On Benchmarking the Capability of Symbolic Execution Tools with Logic Bombs

Hui Xu*†, Zirui Zhao,† Yangfan Zhou‡, 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

***Abstract*—Symbolic execution is an important software testing approach. It has been widely employed in program analysis, such as bug detection and malware analysis. However, the approach is not overwhelming because it suffers many issues, including the well-known scalability issue and other challenges, such as handling floating-point numbers and symbolic memories. Currently, several symbolic execution tools are available off-the-shelf, but they generally do not demonstrate their limitations clearly to users. Yet, we have no effective approach to benchmark their capabilities. Without such information, users would not know which tool to choose, or how reliable their program analysis results are based on particular symbolic execution tools. To address such concerns, this paper proposes a novel approach to benchmark symbolic execution tools. Our approach is based on logic bombs which are guarded by particular challenging problems. If a symbolic execution tool can find test cases to trigger such logic bombs during evaluation, it indicates that the tool can handle corresponding problems. Following the idea, we have designed an automated benchmark framework and a dataset of logic bombs covering 12 different challenges. Then we have conducted real-world experiments on benchmarking three popular symbolic execution tools: KLEE, Angr, and Triton. Experimental results show that our approach can reveal their limitations in handling particular issues accurately and efficiently. The benchmark process generally takes only a few minutes to evaluate a tool. To better serve the community, we release our toolset on GitHub as open source, and we hope it would serve as an essential tool to benchmark symbolic execution tools in the future.**

I. Introduction

Symbolic execution is a white-box software testing approach that can automatically generate test cases for triggering corresponding code paths. Since the approach was firstly proposed by King [1] in 1976, it has become an important research topic in software testing and program analysis fields. The technique has achieved rapid development in the last decade with several open-source symbolic execution tools released, such as KLEE [2], Angr [3], BAP [4], and Triton [5]. Thanks to these tools, there are many fantastic program analysis applications proposed in the literature. For example, Davidson *et al.* [6] proposed to detect the vulnerabilities of firmware based on KLEE; Shoshitaishvili *et al.* [3] proposed to analyze binaries with Angr; Ming *et al.* [7] proposed to deobfuscate codes based on BAP.

However, current symbolic execution tools are still far from perfection. One well-known issue is that symbolic execution suffers path explosion problems, such that it cannot scale up to large-size programs. Moreover, symbolic execution tools often fail even when handling some small-size programs with certain gadgets [8], such as arrays [9] and floating-point numbers [10]. Such gadgets widely exist in real-world programs. Failure in handling them would lead to inaccurate symbolic execution results, which can further affect upper-level program analysis applications. To properly employ the technique, the users of a particular symbolic execution tool should carefully check its limitations. Unfortunately, such limitations are generally not documented well by the tool developers [11]. This paper, therefore, aims to provide users an approach to benchmark symbolic execution tools and to expose their limitations.

Intuitively, a competent benchmark approach should cover all the known issues, such that it can reveal the as many limitations as possible. To this end, we have conducted a systematic study on the challenges of symbolic execution. We categorize the challenges into *accuracy challenges* and *scalability challenges*. Accuracy challenges attack the core symbolic reasoning process, and they may pose a symbolic execution tool to generate incorrect test cases for particular control flows. Such challenges include symbolic variable declaration, covert propagation, parallel program, symbolic memory, contextual symbolic value, symbolic jump, floating-point number, buffer overflow, and integer overflow. We catalog other challenges that do not attack the symbolic reasoning process as scalability challenges, which may cause a symbolic execution tool starving the computational resources or taking very long time to generate test cases. Not only large-size programs can cause scalability issues but also small-size programs as long as they include complex routines. With our approach, all existing challenges discussed in the literature can be well categorized.

Moreover, the benchmark approach should be accurate and efficient. However, the goal is challenging because real programs are very complex that a failure may be triggered by any components. We should be able to locate the root issue and avoid false results. Moreover, symbolic execution itself is inefficient and it may take several hours or days to test a program [12]. In this work, we subtly address the issue by employing the idea of logic bombs. A logic bomb is a code block that can only be executed when certain conditions are met. We can design such logic bombs that can only be triggered when a challenging problem is solved. To make the evaluation accurate, we should keep each logic bomb as simple as possible such that the evaluation result would not be affected by other unexpected issues. Also, a simple program

largely shortens the required symbolic execution time and provides efficiency. Following this method, we have designed a dataset of logic bombs covering 12 challenges. For each challenge, we have designed several logic bombs to provide fine-grained results. To our best knowledge, our dataset has covered all the challenges appeared in the literature. Note that some logic bombs unavoidably contain other challenging issues. For example, a logic bomb with parallel execution issues should call external functions to create threads; but external function itself is also a challenging issue. To mitigate such influence, we draw a chart that demonstrates the challenge propagation relationships among logic bombs.

Besides, the benchmark approach should be easily adoptable by ordinary users. To this end, we have designed an automated benchmark framework and developed the corresponding benchmark toolset. Our toolset employs the dataset of logic bombs as the evaluation metric. It firstly parses the logic bombs and compiles them to object codes or binaries; then it drives a target symbolic execution tool to perform symbolic execution on the logic bombs in a batch mode; finally, it verifies the generated test cases and generates reports. Our toolset and user manuals are available on GitHub[1].

With the toolset, we have conducted real-world experiments on three prevalent symbolic execution tools, KLEE, Triton, and Angr. Although these tools adopt different implementation techniques, our framework can adapt to them with little customization. The benchmark process for each tool generally takes a few minutes. Experimental results show that our benchmark approach can reveal their limitations efficiently and accurately. Angr has achieved the best performance with 21 cases solved, which is roughly one third of the total logic bombs; KLEE and Triton only solved three cases. We have manually checked the reported solutions and confirmed that they are all correct and nontrivial. Besides, the results also demonstrate some interesting findings about such tools. For example, Angr only supports one-leveled array but not two-leveled; Triton does not even support the `atoi` function. Such findings are not available from the tool websites, which further justifies the necessity of our benchmark idea.

The rest of the paper is organized as follows. We first discuss the related work in Section II and the background of symbolic execution in Section III; then we introduce our study about the challenges of symbolic execution in Section IV; we introduce our methodology to benchmark symbolic execution tools in Section V and experimental evaluation in Section VI; finally, we conclude this paper in Section VII.

