STAR: Stack Trace based Automatic Crash Reproduction via Symbolic Execution

Ning Chen, Sunghun Kim, *Member, IEEE,*

Abstract—Software crash reproduction is the necessary first step for debugging. Unfortunately, crash reproduction is often labor intensive. To automate crash reproduction, many techniques have been proposed including record-replay and post-failure-process approaches. Record-replay approaches can reliably replay recorded crashes, but they incur substantial performance overhead to program executions. Alternatively, post-failure-process approaches analyse crashes only after they have occurred. Therefore they do not incur performance overhead. However, existing post-failure-process approaches still cannot reproduce many crashes in practice because of scalability issues and the object creation challenge.

This paper proposes an automatic crash reproduction framework using collected crash stack traces. The proposed approach combines an efficient backward symbolic execution and a novel method sequence composition approach to generate unit test cases that can reproduce the original crashes without incurring additional runtime overhead. Our evaluation study shows that our approach successfully exploited 31 (59.6%) of 52 crashes in three open source projects. Among these exploitable crashes, 22 (42.3%) are useful reproductions of the original crashes that reveal the crash triggering bugs. A comparison study also demonstrates that our approach can effectively outperform existing crash reproduction approaches.

Index Terms—Crash reproduction, static analysis, symbolic execution, test case generation, optimization.

1 Introduction

Software crash reproduction is the necessary first step for crash debugging. Unfortunately, manual crash reproduction is tedious and time consuming [15]. To assist crash reproduction, many techniques have been proposed [12], [20], [33], [38], [43], [52], [63] including record-replay approaches and post-failure-process approaches. Record-replay approaches [12], [43], [52] monitor and reproduce software executions by using software instrumentation techniques or special hardware that stores runtime information. Most record-replay approaches can reliably reproduce software crashes, but incur non-trivial performance overhead [12].

Different from record-replay approaches, post-failure-process approaches [20], [33], [38], [49], [63] analyse crashes only after they have occurred. Since post-failure-process approaches only use the crash data collected by the bug or crash reporting systems [4]–[7], [14], [30], [51], they do not incur performance overhead to program executions. The purpose of post-failure-process approaches varies from explaining program failures [20], [38] to reproducing program failures [33], [63]. For failure explanation approaches, they try to explain the cause of a crash by analyzing the data-flow or crash condition information. For failure reproduction approaches, they try to reproduce a crash by synthesizing the original crash execution. However, the efficiency of existing failure reproduction approaches is not satisfactory due to scalability issues such as the *path explosion* problem [16], [31] where the number of potential paths to analyze grows exponentially to the number of conditional blocks involved. In addition, crashes for object-oriented programs cannot be effectively reproduced by these approaches because of the *object creation challenge* [60] where the desired object states cannot be achieved because non-public fields are not directly modifiable.

This paper proposes a Stack-Trace-based Automatic crash Reproduction framework (STAR), which automatically reproduces crashes using crash stack traces. STAR is a post-failure-process approach as it tries to reproduce a crash only after it has occurred. Compared to existing post-failure-process approaches, STAR has two major advantages: 1) STAR introduces several effective optimizations which can greatly boost the efficiency of the crash precondition computation process; and 2) Unlike existing failure reproduction approaches, STAR supports the reproduction of crashes for object-oriented programs using a novel method sequence composition approach. Therefore, the applicability of STAR is greatly broadened.

To reproduce a reported crash, STAR first performs a backward symbolic execution to identify the preconditions for triggering the target crash. Using a novel method sequence composition approach, STAR creates a test case that can generate test inputs satisfying the computed crash triggering preconditions. Since STAR uses the crash stack trace to reproduce a crash, its goal is to create unit test cases that can crash at the same location as the original crash and produce crash stack traces that are as close to the original crash stack trace as possible. The generated unit test cases can then be executed to reproduce the original crash and help reveal the underlying bug that caused this crash.

