SymCrash: Selective Recording for Reproducing Crashes

Yu Cao[§], Hongyu Zhang[†], and Sun Ding[‡]

[§]Tsinghua University
Beijing 100084, China
cyrainfish@gmail.com

[†]Microsoft Research
Beijing 100080, China
honzhang@microsoft.com

[‡]Nanyang Technological University
Singapore 639798
ding0037@ntu.edu.sg

ABSTRACT

Software often crashes despite tremendous effort on software quality assurance. Once developers receive a crash report, they need to reproduce the crash in order to understand the problem and locate the fault. However, limited information from crash reports often makes crash reproduction difficult. Many "capture-and-replay" techniques have been proposed to automatically capture program execution data from the failing code, and help developers replay the crash scenarios based on the captured data. However, such techniques often suffer from heavy overhead and introduce privacy concerns. Recently, methods such as BugRedux were proposed to generate test input that leads to crash through symbolic execution. However, such methods have inherent limitations because they rely on conventional symbolic execution techniques. In this paper, we propose a dynamic symbolic execution method called SymCon, which addresses the limitation of conventional symbolic execution by selecting functions that are hard to be resolved by a constraint solver and using their concrete runtime values to replace the symbols. We then propose SymCrash, a selective recording approach that only instruments and monitors the hard-to-solve functions. SymCrash can generate test input for crashes through SymCon. We have applied our approach to successfully reproduce 13 failures of 6 real-world programs. Our results confirm that the proposed approach is suitable for reproducing crashes, in terms of effectiveness, overhead, and privacy. It also outperforms the related methods.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging-*Debugging Aids*

General Terms

Reliability, Experimentation

Keywords

Crash reproduction; program instrumentation; symbolic execution; error handling; capture and replay

1. INTRODUCTION

Although software project teams spend much resource and effort on software quality assurance before releasing products, in reality, released software still contains bugs. Some bugs manifest themselves as crashes, which are often considered as severe problems and are typically assigned a high priority for fixing.

Once software crashes in the field, developers should reproduce and fix the problem. However, it is difficult to reproduce field failures that occur in user environment, which is often different from developer's testing environment. Even the users write a bug report, the reproduction of crash could be still difficult due to complex environmental settings [56], sources of non-determinism [8], and poor bug report quality [54].

To help debug field failures, many crash reporting systems such as Windows Error Reporting [22], Apple Crash Reporter [1], and Mozilla Crash Reporter [31] have been proposed and deployed. When a crash happens in field, these system collect crash related information especially call stack trace, and send these information back to the developers upon user permission. Although the stack information is shown to be useful [6, 16, 18, 27, 48], it is often too limited for effective failure reproduction.

In recent years, automated tools have been developed to help developer reproduce field crashes. Many of these tools [2, 8, 25, 26] are based on the concept of capture and replay --- they capture relevant information from the failing code and reproduce the crashes by replaying the recorded information. For example, ChroniclerJ [8] captures all the non-deterministic inputs to reproduce bugs. However, it is difficult to emulate all non-deterministic inputs. Furthermore, recording user input incurs serious privacy concerns.

BugRedux [25] uses different execution data obtained by different level of instrumentations, and reproduces the crashes by performing symbolic execution over the synthesized traces. Conventional symbolic execution can infer program inputs by solving constraints along the paths. However, conventional symbolic execution often fails to solve certain path constraints due to various hard-to-resolve functions such as overly complex functions and nonlinear math functions [51, 52]. Therefore, the effectiveness of BugRedux can be further improved.

In our work, we propose SymCon, a dynamic symbolic technique, which replaces hard-to-resolve functions with concrete runtime values and then performs symbolic execution. In this way, SymCon can solve more path constraints and improves the effectiveness of conventional symbolic execution. Based on SymCon, we present SymCrash, an automated capture-and-replay technique. SymCrash only selects hard-to-resolve functions to instrument and monitor. When a crash happens, SymCrash performs SymCon using the recorded data and generates test cases that can reproduce the crash.

We also develop a tool, SymCrashJ, which implements SymCrash for Java programs. We evaluate SymCrashJ using 14 failures of 6 real-world programs. SymCrashJ can successfully reproduce 13 out of 14 crashes. We also evaluate the runtime performance and

ASE'14, September 15–19, 2014, Vasteras, Sweden.

http://dx.doi.org/10.1145/2642937.2642993

privacy impact of SymCrashJ. Our results confirm that the proposed approach can achieve lower overhead and better privacy, when compared with the related approaches (BugRedux and Chronicler).

This paper provides the following novel contributions:

- We propose SymCon, a dynamic symbolic execution technique that improves the effectiveness of symbolic execution by replacing the hard-to-resolve functions with their concrete values obtained at program runtime.
- We propose SymCrash, which performs selective recording of hard-to-resolve functions and reproduces crashes using SymCon.
- We develop SymCrashJ, a tool that implements SymCrash for Java programs. We also evaluate the effectiveness, overhead, and privacy of SymCrashJ using real programs.

The remainder of this paper is organized as follows. Section 2 describes the background of crash reproduction and gives a motivating example. Section 3 introduces the background of symbolic execution and presents SymCon. We describe the SymCrash approach and the implementation details in Section 4. Section 5 describes our experimental evaluation and discusses the results. Section 7 surveys related work, followed by Section 8 that concludes the paper.

2. BACKGROUND AND MOTIVATION

Before discussing our approach, we briefly provide some necessary background information on general crash-reproducing methods and give a motivating example.

2.1 Capture and Replay Methods

In recent year, researches have proposed many crash reproduction methods. These methods share the similar process: first they capture the program execution information in the field at the time of crash, they then help developers reproduce the crash by replaying the recorded information in the lab. Figure 1 shows an overall structure of such a capture & replay framework.

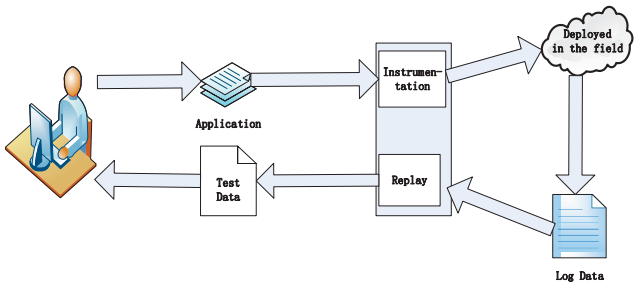

Figure 1. An overview of a crash reproduction

We briefly introduce some of the main methods here:

ReCrash [2, 3] generates multiple unit tests that reproduce a given program failure. It instruments a program to store partial copies of method arguments in memory, and deploys the instrumented program in the field. If the program crashes, ReCrash uses the saved information to create unit tests reproducing the crash. ReCrash can introduce high performance overhead because it captures the states of all objects [8]. Such a high overhead makes ReCrash difficult to be applied in practice.

Chronicler [8] captures all the non-deterministic inputs (such as file, I/O, shared memory, etc.) during program execution and uses the collected data to reproduce the crashes in the lab. Unlike ReCrash, Chronicler performs a lighter recording while still supporting a complete reply. However, the number of non-deterministic methods Chronicler instruments could be overwhelming. Also, recording all non-deterministic inputs, including the user's inputs, could bring serious privacy concerns in practice.

BugRedux [25] collects program execution traces (such as call stacks, method call sequences, complete execution traces) obtained by different level of instrumentations. It can use the collected traces to mimic the observed field failures and to reproduce the crashing faults. Based on the program execution traces, BugRedux utilizes conventional symbolic execution to generate test inputs that can exercise the traces. Their experimental results showed that the method call sequence data are more efficient for crash reproduction. BugRedux may not always return results due to the limitations of conventional symbolic execution (which will be described in Section 3).

