2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks

Concolic Execution on Small-Size Binaries: Challenges and Empirical Study

Hui Xu*†, Yangfan Zhou‡§, Yu Kang‡, Michael R. Lyu*†
* Shenzhen Research Institute, The Chinese University of Hong Kong
† Dept. of Computer Science, The Chinese University of Hong Kong
‡ School of Computer Science, Fudan University
§ Engineering Research Center of Cyber Security Auditing and Monitoring, Ministry of Education, China

***Abstract*—Concolic execution has achieved great success in many binary analysis tasks. However, it is still not a primary option for industrial usage. A well-known reason is that concolic execution cannot scale up to large-size programs. Many research efforts have focused on improving its scalability. Nonetheless, we find that, even when processing small-size programs, concolic execution suffers a great deal from the accuracy and scalability issues.**

This paper systematically investigates the challenges that can be introduced even by small-size programs, such as symbolic array and symbolic jump. We further verify that the proposed challenges are non-trivial via real-world experiments with three most popular concolic execution tools: BAP, Triton, and Angr. Among a set of 22 logic bombs we designed, Angr can solve only four cases correctly, while BAP and Triton perform much worse. The results imply that current tools are still primitive for practical industrial usage. We summarize the reasons and release the bombs as open source to facilitate further study.

I. Introduction

Recently, concolic (concrete and symbolic) execution has become an upsurge of interest for code analysis. As an advanced software testing approach with formal methods, it shows high impact in the research areas of bug detection [1, 2], deobfuscation [3, 4], *etc*, and outperforms other traditional testing approaches (*e.g.,* random testing) when handling small-size programs. There are several popular concolic execution tools available for public usages, such as Angr [5] and Triton [6]. Along with these tools, many showcases have been demonstrated. The famous cases are `crackme` puzzles in Capture the Flag (CTF) contests [7], and Cyber Grand Challenge by Defense Advanced Research Projects Agency (DARPA) [8]. However, concolic execution has not yet gained wide usage in the industrial area.

One well-known reason is that concolic execution does not scale to large-size programs. Many approaches have been proposed to improve the scalability, such as parallel processing [9], state merging [2], and efficient search strategy [10]. However, we observe that even for many small-size programs, real-world concolic execution tools cannot achieve the ideal performance in code coverage. Investigating the limitations of concolic execution tools and the corresponding challenges are essential for attracting the attention of research communities. Also, without a clear understanding of the usability issues, concolic execution users would not know whether the technique meets their needs, or how to engage the technique in a proper way.

In this paper, we systematically investigate the underlying challenges of concolic execution tools on small-size binary programs. To this end, we first discuss the theoretical background of concolic execution, and summarize four error types which may occur during different stages of symbolic reasoning: E_{s0}, which occurs if symbolic variables are not correctly declared; E_{s1}, which occurs during instruction tracing; E_{s2}, which relates to data propagation; and E_{s3}, which relates to constraint modeling. Then we propose seven technical challenges that may raise such errors: symbolic variable declaration, covert symbolic propagation, parallel program, symbolic array, contextual symbolic value, symbolic jump, and floating-point numbers. Incapable of handling the technical challenges would cause reachable code unexplored, or *vice versa*. Further, we propose two scalability challenges and emphasize that small-size programs may also incur scalability issues for concolic execution tools. The essential idea is that small-size programs can have high complexity. The challenges that can increase program complexity while incurring a small overhead in program size include: extensively using external function calls, or using crypto functions (*e.g.,* SHA1) which involve complex problems beyond the capability of computers.

To demonstrate that such challenges are non-trivial for real-world concolic execution tools, we design a set of programs which illustrate the challenges, and then evaluate them against three popular concolic execution tools: BAP [11], Triton [6], and Angr [5]. Our preliminary results show that each of our proposed challenges contain samples that cannot be addressed by all these tools, which implies none of the challenges are trivial. Specifically, Angr can only solve four out of 22 cases, while BAP and Triton perform worse. We further investigate the causes and find that the failures for BAP and Triton are not only due to the challenges but also by their own deficiency and bugs, such as unsupported instructions. In comparison, Angr has better support for instruction lifting, as well as employing some advanced features, such as symbolic memory addressing.

Our work is the first systematic study on the challenges of performing concolic execution on small-size binaries. It would serve as an essential reference for concolic execution researchers to improve their tools, and for the users to properly

2158-3927/17 $31.00 © 2017 IEEE
DOI 10.1109/DSN.2017.11

use the technique. To better serve the community in this area, we release our program set as open source.

The rest of this paper is organized as follows. We first introduce the related work in Section II. Then we discuss the technique of concolic execution in Section III and propose the challenges accordingly in Section IV. We discuss our evaluation approach and results in Section V. Finally, Section VI concludes the paper.

II. Related Work

Concolic execution and symbolic execution have received extensive attention in the last decade. Existing work in this area mainly focuses on using the technique to carry out specific software analysis tasks (*e.g.,* [1, 3, 4]), or proposing new approaches to improve performance issues, such as [9, 12]. In such papers, the limitations and challenges of symbolic executions are occasionally discussed, such as path explosions in [12], system and library calls in [13]; however, they are not systematically studied as we do in this paper.