II. Related Work

Symbolic execution has received extensive attention for decades. Existing work in this area mainly focuses on employing the technique to carry out specific software analysis tasks (*e.g.,* [26], [7], [27]), or proposing new approaches to improve the technology, such as [28], [29], [30]. In such papers, the limitations and challenges of symbolic executions are occasionally discussed, but they are not systematically studied. Figure 1 demonstrates several important papers and the challenges they discussed.

There are several papers that focus on investigating the limitations of symbolic execution tools, including [19], [11], [24]. Kannavara *et al.* [24] have enumerated several challenges that may hinder the adoption of concolic execution in industrial fields. Qu and Robinson conducted a case study on the limitations of concolic testing tools and examined their prevalence in real-world programs. However, none of the two papers provides a method to evaluate concolic execution tools. Cseppento and Micskei [11] proposed several metrics to evaluate source code-based symbolic execution tools. Their metrics are based on specific program syntax of object-oriented codes rather than the language-independent challenges. Such metrics are not very general. More importantly, as shown in Figure 1, our work is more complete than existing papers in discussing the general challenges faced by symbolic execution techniques.

In our previous conference paper [8], we have conducted an empirical study with some of the challenges. This paper notably extends our previous paper with a formal benchmark methodology. It serves as a pilot study on systematically benchmarking symbolic execution tools in handling particular challenges. We provide a novel benchmark framework based on logic bombs, which can facilitate the automation of the benchmark process. We further provide a benchmark toolset that can be easily deployed by ordinary users.

III. Background of Symbolic Execution

A. Theoretical Basis

Symbolic execution is a white-box software testing approach which aims to generate test cases with high code coverage. The core principle of symbolic execution is symbolic reasoning. Informally, given a sequence of instructions along a control path, a symbolic reasoning engine can extract a constraint model and then solve the model to generate a test case for the path.

Formally, we can use Hoare Logic [31] 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 says 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:

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

$\{S_0\}$ 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. 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:

$$\{S_{n-2}\} = wp(I_{n-2}\{S_{n-1}\}), \quad s.t.\ \{S_{n-1}\}\ sat\ \Delta_{n-1}$$

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

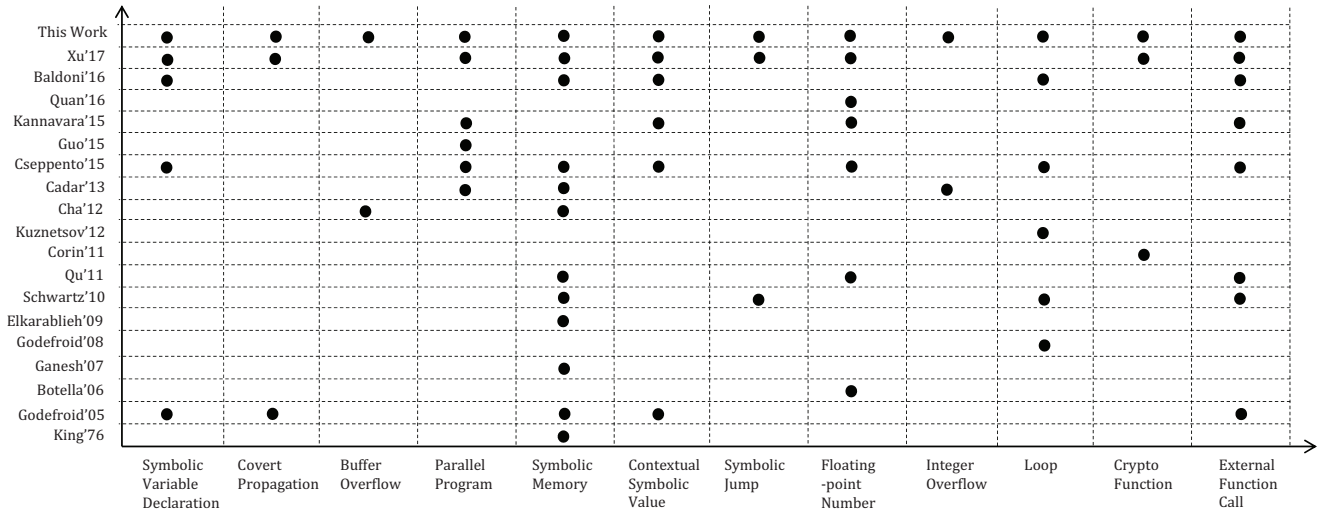

Fig. 1. The challenges of symbolic execution discussed in the literature. The detailed paper references are King'76 [1], Godefroid'05 [13], Botella'06 [14], Ganesh'07 [15], Godefroid'08 [16], Elkarablieh'09 [17], Schwartz'10 [18], Qu'11 [19], Corin'11 [20], Kuznetsov'12 [21], Cha'12 [22], Cadar'13 [9], Cseppento'15 [11], Guo'15 [23], Kannavara'15 [24], Quan'16 [10], Baldoni'16 [25], Xu'17 [8]

$$\{S_{n-3}\} = wp(I_{n-3}\{S_{n-2}\}), \quad s.t.\ \{S_{n-2}\}\ sat\ \Delta_{n-2}$$

$$\ldots$$

$$\{S_1\} = wp(I_1\{S_2\}), \quad s.t.\ \{S_2\}\ sat\ \Delta_2$$

$$\{S_0\} = wp(I_0\{S_1\}), \quad s.t.\ \{S_1\}\ sat\ \Delta_1$$

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

Finally, when sampling $\{I_i\}$, 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. Symbolic Execution Framework

We demonstrate the conceptual framework of a symbolic execution tool in Figure 2. It inputs a program and outputs test cases for the program. The framework includes a core symbolic reasoning engine and a path selection engine.

The symbolic reasoning engine analyzes the instructions along a path and generates test cases that can trigger the path. Based on the theoretical basis of symbolic reasoning, we can divide symbolic reasoning into four stages: *symbolic variable declaration*, *instruction tracing*, *semantic interpretation*, and *constraint modeling and solving*. All the stages should be precise to avoid generating incorrect test cases. The details are discussed as follows:

- *Instruction tracing* (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 are not supported, errors would occur.
- *semantic interpretation* (S_{sem}): This stage translates the semantics of collected instructions with an intermediate language. If some instructions are not correctly interpreted, or the data propagation are incorrectly modeled, errors would occur.
- *constraint modeling and solving* (S_{model}): This stage generates constraint models from IL, and then solve the constraint models. If a required satisfiability modulo theory is unsupported, errors would occur.