We have implemented STAR for the Java language and evaluated it with 52 real world crashes collected from three open source projects. STAR successfully exploited 31 (59.6%) crashes within a maximum time of 100 seconds each on a com-

• *N. Chen and S. Kim are with the Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong.*
E-mail: {ning, hunkim}@cse.ust.hk.

modity platform. Since a successful crash exploitation may not always help reveal the crash triggering bug (Section 5), a detailed investigation was carried out on the usefulness of each crash exploitation. Among all crash exploitations, 22 (42.3%) were identified as useful reproductions of the original crashes, meaning that they could help reveal the actual crash triggering bugs. In comparison, existing approaches such as Randoop [45] and BugRedux [33] could generate useful reproductions of only 8 and 7 crashes, respectively (Section 5.4). We also conducted a developer survey to evaluate how useful are the generated test cases by STAR.

This paper makes the following contributions:

- A novel framework that automatically reproduces crashes based on crash stack traces and implementation, STAR.
- An empirical evaluation of STAR to investigate its usefulness.

The rest of the paper is organized as follows. Section 2 gives an overview of STAR. Sections 3 and 4 present STAR's approaches in detail. Section 5 presents an empirical evaluation. Section 6 discusses the major challenges and potential future work for STAR and identifies threats to validity. Section 7 surveys related work, and Section 8 concludes.

2 OVERVIEW

In this section, we present an overview of STAR. The details of STAR will be presented in Section 3 and Section 4.

Figure 1 is the architecture of STAR. It contains four phases: 1) stack trace processing, 2) initial crash condition inferring, 3) crash precondition computation, and 4) test case generation.

Figure 2 is an example to illustrate STAR's four phases. The *ComponentImpl* class contains three buggy methods. By executing these methods, suppose three different crashes are reported as shown in Figure 3. For crash (a), when the *checkValidity* method is called while *m_length ≠ m_width*, a *RuntimeException* is explicitly thrown at line 07 of the method, causing a program crash. For crash (b), a *NullPointerException* is thrown at line 20 when the *createColor* method is called while *colorID ≤ 0* and *m_defaultColor == null*. For crash (c), a *NullPointerException* is thrown at line 32 when the *add* method is called while *comp1 ≠ null* and *comp2 == null*.

The following sub-sections overview how STAR generates a crash reproducible test case in four phases.

Stack Trace Processing In the first phase, STAR processes the bug report to extract the essential crash information such as: 1) the bug id, 2) the version number, and 3) the crash stack trace. A crash stack trace contains the exception type, names and line numbers of the methods being called at the time of the crash.

For bug reports submitted to bug reporting systems such as Bugzilla [51], their bug id and version number information can be extracted automatically from the "id" and "Version" fields of the bug reports. To extract the crash stack trace information, STAR performs a *regular expression matching* to the bug reports' "Description" and "Comments" fields. Crash stack traces generally have very similar text patterns. Therefore, they can be extracted from the bug reports using regular expression matching.

```
class ComponentImpl implements IComponent {
01  public void checkValidity() {
02    for (IComponent comp : m_subComponents) {
03      if (!comp.isValid())
04        System.out.println("Warning: ...");
05    }
06    if (m_length != m_width)
07      throw new RuntimeException();      // crash (a)
08  }
09
10  public Color createColor(int colorID) {
11    Color color = new Color(colorID);
12    if (m_doLogging)
13      m_logger.log(color);
14
15    Color createdColor = null;
16    if (colorID > 0)
17      createdColor = color;
18    else
19      createdColor = m_defaultColor;
20    createdColor.initialize();          // crash (b)
21    return createdColor;
22  }
23
24  public void add(IComponent comp1, IComponent comp2) {
25    if (m_doLogging)
26      m_logger.log(comp1);
27    if (comp1 != comp2) {
28      if (m_doLogging)
29        m_logger.log(comp2);
30    }
31    int id1 = comp1.getID();
32    int id2 = comp2.getID();             // crash (c)
33    ...
34  }
    ...
    private int          m_width, m_length;
    private Logger       m_logger;
    private boolean      m_doLogging;
    private Color        m_defaultColor;
    private IComponent[] m_subComponents;
}
```

Fig. 2. Motivating example

```
RuntimeException:
  at ComponentImpl.checkValidity(ComponentImpl.java:7)
  at ...
```

(a) Crash stack trace of Figure 2 crash (a)

```
NullPointerException:
  at ComponentImpl.createColor(ComponentImpl.java:20)
  at ...
```

(b) Crash stack trace of Figure 2 crash (b)

```
NullPointerException:
  at ComponentImpl.add(ComponentImpl.java:32)
  at ...
```

(c) Crash stack trace of Figure 2 crash (c)

Fig. 3. Crash stack traces of Figure 2.