The papers most close to our work are [14, 15], which focus on investigating the limitations and challenges of software testing tools with symbolic analysis features. Qu and Robinson conduct a case study on the limitations of concolic testing tools for source code(*e.g.,* KLEE) and examined their prevalence in real-world programs. Concolic testing tools for source code are different from those for binaries, and they suffer different challenges. For example, data structures, and pointers are challenging problems for source code analysis, but binary programs neither exhibit data structures nor employ pointers. Another similar work by Cseppento and Micskei also focuses on symbolic execution techniques for source code [15], which are different from ours. Because we study the challenges of an entirely different area, as a result, our evaluation experiment shares no common tools with any of the two papers. Another work by Kannavara *et al.* [16] also points out several challenges that have hindered the adoption of concolic execution; however, it does not examine the prevalence of the challenges in real-world tools.

In a nutshell, this work is a pilot study on the challenges of concolic execution on small-size binaries. Although some challenges we discussed in this paper are not newly proposed, to our best knowledge, we are the first to systematically study them for small-size binary programs.

III. Background

A. Theoretical Background

Concolic execution has two phases, a concrete execution phase and a symbolic reasoning phase. The concrete execution phase executes the program and generates a trace of executed instructions, while the symbolic reasoning phase analyzes the trace and calculates new test cases that can trigger unexplored control flows. The two phases work alternatively so that all the possible control flows can be explored eventually.

Formally, we can use Hoare Logic [17] to model the symbolic reasoning problem. Hoare Logic is composed of basic triples $\{S_1\}P\{S_2\}$, where $\{S_1\}$ and $\{S_2\}$ are the assertions of symbolic variable states and P is a program spinet or command. The Hoare triple says if a precondition $\{S_1\}$ is met, when executing P, it will terminate with the postcondition $\{S_2\}$. Using Hoare Logic, a concrete execution can be modeled as:

$$\{S_0\}P_0\{S_1, \Delta_1\}P_1...\{S_n, \Delta_n\}P_n$$

$\{S_0\}$ is the initial symbolic state of the program; $\{S_1\}$ is the symbolic state before the first conditional branch with symbolic variables; Δ_i is the corresponding constraint for executing the following instructions, and $\{S_i\}$ satisfies Δ_i; P_i represents a sub-trace of instructions. A symbolic executor can compute an initial state $\{S_0'\}$ (*i.e.,* the concrete values for symbolic variables) which can trigger the same sequence of instructions. This can be achieved by computing the weakest precondition (*aka* wp) backward using Hoare Logic:

$$\{S_{n-1}\} = wp(P_{n-1}\{S_n\}), \quad s.t.\ \{S_n\}\ sat\ \Delta_n$$
$$\{S_{n-2}\} = wp(P_{n-2}\{S_{n-1}\}), \quad s.t.\ \{S_{n-1}\}\ sat\ \Delta_{n-1}$$
$$...$$
$$\{S_1\} = wp(P_1\{S_2\}), \quad s.t.\ \{S_2\}\ sat\ \Delta_2$$
$$\{S_0\} = wp(P_0\{S_1\}), \quad s.t.\ \{S_1\}\ sat\ \Delta_1$$

Recursively, we can get a constraint model in conjunction normal form: $\delta_1 \wedge \delta_2 \wedge ... \wedge \delta_k$. Computing symbolic values that can satisfy the model is a satisfiability problem and the solution is a test case $\{S_0'\}$ that can trigger the same control flow.

Concolic execution searches test cases that can trigger unexplored control flows via generating new constraint models. We may negate δ_i and cut off the tail instructions to generate a new constraint model: $\delta_1 \wedge \delta_2 \wedge ... \wedge \overline{\delta_i}$. Note that, if we do not remove constraints $\delta_{i+1} \wedge ... \wedge \delta_k$, the constraint models may have no solutions.

Finally, when sampling $\{P_i\}$, not all instructions are useful. We only keep the instructions whose parameter values depend on the symbolic variables. We can demonstrate the correctness by expending any irrelevant instruction I_i to $X := E$, which manipulates the value of a variable X with an expression E. Suppose E does not depend on any symbolic value, then X would be a constant, and should not be included in the weakest preconditions. In practice, it can be realized using taint analysis techniques.

B. Conceptual Framework

Now we discuss how the theoretical model can be implemented in practice. Since binary programs do not exhibit explicit variables and types, the symbolic state $\{S_i\}$ and constraint Δ_i are represented with symbolic memories. The program P_i is represented with assembly instructions. A detailed framework which technically synthesizes the whole concolic execution process is shown in Figure 1.