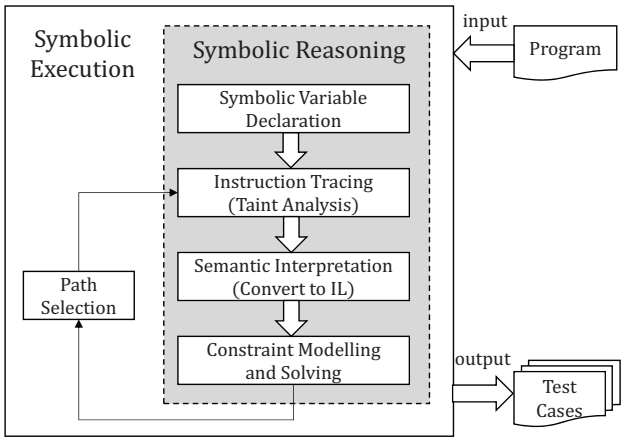

Fig. 2. A conceptual framework for symbolic executing

The path selection engine determines which path should be analyzed in the next round of symbolic reasoning. The selection strategies are important when there are scalability

issues or loops.

C. Implementation Variations

The taxonomy of symbolic execution includes several different implementation techniques, such as static symbolic execution or dynamic symbolic execution, source code-based symbolic execution or binary code-based symbolic execution. All the variations sharing the theoretical backgrounds in symbolic reasoning.

1) Static versus Dynamic: Static symbolic execution mainly differs from dynamic symbolic execution in the strategies of instruction collection.

Static symbolic execution is based on static analysis, which loads a whole program for analysis. In this way, it can select a control path from the control-flow graph of the program and extract all instructions along the path. Then it performes a symbolic reasoning process with the extracted instructions to find test cases. In the next round, it selects an alternative control path for analysis. The process continues until all the paths have been explored. In practice, there are different path selection strategies, or caching approaches to optimize symbolic execution [2].

Dynamic symbolic execution is also known as concolic (concrete and symbolic) execution. It collects instruction traces along control paths via concrete execution. In each concrete execution round, the concolic execution engine executes the program with a concrete value. Then the concrete value is replaced with symbolic variables for symbolic analysis. To deduce test cases that can trigger alternative control paths, the symbolic execution engine only need to negate one clause of the CNF and prone the tails, *e.g.,* $\Delta_1 \wedge \neg\Delta_2$.

2) Source Codes versus Binaries: 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). For source codes, we have to parse the syntax of the codes with a compiler frontend and convert the source codes into IL; for binaries, we have to lift the assembly codes into IL.

Their main difference lies in the converting process. Converting source codes to IL is much easier because the semantics are explicit. However, lifting assembly instructions from binaries are much complicated. Moreover, because there are no variables in binaries but only memories, so the symbolic variables in the resulting constraint model are generally memory addresses.

IV. Challenges of Symbolic Execution

Based on whether a challenge attacks the symbolic reasoning process, we categorize the challenges of symbolic execution into *accuracy challenges* and *scalability challenges*. An accuracy challenge attacks symbolic reasoning and may lead to incorrect symbolic analysis result. A scalability challenge does not attack the symbolic reasoning process, but it may starve the computational resources or may require very long time for symbolic execution.

Table I demonstrates the challenges that we have investigated in this work. We collect such challenges via a careful survey of existing papers. The survey scope covers several survey papers about symbolic execution (*e.g.,* [9], [25]), several investigations that focus on the challenges faced by symbolic execution (*e.g.,* [11], [24]), and other important papers about symbolic execution techniques (*e.g.,* [13], [22]).

A. Accuracy Challenges

Next, we discuss nine challenges that may incur errors to the symbolic reasoning process.

1) Symbolic Variable Declaration: Test cases are the solutions of symbolic variables to constrain models. Therefore, the symbolic variables should be declared before the symbolic analysis process. For example, source code-based symbolic execution tools (*e.g.,* KLEE) require users to declare each symbolic variable manually. Binary-based concolic execution tools (*e.g.,* Triton) 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 problem happens before symbolic execution, the challenge attacks S_{prep}.

2) Covert Propagation: Some data propagation ways are covert because they cannot be traced easily by data-flow analysis tools. For example, if the symbolic value are propagated via other channels outside of the process (*e.g.,* disk write/read), the propagation would be untraceable. Handling some propagation methods is beyond the capability of pure program analysis. Besides, there are propagation ways that may be challenging only to certain implementations. For example, propagating symbolic values via embedded assembly codes should be a problem for source-code based symbolic execution tools. If a symbolic execution tool fails to detect some propagation, the instructions related to the propagated values would be missed from the following analysis. Therefore, the challenge attacks the stages of S_{inst} and S_{sem}.

3) Buffer Overflow: Buffer overflow is a typical software bug that can bring security issues. Due to insufficient boundary check, the input data may overwrite adjacent memories. Adversaries can employ such bugs to inject data and intentionally tamper the semantics of the original codes. Buffer overflow can happen in either stack or heap regions. If a symbolic execution tool cannot detect the overflow issues, it would fail to track the propagation of symbolic values. Therefore, buffer overflow involves a special covert propagation issue. Source code-based symbolic execution tools are prone to be affected by buffer overflow because they are likely to be fooled by the information of data size defined in source codes. Moreover, the stack layout of a program only exists in assembly codes, which may vary for particular compiler options. Therefore, such tools cannot model the stack information with source codes and perform symbolic reasoning. In contrast, binary-based symbolic execution tools should be more powerful in handling buffer overflow issues because there can simulate actual memory operations. However, even if such tools can precisely track the propagation, they still suffer difficulties in analyzing the unexpected program behaviors caused by overflow.

TABLE I
A LIST OF THE CHALLENGES FACED BY SYMBOLIC EXECUTION, AND THE SYMBOLIC EXECUTION STAGES THEY ATTACK.

