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

pbSE: Phase-based Symbolic Execution

Qixue Xiao^1, Yu Chen^{1*}, Chengang Wu^{2*}, Kang Li^3, Junjie Mao^1,
Shize Guo^4 and Yuanchun Shi^1
^{1}Dept. of Computer Science and Technology, Tsinghua University, Beijing, China
{xqx12,maojj12}@mails.tsinghua.edu.cn, {yuchen,shiyc}@tsinghua.edu.cn
^{2}State Key Laboratory of Computer Architecture, Institute of Computing Technology
Chinese Academy of Sciences, Beijing, China. wucg@ict.ac.cn
^{3}Dept. of compute Science, University of Georgia, Georgia, USA. kangli@cs.uga.edu
^{4}School of CyberSpace Security, Beijing University of Posts and Telecommunications,
Beijing, China. nsfgsz@126.com

***Abstract*—**
The study of software bugs has long been a key area in software security. Dynamic symbolic execution, in exploring the program's execution paths, finds bugs by analyzing all potential dangerous operations. Due to its high coverage and abilities to generate effective testcases, dynamic symbolic execution has attracted wide attention in the research community. However, the success of dynamic symbolic execution is limited due to complex program logic and its difficulty to handle large symbolic data. In our experiments we found that phase-related features of a program often prevents dynamic symbolic execution from exploring deep paths. On the basis of this discovery, we proposed a novel symbolic execution technology guided by program phase characteristics. Compared to KLEE, the most well-known symbolic execution approach, our method is capable of covering more code and discovering more bugs. We designed and implemented pbSE system, which was used to test several commonly used tools and libraries in Linux. Our results showed that pbSE on average covers code twice as much as what KLEE does, and we discovered 21 previously unknown vulnerabilities by using pbSE, out of which 7 are assigned CVE IDs.

I. Introduction

Software bugs have long been a major concern in software security. In 1990s, Aleph One became the first one to publish a hacking method using stack overflow[22]. Since then, the analysis and mining of software bugs have gradually become a vital area in academic research and industrial applications.

Recently, there is an increasing trend to use symbolic execution[16] to detect bugs. Instead of using concrete data values as input and to represent the values of program variables as symbolic expressions over the symbolic input values, symbolic values are used as input. When symbolic execution encounters a branch condition, it forks the execution state, following both branch directions and updating the corresponding path constraints on the symbolic input. In doing so, it accomplishes path exploration and bug detection . Dynamic symbolic execution combines concrete execution and symbolic execution, in which path exploration is performed by symbolic execution, and when the solver fails, concrete execution is used to execute the path. Dynamic symbolic execution has attracted wide attention because it draws on the advantages of static, dynamic and symbolic analysis. In recent years, with the development of SMT solver, the unique advantage is becoming increasingly more prominent, and dynamic symbolic execution has been applied in bug detection in real-world programs[3], [14].

Theoretically, dynamic symbolic execution can explore all paths of a program. In practice, the rapidly increasing number of branches will lead to path explosion. This problem is not acute for small programs such as tools in Coreutils. But for those with a large number of code lines and need a sizeable symbolic data input, path explosion can be particularly severe.

The ability to cover more new codes in limited time will enhance the efficiency of bug detection. As software becomes more complex, path explosion becomes inevitable. This led to the emergence of many heuristic path selection researches [18], [7], [20], [1], [21], [27]. However, we observed that these techniques often helped symbolic execution to increase code coverage for a while but soon the increasing speed slows down. This trend is largely attributed to the existence of program *phases*.

For this paper, *phase* is defined as a property that captures the time varying behavior [29] of a program's execution. A group of code that repeatedly being executed in a given time frame is defined as a phase. Phases are not just loops but including broader concepts such as recursive function calls. The phases are inherently defined by the nature of program logic, which moves the program execution from one phase to the other based on the code and input. However, the observation of phases from concrete execution depends on the observing time intervals and the method to cluster similar code repetitions over time. More details about the observation of phases can be found in Section II.

We found that, when the coverage of a dynamic symbolic execution stops increase, the program often stuck in some specific phases of a program. Those phases that make symbolic execution difficult to move on are called trap phase. A trap phase is usually characterized by a loop or embedded loops, or recursive codes.

In this paper, we study the impact of program phases on the effectiveness of dynamic symbolic execution. We suggest to derive phase information from concrete execution and ex-

*To whom correspondence should be addressed plicitly use these derived phase information to guide dynamic symbolic execution. This paper presents a prototype of phase-based symbolic execution, which performs symbolic execution systematically to every phase, covers deeper paths, enhances code coverage and detects more bugs. Through this paper, we make the following contributions:

- A novel symbolic execution approach is proposed to systematically detect bugs by exploring code execution in phases. Phase information are obtained from concrete execution with a few seed inputs. These phase information allows symbolic execution to reach more new code in a limited time. Compared with the latest version of KLEE, code coverage of our approach is increased by more than 100 percent on average.
- We designed and implemented a tool on the basis of KLEE, pbSE(phase-based symbolic execution system), to demonstrate this approach.
- By applying pbSE to several commonly used software, we discovered 21 unknown vulnerabilities, out of which 7 are assigned CVE IDs.

This paper is organized as follows. Section 2 makes a detailed analysis of the influence of program phase's characteristics on symbolic execution. In Section 3 the phase-based symbolic execution approach is described and an overview of the system is given. Section 4 presents implementation and evaluation of our approach. In Section 5 related work is surveyed, and the conclusion is made in Section 6.

II. Program phase and Symbolic Execution

A. Methodology

Dynamic symbolic execution performs quite well when dealing with small programs, but its effectiveness worsens quickly for those with a large number of code lines and more symbolic input data. In the latter case, dynamic symbolic execution will cover a large number of new codes at the beginning, but the speed of finding new paths slows down. This section analyzes this phenomenon and reveals its underlying cause.

We analyze the code distribution during symbolic execution, and the results are compared with those obtained from concrete execution. In order to present the comparison in a clearer way, plots are used to illustrate the distribution patterns.

For each basic block(BB) entered during an execution, we record its index and the entry time. We plot the figures using y-axis to represent the index numbers of BBs, and x-axis to represent entry time.

For further analysis, the basic blocks are arranged according to the time sequence in concrete execution, and each BB is labeled with a number in rising order. BBs called more than once are indexed with the initial label. For convenience of comparison with symbolic execution, BBs already indexed in concrete execution will be labeled with the same number in symbolic execution. In the case of a BB not indexed in concrete execution, in symbolic execution we will give it a new number, following the max number of indexing in concrete execution.

Previous research [25], [36] has found that coverage of concrete execution is highly sensitive to input data (also called seeds). In our experiment we randomly choose seeds from reference file. For example, in the experiment with readelf, we randomly choose 10 from the 900 elf files from the directory /usr/bin in Ubuntu 12.04, whose size varies between 6120 bytes to 131,312 bytes.

Many excellent symbolic execution engines have been acknowledged by the academic community, or have been widely applied in industry, for instance, DART[13], KLEE[3], Crest[1], PEX[33], etc. KLEE is chosen in our experiment for the following reasons:

- KLEE, with its strong support for handling interactions with the external environment, for making input arguments and files symbolic, has been widely used in many research fields, including program analysis[6], [15], [18], software testing[8], [21], and software error detection[24], [7].
- KLEE can detect various kinds of bugs, including out-of-bounds and divide-by-zero. KLEE also provides different built-in search strategies, including traditional DFS, random search and some other heuristic search strategies, which will be used to compare its effectiveness with ours directly.
- KLEE is an open-source project and is constantly being updated, which is quite convenient for us to modify its code.

