SECURITY AND COMMUNICATION NETWORKS
Security Comm. Networks 2015; **8**:3667–3681
Published online 29 June 2015 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/sec.1290

RESEARCH ARTICLE

An empirical investigation into path divergences for concolic execution using CREST

Ting Chen[1]*, Xiaodong Lin[2], Jin Huang[1], Abel Bacchus[2] and Xiaosong Zhang[1]

[1] Institute of Big Data Security, Big Data Research Center, University of Electronic Science and Technology of China, Chengdu, China
[2] Faculty of Business and Information Technology, University of Ontario Institute of Technology, Oshawa, Ontario, Canada

ABSTRACT

Recently, concolic execution has become a hotspot in the domain of software testing and program analysis. However, a practical challenge, called path divergence, impairs the soundness and completeness of concolic execution. A path divergence indicates the tested program runs an unpredicted path. In this work, we carry out a comprehensive empirical study on path divergences using an open-source concolic execution tool, named CREST. To make the investigation representative, we select 120 test units randomly from 21 different open-source programs. The results are interesting, and will provide insight to solve the challenging path-divergence problem. First, about one-half of test units suffer from path divergences, indicating path divergences are so prevalent that the issue is worthy of great attention. Second, quite a number of generated test inputs drive test units to take divergent paths. This means testers need considerable effort to eliminate the misleading test inputs before aggregating them to a test suite. Third, we dig out ten divergent patterns through manual analysis of each path divergence. Among them, the three most prevalent ones, which are exceptions, external calls, and type casts, lead to almost 82% of path divergences. Finally, we discuss several countermeasures to overcome path divergences. Copyright © 2015 John Wiley & Sons, Ltd.

KEYWORDS
concolic execution; path divergences; prevalence; misleading inputs; divergent patterns; countermeasures

***Correspondence**
Ting Chen, Institute of Big Data Security, Big Data Research Center, University of Electronic Science and Technology of China, Chengdu, China.
E-mail: brokendragon@uestc.edu.cn

1. INTRODUCTION

Software testing is a widely accepted approach for improving software security, reliability, and quality. While vendors may purport to create their highest quality software, testing is at times jeopardized by a lack of time, tools, and man power. Manually deliberating over thousands of code lines is both exhausting and error prone. To address these tiresome tasks, a number of automated testing tools have been introduced. Among them is a promising technique dubbed symbolic testing, which dates back to the 1970s [1,2].

Concolic execution [3], also termed dynamic symbolic execution [4] or directed automated random testing [5], is an exciting variation of symbolic execution. As the name suggests, concolic execution combines the techniques of concrete execution and symbolic execution. It consists of instrumenting and running a program while collecting path constraints on test inputs from predicates encountered in branch instructions and of deriving new test inputs from previous path conditions via a theorem prover, so as to steer the next executions toward new program paths. A formal definition can be found in [5].

Symbolic execution runs in a different way. It considers all test inputs as symbols, and it treats concrete execution as symbolic computations. When a symbolic predicate is encountered, it consults a theorem prover to determine which branch is feasible. A feasible branch is then chosen for further exploration according to its path exploration algorithm. After one path is completed, the corresponding test input is generated by invoking a theorem prover.

Concolic execution can degrade gracefully after encountering unsolvable situations through approximating symbolic execution by concrete execution. In which cases, symbolic execution has failed [6]. It is difficult to argue which technique is better. Both of them have their advantages and drawbacks [6]. However, the approximation, noted earlier, may weaken the soundness and completeness of concolic execution and may incur path divergences. Imperfect implementation of concolic execution tools may also lead to path divergences.

Copyright © 2015 John Wiley & Sons, Ltd.

In plain words, a path divergence indicates that the tested program exercises an unexpected path. Accordingly, the associated test input will drive the tested program to exhibit unforeseen behaviors, thus weakening the soundness of concolic execution. As a result, testers have to make an effort to filter out the misleading test inputs to ensure the test suite is succinct. According to Godefroid, Levin and Molnar [7], path divergences can also impair the completeness of concolic execution. More precisely, path exploration may go back to previous paths because of path divergences. As a consequence, path divergences may hinder further progress in path exploration.

The formal definition of a path divergence can be given straightforwardly by introducing a graph theory. A tested program can be represented as a graph. Each vertex represents a basic block consisting of a straight-line sequence of code with only one entry point and only one exit. Each edge illustrates a control flow transfer. Concolic execution aims at exploring every feasible path, which starts from a unique entry block and ends at one exit block. Under normal circumstances, the predicted path of concolic execution should be the prefix of the path actually taken. Otherwise, a path divergence has occurred.

Path divergences have been frequently observed in previous works, such as SAGE [7] and CREST [1]. A recent research in concolic execution for Android Apps also demonstrated path divergences [8]. Furthermore, a few path divergences arose while testing our self-developed concolic execution tool, SMAFE [9]. Because of the difficulties involved in determining the causes, path divergences are hard to eliminate. SAGE, deployed by Microsoft in 2008, was unable to resolve path divergence issues until 2013 because of the defects in implementations [10].

An essential step to overcoming path divergences is to first attain an in-depth understanding of it and the problems it poses. However, there are no existing works focusing on path divergences. To fill the gap, this work digs deep into the path-divergence problem by conducting a comprehensive empirical study on it in the context of concolic execution. In our study, the prevalence of path divergences and the proportion of misleading test inputs are calculated, thereby giving a clear picture of the path divergences.

All experiments are operated on one concolic execution tool, CREST [1]. We do not make a similar comprehensive study on other concolic execution tools, mainly. This is mainly because of the extraordinary cost of manual analysis for each divergence. To make our study representative, all test units are randomly selected from 21 different open-source programs. In total, 120 test units are studied.

Several interesting observations appear in the quantitative study. First, almost one-half of test units suffer from path divergences. The figure is fairly high and corresponds to an observation in [7]. Second, quite a few test inputs generated by CREST are misleading. This is a result of test inputs being generated to coerce test units to run divergent paths. For example, over half of the generated test inputs of 16 test units are misleading.

Last, we investigate the divergent pattern of each path divergence manually. There is currently no existing software that can automatically diagnose the root causes of path divergences. Eventually, we have discovered 10 patterns. Among them, eight are observed in tested programs. The other two can be constructed easily; even if we do not regularly witness them in practice. The three most prevalent patterns, which are exceptions, external calls, and type casts, result in nearly 82% of path divergences.

In summary, our work has three contributions:

- We conduct a comprehensive empirical investigation into path divergences. The popularity of path divergences and the percentage of misleading test inputs are calculated.
- We analyze each path divergence manually and finally dig out 10 divergent patterns.
- We discuss the countermeasures for each divergent pattern in detail.

To the best of our knowledge, this work is the first to study path divergences in the context of concolic execution in such detail. We believe our work is a good reference not only for the researchers who want to understand concolic execution in depth but also for the developers who plan to develop more divergence-tolerant concolic execution tools.

The remainder of this paper is organized as follows. Section 2 shows an example of a path divergence. Section 3 briefly reviews related work. Section 4 explains the design of our investigation. Experimental results are shown in Section 5. Section 6 presents two divergent patterns that are not observed in practice. Section 7 presents possible countermeasures for each divergent pattern. Section 8 discusses the limitations of our investigation. Finally, this paper is concluded in Section 9.

2. AN EXAMPLE OF PATH DIVERGENCE

Figure 1 shows a program that will produce a path divergence when tested by a concolic execution tool like CREST. The tested program, *diverge_fun*, accepts only one input of integer type *x*. Function *abs* is an external function that concolic execution cannot reason about. As its name suggests, function *abs* returns the absolute value of its parameter *x*.

```
1:  int diverge_fun(int x) {
2:      abs(x);
3:      if(x >= 2)
4:          return 1;
        else
5:          return 0;
    }
```

Figure 1. An example for a path divergence.

