Benchmarking the Capability of Symbolic Execution Tools with Logic Bombs

Hui Xu, *Member, IEEE*, Zirui Zhao, Yangfan Zhou, *Member, IEEE*, and Michael R. Lyu, *Fellow, IEEE*

Abstract—Symbolic execution has become an indispensable technique for software testing and program analysis. However, since several symbolic execution tools are presently available off-the-shelf, there is a need for a practical benchmarking approach. This paper introduces a fresh approach that can help benchmark symbolic execution tools in a fine-grained and efficient manner. The approach evaluates the performance of such tools against known challenges faced by general symbolic execution techniques, e.g., floating-point numbers and symbolic memories. We first survey related papers and systematize the challenges of symbolic execution. We extract 12 distinct challenges from the literature and categorize them into two categories: *symbolic-reasoning challenges* and *path-explosion challenges*. Next, we develop a dataset of logic bombs and a framework for benchmarking symbolic execution tools automatically. For each challenge, our dataset contains several logic bombs, each addressing a specific challenging problem. Triggering one or more logic bombs confirms that the symbolic execution tool in question is able to handle the corresponding problem. Real-world experiments with three popular symbolic execution tools, namely, KLEE, angr, and Triton have shown that our approach can reveal the capabilities and limitations of the tools in handling specific issues accurately and efficiently. The benchmarking process generally takes only a few dozens of minutes to evaluate a tool. We have released our dataset on GitHub as open source, with an aim to better facilitate the community to conduct future work on benchmarking symbolic execution tools.

Index Terms—Symbolic execution

1 Introduction

Symbolic execution is a popular technique for software testing and program analysis [1]. It has experienced rapid development over the last decade. As a result, several open-source symbolic execution tools such as KLEE [2] and angr [3] have become available. Current methods for evaluating symbolic execution tools generally rely on the code coverage achieved or the number of bugs detected in real-world programs [2], [4]. However, the performances of such metrics often depend on the particular type of programs being analyzed. Also the metrics cannot fully capture the detailed capabilities or limitations of the symbolic execution tool. This paper proposes a fine-grained benchmarking approach which is less sensitive to targeting programs.

There are certain challenging problems most symbolic execution tools are unable to handle well, e.g., floating-point numbers [5] and loops [6]. Since these point to factors determining the code coverage that a symbolic execution tool can achieve, we develop a benchmarking approach based on known challenges. Specifically, we view each challenge as an evaluation metric such that we can extract more meaningful information concerning the capability of the symbolic execution tool. The benchmarking result is unbiased as it does not depend on particular programs for analysis.

We first conduct a systematic survey of the challenges associated with symbolic execution. This step is essential for ensuring that our benchmarking approach is capable of addressing as many distinct challenges as possible. We categorize existing challenges into two categories: *symbolic-reasoning challenges* and *path-explosion challenges*. Symbolic-reasoning challenges attack the core symbolic reasoning process, whenever it incurs errors for symbolic execution tools to generate incorrect test cases for particular control flows. These challenges include symbolic variable declarations, covert propagations, parallel executions, symbolic memories, contextual symbolic values, symbolic jumps, floating-point numbers, buffer overflows, and arithmetic overflows. Path-explosion challenges introduce too many possible control flows to analyze, which may cause a symbolic execution tool starving the computational resources or spending very long time on exploring the paths. Not only large-sized programs but also small-sized programs can lead to path-explosion issues, arising from complex routines, such as external function calls, loops, and crypto functions. This ensures that all existing challenges discussed in the literature can be well categorized.

Next, we develop an accurate and efficient approach for benchmarking the capability of symbolic execution tools with respect to each of the challenges. Interestingly, we cannot employ real-world programs for testing because they are too complicated and any challenges could lead to a

- *H. Xu and M. R. Lyu are with the Shenzhen Research Institute and Department of Computer Science & Engineering, The Chinese University of Hong Kong, Hong Kong, China. E-mail: {hxu, lyu}@cse.cuhk.edu.hk.*
- *Z. Zhao is with the Chinese University of Hong Kong, Hong Kong, China, and also with the The University of Science and Technology of China, Hefei, Anhui 230000, China. E-mail: zzrcxb@mail.ustc.edu.cn.*
- *Y. Zhou is with the School of Computer Science, Shanghai Key Laboratory of Intelligent Information Processing, Fudan University, Shanghai 200433, China. E-mail: zyf@fudan.edu.cn.*

Manuscript received 2 Jan. 2018; revised 2 Aug. 2018; accepted 12 Aug. 2018. Date of publication 21 Aug. 2018; date of current version 12 Nov. 2020. (Corresponding author: Yangfan Zhou.)
Digital Object Identifier no. 10.1109/TDSC.2018.2866469

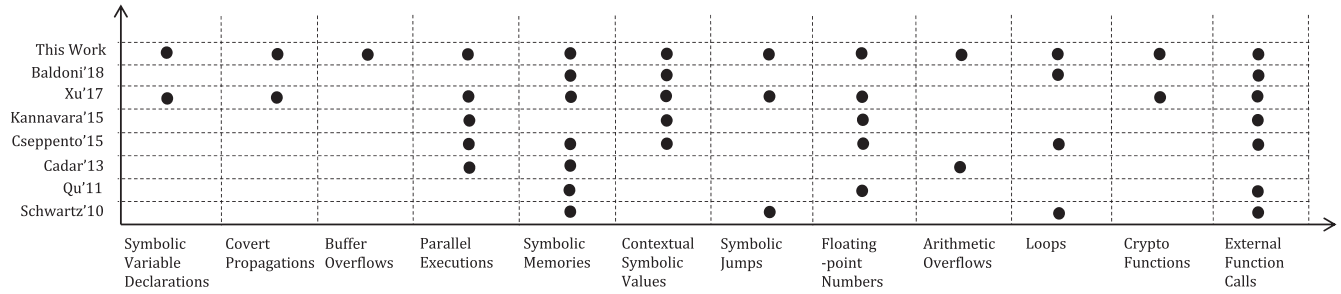


Fig. 1. Major challenges of symbolic execution as discussed in the literature. The detailed paper references are Schwartz'10 [7], Qu'11 [8], Cadar'13 [6], Cseppento'15 [9], Kannavara'15 [10], Quan'16 [5], Xu'17 [11], and Baldoni'18 [12].

failure. Moreover, symbolic execution itself is not very efficient, and benchmarking using real-world programs generally takes a long time. We tackle this problem by designing small programs embedded with logic bombs. A logic bomb is an artificial code block that can only be executed when certain conditions are met. We create logic bombs that can be triggered only when a challenging problem is solved. The benefits of doing this are two-fold. First, since we keep each logic bomb as small as possible, our evaluation result is less likely to be affected by other unexpected issues. Second, employing small programs can shorten the required symbolic execution time and improves efficiency.

Following this method, we designed a dataset of logic bombs covering all the challenges. This lead to the development of a framework for running benchmarking experiments automatically. For each challenge, our dataset contains several logic bombs with different problem settings or different levels of hardness, e.g., a one-leveled array or a two-leveled array is designed for the specific symbolic memory challenge. Our framework employs the dataset of logic bombs as evaluation metrics. It first parses the logic bombs and compiles them to object codes or binaries. Next, it directs a symbolic execution tool to symbolically execute the logic bombs in a batch mode. Finally, it verifies the test cases generated and produces reports. We have released our dataset and framework tools on GitHub.

We have conducted real-world experiments to benchmark three popular symbolic execution tools: KLEE [2], Triton [13], and angr [3]. Although these tools adopt different implementation techniques, our framework can adapt to them with only a little customization. The benchmarking process for each tool usually takes dozens of minutes. Experimental results have shown that our benchmarking approach can reveal their capabilities and limitations accurately and efficiently. Overall, angr has achieved the best performance with 22 cases solved, which is roughly one third of the total logic bombs; KLEE solved nine cases; and Triton could only solve three cases. We manually checked the reported solutions and confirmed that they were all nontrivial and consistent. Moreover, our results lead to certain interesting findings about these tools. For example, angr only supports one-leveled arrays but not two-leveled arrays while Triton does not even support the `atoi` function. Most of our findings are new, which further justifies the value of our benchmarking approach.

The rest of the paper is organized as follows. We first discuss the related work in Section 2 and introduce the preliminary knowledge of symbolic execution in Section 3. Next, in Section 4, we examine the challenges of symbolic execution. Section 5 introduces our benchmarking methodology while Section 6 presents our experiments and results. Finally, Section 7 concludes our paper.

2 Related Work

This section compares our work against papers that either systematize the challenges of symbolic execution or employ the challenges to evaluate symbolic execution tools. Note that although symbolic execution has received extensive attention for decades, only a few papers include a systematic discussion about the challenges associated with the technique being examined. Previous work in this area has focused mainly on how the technique could be used to carry out specific software analysis tasks (e.g., [14], [15], [16]), or proposing new approaches to improve the technology in relation to specific challenges (e.g., [17], [18], [19]).

Papers that focus on systematizing the challenges of symbolic execution tools include [8], [9], [10], [20]. Kannavara et al. [10] enumerated several challenges that may hinder the adoption of symbolic execution in industrial fields. Qu and Robinson [8] conducted a case study on the limitations of symbolic testing tools and examined their prevalence in real-world programs. However, neither paper provided a method to evaluate symbolic execution tools. Cseppento and Micskei [9] proposed several metrics to evaluate source-code-based symbolic execution tools. But their metrics are based on specific program syntax of object-oriented codes rather than on language-independent challenges. Further, these metrics were not general enough to permit symbolic execution. Banescu et al. [20] designed several small programs for evaluation. But their purpose was to evaluate the resilience of code obfuscation transformations against symbolic execution-based attacks. They did not investigate the capability of symbolic execution tools; they simply trusted KLEE as a state-of-the-art symbolic executor. Besides, there have been several survey papers (e.g., [6], [7], [12]) which also include some discussion about the challenges. Fig. 1 provides a more complete view of the challenges discussed in these surveys.