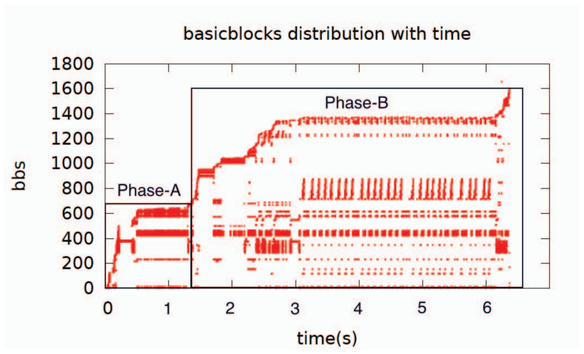
Moreover, our study also chooses several commonly used file-parsing tools and libraries as the test target, including readelf, libpng and libtiff. They are often released with operating systems and used widely, and they have a rich pool of software applications, for instance, various web browsers and editors, etc. In this sense, detecting bugs in these tools and libraries have great significance for enhancing software security.

B. Experiment Results and Analysis

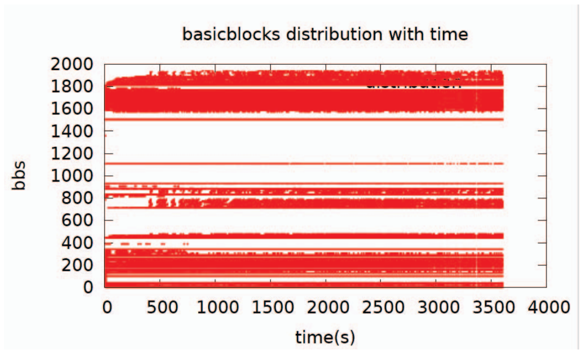
The follow experiment results demonstrate the program phase behavior in real measurements. We used real program execution logs (by recording the executed basic blocks) to show how a program travels through different phases with concrete inputs, and in contract, how much difficulties symbolic execution encounters in order to get out a phase.

The code distribution produced by concrete execution and symbolic execution are given in Fig. 1. Here readelf is taken as example. Readelf holds 727 functions and 16,990 BBs (including uclibc code). Fig 1(a) shows code distribution of concrete execution, and the seed file chosen is 7,960 bytes. Fig 1(b) shows the code distribution after symbolic execution has been running for an hour, with the default path searcher (random-path and covnew interleavingly employed), which is also the best search strategy in KLEE [3]. In these figures y-axis represents the index numbers of BBs, and x-axis represents time.

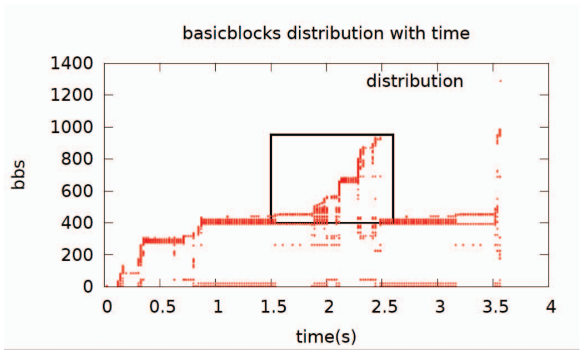
Based on concrete execution behavior of readelf, plus analysis of the source code, the execution can generally be divided into 2 phases (Phase-A and Phase-B as shown in Fig 1(a)).



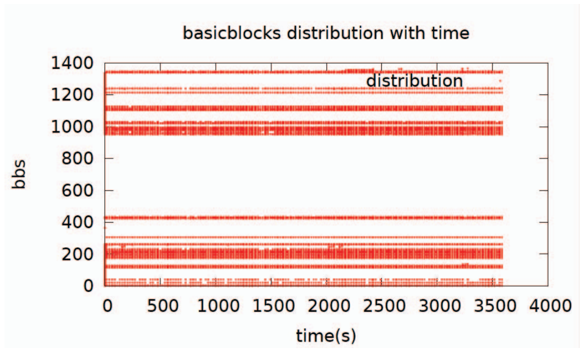
(a) BBs distribution of readelf performing concrete execution



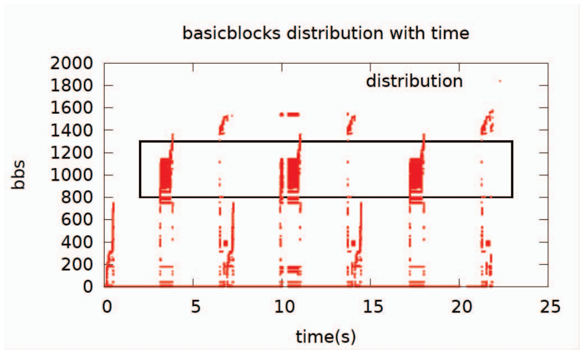
(b) BBs distribution of readelf performing symbolic execution



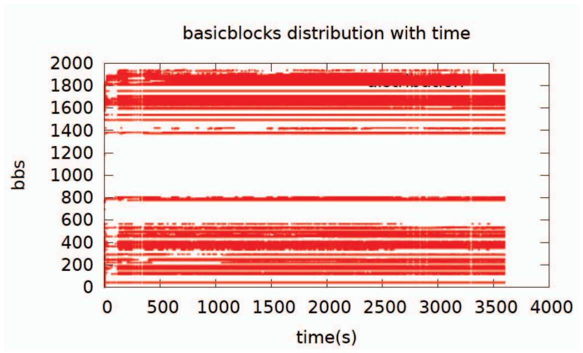
(c) BBs distribution of gif2tiff performing concrete execution



(d) BBs distribution of gif2tiff performing symbolic execution



(e) BBs distribution of pngtest performing concrete execution



(f) BBs distribution of pngtest performing symbolic execution

Fig. 1. Basic Blocks Distribution Analysis

By reading codes, Phase-A mainly includes the file header data handling, the following functions are called, such as *process_file_header*, *process_section_headers*; Phase-B mainly handles other data, includes sections, symbols, etc. The following functions are called, such as *process_dynamic_section*, *process_syminfo*, *process_section_contents*, etc.

It can be seen in Fig 1(b) , all of the BBs numbered between 500 to 700 are not covered by symbolic execution, while there are BBs near 500 which are covered by concrete execution in Phase-A. That means, in terms of execution path, none of the symbolic execution path covers these BBs after an hours running.

Based on code analysis, this is mainly due to the fact that there are at least 5 input depended loops[34] which end with *elf_header.e_phnum* or *elf_header.e_shnum*. *elf_header.e_phnum* indicates the number of program header table entry in the elf file and *elf_header.e_shnum* indicates the number of section header table entry. Both have to be read from files, which means the ending condition of the loops depends on symbolic input. This dependency is the very reason for path explosion, and symbolic execution fails to explore the codes on deeper paths.

