

4.2. Generated test cases

In the case of Acrobat Reader, we generated 1469 test cases in total. In the first run, from the initial input of 743 bytes of '0', we trace Acrobat Reader in 10 minutes and generated 736 new test cases of various lengths from 1 to 738 bytes for Acrobat Reader in around two hours. Post condition variables were chosen from the first to the last variable in the post condition variable list. In the second run, we traced Acrobat Reader in 10 minutes with a 738-byte-long input generated from the first run. BSEE generated 733 test cases which vary from 1 to 733 bytes long in two and a half hours.

In the case of Notepad, in spite that we tried various methods, we failed to generate test cases for Notepad. This is because Vine encountered an error when lifting the execution trace to Vine IR. (see section 5.2)

5. Lessons learned

5.1. Large amount of data to process for analyzing binary program execution

A major challenge of analyzing real-world applications is we need to process a huge amount of low level data. In general, the execution trace contains

Table 1 Test bed

CPU	Intel Core2 Duo E8600 @ 3.33GHz
Memory	8 GB
Host OS	Debian 6.0.3 32-bit
Guest OS	Windows XP SP3(English) 32-bit
Target programs	notepad.exe in Windows XP SP3 (5.1.2500.5512)
	AcroRd32.exe – The main executable file of Adobe Acrobat Reader 9.2.0

millions of instructions. For example, the 10-minute execution trace of Acrobat Reader is 1.2GB large and contains more than 19 million instructions. Though we have information of external libraries and environment but we also have to process all this information in addition to the target program itself. Thus, we need to separate the instructions that belong to the target program from the instructions that belong to external libraries and from the instructions that belong to the operating system. (see the selective symbolic execution technique [3] used in S2E platform [4])

5.2. Limitations of BitBlaze

BitBlaze could not record the execution trace when the target program reads a very small file. Therefore we lost a degree of control to the input of the target program. In our experiments, this problem made us unable to use small inputs generated by BSEE in subsequent runs. This problem may be partly due to the method that BitBlaze used to propagate and monitor tainted bytes. Locations of bytes of file on disk are marked as tainted so that TEMU can monitor and record all operations on the above disk locations. However, the target program may read those bytes from the disk buffer in memory instead of directly from disk and tainted information may be lost.

Furthermore, Vine failed to handle certain binary instructions. For example, when Vine lifts execution trace of Notepad to Vine IR, it failed to handle instruction bytes 0xCD 0x2B 0x0 0x0 which is an interrupt. Though we attempted to use a newer version of VEX library that is responsible for this problem but the problem persisted. Due to this bug we could not generate test cases for Notepad.

Besides, the analysis speed of BitBlaze is too slow for the target programs. For example, to open 743-byte-long file, Acrobat Reader took more than 60 minutes and generated 35GB execution trace. Vine failed to translate traces into Vine IR due to an out-of-memory error. Table 2 shows the statistic of performance of BitBlaze when applying to Acrobat Reader. Besides the experiment in section 4, we performed two more experiments in which Acrobat Reader was traced in a 15 minutes and 60 minutes.

Table 2 BitBlaze Performance

Target program	AcroRd32.exe			notepad.exe
Tracing time	10min	15min	60min	1min
Size of trace file	1.2GB	2.1GB	35.0GB	72MB
Translation time (execution trace to Vine IR)	2min	out of memory	out of memory	1min (interrupted by an error)
Size of Vine IR	23MB	N/A	N/A	N/A

Finally, TEMU can miss propagation of tainted data in the executions of complicated applications. This caused us unable to generate new values for all input bytes. For example, in one experiment, from a 737-byte-long input, we only obtained 98 values for from STP. This is because TEMU missed propagation of tainted data and replaced it by a concrete initialization. Vine indicated this as a warning in the IR file: "WARNING missed prop, using concrete init".

6. Related work

Binary analysis tools are more prevalent these days. CodeSurfer/x86 [5] is a static analysis tool for analyzing memory accesses in x86 executable files. BSEE analyzes execution of the target program in order to generate test cases that cover different execution paths of the target program instead of focusing on memory access analysis as CodeSurfer/x86 does. S2E platform [4] which is built on QEMU [6] and KLEE [7] enables multiple-path analysis of binary programs, libraries and drivers at the operating system level. However, so far, S2E does not support a symbolic disk drive that creates symbolic bytes for the target program. SAGE [8] generates test cases on binary programs by using dynamic symbolic execution. While SAGE only records the target program's user-mode execution, BSEE records all instructions executed in the entire software stack, from the target program to the operating system.

BitBlaze was used in many tools to detect security vulnerabilities. Yin et al. [9] and Liang et al. [10] built tools based on TEMU to detect hooks and analyze hooking behaviors of malwares.

7. Conclusion

In this paper, we developed BSSE, a symbolic execution engine for binary programs based on BitBlaze binary analysis platform and generated test cases for Acrobat Reader, a real-world binary program on Windows. From the experiments, we found that there are still many challenges and limitation of the existing tool that make dynamic symbolic execution not applicable to real-world applications at the operating system level.

References

[1] Y. Kim and M. Kim, "SCORE: a Scalable Concolic Testing Tool for Reliable Embedded Software," in *ACM SIGSOFT Foundation of Software Engineering (FSE) Tool demonstration track*, Szeged, 2011.

[2] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam and P. Saxena, "BitBlaze: A New Approach to Computer Security via Binary Analysis," in *International Conference on Information Systems Security*, 2008.

[3] V. Chipounov, V. Georgescu and C. Zamfir, "Selective Symbolic Execution," in *5th Workshop on Hot Topics in System Dependability (HotDep)*, Lisbon, 2009.

[4] V. Chipounov, V. Kuznetsov and G. Candea, "S2E: A Platform for In Vivo Multi-Path Analysis of Software Systems," in *Conference on Architectural Support for Programming Languages and Operating Systems*, 2011.

[5] G. Balakrishnan, G. Balakrishnan, R. Gruian, T. Reps and T. Teitelbaum, "CodeSurfer/x86 - A Platform for Analyzing x86 Executables," in *Conference on Compiler Construction*, 2005.

[6] F. Bellard, "Qemu, a fast and portable dynamic translator," in *USENIX 2005 Annual Technical Conference*, 2005.

[7] C. Cadar, D. Dunbar and D. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *USENIX conference on Operating systems design and implementation*, 2008.

[8] P. Godefroid, M. Y. Levin and D. A. Molnar, "Automated Whitebox Fuzz Testing," in *Network Distributed Security Symposium*, 2008.

[9] H. Yin, P. Poosankam, S. Hanna and D. Song, "HookScout: Proactive B inary-Centric Hook Detection," in *7th Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'10)*, 2010.

[10] H. Yin, Z. Liang and D. Song, "HookFinder: Identifying and Understanding Malware Hooking Behaviors," in *15th Annual Network and Distributed System Security Symposium*, 2008.