In one of our previous conference papers [11], we have presented an empirical study examining some of the challenges. This paper extends our previous paper with a formal benchmarking methodology and serves as a pilot study systematically benchmarking symbolic execution tools in handling particular challenges. In this paper, we design a novel benchmarking framework based on logic bombs, which can facilitate the automation of the benchmarking process. We

further provide a benchmarking toolset that can be deployed easily by ordinary users.

3 Preliminary

This section reviews the underlying techniques of symbolic execution as a prelude to discussing the challenges and our new benchmarking approach.

3.1 Theoretical Basis

The core principle of symbolic execution is symbolic reasoning. Informally, given a sequence of instructions along a control path, a symbolic reasoning engine extracts a constraint model and generates a test case for the path by solving the model.

Formally, we can use Hoare Logic [21] to model the symbolic reasoning problem. Hoare Logic is composed of basic triples $\{S_1\}I\{S_2\}$, where $\{S_1\}$ and $\{S_2\}$ are the assertions of variable states and I is an instruction. The Hoare triple tells if a precondition $\{S_1\}$ is met, when executing I, it will terminate with the postcondition $\{S_2\}$. Using Hoare Logic, we can model the semantics of instructions along a control path as:

$$\{S_0\}I_0\{S_1, \Delta_1\}I_1...\{S_{n-1}, \Delta_{n-1}\}I_{n-1}\{S_n\}.$$

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

$$\begin{aligned}
\{S_{n-2}\} &= wp(I_{n-2}\{S_{n-1}\}), \quad s.t.\ \{S_{n-1}\}\ sat\ \Delta_{n-1}\\
\{S_{n-3}\} &= wp(I_{n-3}\{S_{n-2}\}), \quad s.t.\ \{S_{n-2}\}\ sat\ \Delta_{n-2}\\
&...\\
\{S_1\} &= wp(I_1\{S_2\}), \quad s.t.\ \{S_2\}\ sat\ \Delta_2\\
\{S_0\} &= wp(I_0\{S_1\}), \quad s.t.\ \{S_1\}\ sat\ \Delta_1.
\end{aligned}$$

Combining the constraints in each line, we can get a constraint model in conjunction normal form: $\Delta_1 \wedge \Delta_2 \wedge ... \wedge \Delta_{n-1}$. The solution to the constraint model is a test case $\{S_0'\}$ that can trigger the same control path.

Finally, while sampling $\{I_i\}$, not all instructions may be found to be 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. If E does not depend on any symbolic value, X would be a constant, and should not be included in the weakest preconditions. In practice, it can be realized by symbolic execution tools (e.g., Mayhem [22] and FuzzBALL [23]) using taint analysis techniques [7].

3.2 Symbolic Execution Framework

Fig. 2 demonstrates the conceptual framework of a symbolic execution tool. It involves inputting a program and outputting test cases for the program. The framework includes a core symbolic reasoning engine and a path selection engine.

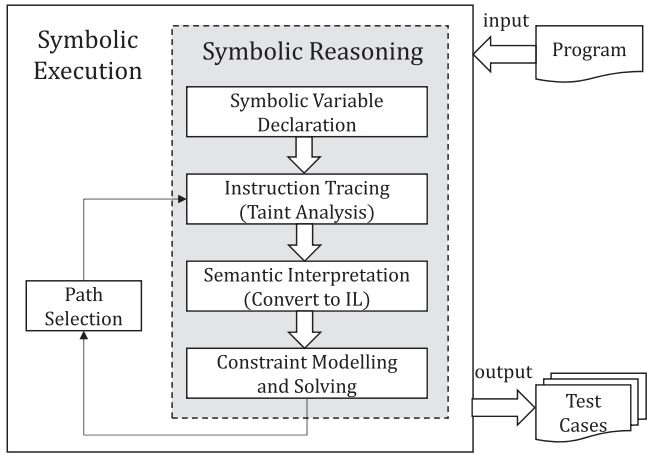


Fig. 2. Conceptual framework for symbolic execution.

The symbolic reasoning engine analyzes the instructions along a path and generates test cases that can trigger the path. Based on the symbolic reasoning, we can identify four stages: symbolic variable declaration, instruction tracing, semantic interpretation, and constraint modeling and solving. The details are as follows:

- *Symbolic variable declaration* (S_{var}): In this stage, we have to declare symbolic variables which will be employed in the following symbolic analysis process. If some symbolic variables are missing from declaration, insufficient constraints can be generated for triggering a control path.
- *Instruction tracing* (S_{inst}): This stage collects the instructions along control paths. If some instructions are missing, or the syntax is not supported, the symbolic reasoning process would be inconsistent.
- *Semantic interpretation* (S_{sem}): This stage translates the semantics of collected instructions with an intermediate language (IL). If some instructions are incorrectly interpreted, or the data propagations are incorrectly modeled, the symbolic execution engine would consequently generate inconsistent constraint models.
- *Constraint modeling and solving* (S_{model}): This stage generates constraint models from IL, and then solves them. If the required satisfiability modulo theory is unsupported, errors are likely.

The path selection engine determines which path should be analyzed in the next round of symbolic reasoning. The favorited strategies include depth-first search, width-first search, random search, etc. [12].

3.3 Implementation Variations

According to the different ways of instruction tracing, we can classify symbolic execution tools into static symbolic execution (e.g., KLEE [2], [24]) and dynamic symbolic execution (e.g., Triton [13]). Static symbolic execution loads a whole program first before extracting instructions along with a path on the program control-flow graph (CFG). Dynamic symbolic execution is also known as concolic (concrete and symbolic) execution. It collects instructions which have been actually executed. In each round, the concolic execution engine executes the program with concrete values to generate instructions [17].

TABLE 1
List of Challenges Faced by Symbolic Execution, and the Symbolic Reasoning Stages They Attack

Challenge		Idea	Stage of Error		
			S_{var}	S_{inst} & S_{sem}	S_{model}
Symbolic-reasoning Challenges	Sym. Var. Declaration	Contextual variables besides program arguments	✓	✓	✓
	Covert Propagations	Propagating symbolic values in covert ways	-	✓	✓
	Buffer Overflows	Writing symbolic values without proper boundary check	-	✓	✓
	Parallel Executions	Processing symbolic values with parallel codes	-	✓	✓
	Symbolic Memories	Symbolic values as the offset of memory	-	✓	✓
	Contextual Symbolic Values	Retrieving contextual values with symbolic values	-	✓	✓
	Symbolic Jumps	Sym. values as the addresses of unconditional jump	-	-	✓
	Floating-point Numbers	Symbolic values in float/double type	-	-	✓
	Arithmetic Overflows	Integers outside the scope of an integer type	-	-	✓
Path-explosion Challenges	Loops	Change symbolic values within loops	-	-	-
	Crypto Functions	Processing symbolic values with crypto functions	-	-	-
	External Function Calls	Processing sym. values with some external functions	-	-	-

We may also classify symbolic execution tools into source-code-based symbolic execution and binary-code-based symbolic execution. In general, we do not perform symbolic reasoning on source codes or binaries directly. A prior step is to interpret the semantics of the program with an intermediate language (IL). Therefore, the main difference between the two implementation methods lies in the translation process. Regarding source codes, we can translate the code directly with the compiler's frontend. As for binaries, we have to lift the assembly codes into IL, which is error-prone due to the complicated features of modern CPUs [25]. The lifting process is challenging and remains as an active research area.

4 Challenges of Symbolic Execution

Based on whether a challenge is associated with the symbolic reasoning process, we can categorize the challenges of symbolic execution into *symbolic-reasoning challenges* and *path-explosion challenges*. A symbolic-reasoning challenge attacks the symbolic reasoning process and leads to incorrect test cases being generated. A path-explosion challenge happens when there are too many paths to analyze. It does not attack a single symbolic reasoning process, but may get starved of computational resources or require a very long time for symbolic execution.

Table 1 lists the challenges that we have investigated in this work. We collected the challenges via a careful survey of existing papers. The survey coverred several survey papers realated to symbolic execution techniques (e.g., [6], [7], [12]), several investigations that focus on systemizing the challenges of symbolic execution (e.g., [9], [10]), and other important papers related to symbolic execution (e.g., [16], [17], [22], [26], [27], [28], [29], [30]).

4.1 Symbolic-Reasoning Challenges

We now discuss nine challenges that may incur errors to symbolic reasoning.

4.1.1 Symbolic Variable Declarations

Since the test cases are the solutions of symbolic variables subject to constrain models, symbolic variables should be declared before a symbolic reasoning process. For example, in source-code-based symbolic execution tools (e.g., KLEE [2]), users can manually declare symbolic variables in the source codes. Binary-code-based concolic execution tools (e.g., Triton [13]) generally assume a fixed length of program arguments from stdin as the symbolic variable. If some symbolic variables are missing from the declaration, the generated test cases would be insufficient for triggering particular control paths. Since the root cause occurs before symbolic execution, the challenge attacks S_{var}.

Fig. 3a is a sample with a symbolic variable declaration problem. It returns a `BOMB_ENDING` only when being executed with a particular process id. To explore the path, a symbolic execution tool should treat `pid` as a symbolic variable and then solve the constraint with respect to `pid`. Otherwise, it cannot find test cases that can trigger the path.