It is noteworthy that symbolic execution does cover part of the codes in Phase-B. This is because in Phase-A there are some paths which can circumvent the loops, thus making it possible to jump into the next phase to cover the codes in Phase-B. Codes in Fig 2 is a good example. In function *process_section_groups*, symbolic execution can return from the function by executing code at line 6017 or 6024, therefore bypassing the loops at line 6046 to reach the next phase.

```
6003 process_section_groups (FILE * file)
6004 {
6005   Elf_Internal_Shdr * section;
6014
6015   /* Don't process section groups unless needed. */
6016   if (!do_unwind && !do_section_groups)
6017     return 1;
6018
6019   if (elf_header.e_shnum == 0)
6020    { printf (_("\nThere are no sections to group.\n"));
6021       if (do_section_groups)
6022
6023
6024       return 1;
6025    }
...
6046   for (i = 0, section = section_headers;
6047        i < elf_header.e_shnum;
6048        i++, section++) {}
...
6272 }
```

Fig. 2. Code snippets of readelf

In all the cases mentioned above, the case that there are BBs covered by concrete execution but not by symbolic execution is the most representative and important. Other programs demonstrated similar behavior. As in Fig 1, Fig 1(c) and Fig 1(d) give the results of the test of gif2tiff in libtiff; Fig 1(e) and Fig 1(f) give the results of the test of pngtest in libpng. By comparing the distribution of concrete execution and symbolic execution, it is easy to notice that there are a lot of BBs covered by concrete execution but not by symbolic execution, which are marked by boxes.

Further analysis shows that some phases are to a great extent responsible for preventing symbolic execution from exploring deeper paths, which was referred as trap phases by our study. One noticeable example is with readelf, which includes 5 loops to handle elf header data, and these 5 loops consist a phase that keeps symbolic execution from entering next phase.

Based on experiment results and code analysis we make the following conclusions.

- A phase of a program is made on the basis of similar or identical behavior. That is, in one phase identical or similar instructions are executed in a continuous long time, and this is usually caused by nested loops and deep recursions.
- For execution to pass from one phase to the next, some conditions have to be met. But in the code of one phase, there are potentially a great number of paths, only a few of which is capable of passing through to reach the next phase. If there are many phases which have to be passed, the chance of successfully reaching a deep phase is very small.

III. Phase-based Symbolic Execution

A. Overview

Our experiment results have shown that symbolic execution may be trapped in a certain phase of the program, making it difficult to jump out of this phase to explore the codes in the next. Therefore, it makes sense to automatically divide the program into phases and identify trap phases, and to perform symbolic execution for each individual phase, in order to enhance code coverage. This study proposes a phase-based symbolic execution test approach(pbSE). First, pbSE divides a concolic execution into different phases, and then symbolic execution is performed for each phase systematical. The overall procedure is given in Algorithm 1. pbSE takes the target program and seed file as input, and then performs concolic execution, phase analysis and symbolic execution, to finish the test and produce bug reports and related test cases. The overall architecture is shown in Fig. 3.

- **Concolic execution.** In pbSE, concolic execution is used to gather information needed by dividing different phases and performing symbolic execution. The information mainly consists of two groups of information: a) basic block vector(BBV), which logs the ids of executed basic block over a given time interval. Section III-B1 provides more details about BBV. b) seedStates, which record the conditions required for the execution to reach a specific state. Section III-B2 provides more details about seedStates. Concolic execution consists of performing concrete and symbolic execution. That is, for every instruction, a concrete execution is performed, which is immediately followed by symbolic execution. The basic block vector is gathered in performing concrete execution, and seedStates are obtained in performing symbolic execution. Symbolic execution performed in this step only records the fork point information while not explore any new state. This process is illustrated in Algorithm 2.
- **Phase parsing.** The second step is phase analysis, which mainly consists of phase division, selection and trap phase identification. pbSE first analyzes BBVs, and then classifies them into different groups, or phases, based on their similarities. For each phase, this step identifies the seedState that brings the execution to each specific phase. Details are provided in Section III-B1 and Section III-B2.
- **Symbolic execution.** In the third step, pbSE performs a systematical symbolic execution using a specific schedule strategy that uses the phase and seedState information gathered in step 2. Test cases are generated if bugs are found during symbolic execution. The schedule strategy is described in Algorithm 3 and more details are in Section III-B3.

B. Design Details

The design of pbSE needs to answer the following key questions: a) how we divide a program into phases using the concrete execution information, b) how pbSE uses the phase information to guide execution to new phases, c) how

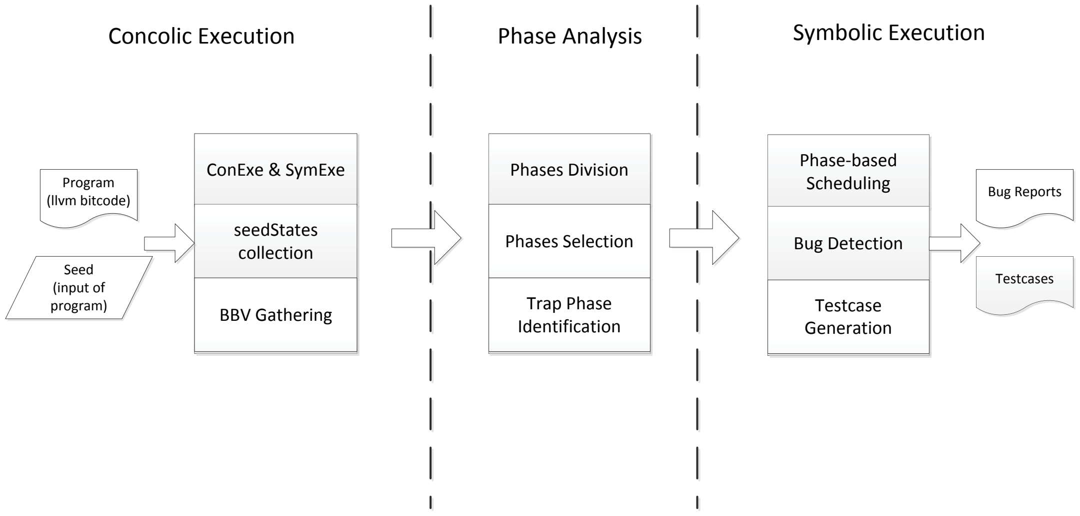


Fig. 3. Architecture of pbSE

```
Algorithm 1 Phase-Based Symbolic Execution Testing
Require: $Seeds, Target$
 1: $Seeds$ : seeds selected using methods similar with fuzzing
    testing
 2: $Target$ : the tested program which compiled in llvm
    bitcode
 3: function PBSYMBOLICEXETESTING($Seeds, Target$)
 4:     while $(!Seeds.empty())$ or $(!TIMEOUT)$ do
 5:         $seed = selectSeeds(seeds);$
 6:         $BBVs$ = CONCOLICEXE(seed);
 7:         $phases = PhaseAnalysis(BBVs);$
 8:         PBSYMBEXE($phases$)
 9:     end while
10: end function
```

to balance the execution time among different phases. The following text describes our design details that address these issues respectively, followed by a brief discussion of the potential enhancement with multiple seed inputs.

1) Phase Dividing: The first key step for pbSE is to divide one execution into phases. We choose to use basic block vector-based distribution analysis to achieve automatic division of program phases. Similar approaches have been applied in hardware simulation and performance analysis.