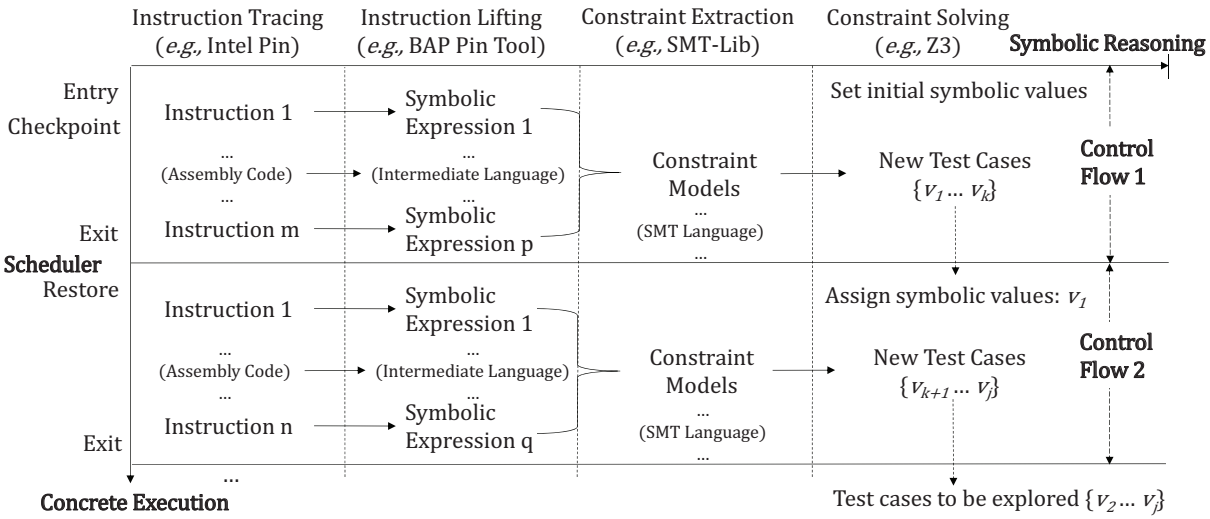

Fig. 1: A conceptual framework for concolic executing binary programs

Vertically, the framework includes several rounds of concrete executions, and each round is initiated with a new setting of symbolic values. Horizontally, each round includes several essential steps for symbolic reasoning.

Instruction Tracing: In this step, the concolic executor records the instructions during concrete execution. This can be achieved based on CPU tools (*e.g.,* Intel Pin [18]), or machine emulation tools (*e.g.,* QEMU [19]). In general, not all instructions are our interests. A forward taint analysis process can be employed to filter out the unrelated instructions.

Instruction Lifting: This step interprets the semantic of each assembly instruction with a form of intermediate language (IL), and lifts the whole trace into intermediate representatives (IR) using the IL. In this way, the original operations on registers and memories can be explicitly modeled.

Constraint Extraction: Each branch with symbolic conditions indicates a new control flow possibility. Concolic execution extracts the constraint model for each branch via recursive symbolic expression substitution. Each concrete execution round may generate several constraint models depending on the number of conditional branches along the trace. The constraint models are generally described with a language of satisfiability modulo theories (SMT), such as SMT-Lib [20].

Constraint Solving: A constraint solver is employed to search solutions for each constraint model. There are several popular constraint solvers off-the-shelf, such as MiniSat [21] and Z3 [22].

After new test cases are generated, a *scheduler* prioritizes the newly generated test cases, and determines which one should be used in the next round of concrete execution. This process is carried on until all test cases are explored. In practical implementations, a *checkpoint mechanism* can be used to further facilitate concolic execution by saving redundant executions.

Note that some concolic execution tools (*e.g.,* Angr) adopt a hybrid concolic execution approach rather than our discussed framework. Such tools lift the whole program into IR first and then perform dynamic symbolic execution on the IR. In this way, the efforts in redundant instruction lifting can be saved. However, such tools also inherit the drawbacks of static analysis and emulation, and they are vulnerable to sophisticated obfuscation techniques (*e.g.,* code mutation). Considering the popularity of camouflages in binaries, our conceptual framework is more reliable for binary analysis.

IV. Challenges

In this section, we first overview the errors that may occur during concolic execution, and then discuss the challenges that may incur such errors; finally, we discuss the scalability issues for small-size programs.

A. Errors for Symbolic Execution

When performing symbolic reasoning, errors can be introduced in four stages.

- E_{s0}: Symbolic variable declaration errors, which happen before symbolic reasoning. As a result, insufficient constraints can be generated for triggering new control flows.
- E_{s1}: Instruction tracing and supporting errors, which happen when some instructions are missing, or are not supported for instruction lifting.
- E_{s2}: Data propagation errors, which can be introduced when some instructions are not correctly interpreted, or when memories are not correctly modeled. As a result, the symbolic states are not correct computed.
- E_{s3}: Constraint modeling errors, which can be introduced when a required satisfiability modulo theory is not supported.

The listed errors are not independent. Any error in one stage can cause other errors in its posterior stages.

B. Accuracy Challenges

Real-world binaries exhibit rich diversities in syntax and semantics. A program (*e.g.,* Malware) can employ rarely used syntax to avoid analysis, which may incur errors in symbolic reasoning. Table I demonstrates a list of such technical challenges with the corresponding errors they may incur.

TABLE I: A list of challenges, and the corresponding errors they may incur.

Challenge	Stage of Error			
	E_{s0}	E_{s1}	E_{s2}	E_{s3}
Symbolic Variable Declaration	✓	✓	✓	✓
Covert Symbolic Propagation	-	-	✓	✓
Parallel Program	-	-	✓	✓
Symbolic Array	-	-	-	✓
Contextual Symbolic Value	-	-	-	✓
Symbolic Jump	-	-	-	✓
Floating-point Number	-	-	-	✓

1) Symbolic Variable Declaration: Symbolic variables are the factors that affect program execution. Such factors include program arguments, and other runtime information from the context, such as time. In general, symbolic variables should be declared before concolic execution. An attainable approach is to consider the arguments to the program (*i.e.,* `argv`) as symbolic variables; however, it is frustrating to consider all possible factors. Besides, even for the symbolic variables from `argv`, concolic executors may not be able to handle varying lengths of symbolic variables automatically.