Initial Crash Condition Inferring STAR then infers the initial crash condition that triggers the reported crash at the crash location. For example, the initial crash condition for Figure 2 crash (b) at line 20 is *{createdColor == null}* as a *NullPointerException* is thrown at this location. Initial crash conditions can vary according to the content of the crash lines and the type of exceptions.

As a proof of concept, STAR currently supports crashes caused by three types of exceptions: 1) *Explicitly thrown exception*: this type of exception has an initial crash condition of *{TRUE}*, meaning that the exception can be triggered when

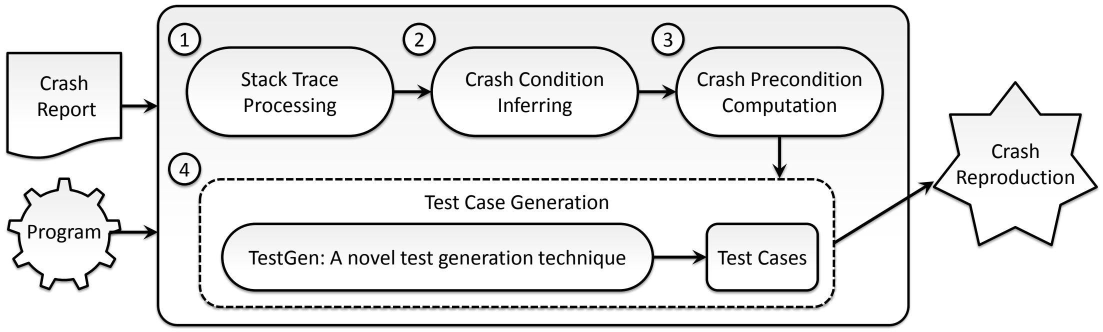
Crash Report
Program
1
Stack Trace Processing
2
Crash Condition Inferring
3
Crash Precondition Computation
4
Test Case Generation
TestGen: A novel test generation technique
Test Cases
Crash Reproduction

Fig. 1. Architecture of STAR which consists of four main phases: stack trace processing, crash condition inferring, crash precondition computation, and test case generation.

the *throw* instruction is reached; 2) *NullPointerException*: this type of exception has an initial crash condition in the form of *{ref == null}*, where *ref* is the variable being dereferenced at the crash location; 3) *ArrayIndexOutOfBoundsException*: this type of exception has an initial crash condition in the form of *{index < 0 or index >= array.length}*, where *array* is the array variable being accessed at the crash location and *index* is the accessing index. Sometimes, multiple initial crash conditions may be inferred. For example, if a *NullPointerException* is raised at a line where two different dereferences (e.g. *x.foo* + *y.bar*) exist, two initial crash conditions will be inferred (i.e. *{x == null}* and *{y == null}*). Each initial crash condition will then be used to reproduce the *NullPointerException* individually.

With additional engineering effort, STAR can support more types of exceptions. However, there are a few types of exceptions whose initial crash conditions are difficult to infer. Some typical examples include *ClassNotFoundException*, *InterruptedException* and *FileNotFoundException*. In general, the initial crash conditions are difficult to infer if the triggering of these exceptions depends on the global program state or the external environment.

Crash Precondition Computation Using the crash stack trace and initial crash condition, STAR next tries to figure out how to trigger the target crash at method entries. We adapt a symbolic execution-based approach to compute a set of weakest preconditions [27] for triggering the target crash. For example, for Figure 2 crash (a), a precondition to trigger the crash at the entry of method *checkValidity* would be: *{m_subComponents ≠ null && m_length ≠ m_width}*.