BBV Gathering: A basic block vector (BBV) maps the program's BBs onto a single dimensional array, and there is an element in the array for each BB. Each element of array is a count of how many times a given BB has been entered during a certain interval. pbSE gathers the information of BBVs when concolic execution is being performed.

Concolic execution maintains a concrete state and a symbolic state. For every instruction, a concrete execution is performed, which is followed immediately by a symbolic execution. When the execution reaches a TerminatorInst instruction, such as br or switch, the number of entries into the BB containing the instruction will increase by one. And it is done by the function *trackBB*(Algorithm 2, Line 17). In concolic execution, BBV data are gathered and recorded once after each interval. That is, the BBV data are recorded, and the elements in the array are set to zero for next interval. If the condition of this branch is symbolic, the symbolic execution would fork at this point and keep record the forked states. It is showed in at line 19-20 of algorithm 2. When concolic execution is finished, pbSE will obtain a set of BBVs and seedStates. This process is illustrated in Algorithm 2.

Phase Division: We use k-means algorithm to cluster BBVs into phases(Algorithm 1, Line 7). Since what we are concerned about is not the number of execution for each BB for a given interval; instead, what really counts is the proportions of BB execution. Every BBV has to be normalized.

In most cases the BBVs belonging to the same phase are highly dispersed in terms of time intervals. However, our purpose is to find phases consisting of densely distributed BBVs, because this distribution pattern is characteristic of trap phases. The change of code coverage over time is a typical reflection of trap phases. Code coverage usually increases with deepening of paths; but as the execution falls into a loop, it keeps repeating the same part of code, thus contributing nothing to the increase of code coverage.

Therefore, we add a new element to the array of BBV, that is, the code coverage at the moment when BBV is being gathered. Phase division in pbSE is made on the basis of the new BBVs with an element of code coverage added.

According to the above analysis, if in a series of continuous time intervals the same part of code is being repeatedly

Algorithm 2 Concolic Execution in pbSE

```
function CONCOLICEXE($seed$)
    $conExecutor.init(seed)$;
    $symExecutor.init(seed)$;
    $pc = MainEntry$;
    while $true$ do
        if $pc.instType! = EXIT$ and $pc.instType! = TerminatorInst$ then
            $nextPC = conExecutor.exec(pc)$;
            $symExecutor.exec(pc)$;
            $pc = nextPC$;
            $continue$;
        end if
        if $pc.instType == EXIT$ or $isFindBug()$ then
            $trackBB(pc.getParent())$;
            $break$;
        end if
        if $pc.instType == TerminatorInst$ then
            $trackBB(pc.getParent())$;
            $nextPC = conExecutor.exec(pc)$;
            if $isForkable()$ then
                $symExecutor.addSeedStates(pc.cond)$;
                $symExecutor.addSeedStates(!(pc.cond)$;
            end if
            $symExecutor.transferToPC(nextPC)$;
            $pc = nextPC$;
            $continue$;
        end if
        if $timeElapse()\%timeInterval == 0$ then
            $logToBBVs(BBVs)$;
        end if
    end while
     return $BBVs$
end function
```

executed, it means a trap phase has been identified. Therefore, a phase consisting of N continuous BBVs is identified as a trap phase, and the value of N can be customized. In pbSE, we empirically set N to 0.05 of the total number of BBVs in one execution.

When k-means is used to cluster phases, we try different values of k(from 1 to 20) to find the fittest value. That is, we choose the k value which can cluster the greatest number of trap phases, because the more traps phases identified, the better symbolic execution will be performed. When two or more values of k produce the same number of trap phases, we choose the smallest one.

Here it should be noted that a phase consists of several similar or identical BBVs; and each BBV is gathered within a specific time interval. Therefore, the BBVs of the same phase can be distributed in different intervals; or a phase may consist of many different intervals.

2) Pass through a phase: Generally, before symbolic execution can travel to a certain phase, it has to pass prior phases before this one. Considering the characteristics of symbolic

Algorithm 3 Symbolic Execution in pbSE

```
function PBSYMBEXE($phases$)
    $i = 0$;
    while $phases.size() > 0$ and $!TIMEOUT$ do
        $i++$;
        $phaseNum = i\%phases.size()$;
        $turnNum = i/phases.size() + 1$;
        $states = \&phases[phaseNum].states$;
        if $states.size() == 0$ then
            $phases.erease(phaseNum)$;
        end if
        while $states.size() > 0$ do
            ExecutionState $es = selectState()$;
            $es.executeInstruction()$;
            $updateStates(es)$;
            if $elapsedTime > turnNum * TimePeriod$ and $!isCoverNewInst()$ then
                $break$;
            end if
        end while
    end while
end function
```

execution, we introduce a lazy pass through.

Phases in pbSE are derived from concolic execution, which involves both concrete execution and symbolic execution. More specifically, the symbolic execution here only includes gathering path constraints and recording fork points.

In pbSE, the path of concolic execution is called as a seedPath. In addition, pbSE records new states at each fork point (Algorithm 2, Line 20-21), and we call them seedStates. The states mentioned here are the same as those in KLEE[3].

In our method, on the basis of the time when seedStates are forked, we map them to different phases. Therefore, symbolic execution performed to different phases can be transformed into one performed to the mapped seedStates. We only need seedStates as initial states to perform symbolic execution. Because seedStates contain the path information before they are forked in concolic execution. In this way, the phases before the forking point are passed through without any extra work.

3) Phase-based Scheduling: According to above analysis, in pbSE symbolic execution can be performed with seedStates in each phase as initial states. The pbSE scheduling needs to address these two issues: a) how to select seedState within a phase, b) when pbSE scheduler should consider moving to a new phase.

In order to enhance code coverage in symbolic execution, and to avoid repeating the same test in one phase, pbSE analyzes and selects the seedStates of a trap phase, mainly based on the location and type of the forking point. For seedStates which have the same forking point, we only keep the earliest seedState. What is more, pbSE adopts a schedule strategy to perform symbolic execution in turn for each phase, as shown in Algorithm 3. When symbolic execution performed to one phase no longer covers any new code in a certain period of time, pbSE begins to test the next phase. When all phases have been tested, pbSE will use a longer time period to perform symbolic execution to these phases again, until all possible execution paths have been covered, or the execution reaches the time-limit set by the user.

The execution order of phases is based on the time when the first BBV of them is gathered. This is because generally the path constraints of earlier phases are simpler than those of latter ones, hence more suitable for symbolic execution.

4) Enhancing code coverage by seeds: The phase information used by pbSE comes from concolic execution with seed inputs. Clearly the ability to capture a program's phase information heavily depends on the selection of seed inputs, and multiple seed inputs can likely generate better observation about a program's phase behavior. However, how to best use multiple seed inputs in symbolic execution is a challenging task, and the issue has attracted some research interests[25], [36]. Our research focused on showing the benefit of applying program phase information on symbolic execution, and in this work, we only consider the situation of a single seed input when multiple seeds are available. The selection of the single seed considers both the input length and the code coverage. The evaluation used in this paper is based on the following heuristic – when multiple seeds are available, we only consider the smallest 10 seed inputs, and we pick the one with top coverage among those 10. We expect to explore the better ways to use multiple seed inputs in the near future.

IV. Test and Evaluation

In this section, we use code coverage and bug detection to evaluate the performance of pbSE.