To declare symbolic variables precisely, a user should know target programs well. However, the task is impossible when analyzing programs on a large scale, e.g., when performing malware analysis. In an ideal case, a symbolic execution tool should automatically detect such variables which can control program behaviors and report the solutions accordingly. To our best knowledge, very few tools have implemented this feature, except DART [26]. Instead, present papers (e.g., [12], [31]) generally discuss the challenge together with other problems related to the computing environment, such as libraries, kernels, and drivers. In reality, there are several challenges of this work referring to the computing environment, such as contextual symbolic variables, covert propagations, parallel executions, and external function calls. We demonstrate that these challenges are different.

4.1.2 Covert Propagations

Some data propagation ways are covert because they cannot be traced easily by data-flow analysis tools. For example, if the symbolic values are propagated via other media (e.g., files) outside of the process memory, the propagation would be untraceable. Such propagation methods are undecidable and can be beyond the capability of pure program analysis. Symbolic execution tools have to handle such cases using ad hoc methods. There are also some propagations challenging only to certain implementations. For example, propagating symbolic values via embedded assembly codes can

```
int logic_bomb() {
  int pid = (int) getpid();
  printf ("current pid is %d\n%", pid);
  if(pid == 4096)
    return BOMB_ENDING;
  else
    return NORMAL_ENDING;
}
```

(a) Symbolic variable declarations.

```
char* shell(const char* cmd){
  char* ret = "";
  FILE *f = popen(cmd, "r");
  char buf[1024];
  memset(buf,'\0',sizeof(buf));
  while(fgets(buf,1024-1,f)!=NULL)
    ret = buf;
  pclose(f);
  return ret;
}
int logic_bomb(char* symvar) {
  int i=symvar[0]-48;
  char cmd[256];
  sprintf(cmd, "echo %d\n", i);
  char* ret = shell(cmd);
  if(atoi(ret) == 7){
    return BOMB_ENDING;
  }
  return NORMAL_ENDING;
}
```

(b) Covert symbolic propagations.

```
int logic_bomb(char* symvar) {
  int flag = 0;
  char buf[8];
  strcpy(buf, symvar);
  if(flag == 1){
    return BOMB_ENDING;
  }
  return NORMAL_ENDING;
}
```

(c) Buffer overflows.

```
int threadprop(int in){
  pthread_t tid[2];
  int rc1 = pthread_create(&tid[0], NULL, Inc, (void *) &in);
  int rc2 = pthread_create(&tid[1], NULL, Mult, (void *) &in);
  rc1 = pthread_join(tid[0], NULL);
  rc2 = pthread_join(tid[1], NULL);
  int out = in;
  return out;
}
int logic_bomb(char* symvar) {
  int i=symvar[0]-48;
  int j=threadprop(i);
  if(j == 50){
    return BOMB_ENDING;
  }
  return NORMAL_ENDING;
}
```

(d) Parallel executions.

```
int logic_bomb(char* symvar) {
  int i=symvar[0]-48;
  int array[] ={1,2,3,4,5};
  if(array[i%5] == 5){
    return BOMB_ENDING;
  }
  else
    return NORMAL_ENDING;
}
```

(e) Symbolic memories.

```
int logic_bomb(char* symvar) {
  FILE *fp = fopen(symvar, "r");
  if(fp != NULL){
    fclose(fp);
    return BOMB_ENDING;
  }else{
    return NORMAL_ENDING;
  }
}
```

(f) Contextual symbolic values.

```
int f0() {return 0;}  ...  int f6() {return 6;}
int logic_bomb(char* symvar) {
  int (*func[7])() = {f0, f1, f2, f3, f4, f5, f6};
  int ret = func[(symvar[0] - 48)%7]();
  if(ret == 5){
      return BOMB_ENDING;
  }
  return NORMAL_ENDING;
}
```

(g) Symbolic jumps.

```
int logic_bomb(char* symvar) {
  int i=symvar[0]-48;
  float a = i/70.0;
  float b = 0.1;
  if(a != 0.1 & a - b == 0){
      return BOMB_ENDING;
  }
  return NORMAL_ENDING;
}
```

(h) Floating-point numbers.

```
int logic_bomb(char* symvar) {
  int i = symvar[0] - 48;
  if (254748364 * i < 0 && i > 0){
    return BOMB_ENDING;
  }
  return NORMAL_ENDING;
}
```

(i) Arithmetic overflows.

```
int logic_bomb(char* symvar) {
  int i = symvar[0] - 48;
  float v = sin(i * PI / 30);
  if(v > 0.5){
    return BOMB_ENDING;
  }
  return NORMAL_ENDING;
}
```

(j) External function calls.

```
int f(int x){
  if (x % 2 == 0)
    return x / 2;
  return 3 * x + 1;
}
int logic_bomb(char* symvar) {
  int i = symvar[0]-48+94;
  int j = f(i);
  int loopcount = 1;
  while(j != 1){
    j = f(j);
    loopcount ++;
  }
  if(loopcount == 25)
    return BOMB_ENDING;
  else
    return NORMAL_ENDING;
}
```

(k) Loops.

```
int logic_bomb(char* symvar) {
  int plaintext = symvar[0] - 48;
  unsigned cipher[5];
  cipher[0] = 0X77de68da;
  cipher[1] = 0Xecd823ba;
  cipher[2] = 0Xbbb58edb;
  cipher[3] = 0X1c8e14d7;
  cipher[4] = 0X106e83bb;
  if(SHA1_COMP(plaintext,cipher)==0){
    return BOMB_ENDING;
  }else{
    return NORMAL_ENDING;
  }
}
```

(l) Crypto functions.

Fig. 3. Logic bomb samples with challenging symbolic execution issues. In each sample, we employ `symvar` to denote a symbolic variable, and `BOMB_ENDING` to denote a macro value indicating a particular program behavior.

be a problem for source-code-based symbolic execution tools only. If a symbolic execution tool fails to detect certain propagations, the instructions related to the propagated values would be missed from the following analysis. This results in the challenge attacking the stages of S_{inst} and S_{sem}.

Fig. 3b shows a covert propagation sample. We define an integer `i` and initiate it with the value of a symbolic variable `symvar`. So `i` is also a symbolic variable. We then propagate the value of `i` to another variable `ret` through a shell command `echo`, and let `ret` control the return value. To find a test case which can return the corresponding `BOMB_ENDING`, a symbolic execution tool should properly track or model the propagation incurred by the shell command.

4.1.3 Buffer Overflows

Buffer overflow is a typical software bug that can bring security issues. Due to insufficient boundary checking, the input data may overwrite adjacent memories. Adversaries can employ such bugs to inject data and intentionally and tamper with the semantics of the original codes. Buffer overflows can happen in either stack or heap regions. If a symbolic execution tool cannot detect the overflow issues arising, it would fail to track the propagation of symbolic values. Therefore, buffer overflow involves a particular covert propagation issue. Source-code-based symbolic execution tools are prone to buffer overflows because the stack layout of a program exists only in the assembly codes, depending on the particular platforms. Therefore, such tools cannot model stack information using source codes only. In contrast, binary-code-based symbolic execution tools should be more potent in handling buffer overflow issues because they can simulate actual memory layouts. However, even if these tools can precisely track propagation, they suffer from difficulties in automatically analyzing the unexpected program behaviors caused by overflow. Otherwise, they would be powerful enough to generate exploits for bugs, which is a problem still requiring solution [32].

Fig. 3c presents an example of buffer overflows. The program returns a `BOMB_ENDING` if the value of `flag` equals one, which is unlikely because the value is zero and should remain unchanged without explicit modification. However, the program has a buffer overflow bug. It has a buffer `buf` of eight bytes and employs no boundary check when copying symbolic values to the buffer with `strcpy`. We can change the value of `flag` to one leveraging the bug, e.g., when `symvar` is "ANYSTRIN\x01\x00\x00\x00".

4.1.4 Parallel Executions

Classic symbolic execution is effective for sequential programs. We can draw an explicit CFG for sequential programs

and let a symbolic execution engine traverse the CFG. However, if the program processes symbolic variables in parallel, classic symbolic execution techniques would face problems. Parallel programs can be undecidable because the execution order of parallel codes does not only depend on the program but may also depend on the execution context. A parallel program may exhibit different behaviors even with the same test case. This poses a problem for symbolic execution to generate test cases for triggering corresponding control flows. If a symbolic execution tool directly ignores the parallel syntax or addresses the syntax improperly, errors would happen during S_{inst} and S_{sem}.

Fig. 3d demonstrates an example with parallel codes. The symbolic variable `i` is processed by another two additional threads in parallel, and the result is assigned to `j`. Then the value of `j` determines whether the program should return a `BOMB_ENDING`.

To handle parallel codes, the symbolic execution tool has to interpret the semantics and track parallel executions, e.g., by introducing extra symbolic variables [33]. However, such an approach may not be scalable because the possibility of parallel execution can be a large number. In practice, there are several heuristic approaches that can be used to improve the efficiency. For example, we may restrict the exploration time of concurrent regions with a threshold [33]; we may conduct symbolic execution with arbitrary contexts and convert multi-thread programs into equivalent sequential ones [34]; or we can prune unimportant paths leveraging some program codes, such as assertion [35].

4.1.5 Symbolic Memories

Symbolic memory is a situation whereas symbolic variables serve as the offsets or pointers to retrieve values from the memory, such as array indexes. While handling symbolic memories, the symbolic execution engine should take advantage of the memory layout for analysis. For example, we can convert an array selection operation to a `switch/case` clause in which the number of possible cases equals the length of the array. However, the number of possible combinations would grow exponentially when there are several such operations along a control flow. In practice, a symbolic execution tool may directly employ the feature of array operations implemented by some constraint solvers, such as STP [36] and Z3 [37]. It may also analyze the alignment of some pointers in advance, such as CUTE [38]. However, the power of pointer analysis is limited because the problem can be NP-hard or even undecidable for static analysis [39]. If a symbolic execution tool cannot model symbolic memories properly, errors would occur during S_{inst} and S_{sem}.