2.2 A Motivating Example

We adapt the WU_FTPD example described in [20] as a motivating example. The WU_FTPD program implements a file transfer server and has a known format string vulnerability. The original program of WU_FTPD is written in C. We modify the original C program (version 2.6.0) and port it into a Java program, as shown in Figure 2. The *site_exec* function allows uses to execute commands remotely. The parameter *cmd* contains the user specified command, such as "/usr/bin/helloworld -l *.c". This program throws an exception when the length of the command string exceeds 32. For example, when the input *cmd* is "!!!!!!!!!!!!!!!!!!!!!", the program crashes.

```
1.      public static void site_exec(String cmd)      {
2.        String PATH = "/home/ftp/bin";
3.        int sp = cmd.indexOf(' ');
4.        double dsp=java.lang.Math.log(sp+2);
5.        int j; String result;
6.        if (dsp == 0) {
7.            j = cmd.lastIndexOf('/');
8.            if (j > -1)
9.                result = cmd.substring(j);
10.           else
11.               result = cmd;
12.       } else {
13.               j = cmd.lastIndexOf('/', sp);
14.               result = cmd.substring(j);
15.       }
16.       if (result.length() + PATH.length() > 32) {
17.           throw new RuntimeException ("Buffer overflow");
18.       }
19.       String buf = PATH + result;
20.       execute (buf);
21.   }
```

Figure 2. The site_exec function of the wu-ftpd program

To reproduce the crash, ChroniclerJ requires to record user input, which can reproduce the crash but could violate user privacy. BugRedux uses symbolic execution to generate test input and can thus avoid the privacy concerns. However, BugRedux has limitations in handling certain functions due to the inherent limitations of conventional symbolic execution. For example, to symbolically execute the path along the lines 2-3-4-5-6-12-13-14-16-17, an SMT constraint solver needs to solve the path constraints containing the return value of *Math.log* function. This function is a nonlinear function and takes a variable as a parameter. Its value cannot be easily determined by an SMT

constraint solver during symbolic execution. Therefore, BugRedux cannot reproduce this crash.

In this work, we propose a new capture and replay technique, which only instruments the selected, hard-to-solve functions and enables symbolic execution to continue by utilizing their runtime values. Test input that reproduces the crashes can be generated from symbolic execution. Our approach can mitigate the privacy and overhead concerns and in the meantime improve the effectiveness of crash reproduction.

3. SYMCON - SYMBOLIC EXECUTION WITH CONCRETE VALUES

3.1 Symbolic Execution

Symbolic execution [28] is a program analysis technique, which can infer the program inputs through analyzing the program. During symbolic execution, an analyzer uses symbolic values instead of actual (concrete) values. The inputs are obtained by solving Path Conditions (PCs), which are conjunctions of constraints over symbolic expressions. The solutions to a PC are the inputs that drive the program through an execution path. A PC can be submitted to an off-the-shelf SMT constraint solver (such as Z3 [17] and Yices [19]), which returns a satisfying assignment for all variables appearing in formulas that can be proven satisfiable. If a path is infeasible, the solver returns unsatisfiable and no solution will be given. If the SMT solver cannot solve a path constraint, it returns unknown.

In our work, we implement symbolic execution using the Symbolic Path Finder (SPF) tool [15, 5, 33], which is a symbolic extension of Java Path Finder [47]. SPF combines symbolic execution and model checking techniques to explore different program paths and to automatically generate test inputs. It also provides advanced features such as partial orders and symmetry reductions to handle the problem of state explosion.

Although symbolic execution is effective in generating test inputs, it has limitations too. For example, it has problems in handling complex math operations and external library calls [15, 52]. These limitations are due to the path explosions and the inherent incompleteness in decision procedures. In this paper, we identify a set of functions that are hard to be solved by conventional symbolic execution tools and propose a technique that can address the limitations of symbolic execution by utilizing the runtime values.

3.2 Hard-to-Resolve Functions

Conventional symbolic execution adopts path-based program analysis, which models program behaviour as a path constraint along each execution path. The path constraint is later evaluated against a constraint solver. Path-based analysis offers high precision and is therefore widely adopted in program optimization and test case generation. However, in general, symbolic execution based on path constraint solving faces many intractable problems, including:

- Limited support for the number of predicates along a path. Existing constraint solvers can only support a limited number of predicates in a path. Submitting overly complex path constraints to a solver could lead to state explosion. To address this problem, many tools, such as SPF, introduce an upper bound to limit the number of predicates along a path [33].
- Limit support for the number of paths in a control flow graph (CFG). In symbolic execution, the program branches, function calls and loops are exhaustively visited in a depth-first manner. The symbolic execution terminates when the number paths it processes exceeds an upper bound. Therefore, many paths in a CFG may be failed to verify. To address this problem, many tools such as CUTE [34] and Pex [37] introduce an upper bound to limit the number paths for symbolic execution.
- Limited support for loops/recursions. To overcome the state explosion issue caused by complex program structures like loops or recursive function calls, several advanced program analysis techniques are proposed, such as program abstraction [4, 30, 36] and loop summarizations [24]. The abstraction techniques can map a program with a large number of states into an abstracted model with limited states. Later symbolic execution can be applied to the abstracted model to reduce search cost. Loop summarization techniques treat a loop as a block and summarize the loop's data dependency impact as inferred invariants [24, 35]. However, not all the loop and recursive structures could be accurately abstracted or summarized. For example, the work in [35] summarizes loops by expressing certain important variables with loop counts, but it cannot handle cases where variables are not linearly updated with the corresponding loop count. Although many other techniques, such as search strategy (using search-guiding heuristics to guide path exploration) [53], have been proposed to address the loop problems, the problems still exist [51].
- Limited support for complex string operations. Modern SMT solvers are capable of solving string constraints by expressing the constraints in terms of bitvectors. Therefore, they support symbolic string analysis. However, these tools still cannot support all complex string operations due to possible state explosion. Redelinghuys [41] compared the ability of 7 different symbolic string analysis tools (including SPF and Pex) and pointed out their limitations in supporting various string operations. He found that some string operations, such as *contains* or *startWith*, are fully supported by all the tools. While other functions, such as *split* and *valueOf*, only receive partial support.
- Limited support for native functions and external library calls. Symbolic execution may fail due to inherent complexity of native functions and external library calls [52]. SPF addresses these limitations by using the Model Java Interface (MJI) mechanism [15], which can model external libraries. However, traditional symbolic execution tools still cannot handle some of the native functions such as nonlinear math functions. This is because most of decision procedures and constraint solvers cannot fully support non-linear arithmetic operations.

In our work, we heuristically identify the functions that could lead to the intractable problems. We treat them as *hard-to-solve functions*, which are hard to be resolved by a conventional SMT solver and can block a symbolic execution. The syntactical characteristics of these hard-to-resolve functions are as follows:

- **Deeply nested predicates:** if a function contains deeply nested predicates, the number of paths and the number of predicates in a path may be large. Therefore, it is more likely to cause the state explosion problem. Existing symbolic

execution techniques impose a bound on the size of the search depth. Following the design of SPF [33], we heuristically consider a set of nested predicates with nested level deeper than 10 a hard-to-solve characteristic.