A. Experiment Setup

Our phase-based symbolic execution system pbSE is built on KLEE, mainly including concolic execution, phase dividing and analysis, and phase-based symbolic execution schedule algorithm. We implement concolic execution by maintaining both concrete state and symbolic state based on KLEE, which is different from the seeded symbolic execution in KLEE[2].

We use pbSE to test several commonly used file parsing programs, including *libtiff*, *libpng*, *readelf*, and *libdwarf*. *libpng* and *libtiff* are programs used to parse pictures; *readelf* and *libdwarf* are used to parse executable binary files. All the experiments were run on a server with Intel(R) Xeon(R) CPU(12 cores, 2.0GHz). The operating system is Ubuntu 12.04.

B. Coverage Evaluation

In coverage evaluation we mainly compare the results obtained by pbSE and those obtained by KLEE to analyze the coverages.

In the first experiment, we apply pbSE and KLEE to *readelf* to perform evaluation of coverage. With KLEE, we perform testing on *readelf* with several search strategies, including DFS, BFS, random-path, random-state, covnew and md2u. DFS and BFS searcher always select the latest or the earliest execution state among all the states to explore next. Random-state randomly selects a pending state to explore; Random-path searcher maintains a binary tree recording the program path followed for all active states, and selects states by traversing the execution tree from the root and randomly picks a direction when it encounters a branch until it reaches a leaf. Covnew and md2u use heuristics to compute which state has better chance to cover new code. Various factors are considered, such as the minimum distance to an uncovered instruction. The default searcher in KLEE employs random-path and covnew interleavingly. Because different size of symbolic files may result different code coverage when symbolic execution performed. We respectively use 10, 100, 1000, 10000 bytes as size of symbolic file in our tests. With pbSE, we choose a seed randomly from our ubuntu linux system.

Table I gives the number of basic blocks covered by pbSE and KLEE with different searchers. The first column gives the name of searchers used in experiment. The second and third columns show number of covered basic blocks when using a 10 bytes symbolic file, the second column is the result at the end of one hour, while the third column give the number at the end of tenth hour. The fourth and fifth columns show the result when using a 100 bytes symbolic file. The sixth and seventh, the eighth and ninth columns represent 1000 bytes and 10000 bytes respectively. The last row in the table shows the result of pbSE. In the last row, 'c-time' shows the time cost of concolic execution step, 'p-time' show the time cost in phase parsing step. 'seed(576)' represents that a seed whose size is 576 bytes used in testing and 'seed(7981)' represents the seed size is 7981 bytes.

TABLE I
Basic blocks covered by performing symbolic execution with different searchers

seacher	sym-10		sym-100		sym-1000		sym-10000	
	1h	10h	1h	10h	1h	10h	1h	10h
default	892	914	976	1079	996	1077	1013	1051
random-path	914	916	1162	1205	1163	1239	1141	1206
random-state	659	659	604	621	619	621	603	621
covnew	662	662	604	604	569	569	571	571
md2u	688	688	687	687	644	652	641	641
dfs	414	876	414	1231	413	1125	413	1103
bfs	757	853	687	687	644	652	634	662
pbSE	c-time		p-time		1h		10h	
seed(576)	3.7s		0.2s		1687		2455	
seed(7981)	407s		2.5s		1801		2597	

From the table we can see, KLEE covers no more than 1239 BBs with different search strategies in 10 hours, while pbSE can cover 2597 BBs in 10 hours. There are less than 10 minutes cost in the concolic execution and phase analysis steps, so we do not consider the time cost in our experiments that consume more than 10 hours.

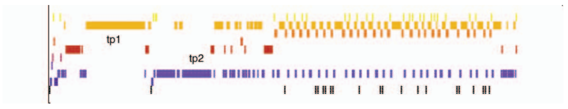
pbSE first divides the execution into phases, and the results are shown in Fig 4(b). And then pbSE performs symbolic execution on every phase according to the schedule strategy mentioned in Section III. Finally, pbSE can perform symbolic execution to cover the basic blocks in different phases that not covered by KLEE, and more new code is also covered, which enhances code coverage by 109% compared with the best results of KLEE.

In the second experiment, We apply pbSE and KLEE to the other three programs mentioned above to perform evaluation of coverage. For *libpng* and *libtiff*, we select seeds from the test cases provided by the project randomly. We choose random-path and covnew search strategy for KLEE, because they are generally considered to be the best ones.

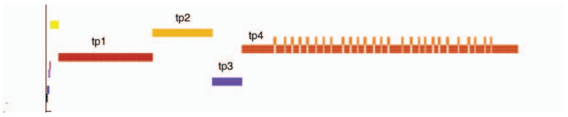
Table II gives the results of code coverage for each program. The first column gives the name of the program and test driver we used. Columns from the second column to the ninth give the results of random-path search strategy of KLEE with different size of symbolic file. Columns from the tenth to seventeenth give the coverage results of covnew strategy. The eighteenth and nineteenth columns represent the number of bbs covered by pbSE. The twentieth column shows the enhancement achieved by pbSE over the best result of KLEE.

Table II shows the results of code coverage. In the *libdwarf* testing, KLEE covers 491 BBs in the best case, while pbSE cover 1045 BBs, which is 112%more than KLEE. For *libpng*, the number of BBs covered by pbSE is higher than that of KLEE by 121%. And for *libtiff*, 134% higher. For *libpng* in the experiment we can see that the results of code coverage for KLEE in performing the 1-hour test and 10-hour test are almost the same. This is because in the 1-hour test of *libpng*, symbolic execution has already entered a trap phase, and code coverage naturally stops increasing. Our method can perform test to each individual phase, therefore can cover far more code.

It is noteworthy that pbSE can identify more trap phases than previous approaches. Phases division of *gif2tiff* are showed in Fig 4, in which different colors represent different phases, and the identified trap phases are marked by text(*tp* represents trap phase). Fig 4(a) gives the phase results obtained by dividing BBVs without code coverage. Fig 4(b) gives the results by combining BBVs and code coverage. From the figures it can be seen that phases obtained by BBVs only are more widely dispersed, so only two trap phases are identified. While four trap phases are obtained by combining BBV and code coverage for more continuously distributed code grouped.



(a) Trap phases identified by using BBVs without code coverage



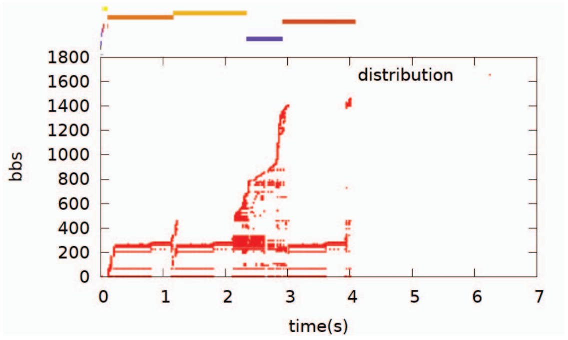
(b) Trap phases identified by combining BBVs and code coverage

Fig. 4. Different results of phase division, different colors show different phases.