In the first run, the initial value of x is selected randomly, for example, $x = 10$. To symbolize it, the variable x is associated with a symbol 'i0' after entering *diverge_fun*. Because the concolic execution tool is not aware of the internal logic of function *abs*, the symbol of variable x remains 'i0' after the execution of statement 2. As a consequence, when *diverge_fun* exits, a path condition like 'i0 >= 2' is collected. So the executed path in the first run should be <1, 2, 3, 4>. To exercise the other path <1, 2, 3, 5>, the concolic execution tool will generate a new path condition, 'i0 < 2', by flipping the original constraint. So a new test input will be computed, for example $x = -5$.

```
1:   int diverge_fun(int x) {
2:     abs(x);
3:     if(x >= 2)
4:       return 1;
       else
5:       return 0;
   }
```

In the second run, *diverge_fun* will exercise the covered path <1, 2, 3, 4> because of the unhandled function *abs*. So a path divergence occurs because the expected path <1, 2, 3, 5> is not the prefix of actually executed path <1, 2, 3, 4>.

This example validates the claim that path divergences can weaken both the completeness and soundness of concolic execution. To be specific, only one path is explored in this example, making path exploration incomplete. Moreover, the second test input $x = -5$ is misleading that it drives the tested program to run an unexpected path, implying concolic execution is unsound. Furthermore, the second test input is redundant. It reveals the same behavior of the tested program with the first test input. Hence, testers need to eliminate the misleading and redundant test input to keep the test suite concise.

3. RELATED WORK

Concolic execution is advantageous in situations where symbolic execution cannot reason about, by approximation with the aid of concrete execution [6]. In such cases, symbolic execution usually gives up. The drawback of such approximation is that it may weaken the soundness and completeness of concolic execution. Actually, approximation is also one of the root causes of path divergences, as explained in Section 5.4.

Note that symbolic execution tools such as Otter do not suffer from path divergences [11]. Take KLEE as an example, it gives up exploring the paths that cannot be reasoned about symbolically in a precise way. Therefore, KLEE maintains soundness at the price of path exploration capability.

In recent years, path divergences have been observed by many researchers in the field of concolic execution [3,6–13]. But none of them have studied the problem in detail. Godefroid *et al.* observed the rate of path divergences were as high as 60%, indicating nearly two-third of generated test inputs were misleading [7]. They reported a practical path divergence incurred by a bitwise operation. Additionally, they conjectured that other divergent patterns may be non-linear operations, symbolic pointers, and incomplete emulation of x86 instructions. However, they did not verify their conjecture in further experiments. Furthermore, they claimed their path exploration algorithm (i.e., generational search) can tolerate path divergences. In essence, generational search could postpone the exploration of divergent paths by prioritizing non-divergent ones. Therefore, generational search was likely to recover from path divergences and continue its search at the price of completeness. In our opinion, generational search is not a general approach to overcome path divergences. The reason being that path exploration for symbolic execution is an active research topic. Thus, we have little reason to limit ourselves to one path exploration algorithm.

A recent work from Microsoft Corporation stated that they fixed several software bugs that previously incur path divergences for SAGE in 2013 [10]. Furthermore, our previous work witnessed a few path divergences and found two patterns; these were external calls and incomplete symbolization of x86 instructions [9]. Like SAGE, SMAFE stopped exploring the divergent paths and searched for other feasible paths instead. The case study for CutiePy exposed a path divergence without further descriptions [13].

4. DESIGN OF THE EMPIRICAL STUDY

4.1. Objective

Our empirical study aims to answer the following questions.

(1) **RQ1**: How prevalent are the occurrence of path divergences? In other words, we want to know the ratio of the programs that suffer from path divergences to the total programs.
(2) **RQ2**: How frequent are the occurrence of misleading test inputs? Obviously, the more misleading test inputs incur, the more manual effort required to maintain the test suite.
(3) **RQ3**: What are the divergent patterns? This question is of greatest interest. The corresponding answer would facilitate future works to address this problem.

4.2. Studied concolic execution tool

The concolic execution tool investigated in this work is CREST [1], which was developed by UC Berkeley in 2008. We download the source of CREST from the website https://code.google.com/p/crest/. CREST works by inserting instrumentation code using CIL [14] into a target program to perform symbolic execution concurrently with

the concrete execution [15]. C sources of tested programs should be available because of the dependency of CIL.

In this study, we choose CREST because of the following reasons. First, CREST is publicly available and open-source, allowing others to replicate our experiments. Second, CREST is comparatively mature. A number of experiments have been carried on by its developers [1] and other researchers [16]. So we have confidence in its code quality. Third, CREST has good performance in code coverage testing. Recent work showed that the effectiveness of CREST is comparable with KLEE [16]. Obviously, our empirical study makes sense only if the studied concolic execution tool is proved excellent. Finally, CREST is able to detect the occurrence of path divergences automatically, even if it cannot analyze them. This allows us to further concentrate on the diagnosis of path divergences.

4.3. Tested programs

To produce convincible results, we have to test an adequate number of practical programs. Furthermore, we must test programs from different categories to guarantee the impartiality of experimental results. We selected 21 practical programs of different categories, including file compression and decompression, image processing, audio processing, file type cast, date library, data structures and associated algorithms, GNU core utilities, and math library. As we know, we are the first to test so many different categories of programs by concolic execution.

We first attempted to test entire programs directly with CREST. Unfortunately, CREST cannot produce enough usable information because of path explosion, which denotes that the number of feasible execution paths of a program increases exponentially with the increase in the length of an execution path [17]. CREST as well as other state-of-the-art concolic execution tools are not scalable with large software. Therefore, we manually partitioned the programs into smaller test units with manageable sizes. Of the test units, 120 are selected randomly. It is worth pointing out that one test unit is not equal to one self-contained function; instead, it may call other functions. These include functions from the same program, functions belonging to other libraries, or the functions of the operating system. In our experiments, we just counted the lines of code (LOCs) belonging to the test units or the tested programs, without taking the external code into counting. The descriptions of tested programs, test units, and experimental results are shown in Appendix A. To our knowledge, we are the first to test so many test units by concolic execution.

According to Qu and Robinson [16], total test time includes the time to partition (*tp*) and the time to test each unit (*tu*). The latter can be separated into three parts: time to setup environment and tool (*ts*), time to create test driver and specify symbolic sources (*tc*), and the time to run the test unit concolically (*tt*). So the total test time is calculated as $tp + \sum_{i=1}^{N}(ts + tc + tt)$, where N is the number of test units. Because of the partition, the time *tt* may be short, but the time *tp*, *ts*, and *tc* would be considerably long. Recent work reports that the sum of *tp*, *ts* and *tc* takes 99% of total test time [16]. That is why we are unable to conduct similar manual studies on other concolic execution tools. Nevertheless, the divergent patterns that are observed through the study of CREST will provide deep insight into solving the challenging path-divergence problem.

Figure 2 illustrates the sizes of tested programs, where all tested programs are presented in descending order. In Figure 2, we can observe that two programs are very large with the sizes exceeding 10 000 lines of code. More than 76% of programs range from 1000 to 10 000 lines. Only three programs in the study are smaller than 1000 lines.

Figure 3 presents the sizes of test units in descending order, where gray bars indicate the sizes. Two test units are larger than 1000 lines; more than 76% of test units range from 50 to 1000 lines; and the others are very small; for example, the test unit *debug_buffer* has only nine lines of C code. We claim that the sizes of test units are appropriate for three reasons. First, current concolic execution tools cannot analyze much bigger programs thoroughly because of path explosion. Second, the sizes of test units in our study are comparable with those in recent empirical studies [16,18]. Finally, divergent patterns must later be discerned manually; we have to restrict test units to relatively small sizes.