Test Case Generation Finally, STAR constructs a test case to reproduce the target crash using a novel method sequence composition approach. This approach generates test inputs that satisfy the crash triggering precondition. Once these test inputs are generated, a unit test case is constructed by invoking the corresponding crashing method in the crash stack trace with these inputs.

3 CRASH PRECONDITION COMPUTATION

To compute the preconditions that trigger the original crash (in short, *crash triggering preconditions*), STAR statically analyzes the crashed methods using a backward symbolic execution, which is inter-procedural, and is both path and context sensitive.

STAR's symbolic execution is implemented following a typical *worklist* approach [44], similar to the ones employed in static analysis frameworks [20], [42]. However, these existing symbolic execution algorithms are not efficient for crash triggering precondition computation because they do not leverage crash stack traces and initial crash conditions to guide their executions. Therefore, STAR adapts and extends an existing symbolic execution algorithm [20] so that it: 1) leverages the input crash stack trace to guide the backward symbolic execution and reduce the search space of the execution, and 2) uses the initial crash condition (e.g. *obj == null*) as the starting condition for the execution since our goal is to compute the crash triggering preconditions.

Furthermore, typical symbolic execution algorithms face challenges such as path explosion [31] and inefficient backtracking. To address these challenges, we introduce several optimization techniques that can improve the performance of symbolic execution when computing crash triggering preconditions (Section 3.2).

3.1 Symbolic Execution Overview

Algorithm 1 presents our backward symbolic execution algorithm. The algorithm takes two inputs: 1) the crash stack trace *crashStack*, and 2) the initial crash condition ϕ_{init}. Its output is a set of crash triggering preconditions computed for the target crash.

Algorithm 1 initiates the backward symbolic execution procedure (i.e. *sym_exec*) with ϕ_{init} as the initial postcondition (Lines 01 - 05). To find out the exact instruction where the symbolic execution should start, a *getInstrPrecedingCrash* procedure is called. This procedure finds and returns the instruction immediately preceding the instruction that causes the target crash (Line 04). Typically, the cause of an *Explicitly thrown exception* is the *throw* instruction, the cause of a *NullPointerException* is the variable dereferencing instruction, and the cause of an *ArrayIndexOutOfBoundsException* is the array accessing instruction. For example, assume a *NullPointerException* has been raised at line *x.foo(y)*, and the input initial crash condition is *{x == null}*. The *getInstrPrecedingCrash* procedure will return the instruction immediately preceding the dereference of *x*.

Algorithm 1 Backward Symbolic Execution

```
Input the crash stack trace crashStack
Input the initial crash condition $\phi_{init}$

Begin
01 Initialize satList to hold satisfiable preconditions.
02 method ← crashStack.frame(1).getMethod()
03 line ← crashStack.frame(1).getLineNum()
04 startInst ← getInstrPrecedingCrash(method, line, $\phi_{init}$)
05 sym_exec(method, startInst, crashStack, 1, $\phi_{init}$, satList)
Output satList.

Procedure sym_exec(m, startInst, cs, frmIndex, $\phi_{post}$, satList)
06 worklist ← initialize stack and push < startInst, $\phi_{post}$ >
07 while worklist ≠ ∅: // depth-first propagation
08   < instruction, $\phi_{post}$ > ← worklist.pop()
09   if instruction is an invocation instruction:
10     $\phi_{pre}$ ← call sym_exec for invocation target method
11   else
12     $\phi_{pre}$ ← transform $\phi_{post}$ according to instruction type
13
14   if instruction ≠ m.ENTRY:
15     predecessors ← findPredecessors(instruction)
16     for each predecessor in predecessors:
17       worklist.push(< predecessor, $\phi_{pre}$ >)
18   else if m is a method in the stack trace:
19     if solver.check($\phi_{pre}$) == SATISFIABLE:
20       model ← solver.getModel($\phi_{pre}$)
21       satList.add(< frmIndex, $\phi_{pre}$, model >)
22
23     if frmIndex < cs.totalFrames:
24       m' ← cs.frame(frmIndex + 1).getMethod()
25       line' ← cs.frame(frmIndex + 1).getLineNum()
26       startInst' ← getInstrPrecedingCallSite(m', line')
27       sym_exec(m', startInst', cs, frmIndex+1, $\phi_{pre}$, satList)
28   else // inside some invocation context
29     return $\phi_{pre}$ // continue propagation in caller
```

The instruction returned by *getInstrPrecedingCrash* is then used as the starting instruction for the *sym_exec* procedure (Line 05).