Challenge		Idea	Stage of Error		
			S_{prep}	S_{inst} & S_{sem}	S_{model}
Accuracy Challenge	Sym. Var. Declaration	Contextual variables besides program arguments	✓	✓	✓
	Covert Propagation	Propagating symbolic values in covert ways	-	✓	✓
	Buffer Overflow	Writing symbolic values without proper boundary check	-	✓	✓
	Parallel Program	Processing symbolic values with parallel codes	-	✓	✓
	Symbolic Memory	Symbolic values as the offset of memory	-	✓	✓
	Cont. Sym. Value	Retrieving contextual values with symbolic values	-	✓	✓
	Symbolic Jump	Sym. values as the addresses of unconditional jump	-	-	✓
	Floating-point Number	Symbolic values in float/double type	-	-	✓
	Integer Overflow	Integers outside the scope of an integer type	-	-	✓
Scalability Challenge	Loop	Change symbolic values within loops	-	-	-
	Crypto Function	Processing symbolic values with crypto functions	-	-	-
	External Function Call	Processing sym. values with some external functions	-	-	-

4) Parallel Program: Classic symbolic execution is only designed for sequential programs. We can draw an explicit control flow graph (CFG) for a sequential program and let symbolic execution parse the graph. If symbolic values are propagated in parallel, classic symbolic execution techniques would suffer problems. Note that the execution order of parallel codes does not only depend on the program, but also the context during execution time. In other words, a parallel program may exhibit different behaviors even with the same test case. This poses a problem for us to calculate test cases and make sure that they can trigger the same control flows. In practice, a symbolic execution tool may directly give up such programs or simply ignore the parallel codes. This would cause errors during S_{inst} and S_{sem}. Even if a tool can support the parallel syntax, the analysis can be very difficult. Supposing a parallel program with m threads and each thread has n instructions, the possibility of parallel execution is m^n, which can be a very large number.

5) Symbolic Memory: Symbolic memory is a situation that symbolic values serve as the offsets or pointers to retrieve values from the memory, such as array indexes. To handle symbolic memories, a symbolic execution engine should take advantage of the memory layout for analysis. Intuitively, it can convert a symbolic memory fetch operation to a `switch/case` clause in which the number of possible cases equals to the size of the symbolic memory. However, such size information may not be directly available in binaries. Moreover, when there are several such fetching operations along a control flow, the number of possible combinations would grow exponentially after conversion. Some theoretical analysis results have shown that symbolic memory involves pointer analysis issues, which can be NP-hard or even non-deterministic for static analysis [32]. If a symbolic execution tool cannot interpret such semantics, it would cause errors during S_{inst} and S_{sem}.

6) Contextual Symbolic Value: The challenge is similar to symbolic memory but more complicated. Other than retrieving values from the memory like symbolic memory, 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, such contextual information is unavailable to the program or process, so the analysis is difficult. Moreover, since such contextual information can be changed any time without informing the program, the problem is non-deterministic. A symbolic tool that doesn't support such semantics would cause errors during S_{inst} and S_{sem}.

7) Symbolic Jump: In general, symbolic execution only extracts constraint models when there are 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 jump. 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 attack the constraint modeling stage S_{model}.

8) Floating-point Number: 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 $f = sign * base^{exp}$. For example, the float type compliant to IEEE-754 is 32-bit (1-bit for $sign$, 23-bit for $base$, and 8-bit for 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 only have limited precision, which causes some unsatisfiable constraints over $\mathbb{R}$ can be satisfied 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. Otherwise, the challenge attacks S_{model}.

9) Integer Overflow: Integer 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 *e.g.,* $[-2^{64}, 2^{64} - 1]$. 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 integer overflow. Handling such integer overflow issues is not as difficult as previous challenges. However, some preliminary symbolic execution tools may fail to consider such cases and suffer errors when extracting and solving constraint models.

B. Scalability Challenges

Now we discuss three gadgets that may cause path explosion issues.

1) Loop: 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 control flows. By default, a symbolic execution tool should explore all the available control flows. If there are too many control flows, a symbolic execution tool would be incapable of handling them. In the literature, several heuristic approaches (*e.g.,* [33], [34], [35]) have been proposed towards mitigating such path explosion issues caused by loops. They generally employ specific path selection strategies to avoid being trapped in one loop, such that the tool can achieve high statement coverage rather than path coverage. However, the side effect is also apparent, *i.e.,* they may miss some important control flows.

2) Crypto Function: Crypto functions generally involve some computationally complex problems to ensure security. For a hash function, the complexity guarantees that adversaries cannot easily compute the plaintext of a hash value. For symmetric encryption function, it promises that one cannot easily 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 such crypto functions can be substantial. For example, the body of the SHA1 algorithm [36] is a loop that iterates 80 rounds, and each round contains several bit-level operations. In general, symbolic execution tools cannot handle such crypto programs.

3) External Function: Shared libraries, such as `libc` and `libm` (*i.e.,* a math 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 when execution. Therefore, such external functions do not enlarge the size of a program, but they can enlarge the code complexity in nature.

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 control 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 an error 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.

So far, we have discussed 12 different challenges in total. 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. Such analysis is essential for us to design the benchmark dataset in Section V-C1.