Figure 2(a) demonstrates an example where a logic bomb (*i.e.,*Bomb()) can only be triggered at a specific time. To explore the bomb path, a concolic executor should declare `tv` as a symbolic variable, and then solve the constraint for `tv`. Failure in handling the challenge incurs E_{s0}.

2) Covert Symbolic Propagation: When extracting constraint models, the symbolic data propagation process should be correctly recognized. However, some data propagation are not explicit. An extreme case happens when the symbolic values are saved outside the process (*e.g.,* into a file), and then read back to the process. Tracking such data propagation would be challenging.

Figure 2(b) demonstrates an example where the value of `argv[1]` determines whether a logic bomb can be triggered. However, `argv[1]` is propagated via a file in a covert way. Failure in handling the challenge incurs E_{s2}.

3) Parallel Program: Traditional data-flow analysis approaches generally cannot handle concurrent programs. Data propagation among threads may have many possibilities, and modeling such data-flow is very expensive.

Figure 2(d) demonstrates a parallel program which implements two threads. The symbolic value from the main thread is processed in another thread with a self-incremental function. Then the symbolic value is evaluated against the condition for trigger the bomb path. If a concolic execution tool does not support concurrent programs, it may raise E_{s2} and generate a wrong test case for triggering the bomb.

4) Symbolic Array: When symbolic values serve as pointers or offsets to access data in memory, challenges arise for generating constraint models. An effective constraint model should include all the data within the memory region so that a solver can infer which data satisfies the model. Otherwise, E_{s3} occurs.

Figure 2(c) demonstrates an example with a one-level symbolic array. To execute the bomb path, 6 should be assigned to the `argv[1]` as an array index.

5) Contextual Symbolic Value: Symbolic values can be used as parameters for retrieving data from the environment (*e.g.,* disk). The challenge is similar to symbolic array and is more complex.

Figure 2(e) shows such an example. If `argv[1]` points to a file that can be opened by the program, the bomb would get triggered. However, it is difficult for concolic executors to interpret the semantic and to know which file exists on disk.

6) Symbolic Jump: After each round of concrete execution, the instructions which indicate branches with symbolic conditions should be extracted, so that the constraint models can be generated respectively. However, such branches can be performed in covert ways. For example, we may use symbolic values as the offset of an unconditional jump. Theoretically, we may assign different values so that the program can jump to any address within the program. In this way, symbolic jump acts similarly as a conditional jump.

Figure 2(f) demonstrates a code snippet where the symbolic value determines the target address of the unconditional jump. To trigger the bomb, we may simply assign 7 to `argv[1]`. Failure in handling symbolic jump incurs E_{s3}.

7) Floating-point Number: When the symbolic conditions involve floating-point numbers, errors may occur. A floating-point number ($f \in \mathbb{F}$) approximates a real number ($r \in \mathbb{R}$) with a fixed number of digits in the form of $f = significand * base^{exp}$. The representation is essential for computers, as the memory spaces are limited in comparison with the infinity of $\mathbb{R}$. As a tradeoff, floating-point numbers have only limited precision, which causes some unsatisfiable constraints over $\mathbb{R}$ can be satisfied over $\mathbb{F}$ with a rounding mode.

Figure 2(g) demonstrates such an example. The conditional expression $1024 + x = 1024 \;\&\&\; x > 0$ has no solutions for x over $\mathbb{R}$, but it has solutions over $\mathbb{F}$, such as 0.00001.

C. Scalability Challenges

Small-size programs may also lead to path explosions. The essential idea is that small-size programs can have high complexity. This can be achieved in two ways: extensively using external function calls, or using a crypto function that has high complexity. We discuss the two challenges in bellow.

1) External Function Call: Shared libraries, such as `libc` and `libm` (*i.e.,* a math library), have been widely used in binaries. They provide some basic function implementations to facilitate software development. External functions become the augmented part of a program once being called, and they enlarge the code complexity in nature.

```c
int main(int argc, char **argv)
{
    struct timeval tv;
    gettimeofday (&tv, NULL);
    if (tv.tv_sec == 2524608000)
        Bomb();
    else
        Foobar();
}
```

(a) Symbolic variable declaration.

```c
int main(int argc, char** argv){
    int j,i=atoi(argv[1]);
    char file[] = "tmp.covpro";
    char cmd[256];
    sprintf(cmd, "echo %d > %s\n", i, file);
    system(cmd);
    FILE *fp = fopen(file, "r");
    fscanf(fp,"%d",&j);
    fclose(fp);
    if(j == 7){
        Bomb();
    } else{
        Foobar();
    }
    remove(file);
}
```

(b) Covert Symbolic propagation.

```c
int main(int argc, char** argv){
    int ary[] = {1,2,3,4,5,6,7,8,9,0};
    int x = atoi(argv[1]);
    if(ary[x] == 7){
        Bomb();
    }
    else
        Foobar();
}
```