Fig. 3e presents a sample of symbolic memories. In this example, the symbolic variable `i` serves as an offset to retrieve an element from the array. The retrieved element then determines whether the program returns a `BOMB_ENDING`.

4.1.6 Contextual Symbolic Values

This challenge is similar to symbolic memories but is more complicated. Other than retrieving values from the memory like symbolic memories, symbolic values can also serve as the parameters to retrieve values from the environment, such as loading the contents of a file pointed by symbolic values. By default, this contextual information is unavailable to the program or process, and the analysis is more complicated. Moreover, since the contextual information can be changed any time without informing the program, the problem is undecidable. A symbolic tool that does not support such operations would cause errors during S_{inst} and S_{sem}.

Fig. 3f is an example of contextual symbolic values. If `symvar` points to an existing file on the local disk, the program returns a `BOMB_ENDING`.

4.1.7 Symbolic Jumps

In general, symbolic execution only extracts constraint models when encountering conditional jumps, such as `var<0` in source codes, or `jle 0x400fda` in assembly codes. However, we may also employ unconditional jumps to achieve the same effects as conditional jumps. The idea is to jump to an address controlled by symbolic values. If a symbolic execution engine is not tailored to handle such unconditional jumps, it would fail to extract corresponding constraint models and miss some available control flows. Therefore, the challenge attacks the constraint modeling stage S_{model}.

Fig. 3g presents an example of symbolic jumps. The program contains an array of function pointers, and each function returns an integer value. The symbolic variable serves as an offset to determine which function should be called during execution. If `f5()` is called, the program would return a `BOMB_ENDING`.

4.1.8 Floating-Point Numbers

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 $f = sign \times base^{exp}$. For example, the 32-bit float type compliant to IEEE-754 has 1-bit for $sign$, 23-bit for $base$, and 8-bit for exp. This 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 limited precision, which turns some unsatisfiable constraints over $\mathbb{R}$ into satisfiable ones over $\mathbb{F}$ with a rounding mode. In order to support reasoning over $\mathbb{F}$, a symbolic execution tool should consider such approximations when extracting and solving constraint models. However, recent studies (e.g., [5], [40], [41], [42]) show that there is still no silver bullet for the problem. Floating-point numbers continue to pose a challenge for symbolic execution tools, and the challenge attacks S_{model}.

Fig. 3h demonstrates an example with floating-point operations. Because we cannot represent 0.1 with float type precisely, the first predicate `a != 1` is always true. If the second condition `a == b` can be satisfied, the program would return a `BOMB_ENDING`. Therefore, one test case to returning a `BOMB_ENDING` is `symvar` equals `'7'`.

4.1.9 Arithmetic Overflows

Arithmetic overflow happens when the result of an arithmetic operation is outside the range of an integer type. For example, the range of a 64-bit signed integer is $[-2^{64}, 2^{64}-1]$. In this case, a constraint model (e.g., the result of a positive integer plus another positive integer is negative)

may have no solutions over $\mathbb{R}$; but it can have solutions when we consider arithmetic overflow. Handling such arithmetic overflow issues is not as difficult as in the case of the previous challenges. However, some preliminary symbolic execution tools may fail to consider these cases and suffer errors when extracting and solving the constraint models.

Fig. 3i shows a sample with an arithmetic overflow problem. To meet the first condition `254748364 * i < 0`, `i` should be a negative value. However, the second condition requires `i` to be a positive value. Therefore, it has no solutions in the domain of real numbers. But the conditions can be satisfied when `254748364 * i` exceeds the max value that the integer type can represent.

4.2 Path-Explosion Challenges

Now we discuss three path-explosion challenges existing in small-size programs.

4.2.1 External Function Calls

Shared libraries, such as `libc` and `libm` (i.e., a maths library), provide some basic function implementations to facilitate software development. An efficient way to employ the functions is via dynamic linkage, which does not pack the function body to the program but only links with the functions dynamically during execution. Therefore, such external functions do not enlarge the size of a program; they just enlarge code complexity.

When an external function call is related to the propagation of symbolic values, the control flows within the function body should be analyzed by default. There are two situations. A simple situation is that the external function does not affect the program behaviors after executing it, such as simply printing symbolic values with `printf`. In this case, we may ignore the path alternatives within the function. However, if the function execution affects the follow-up program behaviors, we should not ignore them. Otherwise, the symbolic execution would be based on the wrong assumption that the new test case generated for an alternative path can always trigger the same control flow within the external function. If a small program contains several such function calls, the complexity of external functions may cause path explosion issues. In practice, there are different strategies (e.g., abstraction [2], strict consistency, and local consistency [31]) that symbolic execution tools may adopt to handle the challenge with a trade-off between consistency and efficiency.

Fig. 3j demonstrates a sample with an external function call. It computes the sine of a symbolic variable via an external function call (i.e., `sin`), and the result is used to determine whether the program should return a `BOMB_ENDING`.

4.2.2 Loops

Loop statements, such as `for` and `while`, are widely employed in real-world programs. Even a very small program with loops can include many or even an infinite number of paths. By default, a symbolic execution tool should explore all available paths of a program, which can be beyond the capability of the tool if there are too many paths. In practice, a symbolic execution tool may employ a search strategy favoring unexplored branches on a program CFG [19], [43], or introduce new symbolic variables as the counters for each loop [44]. Because loop can incur numerous paths, it is difficult to derive a perfect solution for this problem.

Fig. 3k shows a sample with a loop. The loop function is implemented with the Collaz conjecture [45]. No matter what is the initial value of `i`, the loop will terminate with `j` equals 1.

4.2.3 Crypto Functions

Crypto functions generally involve some computationally complex problems to ensure security. For a hash function, the complexity guarantees that adversaries cannot efficiently compute the plaintext of a hash value. For a symmetric encryption function, it promises that one cannot efficiently compute the key when given several pairs of plaintext and ciphertext. Therefore, such programs should also be resistant to symbolic execution attacks. From a program analysis view, the number of possible control paths for the crypto functions can be substantial. For example, the body of the SHA1 algorithm [46] is a loop that iterates 80 rounds with each round containing several bit-level operations.

Fig. 3l demonstrates a code snippet which employs a SHA1 function [46]. If the hash result of the symbolic value is equivalent to a predefined value, the program would return a `BOMB_ENDING`. However, this is difficult since SHA1 cannot be reversely calculated.

In general, symbolic execution tools cannot handle such crypto programs. Malware may employ the technique to deter symbolic execution-based program analysis [47]. When analyzing programs with crypto functions, a common way is to avoid exploring the function internals (e.g.,[48], [49]). For example, TaintScope [48] first discriminates the symbolic variables corresponding to crypto functions from other variables, and then employs a fuzzy-based approach to search solutions for such symbolic variables rather than solving the problem via symbolic reasoning.

So far, we have discussed 12 different challenges. Note that we do not intend to propose a complete list of challenges for symbolic execution. Instead, we collect all the challenging issues that have been mentioned in the literature and systematically analyze them. This analysis is essential while designing the dataset of logic bombs in Section 5.2.2.

5 Benchmarking Methodology

In this section, we introduce our methodology and a framework to benchmark the capability of real-world symbolic execution tools.

5.1 Objective and Challenges

Before describing our approach, we first discuss our design goal and the challenges to overcome.

This work aims to design an approach that can benchmark the capabilities of symbolic execution tools. Our purpose is critical and valid in several aspects. As we have discussed, some challenging issues are only engineering issues, such as arithmetic overflows. With enough engineering effort, a symbolic execution tool should be able to handle these issues. On the other hand, some challenges such as loops are hard from a theoretical viewpoint. However,

some heuristic approaches can tackle certain easy cases. Symbolic execution tools may adopt different heuristics and demonstrate different capabilities in handling them. Therefore, it is worth benchmarking their performances in handling particular challenging issues. Developers generally do not provide much information concerning the limitations of their tools to users.

A useful benchmarking approach should be accurate and efficient. However, it is challenging to benchmark symbolic execution tools accurately and efficiently with real-world programs. First, a real-world program contains many instructions or lines of codes. When a symbolic execution failure occurs, locating the root cause requires much domain knowledge and effort. Since errors may propagate, it is often difficult to conjecture whether a symbolic execution tool fails in handling a particular issue. Second, the symbolic execution itself is inefficient. Benchmarking a symbolic execution tool generally implies performing several designated symbolic execution tasks, which would be time-consuming. Note that existing symbolic execution papers (e.g., [2], [3], [50], [51]) generally evaluate the performance of their tools by conducting symbolic execution experiments with real programs. This process usually takes several hours or even days. They demonstrate the effectiveness of their work using the achieved code coverage and number of bugs detected, while analyzing the root causes of uncovered codes is not a focus.

5.2 Approach based on Logic Bombs

To tackle the challenges of benchmarking symbolic execution tools concerning accuracy and efficiency, we propose an approach based on logic bombs. Below, we discuss our detailed design.

5.2.1 Evaluation with Logic Bombs

A logic bomb is a code snippet that can only be executed when certain conditions have been met. To evaluate whether a symbolic execution tool can handle a challenge, we can design a logic bomb guarded by a particular issue with the challenge. Then we can perform symbolic execution on the program embedded with the logic bomb. If a symbolic execution tool can generate a test case that can trigger the logic bomb, it indicates that the tool can handle the challenging issue, or *vice versa*.