- **Loops/recursions:** a program with loops may cause the number of paths to grow exponentially, and may cause symbolic execution to run out of resources. If a variable is not linearly updated within each iteration, it cannot be symbolically expressed by the loop count (the number of times the loop has executed) [35]. Therefore, we consider a loop containing the following characteristics a hard-to-solve one: a) it updates a variable *v* that is referenced by a path constraint, b) there is an inner loop that also updates *v*, or there are conditional branches within the loop along which the variable *v* is updated, c) the number of loop iterations depends on external input. Furthermore, we consider recursive functions as hard-to-resolve functions.
- **Complex string operations:** in our work, we identify string operations that are not supported or partially supported by SPF [41], and treat them as hard-to-resolve ones. Examples include the *replace*, *split,* and *valueOf* functions.
- **Native functions and external library calls:** we consider the native math functions whose parameters are dependent on external inputs as hard-to-resolve functions. We also consider third-party external library calls (whose source code is not available) as hard-to-resolve functions.

The hard-to-resolve functions can be identified through relatively simple program analysis. Note that our approach also allows users to manually update the hard-to-resolve function list, so that they have flexibility in supporting their specific constraint solvers.

3.3 SymCon

In this work, we propose a dynamic symbolic execution method called SymCon, which can address the limitations of conventional symbolic execution by utilizing the concrete runtime values of the hard-to-resolve functions. The concrete values can be obtained through program instrumentation and are used in constraint solving, together with the symbolic values.

More specifically, in SymCon, if a function *M* is a hard-to-resolve function, we use its return value at runtime, instead of treating it as a symbol. For example, the function *Math.log* is a hard-to-resolve function because it implements nonlinear arithmetic operation. We obtain its return value at runtime via program instrumentation, and use this value in follow-up symbolic execution. If a function is not a hard-to-resolve function, we treat it as a symbol and perform usual symbolic execution.

We use the program in Figure 1 as an example to illustrative SymCon. In this program, the string-related functions such as *indexOf, lastIndexOf, length* can be supported by a modern SMT solver, therefore we treat their return values symbolically. Line 4 contains a hard-to-resolve function (*Math.log*). Suppose we can obtain the concrete value of the *Math.log* function at runtime, we can use this value to perform SymCon. For example, to symbolically execute the path along the lines 2-3-4-5-6-12-13-14-16-17, the path condition to be resolved are as follows:

```
      sp == cmd.indexOf(' ')
∧     dsp == Math.log( sp + 2 )
∧     dsp != 0
∧     j == cmd.lastIndexOf('/', sp)
∧     result == cmd.subString(j)
∧     result.length() + PATH.length() > 32
∧     PATH.length() == 13
```

This PC cannot be solved by an SMT constraint solver such as Z3 due to the existence of a non-linear function *Math.log (sp+2)*. Through program instrumentation, we know the latest return value of this function at the time of crash. So we use this concrete value to replace the *Math.log(sp + 2)* item in the PC, therefore enabling symbolic execution to continue. Finally, the solver returns *"/testtest testtest"*, which satisfies the constraints and reproduces the crash.

Note that the proposed SymCon is different from the mixed concrete-symbolic solving used in SPF [15]. SPF identifies SimplePC (which contains solvable constraints) and ComplexPC (which contains constraints that cannot be solved directly). SPF then forces the solver to generate solutions for the SimplePCs, and use the solutions to solve ComplexPCs. SymCon is also different from existing dynamic symbolic execution techniques such as DART [23], which use randomly generated inputs to enable symbolic executions to continue. SymCon uses the monitored runtime values of hard-to-resolve functions, therefore it can obtain more accurate values for crash reproduction.

4. REPRODUCING CRASHES BASED ON SELECTIVE RECORDING

We propose SymCrash, an approach that reproduces crashes based on SymCon. SymCrash instruments the applications to monitor the return values of hard-to-resolve functions in the field. Using the recorded data, SymCrash can generate test input for the crashes by performing SymCon. Figure 3 shows an overall process of our approach.

SymCrash mainly consists of three parts. The first part is Instrumenter, which instruments the original application to collect program execution information. The second part is Logging, which monitors the program execution in the field and collects necessary log data. The third part is ReExecution, which reproduces crash via SymCon in the lab. During the ReExecution phase, SymCrash replaces the symbolic values of hard-to-resolve functions with the concrete values that are recorded in the log data. In this way, more symbolic executions can be completed and test data that leads to crashes can be generated.

We have implemented a tool called SymCrashJ, which is a realization of SymCrash for Java programs. Note that our approach can be applied to programs written in other languages as well.

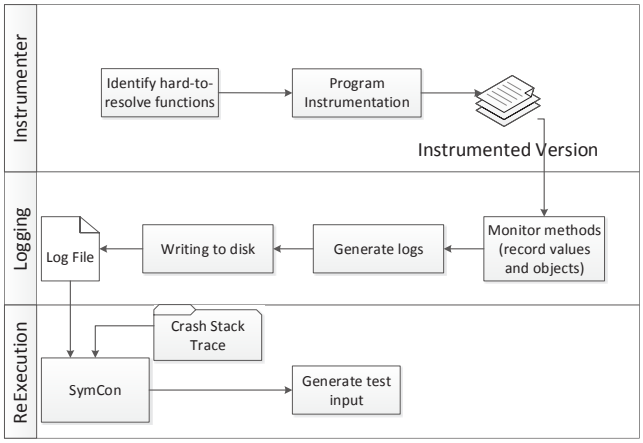

Figure 3. An overview of SymCrash

4.1 Instrumentation and Logging

SymCrashJ first identifies the hard-to-solve functions, which exhibit the characteristics described in Section 3.2 and can be hardly supported by conventional symbolic execution. It then instruments these functions to record the function return values and necessary program execution information. The instrumentation is implemented using the ASM bytecode framework [9]. The instrumented version is deployed in the field, which collects the runtime values of hard-to-resolve functions, generates specific logs according to the method sequence, and writes the log data to disk.

Algorithm 1: ReExecution

Input : *cg* : Call graph of program P
goal_list: Methods in stack trace G_0 G_1 … G_n
log_data : Log data
method_list : A list of hard-to-resolve functions
Output : *input_test :* The test input

```
1  begin
2    cur_goal ← G0
3    state_set ← ( P_entry, initial symbolic values, G0, true )
4    while true do
5      cur_state ← null
6      while cur_state ← null do
7        cur_state ← findNextState( state_set, cg,  cur_goal )
8        if cur_state == null then
9          if cur_goal == G0 then
10           return
11         else
12           cur_goal ← previous goal in goal_list
13           continue
14         end
15       end
16     end
17     if cur_state.m == cur_goal then
18       if cur_goal ≠ Gn then
19         cur_goal ← next goal in goal_list
20         cur_state.m ← cur_goal
21       else
22         input_test ← solver.solve( cur_state.PC )
23         if input_test is found then
24           return input_test
25         else
26           remove ( cur_state, state_set )
27         end
28       end
29     end
30     updateSymCon ( cur_state, state_set , method_list )
31   end
32 end
```

Figure 4. The ReExecution algorithm

There are several issues associated with log data. The first issue is about data type. For immutable types (such as Integer, Double, Float, Byte, Character, Long, String), SymCrashJ simply saves them as pointer references. Other data types require a full copy in order to ensure that the log data reflects the current state of the program, rather than the previous state. Following ChroniclerJ, SymCrashJ uses *System.arrayCopy* provided by JVM to copy arrays that contain immutable types and a runtime reflective cloning library [11] to copy other types. The second issue associated with log data is about log matching. Because the same method can be called multiple times, SymCrashJ does not simply record the method names, it also records the source code line number, the class name, the thread ID, and the current time. The third issue is about the constructor. Java constructor has no return value. As pointed out in [8], in some cases, calling a constructor does not generate a reference to the object. We need to monitor the state of the call stack, and then copy the newly created object until the object is used. In addition, we ensure the correctness of log data in a thread-safe way.