(c) Symbolic array.

```c
void* Inc(void* i){
    ++ *((int*) i);
}

int main(int argc, char** argv){
    pthread_t thread;
    int i = atoi(argv[1]);
    int rc = pthread_create(&thread,
            NULL, Inc, (void *) &i);
    rc = pthread_join(thread, NULL);
    if(i == 7){
        Bomb();
    }
    else
        Foobar();
}
```

(d) Parallel program.

```c
int main(int argc, char** argv) {
    FILE *fp = fopen (argv[1], "r");
    if (fp != NULL) {
        Bomb();
    } else{
        Foobar();
    }
}
```

(e) Contextual symbolic value.

```c
#define jmp(addr) asm("jmp *%0"::"r"(addr):)
int main(int argc, char** argv){
    int addr = argv[0][0] - 26;
    addr = addr - atoi(argv[1]);
flag_0:
    jmp(&&flag_0 + addr);
    Bomb();
    Foobar();
}
```

(f) Symbolic jump.

```c
int main(int argc, char** argv){
    float x = atof (argv[1]);
    if (1024 + x == 1024 && x > 0){
        Bomb();
    }
    else {
        Foobar();
    }
}
```

(g) Floating-point number.

```c
int main(int argc, char** argv){
    int i = atoi (argv[1]);
    if (sin(i*PI/180)== 0.5){
        Bomb();
    }else{
        Foobar();
    }
}
```

(h) External function call.

```c
int main(int argc, char** argv){
    int plaintext = atoi(argv[1]);
    unsigned cipher[5];
    cipher[0] = 0X77de68da;
    cipher[1] = 0Xecd823ba;
    cipher[2] = 0Xbbb58edb;
    cipher[3] = 0X1c8e14d7;
    cipher[4] = 0X106e83bb;
    if(SHA1_COMP(plaintext,cipher)==0){
        Bomb();
    }else{
        Foobar();
    }
}
```

(i) Crypto function.

Fig. 2: Exemplary programs that pose challenges for concolic execution.

A simple situation is that external functions do not return values, or the returned values are not employed in conditions. We demonstrate the idea with Figure 3. When commenting the printing code in line 7, only five instructions propagate the symbolic values (*i.e.,* `argv`), and the solution can be any integers equal to, or greater than 0x32. But when we enable the printing code, 61 more instructions get involved, including some conditional instructions. As a result, 0x32 no longer qualifies the constraint model. In this way, the number of available control flows for checking grows in polynomial to the complexity of `printf`. If such external functions are not our interests, we may ignore their extra constraints in this case. However, it would be incorrect if external functions return values, and the values are used in conditions. Figure 2(h) demonstrates another example, where the sine of a symbolic value is calculated via an external function call (*i.e.,* `sin`), and the result is used to determine whether a bomb should be triggered. In this situation, the conditions within the functions should not be ignored. Otherwise, it is based on an error assumption that a new test case generated under the new constraint model can always trigger the same control flow within the external function. If a program extensively uses such external functions, scalability issues would occur.

Source Code:

```c
int main(int argc, char** argv){
    if (argv[1][0] > '1'){
        printf("%s\n", "yes");
    } else{
        printf("%s\n", "no");
    }
    //printf("%d\n", argv[1][0]);
    return 0;
}
```

Tainted instructions without 'printf':

```
movsbl    (%eax),%eax
cmp       $0x31,%eax
jle       0x0000000008048463
sub       $0x4c,%esp
lea       0x8048510,%eax"
```

Constraint solving result without 'printf':
ASSERT(symb_1_166 = 0x32);

Tainted instructions with 'printf':

```
movsbl    (%eax),%eax
cmp       $0x31,%eax
jle       0x0000000008048463
lea       0x8048530,%eax
sub       $0x4c,%esp
movsbl    (%ecx),%ecx
...
je        0x00000000dc8b4147
cmpl      $0x0,-0x4b0(%ebp)
cmpl      $0x0,-0x4b4(%ebp)
je        0x00000000dc8b1b28
cmpl      $0x0,-0x4ac(%ebp)
movsbl    0x46(%esi),%eax
mov       -0x474(%ebp),%ecx
mov       %esi,(%esp)
push      %edi
```

Constraint solving result with 'printf':
ASSERT(symb_1_166 = 0x37);

Fig. 3: An example of the extra constraints incurred by external function calls. We initiate `argv[1]` to 7 and then concolic executing the program with BAP.

2) Crypto Function: Crypto functions (*e.g.,* hash function) are very complex. When employing crypto functions in a program, the number of conditional branches along the instruction trace of a concrete execution can be very large. More importantly, secure crypto functions are resistance to cryptanalysis, which implies the hardness in reverse computation. For a hash function, we cannot compute the plaintext of a hash value. For a symmetric encryption function, we cannot compute the key when given the pairs of plaintext and ciphertext.

Figure 2(i) demonstrates a code snippet which employs SHA1 function. If the hash result of the symbolic value equivalents to a predefined value, the bomb would be triggered. However, it is difficult since SHA1 cannot be reversely calculated.

Finally, we do not intend to propose a complete list of all challenges. Loop is an exception which we haven't discussed

because it has already gained much attention. Users may extend the list with new challenges following our approach.

V. Evaluation

To show that the proposed challenges are non-trivial for real-world concolic execution tools, we design a set of small-size programs which illustrate the challenges, and then evaluate them against three popular concolic execution tools. Our dataset and testing scripts are available online[1] to facilitate users to repeat our experiment.