4.4. Metrics

Before presenting our findings, it is prudent to first discuss several concepts and metrics used in the analysis of our study. In the beginning, a Boolean function is defined as follows:

$$O(p_i) = \begin{cases} 1\ , & p_i \text{ exposes path divergences} \\ 0\ , & p_i \text{ does not expose path divergences} \end{cases} \quad (1)$$

In equation 1, p_i indicates the ith test unit, $i \in \{1, 2, \ldots, n\}$, where n denotes the total number of tested units and $O(p_i)$ denotes whether p_i exposes path divergences or not. Based on equation 1, we define the prevalence of path divergences as equation 2. The metric is used to estimate the severity of the problem. For example, $prev = 0$ means no test unit suffers from path divergences, while $prev = 1$ indicates every test unit is subjected to path divergences.

$$prev = \sum_{i=1}^{n} O(p_i)/n \quad (2)$$

We use the terms $ND(p_i)$ and $NT(p_i)$ to represent the number of path divergences of test unit i and the number of total test inputs of test unit i, respectively. So the proportion of misleading test inputs to the total test inputs generated for test unit i can be calculated by equation 3. This metric is used to estimate the human effort required to eliminate misleading test inputs. For example, $mis(p_i) = 0$ means

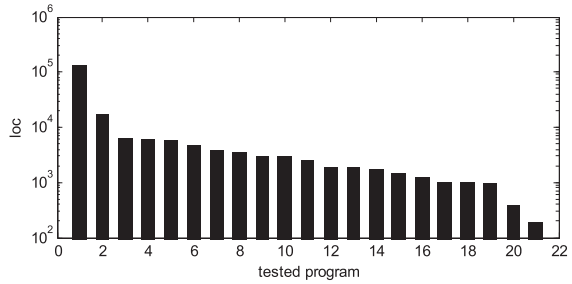

Figure 2. Sizes of tested programs.

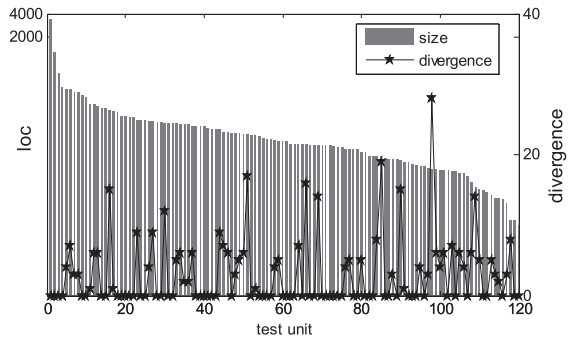

Figure 3. Sizes of test units and number of path divergences.

all test inputs for test unit i are effective; while $mis(p_i) = 1$ indicates all test inputs for test unit i are misleading.

$$mis(p_i) = ND(p_i)/NT(p_i) \tag{3}$$

We then use the term div_{ij} to denote the jth ($j \in \{1, 2, \ldots, ND(p_i)\}$) path divergence when testing unit i. After that, we define another Boolean function in equation 4.

$$C_k(div_{ij}) = \begin{cases} 1\ , & div_{ij} \text{ corresponds to pattern } k \\ 0\ , & div_{ij} \text{ does not correspond to pattern } k \end{cases} \tag{4}$$

$C_k(div_{ij})$ indicates whether the path divergence div_{ij} corresponds to the kth divergent pattern or not, $k \in \{1, 2, \ldots, m\}$, where m indicates the total number of divergent patterns. Hence, the ratio of each divergent patterns can be computed by equation 5. The metric is used to estimate the prevalence of each divergent pattern. For example, $R_k = 0$ means no such divergent pattern is observed, while $R_k = 1$ denotes all path divergences correspond to such pattern.

$$R_k = \sum_{i=1}^{n} \sum_{j=1}^{ND(p_i)} C_k(div_{ij}) / \sum_{i=1}^{n} ND(p_i) \tag{5}$$

5. RESULTS

5.1. Experiment setup

All experiments were conducted on a laptop of one Intel Core i7-2760QM CPU (Lenovo Corp., Guangdong, China) and 8 GB main memory. The operating system is Ubuntu 12.04.

The procedure of our experiments is as follows:

(1) We choose a practical program for testing.
(2) We partition the tested program into many test units with proper sizes manually.
(3) The parameters of CREST are set. For example, a path exploration algorithm is fixed as depth-first. Which path exploration algorithm is best is irrelevant because this work does not focus on the discovery of path divergences. We choose depth-first based on the intuition that long paths are more likely to trigger path divergences. The maximum number of paths needed to explore is set as 1000. It is necessary to terminate concolic execution forcibly because of limited testing resources, even if the test unit has more paths available to explore.
(4) We randomly select one test unit.
(5) We write a test driver for the test unit. The main function of each test driver is to mark the sources of symbols and to run the test unit using CREST. According to the internal logic of test units, data read from files, function parameters, and so on are treated as symbolic sources. To do so, we invoke CREST's built-in routines *CREST_int* and *CREST_char*.
(6) We monitor and capture any occurring path divergences.
(7) If any path divergences occur, we record related information and analyze the divergent patterns manually.
(8) Return to step 4 and repeat the process until all 120 test units are completed.

5.2. Prevalence of path divergences

CREST is able to catch sight of path divergences by its built-in routine *CheckPrediction*. *CheckPrediction* first checks whether the length of original path condition or the length of new path condition is no larger than *branch_idx*, where *branch_idx* denotes the index of the flipped constraint. If the answer is *yes*, *CheckPrediction* reports a path divergence. Otherwise, the first *branch_idx* branches of original path condition are compared with those of the new path condition. If the two are unequal, *CheckPrediction* reports a path divergence. Otherwise, *CheckPrediction* returns that there is no path divergence. *CheckPrediction* works strictly according to the formal definition of path divergence in Section 1.

However, CREST does not know why path divergences occur. Divergent patterns must therefore be analyzed manually. In the experiments, we found that 56 test units suffered from path divergences. So our answer to the first question concerning the prevalence of path divergences is $prev = 56/120 = 47\%$, according to equation 2. The number of path divergences related to test units is depicted in Figure 3 (the line with starlike markers).

The result corresponds to the observation in [7] that path divergences occur frequently. But the prevalence reported in [7] (over 60%) is higher than ours. This is

reasonable because our study differs from that in [7] in many aspects. For example, we study different concolic execution tools. Additionally, we test different programs. Furthermore, the studied concolic execution tools run in different environments.

We assumed that larger test units would expose more path divergences; however, the experimental results do not support the assumption. As shown in Figure 3, the number of path divergences is irrelevant to the sizes of test units. The assumption is based on the following intuition. Bigger programs may have more divergent patterns, such as external calls, type casts, floating-point arithmetic, and non-linear computations (Section 5.4). However, experiments show that big test units also prevent CREST to touch those divergent patterns in limited testing time because of the large number of feasible paths. As evidence, the test unit *json_parse_document* does not expose any path divergences during the test period, while the test unit *json_format_string*, which is invoked by *json_parse_document*, exposes 9. Analysis found that *json_format_string* was not being reached when testing *json_parse_document*, even if *json_parse_document* contains its code body.

5.3. Ratio of misleading test inputs

As mentioned earlier, 64 test units do not expose path divergences, so the ratio of misleading test inputs for those test units should be 0 according to equation 3. Figure 4 presents the ratio of misleading test inputs in descending order of the ratio for those 56 programs that suffer from path divergences.

We have several observations. First, all test inputs for two test units (about 3.6%) are misleading, which indicates each input will lead to a path divergence. Second, more than 50% of test inputs for 16 programs (nearly 29%) are misleading, which denotes more than half of their test inputs will drive test units to run divergent paths. Third, less than 5% of test inputs for 18 programs (about 32%) are misleading. In conclusion, our testers have spent too much effort to eliminate misleading test inputs because of the non-negligible proportion.

5.4. Divergent patterns

We spent much effort in digging out divergent patterns because the outcome would contain crucial information to address the issue. As no automatic method is available to fulfill the task, we have to check each path divergence by hand. After analyzing hundreds of path divergences, we have finally arrived at eight different divergence patterns. These are exceptions, external calls, type casts, symbolic pointers, floating-point arithmetic, 64-bit operations, non-linear computations, and bitwise operations. We do not consider the imperfect implementation of CREST as a single pattern; it is too coarse. For example, the patterns of type casts, 64-bit operations, and bitwise operations can be considered as imperfect implementations.