SymCrashJ adopts a similar strategy for writing log data as ChroniclerJ does. Log data stored in memory is automatically saved to disk at regular intervals. Furthermore, log data is generated automatically when program crashes. SymCrashJ uses a daemon process for logging, so that the monitored program can execute normally. There are two disk formats of log data. Some log data can be serialized (such as basic type) and some cannot. Serializable type is saved using Java serialization mechanism and non-serializable type is saved in XML format. These techniques help improve the efficiency and reduce overhead.

4.2 ReExecution

SymCrashJ reproduces crashes using the log data collected through program instrumentation. It generates test inputs by performing SymCon. The test input leads to the crashes that the original crash stack traces represent.

Figure 4 shows the ReExecution algorithm, which is similar to the test generation algorithm used in BugRedux [25]. The input of this algorithm includes *goal_list* (representing call sequence in crash stack), *log_data* (representing the logged data obtained through program instrumentation), *method_list* (representing the hard-to-resolve functions that are instrumented and monitored). The algorithm outputs the test data that can reproduce the crash as specified by the *goal_list*.

Our algorithm works as follows. First, it performs initializations (lines 2-3), which set *cur_goal* (current goal) to the first goal G_0 in the *goal_list*. It then sets *state_set*, which contains quadruples (*m, state, g, PC*), where *m* is the entry method, *state* is the current symbolic state, *g* is the current target goal, and *PC* is the current path condition. The initial value of *m* can be set by developers and is typically the program entry point.

In lines 4-31, the algorithm performs symbolic execution based on the stack trace given in *goal_list*. It first initializes *cur_state*, which is an element of *state_set* and represents the current state of symbolic execution. The algorithm then finds out the next target state to be explored using the *findNextState* function. To reach the *cur_goal* from the current state, we use call graph, which depicts the caller-callee relationship at the method level. We also use the SOOT tool [44] to generate the call graph. For each state *s* in *state_set*, the *findNextState* function computes the shortest path from *s.m* to *cur_goal* in the call graph. It then returns the state that has minimum distance to *cur_goal,* and assigns it to *cur_state.*

Once the *cur_state* is obtained, the ReExecution algorithm checks if its value is *null*. If it is *null*, it means that the current goal cannot be reached. The algorithm then backtracks to the previous goal in the *goal_list* (lines 8-16). For the new *cur_state*, the algorithm checks if *cur_stat.m* and *cur_goal* are the same. If the *cur_stat.m* and *cur_goal* are the same and G_n (the last goal) is reached, the symbolic execution stops and the current PC is submitted to an SMT constraint solver. Otherwise, the algorithm searches for the next goal in the *goal_list* (lines 19-20) and continues symbolic execution by calling the *updateSymCon*

function. The *updateSymCon* function implements SymCon as described in Section 3.3. Once a method *M* is encountered, *updateSymCon* checks if *M* is within the predefined hard-to-resolve function list. If yes, it searches for the function return value recorded in the log file, and uses the return value at the time of crash to replace the symbolic value of *M*. Also, during the symbolic execution, the symbolic state *cur_state* and *state_set* are updated.

The ReExecution algorithm terminates when a test input is found (i.e., the constraint solver finds a solution to satisfy the path constraints), or there are no more states to explore.

In summary, our approach instruments a program, collects the concrete runtime values of hard-to-resolve functions, and performs SymCon to obtain test input that can lead to the same crash stack trace as the users observe. Unlike ReCrashJ, SymCrashJ does not record all the objects and therefore reduces overhead. Unlike BugRedux, SymCrashJ can reproduce more crashes because SymCon enables more constraints to be solved. Unlike ChroniclerJ, SymCrashJ does not record the user inputs, therefore mitigating the privacy concerns.

5. EXPERIMENTS AND RESULTS

5.1 Research Questions

We perform experiments to evaluate the effectiveness, performance overhead, and privacy impact of the proposed approach. We aim to answer the following research questions:

RQ1: How effective can SymCrashJ reproduce failures?
This RQ evaluates the ability of SymCrashJ to successfully reproduce the observed crashes. We have selected 14 crashes of 6 real-world Java applications, as shown in Table 1. Many of these crashes are also used in related work. We run SymCrashJ to see how many of them can be reliably reproduced.

RQ2: What is the runtime overhead of SymCrashJ?
This RQ evaluates if the performance overhead of SymCrashJ, introduced by its instrumentation mechanism, is suitable to be deployed in the field. To answer RQ2, we evaluate SymCrashJ's performance overhead using the same subject programs as used in ReCrashJ [3]. We also run these programs to perform the same tasks as described in [3]. The programs and the tasks are shown in Table 2.

Table 1. The crashes used in our study

Program	Size (KLOC)	Description	Bug ID	Exception	Reason	Related Work
BSTTree	0.04	An implementation of BST Tree algorithm	1	ClassCastException	Object type conversion	CnC [13]
			2	ClassCastException	Object type conversion	CnC [13]
Apache Commons Math	48.21	An implementation of math library	645	MathRuntimeException	Iterate on the original vector, not on the copy that is modified.	ChroniclerJ [8]
			790	IllegalArgumentException	Intermediate integer values overflow when processing large datasets	ChroniclerJ [8]
			803	Non-exception	"X*0 == 0" returns false if X is an infinite number.	ChroniclerJ [8]
Apache Commons Lang	27.91	A library that provides helper utilities for Java.lang API	72	NullPointerException	When calling the function EqualsBuilder.append(Object[],Object[])	ChroniclerJ [8]
			84	Exception	Missing boundary check when calling RandomStringUtils.randomAlphabetic	N/A
			294	ArrayIndexOutOfBoundsException	The indexOf(String str, int startIndex) function does not check whether it has gone over the actual size of the string being built.	N/A
			300	NumberFormat Exception	Invalid strings are passed to the NumberUtils.createNumber method	ChroniclerJ [8]
Ant	101.20	A Java-based build tool	33446	NullPointerException	The value.length() function throws an exception due to a *null* input value.	Star [45]
			38458	NullPointerException	The NullPointerException is thrown in Task.log. getProject().log(this, msg, msgLevel);	Star [45]
Apache Commons Collections	13.61	An extension of java collections library	28	NullPointerException	The SequencedHashMap.indexOf(Object key) method fails when the object *key* is not in the map.	Star [45]
Joda Time	81.96	A date and time Java class library	88	IllegalArgumentException	Invalid arguments for the constructor method	N/A
			93	NullPointerException	The variable *weekyear* is null	N/A

Table 2. The programs used for evaluating the overhead of SymCrashJ

Program	Task	File	Size
SVNKit Checkout	Checking out a project	880	44 Mb
SVNKit Update	Updating a project	880	44 Mb
Eclipse Content	Compiling	Content.java	48 LOC
Eclipse String	Compiling	StringContent .java	99 LOC
Eclipse Channel	Compiling	ChannelIOSe cure.java	642 LOC
Eclipse JLex	Compiling	JLex version 1.2.4	841 LOC

We also evaluate the overhead of SymCrashJ under two extreme scenarios: 1) heavy I/O workload, and 2) heavy string-processing workload. For scenario 1, we develop special programs to simulate the scenario. For scenario 2, we choose the DaCapo benchmark as used in [2, 8]. The overhead is computed as the percentage increase of the running time due to the instrumentation.