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) then
        // Trigger the bomb
        Bomb();
    end
```

V. Benchmark Methodology

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

A. Objective and Challenges

Before introducing 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 valid in several aspects. As we have discussed, some challenge issues are only engineering challenges, such as integer overflow. With enough engineering effort, a symbolic execution tool should be able to handle such issues. On the other hand, some challenges are difficult in a theoretical view, such as loop. But some heuristic approaches can tackle some easy cases. Symbolic execution tools may adopt different heuristics and demonstrate different capabilities in handling them. Therefore, the failures of symbolic execution tools in handling particular challenging issues can be caused by unimplemented features, approach limitations, or even bugs. Developers do not well demonstrate such problems on the tool website.

A useful benchmark approach should be accurate and efficient. However, it is difficult to benchmark symbolic execution tools accurately and efficiently. Firstly, a real program contains many instructions. When a symbolic execution failure happens, locating the root cause requires much domain knowledge and effort. Since errors may propagate, it is challenging to conclude that whether a symbolic execution tool really fails in handling a particular issue. Secondly, the symbolic execution itself is inefficient. Benchmarking a symbolic execution tool generally implies performing several designated symbolic execution tasks, which should be time-consuming.

Next, we introduce our approach and discuss how we overcome the challenges.

B. Our Approach

Our approach includes a core evaluation idea based on logic bombs and an automated benchmark framework.

1) Evaluation with Logic Bombs: A logic bomb is a code snippet that can only be executed when certain conditions are met. To evaluate whether a symbolic execution tool can handle a challenging issue, we can design a logic bomb guarded by

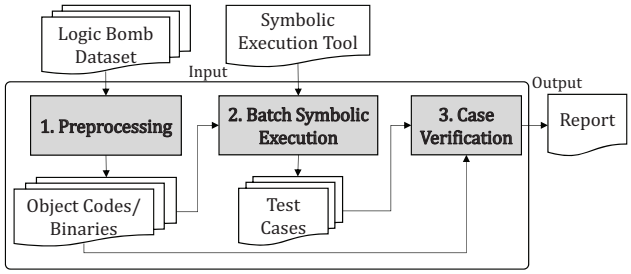

Fig. 3. A framework to benchmark symbolic execution tools

that particular issue. Then we can perform symbolic execution on the program which embeds the logic bomb. If a symbolic execution tool can generate a test case that can trigger the logic bomb, it implies the tool can handle the challenging issue or *vice versa*.

Algorithm 1 demonstrate a general framework to design 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 the condition has been satisfied. Note that because the value of $symvar2$ is propagated from $symvar$, $symvar2$ is also a symbolic value 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 contains only a challenging issue and a logic bomb that tells the evaluation result. Because the object programs for symbolic execution are 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 shorten the symbolic execution time either by controlling the problem complexity or employing a timeout setting.

2) Benchmark Framework: Base on the evaluation idea with logic bombs, we design a benchmark framework as shown in Figure 3. The framework inputs a dataset of carefully designed logic bombs and outputs the benchmark result for a particular symbolic execution tool. There are three critical phases in the framework: dataset preprocessing, batch symbolic execution, and case verification.

In the preprocessing phase, 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. If a target symbolic execution tool requires adding extra instructions to launch tasks, the parser should add such required instructions automatically. For example, KLEE requires adding symbolic variable declaration information within the source codes. The compilation process compiles the processed source codes into binaries or other formats needed by the target symbolic execution tool. 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-based symbolic execution tools, we can directly compile them into binaries, and the tool will lift binary codes into intermediate codes automatically.

In the second phase, we drive a symbolic execution tool to analyze the compiled logic bombs in a batch mode. This phase outputs a set of test cases for each logic bomb program. Some dynamic symbolic execution tools (*e.g.,* Triton) can directly tell which test case can trigger a logic bomb during runtime. However, others static symbolic execution tools cannot do this. Besides, some tools may falsely report that a test case can trigger the logic bomb. Therefore, we need a third phase: case verification.

In the third step, we execute each binary program with corresponding test cases. If a logic bomb can be triggered, it indicates that the challenging case has been solved by the tool. Finally, we can generate a report based on the case verification results.

C. Implementation

1) Logic Bomb Dataset: Following Algorithm 1, we have designed a dataset of logic bombs to evaluate the capability of symbolic execution tools. Our dataset includes over 60 logic bombs for 64-bit Linux platform, which covers all the challenges discussed in Section IV. For each challenge, we implemented several logic bombs. Either each bomb involves a unique challenging issue (*e.g.,* covert propagation via file write/read or via system call), or introduces a problem with a different complexity setting (*e.g.,* one-leveled array or two-leveled array).

To demonstrate how logic bombs work, we demonstrate several samples covering all the discussed challenge types in Figure 4. Figure 4(a) is a sample with a symbolic variable declaration problem. It contains a logic bomb which can be triggered 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 bomb. We can define other logic bombs by modifying the trigger condition, *e.g.,* with a certain time.

Figure 4(b) shows a sample with a covert propagation problem. We first get an integer value `i` from the symbolic value (`symvar`). `symvar` is a pointer that points to `argv[1]` after padding. Note that in this step, we avoid employing the `atoi` function to get an integer because some symbolic execution tools may not support it. Then, `i` is propagated to another variable (`ret`) through the `echo` command, and then the value of `ret` determines whether a logic bomb can be triggered. To find a test case that can trigger the logic bomb, a symbolic execution tool should be able to track the propagation. Similarly, we can design other covert propagation samples that propagate `i` via file write/read, socket, *etc*.

```c
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 declaration.

```c
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 propagation.

```c
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 overflow.

```c
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 program.

```c
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 memory.

```c
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 value.

```c
#define jmp(addr) asm("jmp *%0"::"r"(addr):)

int logic_bomb(char* symvar) {
  int offset = symvar[0] - 48;
  if (offset %6 != 1 || offset < 10|| offset > 40 || offset == 19){
    offset = 13;
  }
  long long addr = &&flag_0 + offset ;
  jmp(addr);
flag_0:
  if (symvar > 0){
    symvar++;
    if(symvar == 0)
      return BOMB_ENDING;
  }
  return NORMAL_ENDING;
}
```

(g) Symbolic jump.

```c
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 number.

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

(i) Integer overflow.

```c
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;
}
```

(j) Loop.

```c
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;
  }
}
```

(k) Crypto function.

```c
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;
}
```

(l) External function call.

Fig. 4. Logic bomb samples with challenging issues.

Figure 4(c) demonstrates a buffer overflow example. The logic bomb can be triggered if the value of `flag` equals 1, which is unlikely possible because the value should always be 0. 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 employ the bug to change the value of `flag`, *e.g.,* when `symvar` is "ANYSTRIN\x01\x00\x00\x00". We may design more complex buffer overflow problems that require modifying the return address on the stack, or design different buffer overflow problems which happen in the heap region.

Figure 4(d) demonstrates a logic bomb 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 logic bomb should be triggered. Similarly, we can design multi-process problems, or harder problems with more threads.

Figure 4(e) demonstrates an example of a symbolic memory problem. The code snippet contains an array, and the symbolic value `i%5` serves as an offset to retrieve an element from the array. Note that we should bound the offset to avoid segmentation issues because segmentation issues may interrupt some symbolic execution tools. To enrich our dataset, we may design dynamically allocated arrays, or increase the problem hardness by introducing two-leveled array as shown in Figure 7(a).