Figure 5 shows the ratio of each divergent pattern (R_k), which is calculated by equation 5. One observation is that the three most prevalent patterns lead to 82% of total path divergences. This information is handy for developers with limited development budgets. That is to say, we suggest developers to handle the most prevalent divergent patterns with high priority. In the following, we further explain each divergent pattern with an accompanying practical case.

(1) **Exceptions**. Exceptions account for 36% of total path divergences. When an exception occurs, execution will be directed to related exception handling routine. So the path actually taken will differ from the expected one. Typical exceptions include assertions, out of bound of array, and dividing zero. The following code snippet presents a practical case of an out of array bound exception. The code snippet is in the test unit *coloncpy*, which belongs to the program *8conv* [19].

```
1: do{
2:   *dst++ = ((ch = *src++) == ':') ? sep : ch;
3: } while (ch != '\0');
```

src points to an array of characters, with each element being a symbol. So after statement 2 executes, the character *ch* should be symbolized. CREST tries to explore more paths by flipping the constraint generated in statement 3. As a result, an out of array bound exception will occur if the loop iterates too many times.

(2) **External calls**. External calls account for 28% of total path divergences because CREST cannot track symbol propagation in external calls. Recall that CREST symbolically executes by instrumenting the C source code of test units, so it is unaware of the

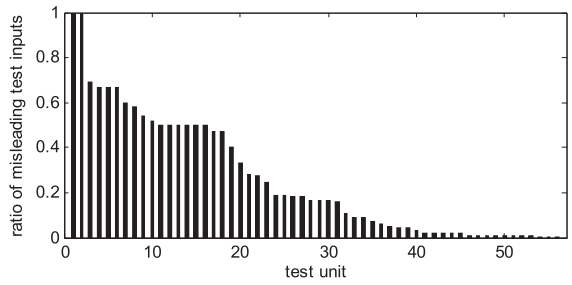

Figure 4. Proportion of misleading test inputs.

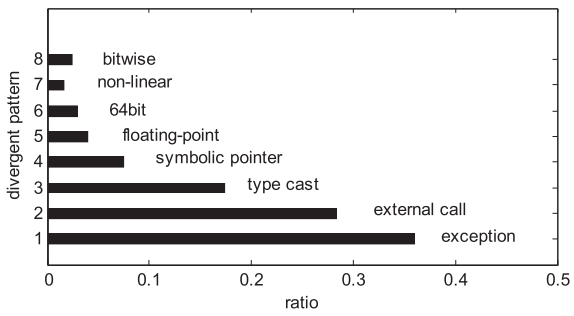

Figure 5. Ratio of each divergent pattern.

Security Comm. Networks 2015; **8**:3667–3681 © 2015 John Wiley & Sons, Ltd.
DOI: 10.1002/sec

execution in external code. The pattern is so frequent because practical programs are not self-contained. Instead, they interact with an external code frequently to get support from C run-time libraries, any third-party libraries, kernel code, device drivers, and so on. The following is a code snippet in the test unit *ini_parse*, which belongs to the program *inih* [20].

```
1:  char* p = s + strlen(s);
2:  while (p > s && isspace((unsigned char)(*--p)))
3:    *p = '\0';
```

The *s* points to an array of characters, with each element being a symbol, and *strlen* is an external call for CREST. So no constraint can be produced on the branch 'p > s'. But actually the outcome of this branch is influenced by *s* because of the external call *strlen*. So in subsequent runs, path divergences appear right on the branch 'p > s'.

(3) **Type casts**. Type casts are extremely common in the C language. For example, a char variable may be used as a 32-bit integer or a 16-bit word. Type casts lead to 17% of total path divergences because CREST does not handle them properly. A practical case is in the test unit *Rice_Compress*, which belongs to the program *Basic Compression Library* [21] as shown in the following.

```
1:  sx = ((int *) ptr)[idx];
2:  x = sx < 0 ? -1-(sx<<1) : sx<<1;
```

Tne *ptr* points to an array of characters with each item being a symbol. In statement 1, the pointer *ptr* is interpreted as a pointer to an integer array. But CREST handles this case in an incorrect way. For example, assume the char variable '((char*)ptr)[idx]' has already been symbolized as 'x0'. But after the execution of statement 1, the integer variable *sx* is associated with a wrong symbol 'x0' by CREST. As a result, the constraint generated from the predicate 'sx < 0?' in statement 2 is incorrect.

(4) **Symbolic pointers**. Symbolic pointers denote the case that pointers themselves are symbols. Symbolic pointers (account for 7%) can lead to path divergences because CREST cannot track symbol propagation from pointers to the pointed variables. We present a case in the test unit *RLE_Compress*, which belongs to the program *Basic Compression Library* [21].

```
1:  for(i = 0; i < insize; ++ i)
2:    ++ histogram[in[i]];
3:  for(i = 1; i < 256; ++ i)
4:    if(histogram[i] < histogram[marker])
5:      marker = i;
```

The *in* points to an array of integers, with each element being a symbol. So *in*[*i*], which acts as an index for the array *histogram*, is a symbol. However, CREST cannot track symbols that flow from *in*[*i*] to *histogram*[*in*[*i*]]. Therefore, after execution of the first loop, the items in array *histogram* are not symbolized. In reality, the elements in array *histogram* depend on inputs, so they should be treated as symbols. As a consequence, no constraints can be generated in statement 4. So in subsequent runs, we found that path divergences on the branch of statement 4.

(5) **Floating-point arithmetic**. CREST cannot execute floating-point operations symbolically because its theorem prover Yices [22] cannot reason about floating-point arithmetic. About 4% of path divergences correspond to the pattern. In the following, we present a practical case in the test unit *FLAC__lpc_quantize_coefficients*, which belongs to the program FLAC [23].

```
1:  double rprecision = (double)precision;
… …
2:  else if ((rprecision-1.0))-maxlog>=precision+1))
```

The *precision* is an integer, which is already a symbol. Because CREST cannot handle floating-point symbolically, the variable *rprecision* will not be symbolized after the execution of statement 1. But as we can see, *rprecision* depends on *precision*, so it should be treated as symbol. Hence, the constraint produced on the branch of statement 2 is incorrect, leading to path divergences.

(6) **64-bit operations**. As we know, all concolic execution tools, including CREST, cannot handle 64-bit operations symbolically, so we find some path divergences (account for 3%) because of the reason. In the following is the practical case we find in the test unit *FLAC__lpc_compute_residual_from_qlp_coefficients*, which belongs to the program FLAC [23].

```
1:  sumo += qlp_coeff[j] * (*(--history));
2:  if(… sumo < -2147483648ll)
```

The *qlp_coeff* is an integer array, with each element being a symbol. So after statement 1 executes, the variable *sumo* is also a symbol. Because the constant −2147483648ll is a 64-bit integer, CREST produces an incorrect constraint on the corresponding branch in statement 2. As a result, the test unit runs unpredicted paths in subsequent executions.

(7) **Non-linear computations**. CREST cannot handle non-linear computations properly because its theorem prover Yices [22] is not capable of reasoning about non-linear arithmetic. From the perspective of implementation, CREST formulates each constraint as a linear expression. If a non-linear computation is encountered, CREST will approximate the non-linear computation through a linear expression or a concrete value by concretizing some symbols. In those cases, concolic execution degrades gracefully with the sacrifice of precision. We found a number of path divergences (approximately 2%) correspond to the pattern. Consider the following

practical case in the test unit *hdate_hdate*, which belongs to the program *Libhdate* [24].

```
1:  l = y*7+1;
2:  l % = 19;
… …
3:  if (l < 12… …)
```

The variable *y* is an integer that is already associated with a symbol. So after statement 1 executes, the variable *l* is symbolized as a linear expression. As the '%' operation is non-linear, CREST concretizes the symbol of *l* with its run-time value. So when statement 2 is executed, the symbol of *l* should be discarded, indicating the variable *l* is no longer a symbol. But this was not the case. We found that CREST could not generate constraints in statement 3; this leads to path divergences in subsequent runs.