A. Dataset

The overall idea is to test whether a code block can be explored by concolic execution tools. In our dataset, each program has been placed with a logic bomb. To trigger the bomb, a problem which illustrates a challenge has to be solved. If the bomb can be triggered by a correct test case, it implies the tool has successfully addressed the problem or *vice versa*.

Our dataset includes over 20 programs for X86_64, which cover all the discussed challenges. For each challenge, we implement several programs. Either each program involves a unique technical problem (*e.g.,* covert propagation via file), or introduces a problem with a different complexity setting (*e.g.,* one-level symbolic array, and two-level symbolic array). Table II demonstrates our program samples.

To avoid noise, each program in our dataset only reserves a simple implementation of the challenge. The sizes of the binary programs in our dataset are within the range of [10K bytes - 25K bytes], with a median of 14K bytes. In this way, the concolic execution tools have lower chances to be affected by other unexpected limitations.

B. Tools and Settings

We choose three popular concolic execution tools for evaluation: BAP, Angr, and Triton. Our choosing strategies are that: 1) the tool should be able to perform concolic execution on binaries; 2) the tool should have high impact in communities and is under maintenance; 3) it should be released as open source so that we can investigate thoroughly via code review. To our best knowledge, only these three tools can meet our standards. They are available on Github and have received hundreds of stars. By default, we use their latest stable versions for evaluation. Note that there are other famous symbolic execution tools which do not meet our requirements. For example, KLEE is a popular symbolic execution tool but cannot process binaries [12]; PySymEmu has not been updated for almost a year [23]; Mayhem is not publicly released [2].

Next, we first briefly introduce the tools and then discuss our experimental settings for them.

1) Concolic Execution Tools: BAP is an OCaml/C++ project maintained by Carnegie Mellon University [11]. It implements a Pin tool [18] for instruction tracing. The instructions are then lifted to BAP IL. BAP adopts CVC as a default constraint modeling language and employs STP [24] for constraint solving.

[1]https://github.com/hxuhack/logic_bombs

Triton is a C++/Python project maintained by QuarksLab [6]. It also leverages Pin to trace instructions. Different from BAP, it directly lifts the instructions into SSA (single static assignment), which is convenient for generating constraint models. Triton employs SMT-Lib [20] as the constraint modeling language and Z3 [22] as the constraint solver.

Angr is a python project maintained by University California of Santa Bara. The instruction lifter is based on VEX [25], which lifts the whole program into VEX IR. Then a symbolic execution engine (*i.e.,* SimuVEX) is employed to perform symbolic execution on the IR directly. To support virtual execution on IR, Andr simulates system calls in SimuVex. Angr also follows SMT-Lib to generate constraint models and uses Z3 as the constraint solver.

2) Settings: Among these tools, Triton dedicates on concolic execution, so we can use its native script for concolic execution. BAP and Angr have rich features for program analysis and require users to customize their own scripts based on tool APIs. So we should customize our testing scripts for BAP and Angr.

For Angr, our script first loads the binaries as VEX IR, and then performs directed symbolic execution [26]. In this way, the script can examine whether a bomb path is reachable and outputs the corresponding symbolic values. Angr provides two operations about whether loading dynamic libraries for analysis. For a better comparison, we report results for both the two settings separately. BAP and Triton do not have the options because all the instructions from dynamic libraries should be traced.

BAP is a primitive tool that provides no systematic support for concolic execution. It can only output values that trigger the current control flow. Therefore, our experiment for BAP includes both concolic execution and manual checking. We first execute the tool with concrete values that can trigger the bomb path. If BAP correctly solves the problem, we think it incurs no errors when handling the challenge. Then we check whether it suffers path explosions by concolic executing the program over several other concrete values and examine whether BAP can merge different values that trigger the same path. Finally, we confirm our result via reviewing the corresponding source code.

C. Evaluation Results

Our results are shown in Table II. Among 22 logic bombs, Angr achieves the best performance with four cases addressed; Triton solves 1 case, and BAP solves 2. If a reachable bomb path is deemed as unreachable, it implies an error occurs and we label the result with a corresponding error type. If a tool exits abnormally with exceptions (*e.g.,* memory out), or gives no feedback for 10 minutes, we label the result with E. It is worth noting that for all the challenges, there exist at least one test case which cannot be handled by all the tools. The results imply all the proposed challenges are non-trivial.

We further investigate the root causes of the results. Angr successfully handles variant lengths of `argv`, because it enables users to specify a fixed length of bits for the symbolic

TABLE II: Experimental results on whether a concolic execution tool can handle our challenging program. The error types which incur the failures are also reported if applicable. ✓: Success; $E_{s\#}$: Fail; E: Exit abnormally; P: Partial success for Angr if the generated symbolic values are insufficient for triggering the path due to system call simulation.