RQ3: What is the impact of SymCrashJ on user privacy?

Privacy concerns can adversely affect the usefulness of crash reproducing tools. Unlike ChroniclerJ, SymCrashJ records only the selected, hard-to-resolve functions, not all non-deterministic functions. In this RQ, we evaluate how SymCrashJ affects user privacy by comparing the original inputs that lead to crashes in Table 1 and the test inputs that reproduce these crashes.

For all the RQs, we compare SymCrashJ with two recent work: BugRedux and ChroniclerJ. Because ReCrashJ can be hardly deployed in practice due to its high performance overhead [8], we did not compare with it in RQ1. For BugRedux, it supports four types of execution data (Point of Failure, Call Stack, Call Sequence, and Complete Trace). In our experiments, we only compare our tool with BugRedux instrumented with Call Sequence data, which is the most cost-effective realization of BugRedux as highlighted in [25]. Furthermore, as the original BugRedux supports only C programs, we also developed a Java version of BugRedux in order to perform the comparison. We perform all the experiments on a Windows 7 system, with 3GB RAM.

5.2 Experimental Results

RQ1 - Effectiveness

We apply SymCrashJ, BugRedux, and ChroniclerJ to reproduce the 14 real crashes shown in Table 1. The experiment results are summarized in Table 3, which shows if the observed crashes can be successfully reproduced ("Y" or "N").

Table 3 shows that SymCrashJ can successfully reproduce 13 out of 14 crashes. The programs terminate with the same exceptions as users would observe.

BugRedux only reproduces 9 crashes. Taking the Apache Commons Lang-294 bug as an example, this bug causes the following *ArrayIndexOutOfBoundsException* exception when the function *StrBuilder.deleteAll* is called:

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException
        at java.lang.System.arraycopy(Native Method)
        at org.apache.commons.lang.text.StrBuilder.
           deleteImpl(StrBuilder.java:1114)
        at org.apache.commons.lang.text.StrBuilder.
           deleteAll(StrBuilder.java:1188)
```

The bug is within a user-defined *indexOf* function called by the *deleteAll* function, which does not check the boundary of the string being built properly. BugRedux cannot reproduce this crash because there are many complex string operations (such as *arraycopy*), which are hard-to-resolve functions that block traditional symbolic executions. SymCrashJ utilizes SymCon, therefore it can successfully reproduce this crash.

Table 3. The results for evaluating the effectiveness of SymCrashJ

Bug ID	ChroniclerJ	BugRedux	SymCrashJ
BSTTree-1	√	√	√
BSTTree-2	√	√	√
Math-645	√	√	√
Math-790	√	√	√
Math-803	√	×	×
Lang-72	√	√	√
Lang-84	×	√	√
Lang-294	√	×	√
Lang-300	√	√	√
ANT-33446	×	×	√
ANT-38458	√	×	√
Collections-28	√	×	√
Joda-Time-88	×	√	√
Joda-Time-93	×	√	√

The results also show that SymCrashJ outperforms ChroniclerJ, which can only reproduce 10 crashes. This could be due to the incomplete list of nondeterministic methods that ChroniclerJ instruments and monitors.

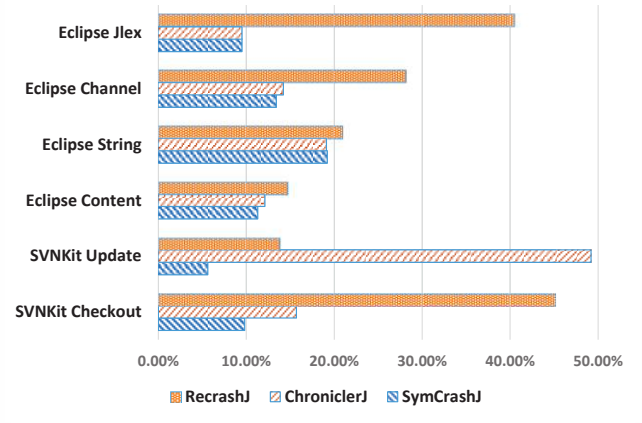

Figure 5. The performance overhead of SymCrashJ

RQ2 - Overhead

We have evaluated the runtime overhead of SymCrashJ using the benchmarks described in [2]. Figure 5 shows the comparison results. Because the Eclipse subjects are small, SymCrashJ and ChroniclerJ achieve similar overhead. The SVN projects are much larger and have a larger number of I/O operations. Therefore ChroniclerJ incurs much higher overhead, while SymCrashJ achieves lower overhead because it only monitors a limited number of hard-to-resolve functions.

We also evaluate the performance of SymCrashJ under two extreme scenarios: 1) heavy I/O workload, and 2) heavy string-processing workload. For scenario 1, we developed a program that reads characters from a disk file, using the *read( )* function in the *java.io.BufferedReader* class. The number of characters to be read is a randomly generated number, ranging from 2MB to 1GB. We run this program 200 times and monitor its execution using SymCrashJ and ChroniclerJ. Figure 6 shows the average overhead of the 200-time executions. SymCrashJ achieves much lower overhead than ChroniclerJ (<2%), while ChroniclerJ causes 40%-88% overhead.

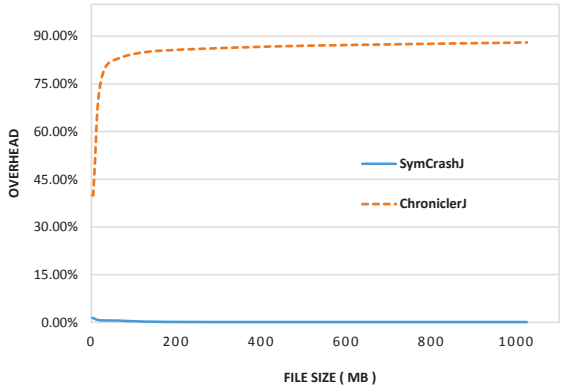

Figure 6. The overhead of SymCrashJ and ChroniclerJ (for heavy I/O workload)

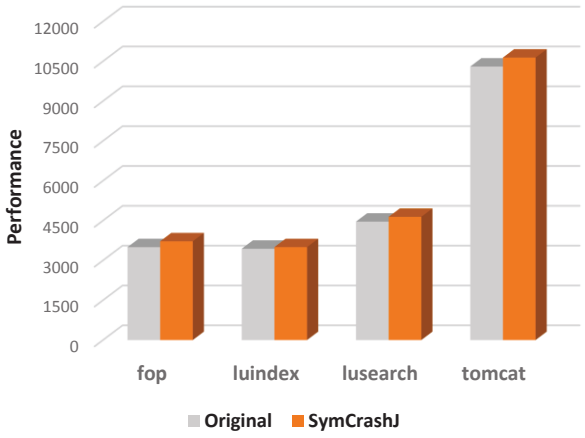

Figure 7. The performance of SymCrashJ and the original program (for heavy string-processing workload)

For scenario 2, we select 4 subject systems from the DaCapo Benchmarks [7], namely Fop, Luindex, Lusearch, and Tomcat. These subject systems are selected because they contain a lot of string-related operations such as text search, file processing, and webpage processing. We use these systems to test the performance of SymCrashJ under heavy string-processing workload. Figure 7 shows the experimental results. SymCrashJ achieves similar performance as the original system's, the overhead is less than 6.5%.

Our evaluation results show that SymCrashJ is a light-weight monitoring tool, which does not incur much performance overhead, even for systems involving heavy I/O and string-processing work.

RQ3 - Privacy

Unlike ChroniclerJ, SymCrashJ records only the selected, hard-to-resolve functions. It generates test data based on symbolic execution and recorded values of the selected functions. Therefore, SymCrashJ does not record all user inputs and the generated test cases could be different from the original user inputs. In this way, SymCrashJ mitigates the privacy issues. As an example, considering the Apache Commons Lang 294 bug, which leads to an *ArrayIndexOutOfBoundsException* exception when the string "\n%BLAH%" is input. Using SymCrashJ, we obtain the input string "\n" through symbolic execution, which can also lead to the same exception, therefore reproducing the crash using a different input, without violating the users' privacy.

Table 4. The results for evaluating the privacy of SymCrashJ

Bug ID	SymCrashJ	ChroniclerJ	Original Input
BSTTree-1	"aaa"	"gbk"	"gbk"
BSTTree-2	"aaa"	"BSTTree"	"BSTTree"
Math-645	-1	0	0
Math-790	1490	1600	1600
Lang-72	3	2	2
Lang-294	"\n"	"\n%BLAH%"	"\n%BLAH%"
Lang-300	"2l"	"1l"	"1l"
ANT-38458	0	82	82
Collections-28	"test"	"s"	"s"

For the 9 bugs that can be reproduced by both SymCrashJ and ChroniclerJ (Table 3), we compare the original inputs that caused the crashes and the test inputs generated by the tools. As ChroniclerJ is based on the recording of the user inputs, the test input generated by ChroniclerJ are exactly the same as the original inputs, while our approach can produce different inputs, therefore mitigating the privacy problem.

5.3 Discussions of the Results

Our evaluation shows that SymCrash is suitable for reproducing crashes, in terms of effectiveness, performance and privacy. SymCrash also outperforms the related methods such as BugRedux [25] and Chronicler [8]. BugRedux supports the collection of method call traces through instrumentations and the generation of test data that exercises the execution traces. Like BugRedux, SymCrash is also based on symbolic execution, but it addresses the limitation of conventional symbolic execution by considering the hard-to-resolve functions. Chronicler [8] reproduces crashes by recording nondeterministic functions including user inputs. Unlike Chronicler, SymCrash does not monitor all non-deterministic methods, instead it only monitors the functions that symbolic execution are likely to fail and generate test input through symbolic execution. Our experimental

results confirm that SymCrash improves crash reproducibility, reduces the overhead and mitigates the privacy concerns.

Our approach has limitation too. Although SymCon can address the limitations of conventional symbolic execution by recording concrete values of hard-to-resolve functions, it may still fail to generate valid solutions. Even though each hard-to-resolve function is replaced with its concrete runtime value, the number of path conditions could be still large, especially for large programs. Symbolic execution could still fail to deal with the state explosion problem even advanced mechanisms (such as partial order, abstraction, loop summarization) are adopted [15, 51, 52]. Furthermore, symbolic execution tools often encounter the object-creation problem, where they fail to generate desirable object states [52]. In Table 3, we show that SymCrashJ cannot reproduce the Apache Commons Math 803 crash. This program contains a function as shown below. This function takes an object of *RealVector* class as an input parameter, performs data copy operations over the object, and then returns an object of *ArrayRealVector* class. SymCon fails to process this function due to the complex nature of symbolic execution.