(8) **Bitwise operations**. Bitwise operations are quite common in C programs. Unfortunately, CREST does not handle them because CREST models all operations in *int* theory. Recall that CREST formulates all symbolic expressions into linear computations. But it is unable to formulate bitwise operations in this way. Finally, CREST falls back to concrete values when it encounters bitwise operations. The divergent pattern accounts for about 2% of total path divergences. In the following is a practical case in the test unit *FLAC__lpc_restore_signal*, which belongs to the program FLAC [23].

```
1:  for(i = 0; i < data_len; i++) {
… …
2:    history = data + i;
3:    for(j = 0; j < order; j++) {
… …
4:      sumo += qlp_coeff[j] * (*history);
5:      if(sumo > 21474836471l … …)
… …
      }
6:    data[i] = residual[i] + sum >> lp_quantization;
  }
```

The *data* points to an integer array, with each item being a symbol. So after statement 2, *history* points to one of the items of *data*. Then after statement 4, the variable *sumo* is symbolized. So CREST produces a constraint for statement 5. In statement 6, the variable *lp_quantization* is a symbolized integer. However, CREST cannot track symbol propagation in this statement correctly because of the bitwise operation '>>'. As a result, in subsequent iterations, CREST will generate incorrect constraints in statement 5, resulting in path divergences.

6. MINOR DIVERGENT PATTERNS

In this section, we identify two minor divergent patterns. These patterns did not appear in any of the practical programs tested. However, they remain a possible obstacle for programs outside of our test set. Hence, we label these patterns as minor.

(1) **Pointer operations**. Pointer operations are common and often very complex in C programs. However, we found that CREST could not handle pointer operations properly. So path divergences can result from the pointer operations as shown in the following sample code.

```
1:  int b = a;
2:  char *p = &a;
3:  *p = 0;
4:  if (b < a - 5)
```

The variable *a* is already a symbol of integer type. After statement 1 executes, the variable *b* is symbolized as the same symbol with variable *a*. Statement 2 defines a pointer *p* of char type, which points to the lowest byte of variable *a*. After statement 3, the lowest byte of *a* is assigned as a constant 0.
We found that CREST incorrectly discards the symbol of variable *a*. As a result, after the execution of statement 4, an incorrect constraint like 'x0 + 5 >= 0' is generated (assume the initial concrete value and the symbol of *a* are 10 and 'x0' respectively). CREST produces a new test input, for example 'x0 = −6', by solving the new path condition 'x0 + 5 < 0', in order to cover the *then* branch of statement 4. But obviously, the new test input will drive the program to run the *else* branch again.

(2) **Inline assembly code**. Programmers at times will embed inline assembly code in their C programs. Unfortunately, we found that inline assembly code could lead to path divergences because CIL does not handle inline assembly code correctly, as shown in the following sample code.

```
1:  asm ("movl %1, %%eax;"
         "movl %%eax, %0;" :"=r"(b) :"r"(a) :"%eax");
2:  if(a > 2 * b + 10)
```

The variable *a* is already a symbol of integer type. The assembly code block is actually equal to the C statement 'b = a'. However, CREST cannot track symbol propagation through the inline assembly code. Thus, CREST generates an incorrect constraint in statement 2, which will lead to a path divergence in the next execution.

In conclusion, divergent patterns like bitwise operations, non-linear operations, external calls, and incomplete symbolization of x86 instructions are observed in previous works. However, the divergent pattern, an incomplete symbolization of x86 instructions, is too coarse to be useful, so it is not considered as a divergent pattern in this study. The other divergent patterns are observed exclusively in our work. All divergent patterns are general, even though different concolic executors suffer from different kinds of divergent patterns because of their different techniques adopted. That is to say, no divergent patterns are specific to CREST because none of them are incurred by specific implementation flaws.

7. COUNTERMEASURES FOR PATH DIVERGENCES

The empirical study aims to assist researchers not only to understand the challenge of path divergences in depth qualitatively and quantitatively but also to solve the problem. To improve the soundness and completeness of concolic execution, this section discusses possible countermeasures to overcome path divergences.

(1) **Exceptions**. We consider the exceptions revealed in concolic execution as a bonus rather than an annoyance. A major goal of software testing is to uncover software bugs. We need to note that an exception does not necessarily indicate a bug. Some exceptions are because of improper settings of test environments and test drivers. So subsequent bug diagnosis is indispensable. To facilitate bug diagnosis, concolic execution tools would better to log all the information needed, such as call stacks, memory states, and register states. Besides, the exception, the divergent branch, and the path to the divergent branch should be recorded to avoid repeated snapshots of the same exception.

(2) **External calls**. It is too expensive (if it is not impossible) to execute all statements concolically. To improve efficiency, S2E [25] proposes a selective symbolic execution that executes the interested code (e.g., the code belonging to the tested program or in the environment) symbolically while executing other code concretely. To do so, S2E runs on a supervisor level, which is lower than tested programs and runtime environments. However, concolic execution tools like CREST cannot employ selective symbolic execution because they can handle the source code of tested programs only. Function summary [26] may be another approach. With pre-computed function summaries, the code in an environment can be executed symbolically. Consequently, the soundness of concolic execution will be improved. However, the major drawback of this method is that function summaries are usually computed manually. This is labor-intensive and error-prone [27]. So in practice, not all the code in environment can be abstracted as summaries. More severely, function summaries exacerbate the solving of path conditions. Besides, how to manage summaries effectively with constantly evolving test programs is a challenge [28].

Function models are another alternative for handling external calls. With pre-computed function models, symbolic execution tools such as KLEE can track symbol propagation in external code. When external calls are invoked, the calls will be redirected to corresponding function models. These function models understand the semantics of the desired action well enough to generate required constraints [29]. However, manual computation of function models is expensive. This is especially true when we model the whole environment. Besides, the whole system is hardly expected to run symbolically because the paths in the environment usually far outnumber the paths in the tested program. Actually, KLEE executes external functions symbolically or concretely, depending on whether the functions are modeled or not [25].

(3) **Type casts**. The key to solve the divergent pattern is to understand the semantics of executed statements precisely. To be specific, when one statement is executed symbolically, the concolic tool should understand which and how variables are influenced precisely. This divergent pattern is usually associated with implementation flaws.

(4) **Symbolic pointer computations**. One way to address this challenge is proposed by Elkarablieh *et al.* [30]. New constraints, which represent the snapshot of the memory region associated with memory dereferences and the bound of symbolic addresses, are generated to handle symbolic pointer operations [30]. However, finding the memory region for each memory dereference is problematic, especially in binary code.

An alternative method is to maintain a list of alias relations, which can be considered as delayed constraints [31]. The delayed constraints are maintained dynamically in a stack-like data structure. A delayed constraint on top of the stack is removed, and an assignment for an index (i.e., pointer) is generated. If the constraint theorem prover fails to generate a test input, the proposed method backtracks to try other possible indices. Only after trying all possible indices unsuccessfully with a test input generation, the proposed method goes to explore other paths. As stated in [32], one drawback of this method is that more constraints will be added, thus, pressure on the theorem prover will be increased.

(5) **Floating-point arithmetic**. The reason for the challenge of floating-point computations is that it does not exist a theorem prover, which is able to reason about floating-point operations perfectly. As stated in [33], it is extremely difficult—if not infeasible—to build a constraint theorem prover for a floating point. Excellent theorem provers such as Z3 [34], which can reason about real numbers, are inadequate at handling floating-point numbers. That is because many common properties of arithmetic over the reals fail over the floating-point numbers [35]. For instance, addition and multiplication over the floating-point numbers are neither associative nor distributive [35].