Algorithm 1. Method to Design Evaluation Samples

```
// Create a function with a symbolic variable
Function LogicBomb(symvar)
  // symvar2 is a value computed from a challenging
     problem related to symvar
  symvar2 ← Challenge(symvar);
  // If symvar2 satisfies a condition
  if Condition(symvar2) than
    // Trigger the bomb
    Bomb();
  end
```

Algorithm 1 demonstrates a general framework for designing such logic bombs. It includes four steps: the first step is to create a function with a parameter $symvar$ as the symbolic variable; the second step is to design a challenging problem related to the symbolic variable and save the result to another variable $symvar2$; the third step is to design a condition related to the new variable $symvar2$; the final step is to design a bomb (e.g., return a specific value) which indicates that the condition has been satisfied. Note that because the value of $symvar2$ is propagated from $symvar$, $symvar2$ is also a symbolic variable and should be considered in the symbolic analysis process.

The magic of the logic bomb idea enables us to make the evaluation much precise and efficient. We can create several such small programs, each containing only a challenging issue and a logic bomb that tells the evaluation result. Because the object programs for symbolic execution are usually small, we can easily avoid unexpected issues that may also cause failures via a careful design. Also, because the programs are small, performing symbolic execution on them generally requires a short time. For the programs that unavoidably incur path explosion issues, we can restrict the symbolic execution time either by controlling the problem complexity or by employing a timeout setting.

5.2.2 Logic Bomb Dataset

Following Algorithm 1, we have designed a dataset of logic bombs to evaluate the capability of symbolic execution tools. Some of the logic bombs are already shown in Fig. 3. Our full dataset is available on GitHub.[1] The dataset contains over 60 logic bombs for 64-bit Linux platform, which covers all the challenges discussed in Section 4. For each challenge, we implement several logic bombs. Either each bomb involves a unique challenging issue (e.g., covert propagation via file write/read or via system calls), or introduces a problem with a different complexity setting (e.g., one-leveled arrays or two-leveled arrays).

When designing logic bombs, we carefully avoid trivial test cases (e.g., `\x00`) that can trigger the bombs. Moreover, we try to employ straightforward implementations, and we hope to ensure that the results would not be affected by other unexpected failures. For example, we avoid using `atoi` to convert `argv[1]` to integers because some tools cannot support `atoi`. However, fully avoiding external function calls is impossible for some logic bombs. For example, we should employ external function calls to create threads when designing parallel codes. Surely the result might be affected if a symbolic execution tool cannot handle external functions. To tackle the interference of challenges, we draw a challenge propagation chart among the logic bombs as shown in Fig. 4. There are two kinds of challenge propagation relationships: *should* in solid lines, and *may* in dashed lines. A *should* relationship means that a logic bomb contains a similar challenging issue in another logic bomb; if a tool cannot solve the precedent logic bomb, it should not be able to solve the later one. For example, the `stackarray_sm_l1` is precedent to `stackarray_sm_l2`. A *may* relationship means a challenge type may be a precedent to other logic bombs, but it is not the determining one. For example, a parallel program generally involves external function calls. However, although a tool is unable to solve the external functions well, it might be able to solve some logic bombs with parallel issues as sequential programs.

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

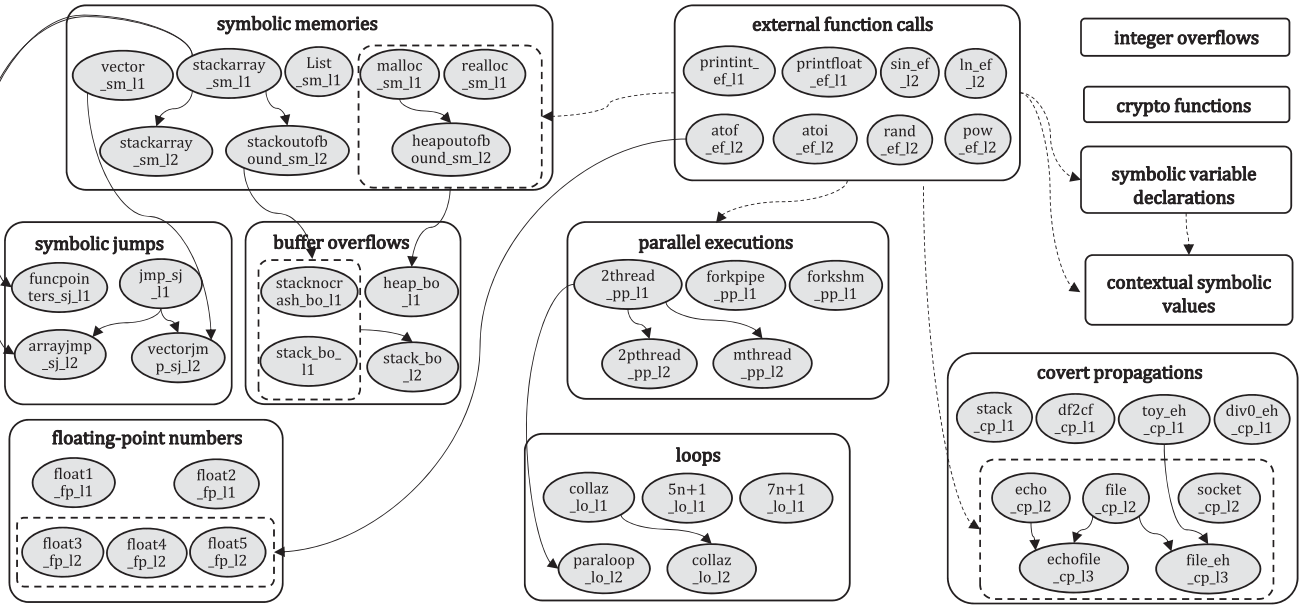


Fig. 4. The challenge propagation relationship among our dataset of logic bombs. A solid line means a logic bomb contains a similar problem defined in another logic bomb; a dashed line means a challenge may affect other logic bombs.

5.3 An Automated Benchmarking Framework

Based on the evaluation idea with logic bombs, we design a benchmarking framework as shown in Fig. 5. The framework inputs a dataset of carefully designed logic bombs and outputs the benchmarking result for a particular symbolic execution tool. There are three critical steps in the framework: dataset preprocessing, batch symbolic execution, and case verification.

In the preprocessing step, we parse the logic bombs and compile them into object codes or binaries such that a target symbolic execution tool can process them. The parsing process pads each code snippet of a logic bomb with a main function and makes it a self-contained program. By default, we employ `argv[1]` as the symbolic variables. If a target symbolic execution tool requires adding extra instructions to launch tasks, the parser should add such required instructions automatically. For example, we can add symbolic variable declaration codes when benchmarking KLEE. The compilation process compiles the processed source codes into binaries or other formats that a target symbolic execution tool supports. Symbolic execution is generally performed based on intermediate codes. When benchmarking source-code-based symbolic execution tools such as KLEE, we have to compile the source codes into the supported intermediate codes. When benchmarking binary-code-based symbolic execution tools, we can directly compile them into binaries, and the tool will lift binary codes into intermediate codes automatically.

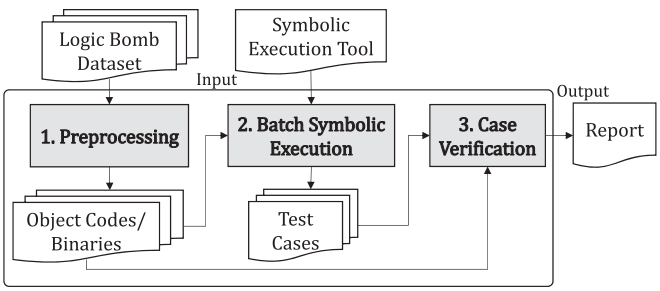


Fig. 5. Framework to benchmark symbolic execution tools.

In the second step, we direct the symbolic execution tool to analyze the compiled logic bombs in a batch mode. This step outputs a set of test cases for each program. Some dynamic symbolic execution tools (e.g., Triton) can directly tell which test case can trigger a logic bomb during runtime. However, other static symbolic execution tools may only output test cases by default, so we need to replay the generated test cases to examine the results further. Besides, some tools may falsely report that a test case can trigger the logic bomb. Therefore, we need a third step to verify the test cases.

In the third step, we replay the test cases with the corresponding programs of logic bombs. If a logic bomb can be triggered, it indicates that the challenging case has been solved by the tool. Finally, we can generate a benchmarking report based on the case verification results.

6 Experimental Study

In this section, we describe an experimental study conducted to demonstrate the effectiveness of our benchmarking approach. Below, we first discuss the experimental setting and then the results.

6.1 Experimental Setting

We choose three popular symbolic execution tools for benchmarking: KLEE [2], angr [3], and Triton [13]. Because our dataset of logic bombs are written in C/C++, we only choose symbolic execution tools for C/C++ programs or binaries. The three tools have all been released as open source and have a high community impact. Moreover, they adopt different implementation techniques for symbolic execution. By supporting variant tools, we show that our approach is compatible with different symbolic execution implementations.