During the backward symbolic execution procedure *sym_exec*, path conditions are collected along the execution paths into symbolic formulae (denoted as ϕ_{pre} or ϕ_{post}). When Algorithm 1 reaches a non-invocation instruction, it evaluates the instruction according to its instruction type to update the current precondition [20] (Lines 11 - 12). For a method invocation instruction, such as *invokevirtual* or *invokestatic*, Algorithm 1 computes its precondition by recursively calling *sym_exec* to descend the current symbolic execution into the invocation target method (Lines 09 - 10). We use the Andersen-style pointer analysis [11] to determine the concrete targets of *invokevirtual* instructions. When multiple possible targets are found at a call site, the algorithm will descend to each possible target method at this call site. Alternatively, an option is also available for specifying the maximum number of targets to consider at each call site.

When *sym_exec* reaches the entry of the method under analysis, it evaluates or returns the computed precondition ϕ_{pre} according to the type of the current method as follows. If the current method is listed in the stack trace frames (i.e., it is a method in *crashStack*), the satisfiability of ϕ_{pre} is evaluated using an SMT solver, Yices [28]. Satisfiable preconditions are saved as the crash triggering preconditions for the current stack frame level (Lines 18 - 21). After that, Algorithm 1 continues the backward execution from the caller (i.e., the previous stack frame in *crashStack*) of the current method to compute the crash triggering preconditions for the caller method (Lines 23 - 27). The symbolic execution in the caller method starts from the location where the caller calls the current method. The satisfiable preconditions obtained in the current stack frame level are used as initial conditions for the caller method's symbolic execution.

If the current method is not listed in the stack trace frames (i.e., it is not a method in *crashStack*), it is a target method of an invocation instruction, which STAR has descended into for computing the precondition of the invocation instruction. In this case, the current precondition ϕ_{pre} is returned to the caller's *sym_exec* procedure (Lines 28 - 29) as the precondition computed for the invocation instruction.

Algorithm 1 stops when it has finished computing crash triggering preconditions for each method in *crashStack*.

Illustrative example To illustrate the symbolic execution steps, consider Figure 2 crash (c). Since a *NullPointerException* is raised at line 32, the initial crash condition input to Algorithm 1 is *{comp2 == null}*. The *sym_exec* procedure initially starts from line 31 which is the first statement preceding to the dereference of *comp2*. Along the backward execution, path conditions are added to the current precondition ϕ_{pre}. For example, at line 31, path condition *{comp1 ≠ null}* is added to ϕ_{pre}. The implicit path condition *{callee ≠ null}* is always added automatically at a dereference statement. At line 27, either *{comp1 ≠ comp2}* or *{comp1 == comp2}* is added to ϕ_{pre} depending on the control flow of the current execution. When reaching an invocation instruction such as line 26, the *sym_exec* procedure is recursively called to compute the precondition for method *log*. The original execution for method *add* is resumed when the *sym_exec* procedure on method *log* returns. Finally, when the execution reaches to the method entry at line 25, since this method (i.e. *add*) is listed in the crash stack trace, the SMT solver is invoked to evaluate the satisfiability of the current precondition ϕ_{pre}. If ϕ_{pre} is satisfiable, it is saved as one of the crash triggering preconditions for the current stack frame. The execution continues until all paths have been traversed or some termination criteria have been satisfied.