The FPSE [36] and its following work [37] can reason about floating-point arithmetic. They are based on interval propagation and projection functions. A limitation of those theorem provers is that they cannot handle the combinations of floating-point computations with the other operations. But in practice, combinations are quite common. FloPSy [38]

does not suffer from the drawback because it is based on search strategies, but it has three shortcomings. First, it is sensible to the rounding-errors incurred in the computation of the branch distance [39]. Second, search-based methods cannot be used to study path feasibility [31]. Last, existing search algorithms are not adequate to solve the path conditions generated from practical programs: local search can be stuck in local minima without being able to find a solution [31]; the alternative, global search is too expensive [40].

(6) **64-bit operations**. We have not found any concolic execution tools that are able to test 64-bit operations. But we believe it is an engineering problem rather than an academic challenge. So we expect this divergent pattern will be addressed by extending current tools with the ability of reasoning about 64-bit operations.

(7) **Non-linear computations**. The decision problem for non-linear arithmetic is generally undecidable. So existing theorem provers cannot fully address non-linear computations. In some cases, theorem provers such as Z3 [34] can reason about non-linear computations. However, those theorem provers cannot ensure to find a solution, even if the solution actually exists. Search-based strategies can address non-linear arithmetic to an extent. However, these methods have their own drawbacks (as mentioned earlier),

(8) **Bitwise operations**. We believe it is not difficult to symbolically execute bitwise operations. First, we need a theorem prover that supports *bitvector* theory. Fortunately, most modern theorem provers are adequate. Second, tested programs should be interpreted in *bitvector* theory rather than *int* theory (like CREST). Third, the semantics of bitwise operations should be interpreted precisely.

(9) **Pointer operations**. Through previous experiments, we also found Fuzzgrind and PEX could not handle pointer operations properly. One method proposed in SAGE [7] addresses pointer operations with subtag and sequence tags. This proposed method is sound but expensive in memory consumption [9].
Another method is to replace sub-expressions with new symbols to reduce overall size [41]. This method will introduce many additional symbols, which would exacerbate constraint solving. SMAFE [9] proposes several heuristics that delete a number of symbols to save memory and dynamically assemble them from other symbols when needed. Experiments have shown its efficiency in memory cost and accuracy in collecting path conditions.

(10) **Inline assembly code**. Binary-level concolic execution tools, such as Fuzzgrind and SMAFE, do not have this divergent pattern. Source-level tools such as CREST are susceptible to inline assembly code because its instrumentation tool CIL [14] cannot handle inline assembly code properly. So to address the problem of CREST, we need to extend the current CIL or replace it with adequate instrumentation tools.

Table I. Divergent patterns and countermeasures.

Patterns	Solved?	Countermeasures
Exceptions	Yes	Human diagnosis
External calls	Partial	Selective symbolic execution [25], function summaries [26], function models [29],
Type casts	Yes	Accurate symbolic interpretation
Symbolic pointer	Partial	Elkarablieh [30], alias analysis [31]
Floating-point	Partial	Interval propagation [36], search-based [38]
64bit	Yes	Implementation extension
Non-linear	Partial	Search-based [38]
Bitwise	Yes	Accurate symbolic interpretation and *bitvector* theory support
Pointer	Yes	Subtag and sequence tags [7], replace with new symbols [41], heuristics [9]
Inline assembly code	Yes	Binary-level, better instrumentation tools

To conclude this section, Table I lists each divergent pattern and its countermeasures. Column 2 presents whether the divergent pattern can be solved using current techniques, but it does not imply the divergent pattern has already been eliminated. For example, we believe 64-bit operations can be solved without academic innovation, but no existing concolic execution tools handle 64-bit operations properly. It is worth pointing out that all countermeasures are new to battle against path divergences, even though some of them such as function summaries and selective symbolic execution have been proposed to resolve other issues.

8. LIMITATIONS OF OUR INVESTIGATION

Although we have tested a number of test units that are extracted from many practical programs of several major categories, we do not claim to have found all divergent patterns. Actually, path divergences depend on the properties of tested programs. For example, Java programs where there are no pointers at all will not introduce the divergent pattern of pointer operations. Additionally, binary programs, which analyzed by binary-level concolic execution tools, are not susceptible to inline assembly code. Furthermore, although we chose as many as 120 test units from different program categories, we cannot guarantee that there was no bias. We naturally prefer to select sequential and non-user-intensive programs because (like most existing concolic execution tools) CREST was not designed to test concurrent or user-intensive programs.

DOI: 10.1002/sec

The different techniques applied by concolic execution tools can also affect experimental results. For example, concolic execution tools such as SMAFE [9] and Fuzzgrind [42] can handle bitwise operations properly, so they will not be exposed to this divergent pattern. Additionally, PEX [43] would also be exposed to fewer path divergences incurred by non-linear computations than CREST. This is because Z3 (the theorem prover of PEX) performs better than Yices in solving non-linear computations.

9. CONCLUSION

In this work, we studied path divergences in concolic execution qualitatively and quantitatively using CREST. As many as 120 test units from 21 different open-source programs are investigated. We first computed the prevalence of path divergences and then calculated the proportion of misleading test inputs. The analysis for divergent patterns is our major contribution. Eventually, we digged out ten patterns, where eight were observed in practice. The three most prevalent patterns found were exceptions, extern calls, and type casts. These lead to almost 82% of total path divergences. Another contribution is the detailed discussion of the countermeasures to overcome path divergences. We believe our work will benefit future works in addressing path divergences.

Through this work, we find it is expensive and time-consuming to diagnose divergent patterns manually. Hence, we plan to design an automatic diagnosis framework for path divergences in our future works.

ACKNOWLEDGEMENTS

This work was supported in part by the National Natural Science Foundation of China, No.61402080 and China Postdoctoral Science Foundation funded project, No.2014M562307.

REFERENCES

1. King JC. Symbolic execution and program testing. *Journal of the ACM* 1976; **19**(7):385–394.
2. Myers GJ. *The Art of Software Testing*. Wiley: New York, USA, 1979.
3. Burnim J, Sen K. Heuristics for scalable dynamic test generation. In Proceedings of 23rd IEEE/ACM International Conference on Automated Software Engineering, 443-446, 2008.
4. Chen T, Zhang XS, Guo SZ, Li HY, Wu Y. State of the art: dynamic symbolic execution for automated test generation. *Future Generation Computer Systems* 2013; **29**(7):1758–1773.
5. Godefroid P, Klarlund N, Sen K. DART: directed automated random testing. In Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2013213-223, 2005.
6. Păsăreanu CS, Rungta N, Visser W. Symbolic execution with mixed concrete-symbolic solving. In Proceedings of the International Symposium on Software Testing and Analysis, 34-44, 2011.
7. Godefroid P, Levin M, Molnar D. Automated whitebox fuzz testing. In Proceedings of the Network and Distributed System Security Symposium. (2008)
8. Anand S, Naik M, Harrold MJ, Yang H. Automated concolic testing of smartphone apps. In Proceedings of the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering, 59–73. (2012)
9. Chen T, Zhang XS, Zhu C, Ji XL, Guo SZ, Wu Y. Design and implementation of a dynamic symbolic execution tool for windows executables. *Journal of Software: Evolution and Process* 2013; **25**(12):1249–1272.
10. Christakis M, Patrice G. Proving memory safety of the ANI windows image parser using compositional exhaustive testing. Technical Report, MSR-TR-2013-120, Microsoft Research. (2013)
11. Ma KK, Phang KY, Foster JS, Hicks M. Directed symbolic execution. In Proceedings of 18th International Symposium on Static Analysis, 95–111. (2011)
12. Davies M, Păsăreanu CS, Raman V. Symbolic execution enhanced system testing. Verified software: theories, tools, experiments, 294–309. (2012)
13. Sapra S, Minea M, Chaki S, Gurfinkel A, Clarke EM. Finding errors in Python programs using dynamic symbolic execution. Testing Software and Systems, 283–289. (2013)
14. Necula GC, McPeak S, Rahul SP, Weimer W. CIL: intermediate language and tools for analysis and transformation of C programs. In Proceedings of Conference on compiler Construction, 213-228. (2002)
15. CREST: automated test generation tool for C. http://jburnim.github.io/crest/.
16. Qu X, Robinson B. A case study of concolic testing tools and their limitations. In Proceedings of the International Symposium on Empirical Software Engineering and Measurement, 117–126. (2011)
17. Joshi P, Sen K, Shlimovich M. Predictive testing: amplifying the effectiveness of software testing. In Proceeding of the 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering, 561–564. (2007)
18. Lakhotia K, McMinn P, Harman M. An empirical investigation into branch coverage for C programs using CUTE and AUSTIN. *Journal of Systems and Software* 2010; **83**(12):2379–2391.
19. 8conv: converts quoted-printable, UTF-8, UTF-16BE, UTF16LE to 8-bit. http://eightconv.sourceforge.net/.