```
1 public ArrayRealVector  ebeMultiply(RealVector v) {
          …
2         checkVectorDimensions(v);
3         double[] out = data.clone();
4         for (int i = 0; i < data.length; i++) {
5              out[i] *= v.getEntry(i);
6         }
7        return new ArrayRealVector(out, false);
8 }
```

Our experimental results as shown in Figures 5-7 confirm that SymCrash achieves relatively low runtime overhead. To evaluate the overhead, we choose the same subject programs used by ReCrashJ and run these programs to complete certain tasks. These tasks include checking out, updating, and compiling a project. We also design experiments to evaluate our tool in the worst and best scenarios. However, it remains possible that there exist some special use cases of the programs that could lead to inconsistent results as we have obtained.

Furthermore, our experimental results as shown in Table 4 confirm that SymCrash can mitigate the privacy concerns. However, in some circumstances, SymCrash may still generate the same test input as users. For example, if the program only crashes at a certain input value, then the test input generated by symbolic execution should be the same as the original input, thus the user privacy could be violated. Furthermore, a hard-to-resolve function might return a value that includes part of the user input. How to further mitigate privacy concerns is an important future work.

6. THREATS TO VALIDITY

We have identified the following threats to validities:

- **Limited number of subjects**. In our experiments, we evaluate SymCrashJ using 14 crashes of 6 real-world programs. Most of these subjects were also used in related works. These subjects were collected by studying bug reports, building the corresponding versions of the programs, and reproducing the crashes. Such a process is tedious and time-consuming. Therefore the number of crashes we evaluated is rather limited and the bugs underlying the crashes may not be representative or comprehensive. We may have accidentally chosen bugs that lead to better (or worse) crash reproducibility.
- **Concurrency.** So far, we have not evaluated our approach for concurrency related failures. Although the implementation of symbolic execution in our algorithm is based on Java Path Finder, which supports multi-threading, our approach does not monitor communications between threads and therefore does not necessarily reproduce races. It would be interesting to combine our approach with certain race-detection technology [32] or thread-level sequential path analysis techniques [42] to support concurrency.
- **User study**. In our work, we evaluate our approach through in-house experiments. Although such evaluation is based on real-world crashes and programs, the ultimate usefulness and effectiveness of the proposed approach should be evaluated by real users. Conducting a user study and obtaining feedbacks from participants will be an important future work.

7. RELATED WORK

Crash Reproduction

Software crashes are a major contributor to system down time and user dissatisfaction. In recent years, many studies have been dedicated to the analysis of crashes of real-world, large-scale software systems. For example, many crash reporting systems [1, 22, 31] are deployed to automatically collect crash stack information from the field. Ganapathi et al. [21] performed an empirical study of Windows OS crashes and discussed major crash types. Several bucketing methods [16, 18, 29] were proposed to group similar crash reports based on call stack similarity. There are also methods for helping developers locate crashing faults based on collected crash stack traces [27, 48]. DebugAdvisor [6] helps developers find a solution to the reported failure by identifying similar problem reported before.

One of the first steps to comprehend and diagnosis a failure is to reproduce the failure. Many crash reproduction techniques have been proposed. We have described BugRedux [25] and Chronicler [8] in Section 2 and compared SymCrash with them in our experiments. Orso et al. [26, 38] also proposed techniques for selectively capturing and replaying of program executions. Their techniques can be used to generate test cases from user executions and reproduce crash. However, their technique lets users specify a subsystem of interest, which requires the users to have prior knowledge about the possible problematic area. ESD [54] proposed by Zamfir et al. automatically synthesizes executions of the program and reproduces bug symptoms based on the point of failure (POF) information given in a bug report. It uses symbolic execution to try to generate inputs that would reach the POF. As we have shown in this paper, conventional symbolic execution has limitations in generating test data. Furthermore, as pointed out by Jin and Orso [25], POF-based crash reproduction is less effective than the method call trace based one.

Our approach is essentially based on the concept of capture & replay. Roehm et al. [43] presented an approach that is complementary to existing capture & replay approaches. Their approach monitors high-level user interactions (such as editing operations or commands), and visualizes the monitored user interaction traces to help developers reproduce failures. Furthermore, our approach aims for helping developer debug field failures in the lab. Tucek et al. [46] proposed an approach to on-site software failure diagnosis at the very moment of failure. Their tool employs lightweight monitoring to detect failures and collect additional information by re-execution on the user's machine.

JCrasher [12] can generate unit test cases for finding crash-inducing bugs based on randomly generated data. Later on, it is combined with a static analysis tool (ESC/Java) to generate better test cases [13]. They also proposed DSD-Crasher [14]: a tool that uses dynamic analysis to infer likely program invariants, explores the space defined by these invariants through static analysis, and finally produces and executes test cases. These tools focus on generating test cases to find crashing faults. Our work generates test cases that reproduce field failures.

Static and Dynamic Symbolic Execution

Symbolic execution has been widely used in software testing and verification. Static symbolic execution focuses on interpreting program behavior using symbolic expressions. An exemplar tool is SPF [15], which is an extension of Java Path Finder for symbolic execution. SPF targets to automatically generate test case for Java programs through model checking and constraint solving. It has good support for math constraints, string operations, data structures and arrays, and pre-conditions. Păsăreanu et al. [40] also proposed a framework that uses annotations in the form of method specifications and loop invariants. Their technique works backward from the property to be checked and systematically applies approximation to achieve termination.