KLEE [2] is a source-code-based symbolic execution tool implemented based on LLVM [52]. It supports programs

TABLE 2
Experimental Results on Benchmarking three Symbolic Execution Tools (KLEE, Triton, and angr) in Handling our Logic Bombs

Challenge	Case ID	KLEE		Triton		angr	
		t = 60s	t = 300s	t = 60s	t = 300s	t = 60s	t = 300s
Covert Propagations	df2cf_cp	pass	pass	fail	fail	pass	pass
	echo_cp	fail	fail	timeout	timeout	timeout	timeout
	echofile_cp	fail	fail	fail	fail	timeout	timeout
	file_cp	fail	fail	timeout	timeout	fail	fail
	socket_cp	fail	fail	fail	fail	fail	fail
	stack_cp	inapplicable	inapplicable	pass	pass	pass	pass
	file_eh_cp	inapplicable	inapplicable	fail	fail	timeout	pass
	div0_eh_cp	inapplicable	inapplicable	fail	fail	timeout	pass
	file_eh_cp	inapplicable	inapplicable	fail	fail	timeout	fail
Buffer Overflows	stacknocrash_bo_l1	fail	fail	fail	fail	pass	pass
	stack_bo_l1	fail	fail	fail	fail	pass	pass
	heap_bo_l1	fail	fail	fail	fail	fail	fail
	stack_bo_l2	fail	fail	fail	fail	fail	fail
Symbolic Memories	malloc_sm_l1	pass	pass	timeout	fail	pass	pass
	realloc_sm_l1	pass	pass	fail	fail	pass	pass
	stackarray_sm_l1	pass	pass	fail	fail	pass	pass
	list_sm_l1	inapplicable	inapplicable	fail	fail	timeout	pass
	vector_sm_l1	inapplicable	inapplicable	fail	fail	timeout	pass
	stackarray_sm_l2	pass	pass	fail	fail	fail	fail
	stackoutofbound_sm_l2	pass	pass	fail	fail	pass	pass
	heapoutofbound_sm_l2	fail	fail	timeout	fail	pass	pass
Symbolic Jumps	funcpointer_sj_l1	pass	pass	fail	fail	fail	fail
	jmp_sj_l1	inapplicable	inapplicable	fail	fail	pass	pass
	arrayjmp_sj_l2	inapplicable	inapplicable	fail	fail	fail	fail
	vectorjmp_sj_l2	inapplicable	inapplicable	fail	fail	timeout	pass
Floating-point Numbers	float1_fp_l1	fail	fail	fail	fail	pass	pass
	float2_fp_l1	fail	fail	fail	fail	pass	pass
	float3_fp_l2	fail	fail	fail	fail	timeout	timeout
	float4_fp_l2	fail	fail	fail	fail	timeout	timeout
	float5_fp_l2	fail	fail	fail	fail	timeout	timeout
Arithmetic Overflows	plus_do	pass	pass	pass	pass	pass	pass
	multiply_do	pass	pass	fail	fail	pass	pass
External Function Calls	printint_ef_l1	fail	fail	pass	pass	pass	pass
	printfloat_ef_l1	fail	fail	fail	fail	fail	fail
	atoi_ef_l2	fail	fail	fail	fail	pass	pass
	atof_ef_l2	fail	fail	fail	fail	timeout	timeout
	ln_ef_l2	fail	fail	fail	fail	timeout	fail
	pow_ef_l2	fail	fail	fail	fail	pass	pass
	rand_ef_l2	fail	fail	timeout	timeout	fail	fail
	sin_ef_l2	fail	fail	fail	fail	timeout	timeout
Symbolic Variable Declarations		7 cases, no pass					
Parallel Executions		5 cases, no pass					
Contextual Symbolic Values		4 cases, no pass					
Loops		5 cases, no pass					
Crypto Functions		2 cases, no pass					
pass #	63 cases	9	9	3	3	17	22

Pass means the tool has successfully triggered the bomb; fail means the tool cannot find test cases to trigger the bomb; timeout means the tool cannot find test cases to trigger the bomb within a given period of time; inapplicable means the program contains unsupported languages for the tool, e.g., C++ and assembly codes for KLEE. For each tool, we adopt two timeout settings: 60 seconds and 300 seconds.

written in `C`. By default, our benchmarking script uses a `klee_make_symbolic` function to declare the symbolic variables of logic bombs in the source-code level. Then, it compiles the source codes into intermediate codes for symbolic execution. The symbolic execution process outputs a set of test cases. Our script finally examines the test cases by replaying them with the binaries. The whole process is automated with our benchmarking script. The version of KLEE we benchmark is 1.3.0. Note that this paper does not intend to find the best tool for particular challenges, so we do not

consider the patches or plugins developed by other parties before they have been merged into the project.

Triton [13] is a dynamic symbolic execution tool based on binaries. It automatically accepts symbolic variables from the standard input. During symbolic execution, it first runs the programs with concrete values and leverages Intel Pin-Tool [53] to trace related instructions; then it lifts the traced instructions into the SSA (single static assignment) form and performs symbolic analysis. If there are alternative paths found in the trace, Triton generates new test cases via symbolic reasoning and employs them as the concrete values in the following rounds of concrete execution. This symbolic execution process continues until no alternative path can be found. The version of Triton we adopted is the one released on GitHub on Jul 6, 2017.

Angr [3] is also a tool for binaries but employs different implementations. Before performing any symbolic analysis, angr first lifts the binary program into VEX IR [54]. Then it employs a symbolic analysis engine (SimuVEX) to analyze the program based on the IR. Angr does not provide ready-to-use symbolic execution script for users but only some APIs. Therefore, we have to implement our own symbolic execution script for angr. Our script collects all the paths to the CFG leaf nodes and then solves the corresponding path constraints. Angr provides all the critical features via APIs, and we only assemble them. Finally, we check whether the generated test cases can trigger the logic bombs. In our experiment, we employ angr version 7.7.9.21.

Note that all our benchmarking scripts for these tools follow the framework proposed in Fig. 5. During the experiments, we employ our logic bomb dataset for evaluation. A tool can pass a test only if the solution generated can correctly trigger a logic bomb. We finally report which logic bombs can be triggered by the tools.

We conduct our experiments on an Ubuntu 14.04 X86_64 system with Intel i5 CPU and 8G RAM. Because some symbolic execution tasks may take very long time, our tool allows users to configure a timeout threshold which ensures benchmarking efficiency. However, the timeout mechanism may incur some false results if it is too short. To mitigate the side effects, we adopt two timeout settings (60 seconds and 300 seconds) for each tool. In this way, we can observe the influence of the timeout settings and decide whether we should conduct more experiments with an increased timeout value.

6.2 Benchmarking Results

6.2.1 Result Overview

Table 2 presents our experimental results. We label the results with four options: pass, fail, timeout, and inapplicable. While 'pass' and 'fail' imply the symbolic execution has finished, 'timeout' implies our benchmarking script has terminated the symbolic execution process when a timeout threshold is triggered. We label several results as inapplicable because the logic bombs contain `C++` or assembly codes, which KLEE does not support.

We can observe that angr has achieved the best performance with 22 cases solved when the timeout was 300 seconds. Comparatively, it only solved 17 cases when the timeout is 60 seconds. KLEE solved nine cases and the result remains the same with different timeout settings. Triton performed much worse with just three cases being solved. To further verify the correctness of our benchmarking results, we compared our experimental results with the previously declared challenge propagation relationships in Fig. 4. The results were all consistent, showing that our dataset can distinguish the capability of different symbolic-execution tools accurately.

The efficiency of our benchmarking approach largely depends on the timeout setting. Note that Table 2 includes some timeout results; they account for most of our experimental time. Although we try to keep each logic bomb as succinct as possible, our dataset still contains some complex but unavoidable problems or path explosion issues. When the timeout value is 60 seconds, our benchmarking process for each tool takes only dozens of minutes. When extending the timeout value to 300 seconds, the benchmark takes a bit longer time. However, the benefit is not very obvious, and only angr can solve 5 more cases. Can the result get further improved by allowing more time? We have tried another group of experiments with 1,800 seconds timeout. But the results remain unchanged. Therefore, 300 seconds should be a marginal timeout setting for our benchmarking experiment. Considering that symbolic execution is computationally expensive, which may take several hours or even several days to test a program, our benchmarking process is very efficient. We may further improve the efficiency by employing a parallel mode, such as assigning several logic bombs for each process.

6.2.2 Case Study

We now discuss the detailed benchmarking results for each challenge. First, there are several challenges that none of these tools can trigger even one logic bomb, including symbolic variable declarations, parallel executions, contextual symbolic values, loops, and crypto functions. Because crypto functions involve tough problems, it can be expected that all the tools fail in handling them. It is a bit surprising that none of the tools can handle parallel executions and loops. For the problems of symbolic variable declarations and contextual symbolic values related to files, KLEE can generate test cases which may trigger the bombs when enabling the `--sym-files` option. However, in the case verification step, our script cannot trigger such logic bombs with the test cases. The reason is that KLEE does not provide a feature for simulating the environment required to replay the test cases.

Covert Propagations. Angr passed four test cases: `df2cf_cp`, `stack_cp`, and two exception handling cases. `df2cf_cp` propagates the symbolic values indirectly by substituting a data assignment operation with equivalent control-flow operations. KLEE also solved the case, but Triton failed. `stack_cp` propagates symbolic values via direct assembly instructions `push` and `pop`. Triton also solved the case. Besides, angr also passed two test cases that propagate symbolic values via the C++ exception handling mechanism, which Triton failed. We further break down the details of an exception handling program (see Fig. 6). As shown in the box region of Fig. 6b, the mechanism relies on two function calls, which might be the problem that fails Triton. All the tools failed other covert propagation cases that propagate values via `fread/fwrite`, echo, socket, etc.

```
double division(int numerator, int denominator) {
  if( denominator == 0 ) {
    throw "Division by zero condition!";
  }
  return (numerator/denominator);
}

int logic_bomb(char* s) {
   int symvar = s[0] - 48;
   try {
     division(10, symvar-7);
     return NORMAL_ENDING;
   }catch (const char* msg) {
     return BOMB_ENDING;
   }
}
```

(a) Source codes.

```
0x0000000000400aa9 <+41>:    callq  0x400a10 <_Z8divisionii>
0x0000000000400aae <+46>:    movsd  %xmm0,-0x40(%rbp)
0x0000000000400ab3 <+51>:    jmpq   0x400ab8 <_Z10logic_bombPc+56>
0x0000000000400ab8 <+56>:    movl   $0x0,-0x4(%rbp)
0x0000000000400abf <+63>:    jmpq   0x400afd <_Z10logic_bombPc+125>
0x0000000000400ac4 <+68>:    mov    %edx,%ecx
......                       ......
0x0000000000400ae1 <+97>:    callq  0x4008a0 <__cxa_begin_catch@plt>
0x0000000000400ae6 <+102>:   mov    %rax,-0x30(%rbp)
0x0000000000400aea <+106>:   movl   $0x1,-0x4(%rbp)
0x0000000000400af1 <+113>:   movl   $0x1,-0x34(%rbp)
0x0000000000400af8 <+120>:   callq  0x400890 <__cxa_end_catch@plt>
0x0000000000400afd <+125>:   mov    -0x4(%rbp),%eax
0x0000000000400b00 <+128>:   add    $0x40,%rsp
0x0000000000400b04 <+132>:   pop    %rbp
0x0000000000400b05 <+133>:   retq
0x0000000000400b06 <+134>:   mov    -0x20(%rbp),%rdi
0x0000000000400b0a <+138>:   callq  0x4008c0 <_Unwind_Resume@plt>
```

(b) Assembly codes.