Figure 4(f) is an example of a contextual symbolic value problem. If `symvar` points to am existed file on the local

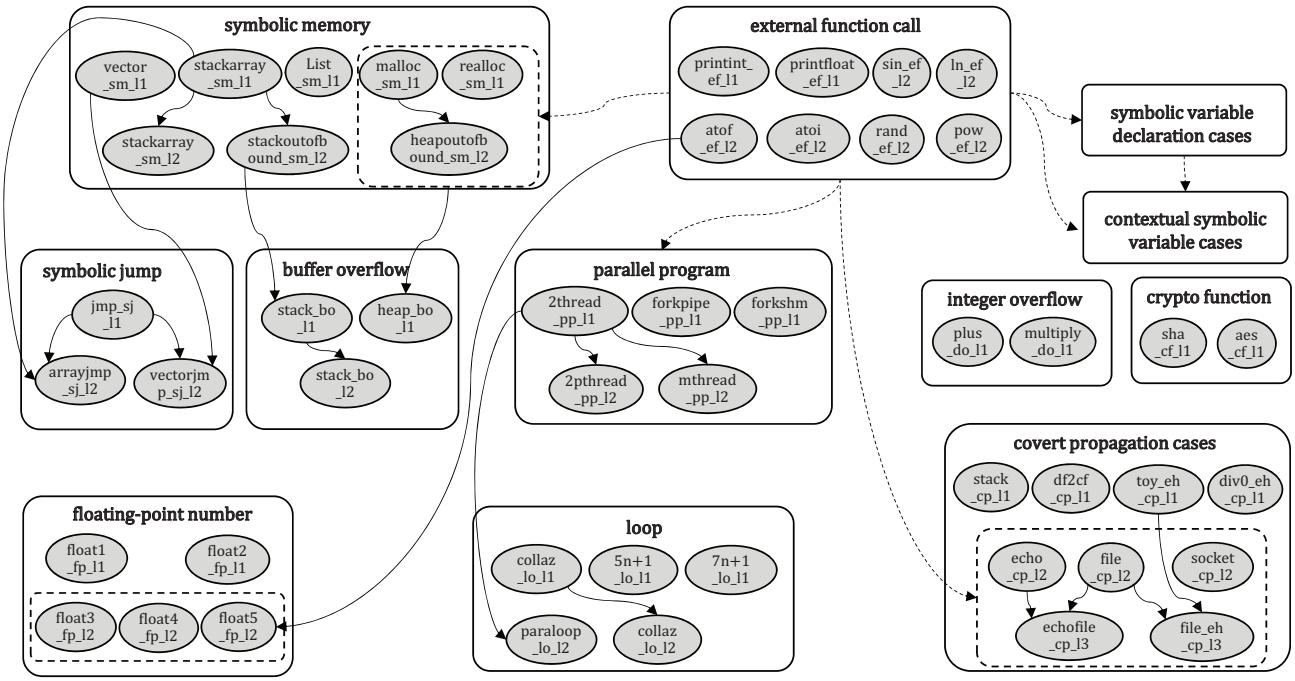

Fig. 5. The challenge propagation relationship among our dataset of logic bombs. A solid line means a logic bomb contains a similar problem defined in another logic bomb; a dashed line means a challenge may affect other logic bombs.

disk, the logic bomb would get triggered.

Figure 4(g) is an example of a symbolic jump problem. There is logic bomb guarded by a condition that cannot be satisfied. However, we still can trigger the bomb. The program first computes an address with the symbolic value. Then an embedded assembly instruction `jmp` takes the address as the parameter. If the address points to the logic bomb, the bomb would be triggered. Note that when designing such cases, we should also avoid segmentation issues by restricting the values of the address.

Figure 4(h) 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. The logic bomb will be triggered if the second condition $a == b$ can be satisfied. Therefore, one solution to trigger the logic bomb is '7'.

Figure 4(i) shows a sample with an integer 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.

Figure 4(j) shows a sample logic bomb with a loop. The loop function is implemented with the Collaz conjecture. No matter what is the initial value of `i`, the loop will terminate with $j = 1$. We may increase the expected loop `loopcount` to increase the problem complexity.

Figure 4(k) 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.

Figure 4(l) demonstrates a sample with an external function call. It computes the sine of a symbolic value via an external function call (*i.e.,* `sin`), and the result is used to determine whether the bomb should be triggered.

When designing the logic bombs, 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 when designing parallel codes, such to create threads. Surely that if a symbolic execution tool cannot handle external functions, the result might be affected. Therefore, we hope to demonstrate such potential influences for our dataset. To this end, we draw a challenge propagation chart among the logic bombs as shown in Figure 5. There are two kinds of challenge propagation relationships: $should$ in solid lines, and may in dashed lines. A $should$ relationship means 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 determinant one. For example, a parallel program generally involves external function calls. However, a tool may not be able to solve the external functions well, but it can solve the logic bombs with parallel issues.

2) Automated Benchmark Framework: We have implemented a framework prototype as described in Figure V-B2

with python. In the parsing step, the tool can automatically pad the logic bomb with a main function, and accepts `argv[1]` as the symbolic values. Then we customize batch symbolic execution scripts for several different symbolic execution tools. Because the dataset of logic bombs are written in C/C++, our prototype only supports symbolic execution tools for C/C++ programs or binaries. Currently, it supports three popular symbolic execution tools: KLEE [2], Angr [3], and Triton [5]. These tools are released as open source and have high community impact. Moreover, they adopt different implementation techniques for symbolic execution. By supporting such various tools, we wish to show that our framework can be extended to embrace other new tools.

KLEE[2] is a static symbolic execution tool implemented based on LLVM [37]. It requires users to manually declare the symbolic variables within the source codes by employing a `klee_make_symbolic` function. Then the source codes are compiled into intermediate codes for symbolic execution. Our tool can automate the process during the preparation phase. Because KLEE is a static analysis tool, it only generates test cases without running the program. Therefore, we have to run the binaries with the test cases and check the execution results. Our tool can automate such procedures of concrete executions and result checking.

Triton [5] is a dynamic symbolic execution tool based on binaries. It does not require users to declare symbolic variables manually, but automatically accepts symbolic variables from the program arguments. During symbolic execution, Triton firstly run the programs with concrete values and leverages Intel PinTool [38] to trace related instructions, then it lifts the traced instructions into the SSA (single static assignment) form and perform symbolic analysis. If there are alternative paths found in the trace, Triton will generate new test cases and employ them as the concrete values in the next rounds of concrete execution. Such a symbolic execution process goes on until no alternative path can be found. The symbolic execution script provided by Triton is ready-to-use and requires little customization.

Angr [3] is also a tool for binaries but employs different implementations. Before performing any symbolic analysis, Angr firstly lifts the binary program into VEX IR [39]. Then it employs a symbolic analysis engine (*i.e.,* 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 key features via APIs, and we only assemble them. Finally, we check whether the generated test cases can trigger logic bombs.