Termination handling Due to the undecidability problem [13], *sym_exec* could run forever. This problem is mainly caused by the existence of loops and recursive calls in the target code. To handle this problem, we use two global options to limit the maximum number of times loops can be unrolled and the maximum recursive invocation depth. With these restrictions, *sym_exec* could traverse each program path in finite steps. In addition, to ensure the termination of the algorithm within a reasonable time, a maximum time can be assigned to the crash precondition computation process.

Field and array modeling Since STAR's symbolic execution runs backward, field owners and array indices can only be determined during executions after the use of the fields or arrays. Therefore, they cannot be modeled like ordinary local variables. STAR models fields and arrays during the symbolic execution as functions. Fields are modeled as a function $f : X \rightarrow Y$, where X = {x | x ∈ the set of object references} and Y = {y | y ∈ the set of object references ∨ y ∈ the set of primitive values}. For example, a field *foo.str* is modeled as a function $str(foo)$, where *foo* is the object reference and *str* is the function for all fields under the name *str*. Arrays are also modeled as a function $array : (X \times I) \rightarrow Y$ where (X × I) = {(x, i) | x ∈ the set of array references and i ∈ the set of natural numbers}, and Y = {y | y ∈ the set of object references ∨ y ∈ the set of primitive values}. For example, an array element *foo.str[0]* is modeled as a function $array(str(foo), 0)$, where *str(foo)* is the array reference and *0* is the array index. All array elements are modeled under the function *array*. Both field and array access operations (i.e., read and write) are modeled by functional *read* and *update* from the theory of arrays [39], [40].

Other language features Besides the basic features, STAR also supports or partially supports other language features such as floating point arithmetic, bitwise operations and string operations. We discuss each of these features individually.

As floating point arithmetic is not natively supported by the SMT solver, Yices, STAR converts floating point values to rational numbers whose arithmetic is supported. This might cause some potential imprecision to the arithmetic, but has not affected the correctness of the crash reproductions in our evaluation study.

Similar to the floating point arithmetic, bitwise operations on integers are not directly supported by Yices. Fortunately, since bitwise operations for bit-vectors are natively supported, we are able to simulate bitwise operations on an integer by creating a bit-vector variable whose value equals to the bit level representation of the integer. We then perform the desired bitwise operations directly on the auxiliary bit-vector variable, and convert the result back to an integer. It is worth noting that bitwise operations in Yices are expensive. Therefore, a formula involving many bitwise operations either takes a long time to evaluate or the evaluation simply runs into a timeout.

Finally, string operations are only partially supported by STAR. In general, off-the-shelf SMT solvers such as Yices and Z3 do not have or have only very limited support for string operations. Therefore, STAR represents string variables during symbolic execution as an array of characters (integers). This allows STAR to perform some basic string operations such as *equals*, *length* and even partial of some more complex operations like *indexOf* and *contains*. However, as discussed in Section 6.1, the lack of (complete) support for complex string operations and regular expressions is one of the major challenges for irreproducible crashes. Thus, having a more specialized solver for string constraints may greatly improve the capability of STAR to reproduce string related crashes. The integration of more specialized constraint solvers such as HAMPI [34] or Z3-str [64] to STAR remains as future work.

3.2 Optimizations

Typical *worklist* style symbolic execution algorithms face challenges that can sometimes cause executions to be inefficient. Therefore, in this section, we propose several optimization approaches to improve the efficiency of STAR's symbolic execution.

3.2.1 Static Path Pruning

A major challenge for typical worklist style symbolic execution algorithms is the *path explosion* problem [16], [31]. The number of potential execution paths grows exponentially to the number of conditional blocks. For a loop containing a conditional block, it has the same effect on the path explosion as a sequence of conditional blocks. Figure 2 crash (a) demonstrates one such example. In this Figure, the number of potential execution paths for the *checkValidity* method grows exponentially because of the presence of the loop from Line 02 to Line 05. The path explosion problem is challenging, but it could be alleviated in the case of finding a crashing path since we are more interested in the conditions that could trigger a crash. Therefore, we propose the *static path pruning* approach to heuristically reduce potential paths by pruning *non-contributive* conditional blocks during symbolic execution.