20. inih: Simple .INI file parser in C, good for embedded systems. https://code.google.com/p/inih/.
21. Basic Compression Library. http://bcl.comli.eu/.
22. Yices: An SMT Solver. http://yices.csl.sri.com/
23. FLAC: Free lossless audio codec. http://xiph.org/flac/.
24. libhdate, hcal and hdate: C library for Hebrew dates / times of day / solar times. http://libhdate.sourceforge.net/
25. Chipounov V, Kuznetsov V, Candea G. S2E: a platform for in-vivo multi-path analysis of software systems. *Computer Architecture News* 2011; **39**(1): 265–278.
26. Godefroid P. Compositional dynamic test generation. In Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 47–54. (2007)
27. Schwartz EJ, Avgerinos T, Brumley D. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In Proceedings of IEEE Symposium on Security and Privacy, 317–331. (2010)
28. Godefroid P, Lahiri SK, Gonz´alez CR. Incremental compositional dynamic test generation. Technique report MSR-TR-2010-11. (2010)
29. Cadar C, Dunbar D, Engler D. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In Proceedings of the USENIX Symposium on Operating System Design and Implementation. (2008)
30. Elkarablieh B, Godefroid P, Levin MY. Precise pointer reasoning for dynamic test generation. In Proceedings of International Conference on Software Testing and Analysis, 129–140. (2009)
31. Kosmatov N. All-paths test generation for programs with internal aliases. In Proceedings of IEEE International Symposium on Software Reliability Engineering, 147–156. (2008)
32. Papadakis M, Malevris N. Mutation based test case generation via a path exploration strategy. *Information and Software Technology* 2012; **54**(9):915–932.
33. Collingbourne P, Cadar C, Kelly PHJ. Symbolic crosschecking of floating-point and SIMD code. In Proceedings of the EuroSys, 315–328. (2011)
34. Leonardo DM, Nikolaj B. Z3: an efficient SMT solver. *Lecture Notes in Computer Science* 2008; **4963**: 337–340.
35. Blanc B, Bouquet F, Gotlieb A, Jeannet B, Jeron T, Legeard B, Marre B, Michel C, Rueher M. The V3F project. In Proceedings of the 1st Workshop on Constraints in Software Testing, Verification and Analysis, 2006
36. Botella B, Gotlieb A, Michel C. Symbolic execution of floating-point computations. *Software Testing, Verification and Reliability* 2006; **16**(2):97–121.
37. Bagnara R, Carlier M, Gori R, Gotlieb A. Symbolic path-oriented test data generation for floating-point programs. In Proceedings of the 6th IEEE International Conference on Software Testing, Verification and Validation. (2013)
38. Lakhotia K, Tillmann N, Harman M, de Halleux J. FloPSy - search-based floating point constraint solving for symbolic execution. In Proceedings of 22nd IFIP WG6.1 International Conference on Testing Software and Systems, 142–157. (2010)
39. Arcuri A. Theoretical analysis of local search in software testing. In Proceedings of the 5th International Symposium on Stochastic Algorithms: Foundations and Applications, 156–168. (2009)
40. Harman M, McMinn P. A theoretical and empirical study of search-based testing: Local, global, and hybrid search., IEEE Transactions on Software Engineering, Vol. 36, No.2, 226–247. (2010)
41. Brumley D, Hartwig C, Kang MG, Liang ZK, Newsome J, Poosankam P, Song D. Bitscope: automatically dissecting malicious binaries. Technical Report, CS-07-133. (2007)
42. Fuzzgrind: an automatic fuzzing tool. Fuzzgrind. http://seclab.sogeti.com/dotclear/index.php?pages/Fuzzgrind.
43. Tillmann N, De Halleux J. Pex-white box test generation for .NET. In Proceedings of 2nd International Conference on Tests and Proofs, 134–153. (2008)

APPENDIX
Descriptions of test units, tested programs and experimental results

Number	Name	Description Divergence	Loc
1	Buffer compression library	memory compression/decompression	192
1	debug_buffer	0 0 0 0 0 0 0 0 0	9
2	compress	0 0 0 0 0 0 0 0 0	18
3	decompress	0 0 0 0 0 0 0 0 0	19
4	rle_compress	0 0 0 0 0 0 0 0 0	21
5	rle_decompress	9 0 0 0 0 0 0 0 0	17
2	Inih v. r27	ini file parser	390
6	ini_parse	0 17 0 0 0 0 0 0	390
3	Libhdate v.0.1	date library	1018
7	hdate_hdate	0 0 0 0 0 0 0 3 0	101
8	hdate_gdate	0 0 0 0 0 0 0 0 0	85
9	hdate_get_day_name	0 0 0 0 0 0 0 0 0	93
10	hdate_int_to_hebrew	0 0 0 0 0 0 0 0 0	86
11	hdate_get_hebrew_month_name	0 0 0 0 0 0 0 0 0	88
12	hdate_get_holyday	0 0 0 0 0 0 0 0 0	148
13	hdate_get_holyday_type	0 0 0 0 0 0 0 0 0	110
4	Littleutils v. 1.0.28	command-line utilities	6540
14	first_marker	0 7 0 0 0 0 0 0	18
15	next_marker	0 0 0 0 0 0 0 0	40
16	skip_variable	2 2 0 0 0 0 0 0	33
17	process_COM	0 16 12 0 0 0 0 0	54
18	bmp	0 0 0 0 0 0 0 0	78
19	tiff_be	0 0 5 0 0 0 0 0	121
20	xpm	0 0 0 0 0 0 0 0	68
21	tiff_le	0 0 0 0 0 0 0 0	121
22	png	0 0 0 0 0 0 0 0	84
23	jpeg	0 0 0 0 0 0 0 3	154
5	Coreutils v. 8.21	utilities of GNU operating system	136825
24	base64_encode	0 0 0 0 0 0 0 0	42
25	decode_4	5 0 0 0 0 0 0 2	156
26	dupfd	2 0 4 0 0 0 0 0	228
27	filemodestring	0 0 0 0 0 0 0 0	93
28	hash_pjw	0 0 0 0 0 0 1 0	41
29	md5_stream	0 0 0 0 0 0 0 0	447
6	8conv v. 1.0b	file decoder	975
30	octesc	0 0 0 0 0 0 0 0	23
31	hexval	0 0 0 0 0 0 0 0	9
32	esccpy	0 0 0 0 0 0 0 0	73
33	tr_newspec	0 14 0 0 0 0 0 0	108
34	t8_trans	0 0 0 0 0 0 0 0	24
35	coloncpy	15 0 0 0 0 0 0 0	8
7	FLAC - free lossless audio codec v. 0.3	Audio codec	5871
36	FLAC_bitbuffer_write_raw_uint32	0 0 0 0 0 0 0 0	115
37	FLAC_bitbuffer_read_raw_uint32	7 0 0 0 0 0 0 0	94
38	FLAC_bitbuffer_dump	2 0 0 4 0 0 0 0	37
39	FLAC__bitbuffer_write_zeroes	0 0 1 0 0 0 0 0	82
40	FLAC__bitbuffer_write_raw_uint64	2 0 0 0 0 0 0 0	124