However, real-world programs are usually large and complex. As described in Section 3, in general, symbolic execution based on path constraint solving faces many intractable problems such as state explosion [15, 52]. To minimize the impact of such intractable problems, most existing symbolic execution approaches, including those reviewed above, are bounded: they provide parameters or mechanisms to limit the symbolic execution under a controllable range [15, 37].

To overcome the intractable problems of symbolic execution, Dynamic Symbolic Execution (DSE) techniques (also known as concolic execution or directed random testing) have been proposed [10, 23, 34]. DSE executes the program under test symbolically and replaces the hard-to-resolve expressions with the concrete values generated by random or default inputs. For example, DART [23] is a DSE technique that runs the program under test both concretely (executing the actual program with random inputs) and symbolically (calculating constraints on values at memory locations expressed in terms of input parameters). CUTE [34] is a DSE technique that attempts to cover all feasible paths: it traverses a program in a depth-first search, and generates path constraint along each traversed path. It forces symbolic operations to be performed as if some symbolic variables are temporarily concrete (non-symbolic). SymCon is also a DSE technique. However, it uses the monitored runtime values of hard-to-resolve functions, instead of using the values generated by random inputs.

Pex [37] is a white-box test case generation tool, which explores programs under test by dynamic symbolic execution and builds automated tests with high code coverage. It supports reasoning over pointer arithmetic and object-oriented programs. To address the space-explosion issue in path exploration, Xie et al. [53] proposed an extension of Pex, which adopts a search strategy that uses state-dependent fitness values (computed through a fitness function) to guide path exploration. Recently, they also proposed to involve human cooperation in DSE. For example, in Pex4Fun [49] and CodeHunt [50], they allow programmers to modify the given working implementation to match the behavior of the secret implementation.

8. CONCLUSIONS

Reproducing field failures is an important step of debugging. In this paper, we first identify a set of methods that are hard to be solved by conventional symbolic execution. We then propose SymCon, a dynamic symbolic execution technique that replaces hard-to-resolve functions with concrete runtime values. Based on SymCon, we present SymCrash, which is an automated capture-and-replay technique. SymCrash only instruments and monitors the selected, hard-to-solve functions. Developers can use the recorded log data to perform SymCon and to reproduce the crashes. We develop SymCrashJ, which is an implementation of SymCrash for Java. We have applied SymCrashJ to successfully reproduce 13 failures of 6 real-world programs. Our results also confirm that SymCrashJ can introduce less runtime overhead and achieves better privacy, when compared to the related tools.

In the future, we will apply our tool to a variety of real programs to further evaluate and improve the tool, and to seek feedback from real developers. In addition, we will consider how to further reduce the impact of state explosion, through a combination of relevant work such as program annotation [40]. Other techniques such as random testing can be also used to complement symbolic execution. In this paper, we identify and instrument all the hard-to-resolve functions. How to perform program instrumentation in order to achieve an optimal balance among effectiveness, overhead, and privacy is also an interesting future work.

9. ACKNOWLEDGMENTS

We thank Huiyong Huo, Rongxin Wu, and Hee Beng Kuan Tan for the helpful discussions on the experiments. This research is supported by the NSFC grant 61272089.

10. REFERENCES

[1] Apple, Technical Note TN2123: CrashReporter. http://developer.apple.com/library/mac/#technotes/tn2004/tn2123.html, 2010.

[2] S. Artzi, S. Kim, and M. D. Ernst. ReCrashJ: a tool for capturing and reproducing program crashes in deployed applications. In *Proc. ESEC/FSE'09*, pp. 295-296, August 2009.

[3] S. Artzi, S. Kim, and M. D. Ernst. ReCrash: Making software failures reproducible by preserving object states. In *Proc. 22nd European Conference on Object-Oriented Programming (ECOOP 2008)*, Paphos, Cyprus, July 2008.

[4] S. Anand, C. Pasareanu, and W. Visser. Symbolic execution with abstract subsumption checking. In *Proc. SPIN*, 2006.

[5] S. Anand, C. S. Păsăreanu, and W. Visser. JPF–SE: A symbolic execution extension to java pathfinder. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems*, Springer, pp. 134-138, 2007.

[6] B. Ashok, J. Joy, H. Liang, S. K. Rajamani, G. Srinivasa, and V. Vangala. DebugAdvisor: A recommender system for debugging. In *Proc. of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (ESEC/FSE'2009)*, pp. 373-382, 2009.

[7] S. M. Blackburn, R. Garner, et al. The DaCapo benchmark suite. http://www.dacapobench.org/benchmarks.html.

[8] J. Bell, N. Sarda, and G. Kaiser. Chronicler: Lightweight recording to reproduce field failures. In *Proc. of the 34th*

International Conference on Software Engineering (ICSE'13), pp. 362-371, May 2013.

[9] E. Bruneton, R. Lenglet, and T. Coupaye. Asm: A code manipulation tool to implement adaptable systems. In *Adaptable and extensible component systems*, 2002.

[10] C. Cadar, V. Ganesh, P. Pawlowski, D. Dill, D. Engler. EXE: automatically generating inputs of death. In *Proc. of the 13th ACM conference on computer and communications security (CCS 2006)*, pp. 322-335. ACM, 2006

[11] Cloning – a Java Deep-Cloning library. http://code.google.com/p/cloning/, 2014.

[12] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. In *Software: Practice and Experience*, vol. 34, pp. 1025-1050, 2004.

[13] C. Csallner and Y. Smaragdakis. Check 'n' Crash: Combining static checking and testing. In *Proc. of the 27th International Conference on Software Engineering (ICSE 2005)*, pp. 422-431, 2005.

[14] C. Csallner, Y. Smaragdakis, and T. Xie. DSD-Crasher: A hybrid analysis tool for bug finding. In *ACM Transactions on Software Engineering and Methodology*, 17(2):345-371, April 2008.

[15] C. Păsăreanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz, and N. Rungta. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. In *Automated Software Engineering Journal*, 20:391-425, 2013.

[16] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel. ReBucket: A method for clustering duplicate crash reports based on call stack similarity. In *Proc. of the 34th International Conference on Software Engineering (ICSE 2012)*, pp.1084-1093, Zurich, Switzerland, June 2012.

[17] L. de Moura and N. Bjorner. Z3: An Efficient SMT Solver. Technical report, Microsoft, 2008.