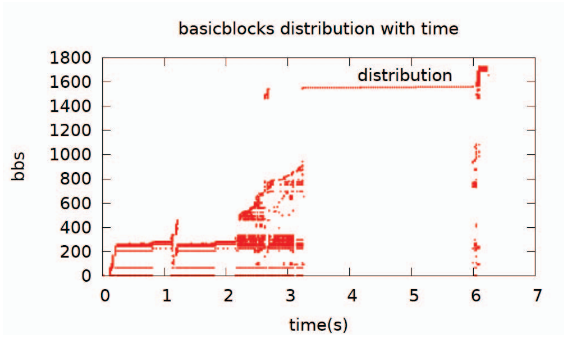
C. Bug Detection

Due to more new code is covered, pbSE can find more bugs. In our experiments, we discover 21 unknown bugs, 2 in *libpng*, 5 in *libtiff*, 4 in *readelf*, and 10 in *libdwarf*. Of the 21 bugs, 7 are assigned CVE ID. Table III presents the details. The first column gives the name of the tested packet. The second column shows test drivers used in the experiment. The third column (s-size) gives the size of seeds used by pbSE. The fourth column (t-p) shows the number of trap phases identified in the tests. The fifth column (b-p) shows index of the phase in which the bug is found. The rightest column shows CVE ID of the bug if assigned.

We describe below two representative errors found by pbSE, and compare its bug-finding ability against standard symbolic execution.



(a) Concrete execution using normal seed



(b) Concrete execution using buggy seed

Fig. 5. Code distribution of concrete execution using normal and buggy seed of tiff2rgba, (a) shows the code distribution using a normal seed, and its phases divded by pbSE are showed on top portion of the figure, (b) shows the code distribution using a buggy seed.

libtiff case study: libtiff is a library program to read tag information of TIFF image files, and it includes 24 test tools such as gif2tiff, tiff2pdf, tiff2rgba, etc. These tools analyze TIFF image files by calling *libtiff*. The version we tested in our experiments is *libtiff*-4.0.6, which has 62373 lines of code counted by cloc.

In the test of *libtiff*, pbSE reports more than 20 errors, which include at least 5 unknown bugs according to our analysis. Of them, 3 are error of memory-out-of-bound-read, 1 memory-out-of-bound-write, and 1 integer-overflow. Here we take one memory-out-of-bound-read as example. This bug is detected when tiff2rgba is being tested, and the seed is chosen randomly from internet.

TABLE II
RESULTS OF BASIC BLOCKS' NUMBER COVERED IN TESTING *libtiff*, *libpng* AND *libdwarf*

program	random-path								covnew								pbSE		
	sym-10		sym-100		sym-1000		sym-10000		sym-10		sym-100		sym-1000		sym-10000		1h	10h	inc
	1h	10h	1h	10h	1h	10h	1h	10h	1h	10h	1h	10h	1h	10h	1h	10h			
binutils-2.26 readelf	914	916	1162	1205	1163	1239	1141	1206	662	662	604	604	569	569	571	571	1801	2597	109%
libtiff-4.06 gif2tiff	541	541	573	573	567	567	567	567	541	541	558	561	566	567	544	544	1012	1457	134%
libpng-1.2.56 pngtest	606	606	843	843	846	848	846	848	606	606	825	843	843	843	825	835	1445	1878	121%
libdwarf-20150115 dwarfdump	460	465	462	495	465	492	453	482	460	470	455	488	453	506	457	502	932	1045	112%

TABLE III
BUGS FOUND BY PBSE

package	test-driver	s-size	t-p	b-p	CVEID
libpng	pngtest	8796	9	3	CVE-2015-7981
				5	CVE-2015-8540
libtiff	gif2tiff	407	4	3	N
				3	N
	tiff2rgba	243	4	3	N
		1428	5	3	N
	tiff2bw	1428	5	3	N
libdwarf	dwarfdump	9456	11	3	CVE-2015-8538
				2	N
		8336	10	5	CVE-2015-8750
				6	CVE-2016-2050
				3	N
		6908	7	5	N
				5	N
		8200	9	5	CVE-2016-2091
				4	N
		43600	9	3	CVE-2014-9482
binutils	readelf	7960	5	3	N
				4	N
		8336	7	5	N
				4	N

The code which includes the bug is shown in Fig 6. In function *putconig8bitCIELab*(at line 1647), the *w* and *h* values can be read from the file and can be a symbolic value when symbolic execution is performed. The size of memory which *pp* points to is a fixed value. In our test it is 257 bytes. When *h*w*3* is larger than 257, the memory read by the *pp* will be out of bound.

Here we mention this bug because the test of this bug is an excellent illustration of the advantage of pbSE. In our experiment, pbSE identifies 4 trap phases, and it is in testing the third phase that this bug is found. Fig 5(a) gives BB distribution in concrete execution with a normal seed and the division of phases. Fig 5(b) gives BB distribution in concrete execution with a seed which triggers the bug.

A comparison of Fig 5(a) and Fig 5(b) shows that the bug is not triggered until the program has been executed for some time (about 4 seconds). From the phases divided by pbSE(top portion of Fig 5(a)) we can see that it is the third trap phase that triggers the bug. In the test using KLEE, after 10 hours of testing KLEE only reaches the second phase; while pbSE detects the bug in one hour.

libpng case study: libpng is an official PNG image library, which is widely used in many operating systems and applications. In the test of *libpng* we use pngtest program as the test driver, and pngtest.png as the seed. Both of them are provided by the project.

```
1 #define DECLAREContigPutFunc(name) \
2 static void name(\
3     TIFFRGBAImage* img, \
4     uint32* cp, \
5     uint32 x, uint32 y, \
6     uint32 w, uint32 h, \
7     int32 fromskew, int32 toskew, \
8     unsigned char* pp \
9 )

1640 DECLAREContigPutFunc(putcontig8bitCIELab)
1641 {
1642     float X, Y, Z;
1643     uint32 r, g, b;
1644     (void) y;
1645     fromskew *= 3;
1646     while (h-- > 0) {
1647         for (x = w; x-- > 0;) {
1648             TIFFCIELabToXYZ(img->cielab,
1649                     (unsigned char)pp[0],
1650                     (signed char)pp[1],
1651                     (signed char)pp[2],
1652                     &X, &Y, &Z);
1653             TIFFXYZToRGB(img->cielab, X, Y, Z, &r, &g, &b);
1654             *cp++ = PACK(r, g, b);
1655             pp += 3;
1656         }
1657         cp += toskew;
1658         pp += fromskew;
1659     }
1660 }
```

Fig. 6. Code snippets of *libtiff*

We find 2 unknown bugs in *libpng*. Its developer patches the bug within 24 hours, and issued a warning of important threat on their website[12]. The CVEIDs are CVE-2015-7981 and CVE-2015-8540.

The execution process in which pngtest reads the file pngtest.png is divided into 9 phases, and the bugs we find are in phase 3 and 5. Here CVE-2015-8540 is taken as example. This is a buffer overflow, which may leak sensitive information in memory if exploited successfully by an attacker. This bug is found in *libpng* version 1.2.54, but it can also affect other versions, such as *libpng*1.2.55, 1.4.18, 1.5.25, etc. The bug is located in function *png_check_keyword* in pngwutil.c, and the code is shown as Fig 7.

The code mainly includes a loop which reads a byte from *kp* (at line 1284) and checks if it is a space; if yes, the code writes a zero into kp, and *kp* is decreased by one. This process is repeated until the byte is not a space. When the first byte of parameter key is a space, a under buffer overflow occurs(at line 1290).