Fig. 6. An exemplary program that raises an exception when divided by zero. The assembly codes demonstrates how the `try/catch` mechanism works in low level.

Note that KLEE supports modeling file operations in POSIX standard such as `read/write`, but it cannot support `C` libraries directly.

Buffer Overflows. Only angr could solve two easy buffer overflow problems: `stacknocrash_bo_l1` and `stack_bo_l1`. The cases share a simple stack overflow issue. Their solutions require modifying the value of the stack that might be illegal. However, angr could not solve the heap overflow issue `heap_bo_l1`. It also failed on another harder stack overflow issue `stack_bo_l2`, which requires composing sophisticated payload, such as employing return-oriented programming methods [55]. We are surprised that Triton failed all the tests because binary-code-based symbolic execution tools should be resilient to buffer overflows in nature.

Symbolic Memories. The results show that Triton does not support symbolic memory, but KLEE and angr provide very good support. Angr has solved seven cases out of eight. It only failed in handling the case depicted in Fig. 7a with a two-leveled array `stackarray_sm_l2`. This implies that angr would fail when there are multi-leveled pointers. In comparison, KLEE is able to solve the two-leveled array problem because it is based on STP [36], which is designed for solving such problems related to arrays. Fig. 7c presents the assembly codes that initialize the arrays, while Fig. 7b presents the stack layout after initialization. We note that the information about array size or boundary does not exist in assembly codes. This explains why binary-code-based symbolic execution tools do not suffer from problems when a challenge requires an out-of-boundary access, e.g., `stackoutofbound_sm_l2`.

Symbolic Jumps. Since symbolic jump demonstrates no explicit conditional branches in the CFG, it should be a hard problem for symbolic execution. However, KLEE and angr are not likely to be affected much by the trick. KLEE tackled the problem which has an array of function pointers `funcpointer_sj_l1`. Angr successfully handled two cases with assembly `jmp`, but it failed `funcpointer_sj_l1`.

Floating-Point Numbers. The results indicate that KLEE and Triton do not support floating-point operations, and angr can support some. During our tests, Triton directly reported that it could not interpret such floating-point instructions. Angr has solved two out of the five designated cases. The two passed cases are easier ones, which only require integer values as the solution. All the failed cases require decimal values as the solution, and they employ the `atof` function to convert `argv[1]` to decimals. Since angr has also failed the test in handling `atof` in `atof_ef_l2`, the failures are likely to be caused by the `atof` function.

Arithmetic Overflows. Arithmetic overflow is not a very hard problem since it only requires symbolic execution tools to handle such cases carefully. In our test, KLEE and angr

```
int logic_bomb(char* s) {
   int symvar = s[0] - 48;
   int l1_ary[] ={1,2,3,4,5};
   int l2_ary[] ={6,7,8,9,10};

   int x = symvar%5;
```

```
if(l2_ary[l1_ary[x]] == 9){
    return BOMB_ENDING;
  }
  else
    return NORMAL_ENDING;
}
```

(a) Source codes.

```
(gdb) break *logic_bomb+97
(gdb) run
(gdb) x/20xw $rsp-80
0x7fffffffe2d0: 0x00000006   0x00000007   0x00000008   0x00000009
0x7fffffffe2e0: 0x0000000a   0x00000000   0x004003e5   0x00000000
0x7fffffffe2f0: 0x00000001   0x00000002   0x00000003   0x00000004
0x7fffffffe300: 0x00000005   0x00007fff   0xf7fe2000   0x00000001
0x7fffffffe310: 0xffffe6db   0x00007fff   0x00000000   0x00000000
```

(b) Memory layout after array initialization.

```
text: 0x000000000040060d <+29>:    mov   0x4007c0,%rdi
text: 0x0000000000400615 <+37>:    mov   %rdi,-0x30(%rbp)
text: 0x0000000000400619 <+41>:    mov   0x4007c8,%rdi
text: 0x0000000000400621 <+49>:    mov   %rdi,-0x28(%rbp)
text: 0x0000000000400625 <+53>:    mov   0x4007d0,%ecx
text: 0x000000000040062c <+60>:    mov   %ecx,-0x20(%rbp)
text: 0x000000000040062f <+63>:    mov   0x4007e0,%rdi
text: 0x0000000000400637 <+71>:    mov   %rdi,-0x50(%rbp)
text: 0x000000000040063b <+75>:    mov   0x4007e8,%rdi
text: 0x0000000000400643 <+83>:    mov   %rdi,-0x48(%rbp)
text: 0x0000000000400647 <+87>:    mov   0x4007f0,%ecx
text: 0x000000000040064e <+94>:    mov   %ecx,-0x40(%rbp)
......                             ......
.rodata:00000000004007C0    dq 200000001h
.rodata:00000000004007C8    dq 400000003h
.rodata:00000000004007D0    dd 5
.rodata:00000000004007D4    align 20h
.rodata:00000000004007E0    dq 700000006h
.rodata:00000000004007E8    dq 900000008h
.rodata:00000000004007F0    dd 0Ah
```

(c) Assembly codes.

Fig. 7. A program that demonstrates how the stack works with arrays. There is no information about the size of each array left in assembly codes.

have been able to solve all the cases. However, Triton failed in handling the integer overflow case in Fig. 3i. The result shows there is still much room for Triton to improve for this problem.

External Function Calls. In this group of logic bombs, each case only contains one external function call. However, this result is very disappointing. Triton only passed a very simple case that print (with `printf`) a symbolic value of integer type. It does not even support printing out floating-point values. Angr has solved the `printf` cases and two more complicated cases, `atoi_ef_l2` and `pow_ef_l2`. It failed the `atof_ef_l2` and other cases. The results show that we should be cautious when designing logic bombs. Even when involving straightforward external function calls, the results could be affected.

7 Conclusion

This work has proposed an approach capable of benchmarking the capabilities of symbolic execution tools while handling particular challenges. We have studied the taxonomy of challenges faced by symbolic execution tools, including nine symbolic-reasoning challenges and three path-explosion challenges. Such a study is essential for us to design the benchmarking dataset. Next, we proposed a promising benchmarking approach based on logic bombs. The idea has been to design logic bombs that can only be triggered if a symbolic execution tool solves specific challenging issues. By making the programs of logic bombs as small as possible, we have been able to speed up the benchmarking process. Also, by making them as straightforward as possible, we have been able to sidestep unexpected reasons that could affect the benchmarking results. All this has rendered our benchmarking approach both accurate and efficient. Following this idea, we have implemented a dataset of logic bombs and a prototype benchmarking framework which automates the benchmarking process. Then, we conducted real-world experiments on three symbolic execution tools. Experimental results have shown that the benchmarking process for each tool generally takes dozens of minutes. Angr achieved the best benchmarking results with 22 cases solved, KLEE solved nine, and Triton only solved three. These results point to the value of a third-party benchmarking toolset for symbolic execution tools. Finally, we released our dataset as open source on GitHub for public usage. We hope it would be seen as an essential tool for the community to benchmark symbolic execution tools and could facilitate the development of more comprehensive symbolic execution techniques.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (Project Nos. 61332010,61672164), the Research Grants Council of the Hong Kong Special Administrative Region, China (No. CUHK 14234416 of the General Research Fund), and Microsoft Research Asia (2018 Microsoft Research Asia Collaborative Research Award).

References

[1] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, pp. 385–394, 1976.

[2] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. 8th USENIX Symp. Operating Syst. Des. Implementation*, 2008, pp. 209–224.

[3] Y. Shoshitaishvili and et al., "SoK: (State of) the art of war: Offensive techniques in binary analysis," in *Proc. IEEE Symp. Security Privacy*, 2016, pp. 138–157.

[4] D. A. Ramos and D. R. Engler, "Under-constrained symbolic execution: Correctness checking for real code," in *Proc. USENIX Security Symp.*, 2015, pp. 49–64.

[5] M. Quan, "Hotspot symbolic execution of floating-point programs," in *Pro. 24th ACM SIGSOFT Int. Symp. Foundations Softw. Eng.*, 2016, pp. 1112–1114.

[6] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later," *Commun. ACM*, vol. 56, pp. 82–90, 2013.

[7] 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. IEEE Symp. Security Privacy*, 2010, pp. 317–331.

[8] X. Qu and B. Robinson, "A case study of concolic testing tools and their limitations," in *Proc. IEEE Int. Symp. Empirical Softw. Eng. Meas.*, 2011, pp. 117–126.

[9] L. Cseppento and Z. Micskei, "Evaluating symbolic execution-based test tools," in *Proc. 8th IEEE Int. Conf. Softw. Testing Verification Validation*, 2015. pp. 1–10.