(Continues)

Appendix (Continued)

Number	Name	Description Divergence	Loc
41	FLAC__bitbuffer_write_utf8_uint32	0 0 0 0 0 0 0 0	163
42	FLAC__bitbuffer_write_utf8_uint64	0 0 0 0 0 0 0 0	172
43	FLAC__fixed_compute_best_predictor	6 0 0 0 0 0 0 0	51
44	FLAC__fixed_compute_residual	2 0 0 0 0 0 0 0	40
45	FLAC__fixed_restore_signal	0 0 0 0 0 0 0 0	40
46	FLAC__lpc_compute_autocorrelation	16 0 0 0 0 0 0 0	22
47	FLAC__lpc_compute_lp_coefficients	0 0 0 0 0 0 0 0	38
48	FLAC__lpc_quantize_coefficients	0 0 0 0 15 0 0 0	46
49	FLAC__lpc_compute_residual_from_qlp_coefficients	0 0 0 0 0 8 0 0	44
50	FLAC__lpc_restore_signal	0 0 0 0 0 3 0 2	43
8	Basic compression library v. 1.2.0	compression / decompression	1902
51	RLE_Compress	0 0 0 14 0 0 0 0	133
52	RLE_Uncompress	6 0 0 0 0 0 0 0	38
53	LZ_Compress	0 0 0 5 0 0 0 0	117
54	LZ_CompressFast	0 0 0 5 0 0 0 0	136
55	LZ_Uncompress	0 0 0 0 0 0 0 0	55
56	Rice_Compress	0 0 3 0 0 0 0 0	188
57	Rice_Uncompress	0 0 0 0 0 0 0 0	178
9	Bmp2rle v. 0.0.1	bitmap encoder	3028
58	get_rle16_line	0 0 0 0 0 0 0 0	171
59	decompress_rle_file	0 4 2 0 0 0 0 0	705
60	save_line_data	0 0 0 0 0 0 0 0	173
61	save_line_data_pal	4 0 0 0 0 0 0 0	191
62	insert_diff_pixel	1 0 0 0 0 0 0 0	53
10	Dozenal v. 2.0	data type converter	3856
63	errorclear	0 4 0 0 0 0 0 0	48
64	dec	0 2 0 0 0 0 0 0	131
65	negexp	6 0 0 0 0 0 0 0	72
66	doz	0 12 0 0 0 0 0 0	160
67	getop	0 0 0 0 0 0 0 0	62
68	commandops	4 0 0 0 0 0 0 0	37
69	operate	0 0 0 0 0 0 0 0	247
70	dometric	0 4 0 0 0 0 0 0	165
71	dotgm	0 0 0 0 0 0 0 0	161
72	dealunit	0 9 0 0 0 0 0 0	399
73	parse	0 0 0 0 0 0 0 0	441
74	domain	0 0 0 0 0 0 0 0	451
11	Libsmacker v. 1.0	video decoder	1820
75	smk_render_palette	3 0 0 0 0 0 0 0	134
76	smk_render_audio	6 0 0 0 0 0 0 0	275
12	Hubbub v.0.0.1	HTML5 file parser	17361
77	hubbub_dict_insert	0 0 19 0 0 0 0 0	62
78	hubbub_dict_search_step	0 0 0 0 0 0 0 0	67
79	hubbub_string_match	0 0 0 0 0 0 0 0	15
13	Clzip v. 1.4	data compression	3011
80	Re_encode_bit	0 4 0 0 0 0 0 2	98
81	Mf_normalize_pos	0 0 0 0 0 0 0 0	76
82	Re_encode_tree	0 3 0 0 0 0 2 0	112
83	Rd_decode_bit	0 0 0 0 0 0 0 0	102
84	get_dict_size	0 0 0 0 0 0 0 0	85
85	parse_short_option	3 0 0 0 0 0 0 0	86
14	SparthMP v. 1.1	math library	1881
86	array_from_sparth	0 0 6 0 0 0 0 0	83

(Continues)

Security Comm. Networks 2015; **8**:3667–3681 © 2015 John Wiley & Sons, Ltd.
DOI: 10.1002/sec

Appendix (Continued)

Number	Name	Description Divergence	Loc
87	fork_ring_next	0 0 0 0 0 0 0 0	78
88	push_ring	0 0 0 0 0 0 0 0	87
89	adjust_sparth_int	0 4 0 0 0 0 0 0	251
15	M's JSON parser	JSON file parser	3557
90	json_unescape	8 0 0 0 0 0 0 0	125
91	json_escape	0 0 0 0 0 0 0 0	146
92	json_format_string	9 0 0 0 0 0 0 0	178
93	json_strip_white_spaces	0 0 0 0 0 0 0 0	44
94	json_parse_document	0 0 0 0 0 0 0 0	3376
16	C algorithm v. 1.2.0	common algorithms	6021
95	arraylist_insert	0 0 0 0 0 0 0 0	57
96	binary_heap_insert	0 0 0 0 0 0 0 0	60
97	bloom_filter_query	0 0 0 0 0 0 0 0	52
98	bloom_filter_intersection	0 0 0 0 0 0 0 0	84
99	bloom_filter_union	0 0 0 0 0 0 0 0	83
100	list_sort	0 0 0 0 0 0 0 0	99
17	Libcsv v. 3.0.3	CSV library	1028
101	csv_parse	4 0 0 0 0 0 0 0	232
102	csv_write	0 1 0 0 0 0 0 0	62
103	csv_write2	0 0 0 0 0 0 0 0	88
18	uriparser v. 0.7.7	URI processing library	4785
104	ParsePctSubUnres	0 0 0 0 0 0 0 0	194
105	EqualsUri	0 0 0 0 0 0 0 0	195
106	FilenameToUriString	0 0 0 0 0 0 0 0	258
107	UriStringToFilename	0 0 0 0 0 0 0 0	217
108	uriWriteQuadToDoubleByte	0 0 0 0 0 0 0 0	28
19	zzjosn v. 1.0.0	JSON library	1470
109	parse_string	0 0 0 0 0 0 0 0	146
110	parse_number	0 3 0 0 0 0 0 0	158
111	zzjson_parse	0 0 7 0 0 0 0 0	1288
20	Libroxml v. 1.0	xml file parser	630
112	roxml_load_doc	5 0 0 0 0 0 0 0	368
113	roxml_get_name	0 0 0 0 0 0 0 0	155
114	roxml_get_parent	5 0 0 0 0 0 0 0	54
115	roxml_get_content	0 0 0 0 0 0 0 0	72
116	roxml_get_nb_attr	3 0 0 0 0 0 0 0	147
21	Libogg v. 1.3.0	multimedia format library	2574
117	oggpack_writecopy	0 0 0 0 0 0 0 0	118
118	ogg_stream_flush	0 0 6 0 0 0 0 0	348
119	ogg_stream_iovecin	0 0 0 0 0 0 0 0	283
120	ogg_stream_flush_i	0 0 0 0 0 0 0 0	153

The rows with a blue background present the information of tested programs: column 1 gives sequence numbers; column 2 lists program names and associated version numbers; column 3 briefly introduces tested programs; and column 4 shows the lines of code. The other rows present the information of test units and experimental results: column 1 also gives sequence numbers; column 2 lists the names of test units; column 3 shows the numbers of each divergent patterns; and column 4 also presents the lines of code.

Specifically, the figures in column 3 of the rows with a white background correspond to the numbers of eight divergent patterns observed in practical test units. These are exceptions, external calls, type casts, symbolic pointers, floating-point arithmetic, 64-bit operations, nonlinear computations, bitwise operations in order. Take the test unit *decode_4* as an example: it has 156 lines of C code and it belongs to the tested program *Coreutils*, which consists of 136825 lines of C code. The total number of path divergences is 7 when testing *decode_4* with CREST. Two divergent patterns are found, which are exceptions and bitwise operations, whose numbers are 5 and 2, respectively.