It is in phase 5 that pbSE detects the bug. There are several

```
1231 png_size_t /* PRIVATE */
1232 png_check_keyword(png_structp png_ptr,
        png_charp key, png_charpp new_key)
1233 {
1234    png_size_t key_len;
1235    png_charp kp, dp;
......
1259  for (kp = key, dp = *new_key; *kp != '\0'; kp++, dp++)
1260    {
......
1279    }

1283    kp = *new_key + key_len - 1;
1284    if (*kp == ' ')
1285    {
1288       while (*kp == ' ')
1289       {
1290            *(kp--) = '\0';
1291            key_len--;
1292       }
1293    }
```

Fig. 7. Code snippets of libpng affected by CVE-2015-8540

loops before the vulnerable code, such as line 1259. The end condition of the loop is to check if the byte in key is zero, while the value of key is obtained from the file. Therefore when symbolic execution is performed, these are symbolic values, which result in path explosion. It is difficult for KLEE to jump out of the loop in limited time to cover the code behind.

Another bug found by pbSE is a memory-out-of-bound-read, which is located in function *png_convert_to_rfc1123* in png.c, The attacker can obtain sensitive information in memory by counterfeiting tIME chunk data in a png image. The bug affects several versions of *libpng*, including 1.0.X before 1.0.64, 1.2. 1.2.x before 1.2.54, and 1.4.x before 1.4.17, etc. When the month value of tIME chunk is set to be zero, the index value of the array *"short_month"* will be -1, therefore trigger a memory-out-of-bound-read. The vulnerable code is shown as Fig 8.

Others: libdwarf is an open source library inspecting the DWARF debug information in object file. The version we use in test is *libdwarf*-20151114, with 53857 lines of C code. We use dwarfdump provided by the project as test driver, and use the same seeds as the one in testing *readelf*. 10 unknown bugs are found in *libdwarf*, including 7 memory-out-of-bound-read errors, 2 memory-out-of-bound-write errors, and 1 null pointer reference. Of these, 5 have been assigned CVE IDs.

In the test of *tcpdump* we do not find new bugs. This is because *tcpdump* is mainly used to capture network packets, and no complex analysis is performed to network packets. The data read out are simpled printed. What is more, the other reason may be that pbSE's interaction and handling abilities of network related operations are quite limited.

V. Related Works

This section introduces research works related to our study from two aspects: dynamic symbolic execution and program phase.

```
png_charp PNGAPI
png_convert_to_rfc1123(png_structp png_ptr,
    png_timep ptime)
{
static PNG_CONST char short_months[12][4] =
{"Jan", "Feb", "Mar", "Apr", "May", "Jun",
"Jul", "Aug", "Sep", "Oct", "Nov", "Dec"};
...
png_snprintf6(png_ptr->time_buffer, 29,
   "%d %s %d %02d:%02d:%02d +0000"
         ptime->day % 32,
         short_months[(ptime->month-1)% 12],
         ptime->year, ptime->hour % 24,
         ptime->minute % 60,
         ptime->second % 61);
...
```

Fig. 8. Code snippets of libpng affected by CVE-2015-7981

A. Dynamic Symbolic Execution

Dynamic symbolic execution (DSE) is an improvement on traditional symbolic execution, which is characterized mainly by mixing concrete execution and symbolic execution. Representative examples are concolic testing and Execution Generate Testing(EGT)[5]. Two early applications of concolic testing are DART[13] and CUTE[28], which explore all possible paths and check various errors by combing dynamic test case generation and random testing. EXE[4] and KLEE[3] are examples using EGT. They mix concrete execution and symbolic execution in a way other than concolic testing, and implement a memory model based on bit-level accuracy while at the same time providing strong support in interaction with external environment.

Path explosion remains a major challenge faced by dynamic symbolic execution. As programs become more powerful and complex, it is impossible to explore all paths in a program. To cover as many codes as possible or to cover a specified part of code in limited time has become research focus, which has given birth to many heuristic path search strategies. Hybrid Concolic Testing[20] is an execution approach which interleavingly uses random testing and concolic execution to achieve deep and wide program state space. SAGE[14] provides a generational search strategy exploring new paths by generating test cases based on symbolic execution. In this process the test case is constantly changed to guide the exploration of different paths. ZESTI[8] explores new paths that slightly diverge from paths with sensitive instructions already explored using symbolic execution. In this way more paths leading to the sensitive instructions will checked, thus accomplishing thorough testing of them.

Loops (especially unbounded loops) can cause an enormous or even infinite number of paths to be explored in DSE[34]. Bounded iteration and search-guiding heuristics are the most widely used loop techniques across various tasks for their relatively low analysis cost, and yet are incomplete or unsound. However, bounded iteration may cause certain subsequent branches not to be covered, and search-guiding heuristics may not handle different loops well. For examples, [1] uses weighted control flow graph to guide symbolic execution to cover unexplored code. The fitness guided search strategy[35] chooses the best path using a fitness function to calculate the distance to a specified test target. Subpath Guided Path search strategy[18] makes decisions guided by information from explored paths, which proved better guidance.

Driller[32] and MoWF[23] are two effective hybrid fuzzers which detecting bugs combining fuzzing and concolic execution. Approaches such as Driller use symbolic execution to guide fuzzing efforts (which are concrete execution), whereas our approach focuses on using information obtained through concrete execution, such as program phase information, to guide symbolic execution. Therefore, our pbSE approach is orthogonal to Driller and those previous hybrid fuzzers.

B. Program Phase

The execution of a program is characterized by different phases. As early as 1999 Sherwood started a research about program phase[29]. In his opinion, program phase can be regarded as time varying behavior of the program; a phase can be obtained by grouping execution codes with identical or similar behavior[30], [10], [11]. The execution of program can be divided into a series of phases, with each phase different from others. On account of the different behavior of various phases, each phase can be regarded as a specific function[31]. Phase is often used in hardware simulation researches, in which simulation of representative phases can be used instead of the entire execution. ADORE[9] uses phase information to test the performance bottleneck of BLAST, and optimizes it at runtime.

It is a common practice to gather information during execution intervals, which are then grouped into phases. The gathered information can be IPC, branch miss rate, cache miss rate, value misprediction, conditional branch counts and basic block vectors, etc.[29], [30], [10], [11]. It can even be resource type vector, eDoctor[19] divides phases according to resource usage and captures an app's behavior on this basis to obtain information on power consumption of Apps. In [10], the authors compare phase division techniques using instruction working sets, basic block vectors and conditional branch counts. They find that BBV techniques perform better than other techniques providing higher sensitivity and more stable phases. In our research, a technique combining BBV and code coverage is used to more accurately detect trap phases.

Both phase information and symbolic execution have been extensively employed in many researches. But to our knowledge, the combination of them is the first proposal in literature.

The pbSE approach works well for programs that inherently progress in pipelined stages. Each stage consume a chunk of input that is independent from inputs from those consumed by other stages. The program leaves a stage and enters the next only when a certain condition is met, such as requiring a specific header field. Symbolic execution often get stuck with in one of the stage due to the path explosion problem. The pbSE approach helps by detecting the pipeline stages (with maps to the phases detected by our algorithm), and explicitly guide the symbolic execution to deeper pipeline stages.