Category	Challenge	Sample Case	Tool Performance			
			BAP	Triton	Angr	Angr-NoLib
Accuracy Challenge	Symbolic Variable Declaration	Employ time info in conditions for triggering a bomb	E_{s0}	E_{s0}	E_{s0}	E_{s0}
		Employ web contents in conditions for triggering a bomb	E_{s0}	E_{s0}	E	E
		Employ the return values of system calls in conditions	E_{s0}	E_{s0}	P	P
		Employ the length of argv[1] in conditions	E_{s2}	E_{s0}	✓	✓
	Covert Symbolic Propagation	Push symbolic values into the stack and pop out	E_{s1}	✓	✓	✓
		Save symbolic values to a file and then read back	E_{s2}	E_{s2}	E	E_{s2}
		Save symbolic values via system call and then read back	E_{s2}	E_{s2}	P	P
		Change symbolic values in an exception (argv[1] = 0)	✓	E_{s1}	E	E_{s2}
		Change symbolic values in an file operation exception	E_{s2}	E_{s2}	E_{s2}	E_{s2}
	Parallel Program	Change symbolic values in multi-threads via pthread	✓	E_{s2}	E_{s2}	E_{s2}
		Change symbolic values in multi-processes via fork/pipe	E_{s2}	E_{s2}	E_{s2}	✓
	Symbolic Array	Employ symbolic values as offsets for a level-one array	E_{s3}	E_{s3}	✓	✓
		Employ symbolic values as offsets for a level-two array	E_{s3}	E_{s3}	E_{s3}	E_{s3}
	Contextual Symbolic Value	Employ symbolic values as the name of a file	E_{s2}	E_{s3}	E_{s2}	E_{s2}
		Employ symbolic values as the name of a system call	E_{s2}	E_{s3}	E_{s2}	E_{s2}
	Symbolic Jump	Employ symbolic values as unconditional jump addresses	E_{s3}	E_{s3}	E_{s2}	E_{s2}
		Employ symbolic values as offsets to an address array	E_{s3}	E_{s3}	E_{s3}	E_{s3}
	Floating-point Number	Employ floating-point numbers in symbolic conditions	E_{s1}	E_{s1}	E	E_{s3}
Scalability Challenge	External Function Call	Employ symbolic values as the parameter of sin	E_{s1}	E_{s1}	E	E_{s2}
		Employ symbolic values as the parameter of srand	E_{s2}	E	E	E_{s2}
	Crypto Function	Infer the plain text from an SHA1 result	E	E	E	E_{s2}
		Infer the key from an AES encryption result	E_{s2}	E_{s2}	E_{s2}	E_{s2}

variables. The higher bits can be fill up with 0 if they are not useful in the final symbolic values. Angr also solved the case with one-level symbolic array, because it can model memory with a map from indexes to expressions. This feature enables it to store and load values based on the index. However, Angr cannot handle more complex cases, such as two-level symbolic array, and using symbolic jump address within arrays. This implies that symbolic array still cannot be fully supported.

When Angr thinks a bomb can be triggered but generates insufficient symbolic values for triggering the bomb, we label the result as P (partial success). This is because Angr adopts system call simulation, and may simply think a system call can return any value that satisfies a constraint, which is not true actually. When there are unsupported system calls, it has higher chances to incur errors. For example, Angr cannot handle the fork case due to unsupported system calls when dynamic libraries are loaded. There are similar issues when unloading dynamic libraries into SimuVEX. In this mode, Angr doesn't need to explore the details of external functions but may think any values can be returned by external functions. Such an approach facilitates Angr to explore more paths, but it also has drawbacks in generating wrong symbolic values. To verify our idea, we design a negative bomb which is guarded under a constant false predict using square operations (*e.g.,* $x^2 == -1$). Theoretically, the bomb should not be triggered; however, Angr aggressively assigns return values to the `pow` function, and thinks the bomb path can be triggered.

For Triton and BAP, they both employ no sophisticated techniques to handle the proposed challenges and suffer many failures related to E_{s2} and E_{s3}. Besides, there are several E_{s1}, because the binaries contain instructions that cannot be lifted. Specifically, Triton does not support the floating-point instructions, such as `cvtsi2sd` and `ucomisd`. When symbolic variables are propagated in such instructions, the error occurs and propagates. As a result, the constraint models may either contain no symbolic variables or have wrong expressions.

For scalability challenges, no tool can solve any of the cases correctly. Our expected result for scalability problem is E, which means the concolic executor exits abnormally due to resource constraint. However, only half of our results are E, while the other half of them are $E_{s\#}$. This is because some errors happen during symbolic reasoning.

Finally, even for the same case, the failures of different tools may be incurred with different error types. For example, Angr thinks the bomb path in the symbolic jump case is reachable but calculates a wrong result. It is likely that the error is incurred during data propagation. In comparison, BAP and Triton even do not have mechanisms to handle such jump, and the failure should be caused by a constraint extraction issue.

D. Lessons Learnt

1) Limitations of Concolic Execution: Our experimental result shows that even the state-of-the-art concolic execution tools are far less than perfect. There are several non-trivial accuracy challenges which cannot be easily addressed. As a result, real-world concolic execution tools are not as reliable as their theoretical models. Also, even small-size programs

may incur scalability issues. Understanding the limitations of concolic execution tools and knowing the characteristics of the target codes beforehand would be helpful for users to employ the technique appropriately.

2) Application Issue: We use two major application scenarios to illustrate the impacts of the challenges. The first scenario is bug detection [1]. Bugs may happen at any place in any control flow. Thus achieving high control flow coverage is essential for detecting bugs. However, our discussed challenges are prevalent in real-world programs [15], and thus pose concolic execution fails in achieving an ideal coverage. Integrating other testing approaches, such as random testing or fuzz testing, are helpful for bug detection. Besides, some concolic execution tools leverage simulation techniques to improve the coverage, but such tools may incur many false positives.