[18] T. Dhaliwal, F. Khomh, and Ying Zou. Classifying field crash reports for fixing bugs: A case study of Mozilla Firefox. In *Proc. of the 27th IEEE International Conference on Software Maintenance (ICSM'11)*, pp. 333-342, Williamsburg, VA, USA, Sep 2011.

[19] B. Dutertre and L. de Moura. The Yices SMT solver. Tool paper at http://yices.csl.sri.com/tool-paper.pdf, August 2006.

[20] P. Hooimeijer, D. Molnar, P. Saxena, M. Veanes. Modeling imperative string operations with transducers. *Tech. Rep. MSR–TR–2010–96*, Microsoft, 2010.

[21] A. Ganapathi, V. Ganapathi, and D. Patterson. Windows XP kernel crash analysis. In *Proc. of the 20th conference on Large Installation System Administration. Washington*, DC: USENIX Association, pp. 12-12, 2006.

[22] K. Glerum, K. Kinshumann, S. Greenberg, G. Aul, V. Orgovan, G. Nichols, D. Grant, G. Loihle, and G. Hunt. Debugging in the (very) large: ten years of implementation and experience. *In Proc. of 23th ACM SIGOPS Symp. on Operating System Principles (SOSP'09)*, Big Sky, Montana, USA, pp. 103-116, 2009.

[23] P. Godefroid, N. Klarlund, K. Sen. DART: directed automated random testing. In *Proc. 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 213-223. ACM Press, 2005.

[24] P. Godefroid and D. Luchaup. Automatic partial loop summarization in dynamic test generation. In *Proc. International Symposium on Software Testing and Analysis (ISSTA'11)*, pp. 23-33, 2011.

[25] W. Jin and A. Orso. BugRedux: Reproducing field failures for in-house debugging. In *Proc. of the 34th International Conference on Software Engineering (ICSE'12)*, pp. 474-484, Zurich, Switzerland, 2012.

[26] S. Joshi and A. Orso. SCARPE: A Technique and Tool for Selective Capture and Replay of Program Executions. In *Proc. of the 23rd IEEE International Conference on Software Maintenance (ICSM 2007)*, pp. 234-243, 2007.

[27] W. Jin and A. Orso. F3: Fault Localization for Field Failures. In *Proc. International Symposium on Software Testing and Analysis (ISSTA'13)*, pp.213-223, Lugano, Switzerland, 2013.

[28] J. C. King, Symbolic execution and program testing. *Communications of the ACM*, volume 19, number 7, pp. 385-394, 1976.

[29] S. Kim, T. Zimmermann, and N. Nagappan, Crash graphs: An aggregated view of multiple crashes to improve crash triage. In *Proc. of the 41st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2011)*, pp. 486 - 493, Hong Kong, June 2011.

[30] K. L. McMillan, Lazy annotation for program testing and verification. In *Proc. of the 22nd international conference on Computer Aided Verification (CAV'10)*, pp. 104-118, 2010.

[31] Mozilla. Mozila Crash Reporting. http://crash-stats.mozilla.com, 2012.

[32] M. Naik, A. Aiken, and J. Whaley, Effective static race detection for Java. In *Proc. of the 2006 ACM SIGPLAN conference on Programming language design and implementation (PLDI '06)*, ACM, pp. 308-319, 2006.

[33] Symbolic PathFinder -- Tool Documentation. http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc/doc, 2013.

[34] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *Proc. of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE-13)*, pp.263-272, 2005.

[35] P. Saxena, P. Poosankam, S. McCamant, and D. Song. Loop-extended symbolic execution on binary programs. In *Proc. of the 18th international symposium on software testing and analysis (ISSTA'09)*, pp. 225-236, 2009.

[36] N. Sharygina and J. C. Browne. Model checking software via abstraction of loop transitions. In *Proc. Fundamental Approaches to Software Engineering*, Springer, pp. 325-340, 2003.

[37] N. Tillmann and J. De Halleux. Pex–white box test generation for. Net. In *Proc. Tests and Proofs (TAP'08)*, LNCS 4966, Springer, pp. 134-153, 2008.

[38] A. Orso and B. Kennedy. Selective capture and replay of program executions. In *Proc. of the 3rd Int. Workshop on Dynamic Analysis*, pp. 1-7, ACM, 2005.

[39] C. Pacheco, S. Lahiri, M. Ernst, T. Ball, Feedback-directed random test generation. In *Proc. of the 29th International Conference on Software Engineering (ICSE 2007)*, Minneapolis, MN, USA, May 2007.

[40] C. S. Pasareanu, W. Visser. Verification of Java Programs Using Symbolic Execution and Invariant Generation. In *Proc. SPIN*, pp. 164-181, 2004.

[41] G. Redelinghuys. Symbolic String Execution, Master thesis, University of Stellenbosch. http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc, 2013.

[42] M. Russinovich and B. Cogswell. Replay for concurrent nondeterministic shared-memory applications. In *Proc. of Conf. on Programming Languages and Implementation (PLDI'96)*, pp. 258-266, 1996.

[43] T. Roehm, N. Gurbanova, B. Bruegge, C. Joubert, W. Maalej. Monitoring user interactions for supporting failure reproduction. In *Proc. 21st International Conference on Program Comprehension (ICPC 2013)*, 73-82, May 2013.

[44] Soot: a Java Optimization Framework. http://www.sable.mcgill.ca/soot/, 2014.

[45] The Star Project. https://sites.google.com/site/starcrashstack/, 2013.

[46] J. Tucek, S. Lu, C. Huang, S. Xanthos, and Y. Zhou. Triage: Diagnosing production run failures at the user's site. In *Proc. of 21st ACM SIGOPS Symp. on Operating System Principles (SOSP'07)*, 2007.

[47] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2):203-232, April 2003.

[48] R. Wu, H. Zhang, S.C. Cheung and S. Kim, CrashLocator: Locating Crashing Faults based on Crash Stacks. In *Proc. International Symposium on Software Testing and Analysis (ISSTA'14),* San Jose, CA, July 2014.

[49] N. Tillmann, J. de Halleux, and T. Xie. Pex4Fun: Teaching and learning computer science via social gaming. In *Proc. CSEET, Practice and Methods Presentations, & Tutorials (PMP&T)*, pp. 546-548, 2011.

[50] N. Tillmann, J. Bishop, N. Horspool, D. Perelman, and T. Xie. Code Hunt - searching for secret code for fun. In *Proc. of the 7th International Workshop on Search-Based Software Testing (SBST'14)*, 2014.

[51] X. Xiao, S. Li, T. Xie, and N. Tillmann. Characteristic studies of loop problems for structural test generation via symbolic execution. In *Proc. of the 28th IEEE/ACM international conference on automated software engineering (ASE'13)*, pp. 246-256, Nov 2013.

[52] X. Xiao, T. Xie, N. Tillmann, and J. de Halleux, Precise identification of problems for structural test generation. In *Proc. International Conference on Software Engineering (ICSE'11)*, pp. 611-620, 2011.

[53] T. Xie, N. Tillmann, J. de Halleux, W. Schulte, Fitness-Guided Path Exploration in Dynamic Symbolic Execution. In *Proc. of the 39th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2009)*, Lisbon, Portugal, June-July 2009.

[54] C. Zamfir and G. Candea, Execution synthesis: a technique for automated software debugging. In *Proc. of the 5th European conference on Computer systems (EuroSys'10)*, ACM, pp. 321-334, 2010.

[55] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schröter, and C. Weiss. What makes a good bug report? *IEEE Transactions on Software Engineering*, 36(5):618-643, 2010.

[56] A. Zeller. Why does my program fail? A guide to automated debugging. Morgan Kaufmann, May 2005.