However, program phases could not be easy to identify on nondeterministic execution, like for example in device drivers. DDT[17] mitigates the problem of polling loops (which really stress symbolic execution of device drivers) in device drivers through static analysis (loop-analysis) on disassembled driver code. The approach analyzes the disassembly of the target drivers and reports a list of paths which the symbolic execution engine should kill to enable the progress in the execution. SymDrive[26] uses favor-success path-selection algorithm that prioritizes paths which arrive before at the return instruction of the function.

VI. Conclusion

On the basis of extensive experiments, we find that trap phases of programs often prevent dynamic symbolic execution from going deeper. To solve this problem, we propose the phase-based symbolic execution approach to guide symbolic execution to reach individual phases. We have implemented pbSE based on the state-of-the-art symbolic execution engine KLEE. In our experiments, pbSE is used to test several commonly used Linux tools and libraries. The reuslts are quite encouraging: pbSE can cover more new code and detect more bugs. Moreover, 21 unknown bugs, 7 out of which have been assigned CVE IDs.

VII. Acknowledgements

We would like to thank Zhe Wang, Jianjun Li, and the anonymous referees, for their valuable suggestions and helpful comments. This research is supported by the National Natural Science Foundation of China (NSFC) under grant 61332009, the Beijing Municipal Science & Technology Commission Program under grant D161100001216002, and H863-015. Dr. Kang Li's effort is partially supported by National Science Foundation (NSF) under award No. 1319115 and a gift award from Intel Corp.

References

[1] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation (Crest). *ASE 2008, 23rd IEEE/ACM International Conference on Automated Software Engineering, Proceedings*, 2008.

[2] C. Cadar. Klee llvm execution engine. https://klee.github.io/, 2016.

[3] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. *OSDI*, 2008.

[4] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE : Automatically Generating Inputs of Death. *CCS*, 2006.

[5] C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, pages 1–8, 2013.

[6] V. Chipounov, V. Kuznetsov, and G. Candea. The S2E Platform. *ACM Transactions on Computer Systems (TOCS)*, pages 1–49, Feb. 2012.

[7] H. Cui, G. Hu, J. Wu, and J. Yang. Verifying systems rules using rule-directed symbolic execution. *Proceedings of the eighteenth international conference on Architectural support for programming languages and operating systems (ASPLOS)*, page 329, 2013.

[8] P. Dan Marinescu and C. Cadar. make test-zesti: A symbolic execution solution for improving regression testing. *ICSE*, pages 716–726, June 2012.

[9] A. Das, J. Lu, H. Chen, J. Kim, P. C. Yew, W. C. Hsu, and D. Y. Chen. Performance of runtime optimization on BLAST. *Proceedings of the 2005 International Symposium on Code Generation and Optimization, CGO 2005*, 2005:86–95, 2005.

[10] A. S. Dhodapkar and J. E. Smith. Comparing Program Phase Detection Techniques. *Proceedings of the 36th Annual International Symposium on Microarchitecture, San Diego, CA, USA, December 3-5, 2003*, pages 217–227, 2003.

[11] E. Duesterwald, C. Caşcaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *Parallel Architectures and Compilation Techniques, 2003. PACT 2003. Proceedings. 12th International Conference on*, pages 220–231. IEEE, 2003.

[12] glennrp Rander-Perhron. Libpng: Png reference library. http://www.libpng.org/pub/png/libpng.html, 2015.

[13] P. Godefroid. DART : Directed Automated Random Testing. *SIG-PLAN Conference on Programming Language Design and Implementation(PLDI)*, pages 213–223, 2005.

[14] P. Godefroid, M. Y. Levin, and U. C. Berkeley. Automated Whitebox Fuzz Testing. *NDSS*, 2008.

[15] W. Jin and A. Orso. BugRedux: Reproducing field failures for in-house debugging. *34th International Conference on Software Engineering (ICSE)*, pages 474–484, June 2012.

[16] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, pages 385–394, 1976.

[17] V. Kuznetsov, V. Chipounov, and G. Candea. Testing Closed-Source Binary Device Drivers with DDT. *USENIX Annual Technical Conference*, (June), 2010.

[18] Y. Li, Z. Su, L. Wang, and X. Li. Steering symbolic execution to less traveled paths. *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications(OOPSLA)*, pages 19–32, 2013.

[19] X. Ma, P. Huang, X. Jin, P. Wang, S. Park, D. Shen, Y. Zhou, L. K. Saul, and G. M. Voelker. edoctor: Automatically diagnosing abnormal battery drain issues on smartphones. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, Lombard, IL, USA, April 2-5, 2013*, pages 57–70, 2013.

[20] R. Majumdar. Hybrid Concolic Testing . *ICSE*, 2007.

[21] P. Marinescu and C. Cadar. KATCH: High-coverage testing of software patches. *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering(FSE)*, 2013.

[22] A. One. Smashing the stack for fun and profit. In *Phrack magazine volume 49*, 1996.

[23] V.-T. Pham, M. Böhme, and A. Roychoudhury. Model-based whitebox fuzzing for program binaries. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE, pages 552–562, 2016.

[24] D. a. Ramos and D. Engler. Under-Constrained Symbolic Execution : Correctness Checking for Real Code. *USENIX Security Symposium*, pages 49–64, 2015.

[25] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, S. Engineering, G. Grieco, C. I. Franco, I. Científicas, and D. Brumley. Optimizing Seed Selection for Fuzzing. In *23rd USENIX Security Symposium*, 2014.

[26] M. J. Renzelmann, A. Kadav, and M. M. Swift. SymDrive : Testing Drivers without Devices. *Osdi'12*, pages 279–292, 2012.

[27] P. Saxena, P. Poosankam, S. McCamant, and D. Song. Loop-extended symbolic execution on binary programs. pages 225–236. ACM, 2009.

[28] K. Sen, D. Marinov, and G. Agha. Cute: A concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.

[29] T. Sherwood, B. Calder, and S. Diego. Time Varying Behavior of Programs. *Technical Report UCSD-CS99-630*, pages 1–16, 1999.

[30] T. Sherwood, E. Perelman, and B. Calder. Basic block distribution analysis to find periodic behavior and simulation points in applications. *Parallel Architectures and Compilation Techniques, 2001. Proceedings. 2001 International Conference on*, (September):3–14, 2001.

[31] T. Sherwood, S. Sair, and B. Calder. Phase tracking and prediction. In *30th International Symposium on Computer Architecture (ISCA) 2003), 9-11 June 2003, San Diego, California, USA*, pages 336–347, 2003.

[32] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: Augmenting Fuzzing Through Selective Symbolic Execution. *NDSS*, pages 21–24, 2016.

[33] N. Tillmann and J. de Halleux. Pex: White Box Test Generation for .NET. *Proc. TAP*, pages 134–153, 2008.

[34] X. Xiao, S. Li, T. Xie, and N. Tillmann. Characteristic studies of loop problems for structural test generation via symbolic execution. *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013 - Proceedings*, pages 246–256, 2013.

[35] T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. Fitness-guided path exploration in dynamic symbolic execution. *2009 IEEE/IFIP International Conference on Dependable Systems & Networks*, pages 359–368, jun 2009.

[36] C. Zhang, A. Groce, and M. A. Alipour. Using test case reduction and prioritization to improve symbolic execution. *Proceedings of the 2014 International Symposium on Software Testing and Analysis - ISSTA 2014*, pages 160–170, 2014.