Another scenario is deobfuscation. Obfuscation generally increases code complexity by introducing opaque predicates (*e.g.,* constant) and bogus codes, while deobfuscation removes the obscurity and redundancy. Theoretically, concolic execution is effective for deobfuscating such programs via dead code (*i.e.,* bogus codes) elimination. However, when composing opaque predicates leveraging the challenges we investigated in this paper, it would incur troubles for deobfuscation.

VI. Conclusion

To summarize, this paper serves as a first attempt to investigate the challenges of concolic execution on small-size binary programs. We have systematically proposed four types of errors which may occur in different symbolic reasoning stages, and seven challenges that can incur such errors in real-world concolic execution tasks. We have also proposed two challenges that may incur scalability issues when performing concolic execution on small-size programs. To show that the proposed challenges are non-trivial, we have conducted real-world experiments, which includes a set of 22 binary programs and three most popular concolic execution tools. To facilitate further study in this area, we release our dataset as open source. This paper would serve as an essential reference for concolic execution researchers to improve the technique, and for the users to properly use it.

Acknowledgments

This work was supported by the Key Project of National Natural Science Foundation of China (Project No. 61332010), the National Basic Research Program of China (973 Project No. 2014CB347701), and the Research Grants Council of the Hong Kong Special Administrative Region, China (No. CUHK 14234416 of the General Research Fund). Yangfan Zhou is the corresponding author.

References

[1] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, "Aeg: Automatic exploit generation," in *Proc. of the 2011 ACM the Network and Distributed System Security Symposium*, 2011.

[2] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *Proc. of the 2012 IEEE Symposium on Security and Privacy*, 2012.

[3] J. Ming, D. Xu, L. Wang, and D. Wu, "Loop: Logic-oriented opaque predicate detection in obfuscated binary code," in *Proc. of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[4] B. Yadegari and S. Debray, "Symbolic execution of obfuscated code," in *Proc. of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[5] Y. Shoshitaishvili and *et al.*, "Sok: (state of) the art of war: Offensive techniques in binary analysis," in *Proc. of the IEEE Symposium on Security and Privacy*, 2016.

[6] F. Saudel and J. Salwan, "Triton: a dynamic symbolic execution framework," in *Symposium sur la sécurité des technologies de linformation et des communications, SSTIC, France, Rennes*, 2015.

[7] "Crackme puzzles solved with Angr," https://github.com/angr/angr-doc/blob/master/docs/examples.md/.

[8] DAPRA, "Cyber Grand Challenge," http://www.cybergrandchallenge.com/.

[9] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, and G. Candea, "Cloud9: a software testing service," *ACM SIGOPS Operating Systems Review*, vol. 43, no. 4, pp. 5–10, 2010.

[10] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, "Fitness-guided path exploration in dynamic symbolic execution," in *Proc. of the 39th IEEE/IFIP International Conference on Dependable Systems & Networks*, 2009.

[11] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "Bap: A binary analysis platform," in *Proc. of the International Conference on Computer Aided Verification*. Springer, 2011.

[12] C. Cadar, D. Dunbar, and D. R. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. of the 8th USENIX Symposium on Operating Systems Design and Implementation*, 2008.

[13] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Proc. of the 2010 IEEE Symposium on Security and Privacy*, 2010.

[14] X. Qu and B. Robinson, "A case study of concolic testing tools and their limitations," in *Proc. of the IEEE International Symposium on Empirical Software Engineering and Measurement*, 2011.

[15] L. Cseppento and Z. Micskei, "Evaluating symbolic execution-based test tools," in *Proc. of the IEEE 8th International Conference on Software Testing, Verification and Validation*, 2015.

[16] R. Kannavara, C. J. Havlicek, B. Chen, M. R. Tuttle, K. Cong, S. Ray, and F. Xie, "Challenges and opportunities with concolic testing," in *Aerospace and Electronics Conference (NAECON), 2015 National*. IEEE, 2015, pp. 374–378.

[17] C. A. R. Hoare, "An axiomatic basis for computer programming," *Communications of the ACM*, 1969.

[18] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *ACM Sigplan Notices*, vol. 40, no. 6, 2005, pp. 190–200.

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

[20] C. Barrett, A. Stump, and C. Tinelli, "The SMT-LIB Standard: Version 2.0," in *Proc. of the 8th International Workshop on Satisfiability Modulo Theories*, 2010.

[21] N. Sorensson and N. Een, "Minisat v1.13-a sat solver with conflict-clause minimization," 2005.

[22] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," in *Proc. of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008.

[23] "PySymEmu," https://github.com/feliam/pysymemu/.

[24] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *Proc. of the International Conference on Computer Aided Verification*. Springer, 2007.

[25] N. Nethercote, "Dynamic binary analysis and instrumentation," Ph.D. dissertation, PhD thesis, University of Cambridge, 2004.

[26] K. Ma, K. Y. Phang, J. S. Foster, and M. Hicks, "Directed symbolic execution," in *Proc. of the International Static Analysis Symposium*. Springer, 2011.