VI. Experimental Study

In this section, we conduct an experimental study to demonstrate the effectiveness of our benchmark approach and toolset. Below we discuss the experimental setting and results.

A. Experimental Setting

The objective of our experiment is to benchmark the capabilities of KLEE, Triton, and Angr in handling particular challenges. We employ our logic bomb dataset as the evaluation metric, and let each of the symbolic execution tools perform symbolic execution on the logic bombs. A tool can pass a test only if the generated solution can correctly trigger a logic bomb. We finally report which logic bombs can be solved 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 value which ensures benchmark 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 tools. 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.

B. Benchmark Results

1) Result Overview: Table II summarizes of our experimental results. Angr has achieved the best performance with 21 cases solved when the timeout value is 300 seconds. Such a number is only 16 when the timeout setting is 60 seconds. The performances of KLEE and Triton are much worse. They both have solved only three cases out of the 61, and their solved cases are distinctive. The results justify that our dataset can distinguish the capability of different symbolic-execution tools effectively.

To examine the accuracy of our benchmark results, we have manually verified each reported test case to trigger the logic bomb. Such test cases all meet our expectations, and there are no trivial test cases (*e.g.,* `\x00`) that can trigger the bombs. Moreover, the results are consistent with our declared challenge propagation relationships among logic bombs in Figure 5. This further verifies the correctness of our benchmark results.

The benchmark efficiency largely depends on the timeout setting. When the timeout value is 60 seconds, our benchmark process for each tool takes only a few minutes. When extending the timeout value to 300 seconds, the benchmark takes a bit longer time, and only Angr solves 5 more cases. Can the result get further improved with more time? We have tried another group of experiments with 1800 seconds timeout and the results remain the same. Therefore, 300 seconds should be a reasonable marginal timeout setting for our benchmark. Considering that symbolic execution is computational expensive which may take several hours or even several days to test a program [12], our benchmark process is efficient. Moreover, we may further improve the efficiency by employing a parallel mode, such as assigning each process several logic bombs.

2) Case Study: Now we discuss the detailed benchmark results for each challenge. Firstly, there are several challenges that none of the tools can trigger even one logic bomb, including symbolic variable declaration, parallel program, contextual symbolic value, loop, and crypto function. All the three

TABLE II
EXPERIMENTAL RESULTS ON BENCHMARKING THREE SYMBOLIC EXECUTION TOOLS (KLEE, TRITON, AND ANGR) IN HANDLING OUR LOGIC BOMBS. 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. FOR EACH TOOL, WE ADOPT TWO TIMEOUT SETTINGS: 60 SECONDS AND 300 SECONDS.

Challenge	Case ID	KLEE		Triton		Angr	
		t = 60s	t = 300s	t = 60s	t = 300s	t = 60s	t = 300s
Covert Propagation	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	fail	fail	pass	pass	pass	pass
	file_eh_cp	fail	fail	fail	fail	timeout	pass
	div0_eh_cp	fail	fail	fail	fail	timeout	pass
	file_eh_cp	fail	fail	fail	fail	timeout	fail
Buffer Overflow	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 Memory	malloc_sm_l1	fail	fail	timeout	fail	pass	pass
	realloc_sm_l1	fail	fail	fail	fail	pass	pass
	stackarray_sm_l1	fail	fail	fail	fail	pass	pass
	list_sm_l1	fail	fail	fail	fail	timeout	pass
	vector_sm_l1	fail	fail	fail	fail	timeout	pass
	stackarray_sm_l2	fail	fail	fail	fail	fail	fail
	stackoutofbound_sm_l2	fail	fail	fail	fail	pass	pass
	heapoutofbound_sm_l2	fail	fail	timeout	fail	pass	pass
Symbolic Jump	jmp_sj_l1	fail	fail	fail	fail	pass	pass
	arrayjmp_sj_l2	fail	fail	fail	fail	fail	fail
	vectorjmp_sj_l2	fail	fail	fail	fail	timeout	pass
Floating-point Number	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
Integer Overflow	plus_do	pass	pass	pass	pass	pass	pass
	multiply_do	pass	pass	fail	fail	pass	pass
External Function Call	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 Declaration	7 cases, all fail						
Contextual Symbolic Value	4 cases, all fail						
Parallel Program	5 cases, all fail						
Loop	5 cases, all fail						
Crypto Function	2 cases, all fail						
pass #	61 cases	3	3	3	3	16	21

tools only accept `argv` or manually labeled variables as the symbolic variable. They cannot automatically handle the symbolic values introduced during program execution. Therefore, they failed all the tests designed for the symbolic variable declaration challenge. The challenges of contextual symbolic value and crypto function involve very hard problems, and it is ordinary that the tools fail in handling them. However, it is a bit astonishment that all the tools neither support parallel programs nor even very simple loops. Such limitations are not well demonstrated on their websites.

Covert Propagation: Angr have 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 has also solved the case, but Triton failed. `stack_cp` propagates symbolic values via direct assembly instructions `push` and `pop`. Only KLEE failed the test because it is a source-based analysis tool which does not support assembly codes. Besides, Angr has also passed two test cases that propagate symbolic values via the C++ exception handling mechanism. Not only such exception haling mechanism can be covert for some source-code based analysis tools, such as KLEE, but also it can be covert for binary-based symbolic execution tools, such as Triton. We further break

```
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.

```
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. An exemplary program that demonstrates how the stack works with arrays. There is no information about the size of each array left in assembly codes.

down the details of an exception handling program in Figure 6. As shown in the box region of Figure 6(b), 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 file read/write, echo, socket, *etc*. Such detailed limitation reports are not available on the tool websites.

Buffer Overflow: Only Angr has solved one easy buffer overflow problem `stack_bo_l1`. The case has a simple stack overflow issue. Its solution requires modifing the value of the stack that might be illegal. However, Angr cannot solve the heap overflow issue `heap_bo_l1`. It also failed another harder stack overflow issue `stack_bo_l2` , which requires composing sophisticated payload, such as employing return-oriented programming methods [40]. We are surprised that Triton failed all the test because binary-based symbolic execution tools should be resilient to buffer overflows in nature.