[10] R. Kannavara, C. J. Havlicek, B. Chen, M. R. Tuttle, K. Cong, S. Ray, and F. Xie, "Challenges and opportunities with concolic testing," in *Proc. Nat. Aerosp. Electron. Conf.*, 2015, pp. 374–378.

[11] H. Xu, Y. Zhou, Y. Kang, and M. R. Lyu, "Concolic execution on small-size binaries: challenges and empirical study," in *Proc. 47th Annu. IEEE/IFIP Int. Conf. Depend. Syst. Netw.*, 2017, pp. 181–188.

[12] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Survey*, vol. 51, 2018, Art. no. 50.

[13] F. Saudel and J. Salwan, "Triton: A dynamic symbolic execution framework," in *Proc. Symp. Sur La Sécurité Des Technologies de Linformation et Des Communications*, 2015, pp. 31–54.

[14] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, "AEG: Automatic exploit generation," in *Proc. Netw. Distrib. Syst. Security Symp.*, 2011, pp. 74–84.

[15] J. Ming, D. Xu, L. Wang, and D. Wu, "LOOP: Logic-oriented opaque predicate detection in obfuscated binary code," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Security*, 2015, pp. 757–768.

[16] B. Yadegari and S. Debray, "Symbolic execution of obfuscated code," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Security*, 2015, pp. 732–744.

[17] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte, "Fitness-guided path exploration in dynamic symbolic execution," in *Proc. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2009, pp. 359–368.

[18] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, and G. Candea, "Cloud9: A software testing service," *ACM SIGOPS Operating Syst. Rev.*, vol. 43, pp. 5–10, 2010.

[19] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, "Enhancing symbolic execution with veritesting," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 1083–1094.

[20] S. Banescu, C. Collberg, V. Ganesh, Z. Newsham, and A. Pretschner, "Code obfuscation against symbolic execution attacks," in *Proc. 32nd Annu. Conf. Comput. Security Appl.*, 2016, pp. 189–200.

[21] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, pp. 576–580, 1969.

[22] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *Proc. IEEE Symp. Security Privacy*, 2012, pp. 380–394.

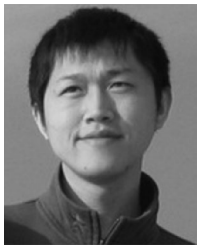
[23] L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis, "Path-exploration lifting: Hi-fi tests for lo-fi emulators," in *Proc. 17th Int. Conf. Architectural Support Program. Languages Operating Syst.*, 2012, pp. 337–348.

[24] T. Su, Z. Fu, G. Pu, J. He, and Z. Su, "Combining symbolic execution and model checking for data flow testing," in *Proc. 37th Int. Conf. Softw. Eng.*, 2015, pp. 654–665.

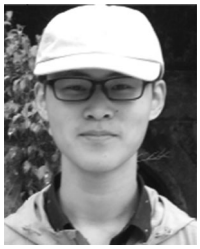
[25] S. Kim, M. Faerevaag, M. Jung, S. Jung, D. Oh, J. Lee, and S. K. Cha, "Testing intermediate representations for binary analysis," in *Proc. 32nd IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2017, pp. 353–364.

[26] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," *ACM Sigplan Notices*, vol. 40, pp. 213–223, 2005.

[27] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "BAP: A binary analysis platform," in *Proc. Int. Conf. Comput. Aided Verification*, 2011, pp. 463–469.
[28] N. Razavi, F. Ivančić, V. Kahlon, and A. Gupta, "Concurrent test generation using concolic multi-trace analysis," in *Proc. Asian Symp. Program. Languages Syst.*, 2012, pp. 239–255.
[29] D. Davidson, B. Moench, T. Ristenpart, and S. Jha, "FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution," in *Proc. USENIX Security Symp.*, 2013, pp. 463–478.
[30] S. Guo, M. Kusano, and C. Wang, "Conc-iSE: Incremental symbolic execution of concurrent software," in *Proc. 31st IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2016, pp. 531–542.
[31] V. Chipounov, V. Kuznetsov, and G. Candea, "S2E: A platform for in-vivo multi-path analysis of software systems," in *Proc. 16th Int. Conf. Architectural Support Program. Languages Operating Syst.*, 2011, pp. 265–278.
[32] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley, "Automatic exploit generation," *Commun. ACM*, vol. 57, pp. 74–84, 2014.
[33] A. Farzan, A. Holzer, N. Razavi, and H. Veith, "Con2colic testing," in *Proc. 9th Joint Meeting Foundations Softw. Eng.*, 2013, pp. 37–47.
[34] T. Bergan, D. Grossman, and L. Ceze, "Symbolic execution of multithreaded programs from arbitrary program contexts," in *Proc. ACM Int. Conf. Object Oriented Program. Syst. Languages Appl.*, 2014, pp. 491–506.
[35] S. Guo, M. Kusano, C. Wang, Z. Yang, and A. Gupta, "Assertion guided symbolic execution of multithreaded programs," in *Proc. 10th Joint Meeting Foundations Softw. Eng.*, 2015, pp. 854–865.
[36] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *Proc. Int. Conf. Comput. Aided Verification*, 2007, pp. 519–531.
[37] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Proc. Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2008, pp. 337–340.
[38] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," in *ACM SIGSOFT Softw. Eng. Notes*, vol. 30, pp. 263–272, 2005.
[39] W. Landi and B. G. Ryder, "Pointer-induced aliasing: A problem classification," in *Proc. 18th ACM SIGPLAN-SIGACT Symp. Principles Program. Languages*, 1991, pp. 93–103.
[40] A. Solovyev, C. Jacobsen, Z. Rakamarić, and G. Gopalakrishnan, "Rigorous estimation of floating-point round-off errors with symbolic taylor expansions," in *Proc. Int. Symp. Formal Methods*, 2015, pp. 532–550.
[41] D. Liew, D. Schemmel, C. Cadar, A. F. Donaldson, R. Zähl, and K. Wehrle, "Floating-point symbolic execution: A case study in n-version programming," in *Proc. 32nd IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2017, pp. 601–612.
[42] D. S. Liew, "Symbolic execution of verification languages and floating-point code," Ph.D. thesis, Dept. of Computing, Imperial College London, 2018.
[43] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *Proc. 23rd IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2008, pp. 443–446.
[44] P. Saxena, P. Poosankam, S. McCamant, and D. Song, "Loop-extended symbolic execution on binary programs," in *Proc. 18th Int. Symp. Softw. Testing Anal.*, 2009, pp. 225–236.
[45] J. C. Lagarias, "The 3x + 1 problem and its generalizations," *The Amer. Math. Monthly*, vol. 92, pp. 3–23, 1985.
[46] D. Eastlake 3rd and P. Jones, "Us secure hash algorithm 1 (SHA1)," standard RFC document. No. RFC 3174, 2001.
[47] M. I. Sharif, A. Lanzi, J. T. Giffin, and W. Lee, "Impeding malware analysis using conditional code obfuscation," in *Proc. 15th Annu. Netw. Distrib. Syst. Security Symp.*, 2008.
[48] T. Wang, T. Wei, G. Gu, and W. Zou, "TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *Proc. IEEE Symp. Security Privacy*, 2010, pp. 497–512.
[49] R. Corin and F. A. Manzano, "Efficient symbolic execution for analysing cryptographic protocol implementations," in *Proc. Int. Symp. Eng. Secure Softw. Syst.*, 2011, pp. 58–72.
[50] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," *ACM Sigplan Notices*, vol. 47, pp. 193–204, 2012.
[51] N. Hasabnis and R. Sekar, "Extracting instruction semantics via symbolic execution of code generators," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2016, pp. 301–313.
[52] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. Int. Symp. Code Generation Optimization: Feedback-Directed Runtime Optimization*, 2004, pp. 75–86.
[53] 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," *ACM SIGPLAN Notices*, vol. 40, pp. 190–200, 2005.
[54] N. Nethercote, "Dynamic binary analysis and instrumentation," Ph.D. dissertation, PhD thesis, Dept. of Computer Laboratory, University of Cambridge, Cambridge, U.K., 2004.
[55] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Trans. Inf. Syst. Security*, vol. 15, 2012, Art. no. 2.



Hui Xu received the MSc degree from the University of Hong Kong, in 2008. He is currently working toward the PhD degree in The Chinese University of Hong Kong.. He is a CISSP and has served in cybersecurity industry for several years. He is a member of the IEEE.



Zirui Zhao is a research assistant with the Chinese University of Hong Kong and a student from The University of University of Science and Technology of China.

Yangfan Zhou received the BS degree from Peking University, in 2000, and the MPhil and PhD degrees from the Chinese University of Hong Kong, in 2006 and 2009, respectively. He is currently an associate professor with Fudan University. His research interests include distributed computing and networking, particularly in mobile computing, cloud computing, and the Internet of Things. Before joining Fudan, he was a research staff member with The Chinese University of Hong Kong from 2009 to 2014. He has also been working as an engineer in information technology industry for many years, where he is now also active in technology consulting. He is a member of the IEEE.

Michael R. Lyu received the BS degree in electrical engineering from National Taiwan University, Taipei, Taiwan, R.O.C., in 1981, the MS degree in computer engineering from the University of California, Santa Barbara, in 1985, and the PhD degree in computer science from the University of California, Los Angeles, in 1988. He is currently a professor of Department of Computer Science and Engineering, The Chinese University of Hong Kong. His research interests include software reliability engineering, distributed systems, fault-tolerant computing, and machine learning. He is an ACM fellow, an IEEE fellow, an AAAS fellow, and a Croucher Senior Research fellow for his contributions to software reliability engineering and software fault tolerance.