Symbolic Memory: The results show that KLEE and Triton do not support symbolic memory, but Angr has very good support. Angr has solved seven cases out of eight. It only failed in handling the case (Figure 7(a)) with two leveled array `stackarray_sm_l2`. Also, it implies that when there are multi-leveled pointers, Angr would fail. Figure 7(c) demonstrates the assembly codes that initialize the arrays, and Figure 7(b) demonstrates the stack layout after initialization. We can observe that the information about array size or boundary does not exist in assembly codes. This justifies why binary-based symbolic execution tools do not suffer problems when a problem requires out-of-boundary access, *e.g.,* `stackoutofbound_sm_l2`.

Symbolic Jump: Since symbolic jump demonstrates no explicit conditional branches in the CFG, it should be a hard problem for symbolic execution. As we have expected, Triton and KLEE failed in all the tests. However, Angr is not likely to be affected by the trick. It has successfully handled two cases solved out of the three.

Floating-point Number: The results show KLEE and Triton

do not support floating-point operations, and Angr can support some. During our test, Triton directly reported that it cannot interpret such floating-point instructions. Angr has solved two cases out of the five. 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.

Integer Overflow: Integer overflow is not very a hard problem, and it only requires symbolic execution tools to handle such cases carefully. In our test, KLEE and Angr have solved all the cases. However, Triton failed in handling the integer overflow case in Figure 4(i). The result shows there is still much room for Triton to improve.

External Function Call: In this group of logic bombs, each case only contains one external function call. However, the result is very disappointing. Triton only passed a very simple case that print out (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 cannot support `atof_ef_l2` and other cases. The results show that we should be cautious when design logic bombs. Even when involving straightforward external function calls, they results could be affected.

VII. Conclusion

In conclusion, this work proposes an approach to benchmark the capability symbolic execution tools in handling particular challenges. To this end, we have studied the taxonomy of challenges faced by symbolic execution tools, including nine accuracy challenges and three scalability challenges. Such a study is essential for us to design the benchmark dataset. Then we have proposed a promising benchmark approach based on logic bombs. The idea is to design logic bombs that can only be triggered if symbolic execution solves specific challenging issues. By posing the programs of logic bombs as small as possible, we can speed up the benchmark process; by making them as straightforward as possible, we can avoid unexpected reasons that may affect the benchmark results. In this way, our benchmark approach can be accurate and efficient. Following the idea, we have implemented a dataset of logic bombs and a prototype benchmark framework which can automate the benchmark process. Then, we have conducted real-world experiments on three symbolic execution tools. Experimental results show that the benchmark process for each tool generally takes a few minutes. Angr has achieved the best benchmark results with 21 cases solved, and the number is only three for both KLEE and Triton. Moreover, our benchmark results have revealed several limitations about the tools but which are not well demonstrated on the tool websites. Such results justify the necessity of a third-party benchmark toolset for symbolic execution tools. Finally, we have released our toolset as open source on GitHub for public usage. We hope it would serve as an essential tool for the community to benchmark symbolic execution engines and would facilitate the development of symbolic execution techniques.

References

[1] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, 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. of the 8th USENIX Symposium on Operating Systems Design and Implementation*, 2008.

[3] 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.

[4] 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.

[5] 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.

[6] D. Davidson, B. Moench, T. Ristenpart, and S. Jha, "Fie on firmware: Finding vulnerabilities in embedded systems using symbolic execution." in *USENIX Security Symposium*, 2013.

[7] 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.

[8] H. Xu, Y. Zhou, Y. Kang, and M. R. Lyu, "Concolic execution on small-size binaries: challenges and empirical study," in *Proc. of the 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2017.

[9] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Communications of the ACM*, 2013.

[10] M. Quan, "Hotspot symbolic execution of floating-point programs," in *Proc. of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2016.

[11] 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.

[12] N. Hasabnis and R. Sekar, "Extracting instruction semantics via symbolic execution of code generators," in *Proc. of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016.

[13] P. Godefroid, N. Klarlund, and K. Sen, "Dart: directed automated random testing," in *ACM Sigplan Notices*, 2005.

[14] B. Botella, A. Gotlieb, and C. Michel, "Symbolic execution of floating-point computations," *Software Testing, Verification and Reliability*, 2006.

[15] 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.

[16] P. Godefroid, M. Y. Levin, D. A. Molnar *et al.*, "Automated whitebox fuzz testing." in *Proc. of the 15th Annual Network and Distributed System Security Conference*, 2008.

[17] B. Elkarablieh, P. Godefroid, and M. Y. Levin, "Precise pointer reasoning for dynamic test generation," in *Proc. of the 18th International Symposium on Software Testing and Analysis*. ACM, 2009.

[18] 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.

[19] 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.

[20] R. Corin and F. A. Manzano, "Efficient symbolic execution for analysing cryptographic protocol implementations." *ESSoS*, 2011.

[21] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea, "Efficient state merging in symbolic execution," *ACM Sigplan Notices*, 2012.

[22] 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.

[23] S. Guo, M. Kusano, C. Wang, Z. Yang, and A. Gupta, "Assertion guided symbolic execution of multithreaded programs," in *Proc. of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015.

[24] 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.

[25] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *arXiv preprint arXiv:1610.00502*, 2016.

[26] 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.

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

[28] 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.

[29] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, "Enhancing symbolic execution with veritesting," in *Proc. of the 36th International Conference on Software Engineering*. ACM, 2014.

[30] D. A. Ramos and D. R. Engler, "Under-constrained symbolic execution: Correctness checking for real code." in *USENIX Security Symposium*, 2015.

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

[32] W. Landi and B. G. Ryder, "Pointer-induced aliasing: A problem classification," in *Proc. of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1991.

[33] R. Majumdar and K. Sen, "Hybrid concolic testing," in *Proc. of the 29th IEEE International Conference on Software Engineering*, 2007.

[34] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *Proc. of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008.

[35] P. Saxena, P. Poosankam, S. McCamant, and D. Song, "Loop-extended symbolic execution on binary programs," in *Proc. of the 18th International Symposium on Software Testing and Analysis*. ACM, 2009.

[36] D. Eastlake 3rd and P. Jones, "Us secure hash algorithm 1 (sha1)," Tech. Rep., 2001.

[37] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *Proc. of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization*. IEEE Computer Society, 2004.

[38] 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.

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

[40] R. Roemer, E. Buchanan, H. Shacham, and S. Savage, "Return-oriented programming: Systems, languages, and applications," *ACM Transactions on Information and System Security (TISSEC)*, 2012.