The key idea of the static path pruning approach is that conditional blocks may not always contribute to the triggering of the target crash. Our static path pruning approach tries to slow down the growth of potential paths in the worklist by pruning *non-contributive* conditional blocks. Essentially, this approach can be viewed as an instance of backward static program slicing [8], [59].

However, a typical backward static program slicing approach is not directly applicable to STAR. The main reason is scalability. During the backward symbolic execution of STAR, the set of variables of interest in the slicing criterion is constantly expanding. For example, for an *Explicit thrown exception*, the initial set of variables of interest is empty because the initial symbolic formula (i.e., ϕ_{pre} in Algorithm 1) is *{TRUE}* at the *throw* statement. As the symbolic execution goes on, more conditions are added to ϕ_{pre}, so more variables of interest are added to the slicing criterion. As a result, it is necessary to re-compute a new program slice whenever the set of interesting variables expends, which is not scalable. Therefore, instead of using a more precise but expensive static program slicing approach, STAR introduces a static path pruning approach, which is relatively conservative but more scalable.

In the static path pruning approach, a conditional block is considered *contributive* to the target crash if it satisfies any of the following criteria. 1) A conditional block is considered contributive if it defines a local variable which is referenced in the current symbolic formula ϕ_{pre}. This criterion is obvious since a definition of any local variable referenced in ϕ_{pre} by the block would establish a direct data dependency between ϕ_{pre} and the conditional block. 2) A conditional block is considered contributive if it assigns a value to a field whose field name is referenced in ϕ_{pre}. This criterion is similar to the first one except that it is for object fields. Due to the nature of the backward symbolic execution, the owner objects of fields may not be decidable at the time of use. Therefore, instead of comparing two fields directly, we only compare if the names of the fields assigned by the conditional block match with the names of the fields referenced in ϕ_{pre}. This makes sure that the identification of a non-contributive block is *conservative* but *safe*. Note that this criterion may not be safe when porting STAR to languages where pointer arithmetics are allowed. 3) A conditional block is considered contributive if it assigns a value to an array element when there is an array element with compatible type referenced in ϕ_{pre}. This criterion is similar to the first two except that it is for array elements. However, unlike criterion two, we do not compare the array names because it is possible to have two arrays with different names sharing the same memory location. Therefore, our approach conservatively considers two array elements are the same if they have compatible types.

This static path pruning approach is more conservative than a typical static program slicing approach. However, it has some desirable advantages. First, it takes only linear time to retrieve all assignable information (i.e. defined local variables, field names, array types) for each conditional block in a method. Second, the assignable information only needs to be computed once for each method and is reusable afterwards. Finally, it is easy to identify if a conditional block is contributive or not using the previous definition.

Retrieving the assignable information for each conditional block in a method is straightforward. The approach examines each instruction in the method to see if the current instruction defines a local variable, assigns a field or modifies an array location. For such cases, the defined variable, assigned field name or array type is saved as the assignable information of the conditional blocks where the current instruction is in. When an invocation instruction is reached and the assignable information for the invocation target method has not been computed, the approach performs inter-procedural analysis to compute the assignable information for the invoked method and then adds the result back to the current computation. Recursive calls do not need to be analyzed since their instructions are already being examined. The assignable information for each method is computed only once. Afterwards, the information can be reused during the computation of their caller methods.

Figure 4 presents the pseudo-code for the *skip_conditional* procedure, which uses the assignable information to determine whether the current symbolic execution can skip a conditional block and where it should skip to. Given the current executing *instruction*, procedure *skip_conditional* first invokes *find_assignable* to find the assignable information for each conditional block in the current method as previously stated. After retrieving the assignable information (i.e., defined local variables, field names and array types), the procedure then finds all conditional blocks merging at *instruction*. For example, the conditional block from line 03 to 04 merges at line 05 in Figures 2 crash (a). These conditional blocks

```
Input the current instruction instruction
Input the current method method
Input the current symbolic formula $\phi$
Input the current assignable_info

if assignable information for method has not been computed:
  find_assignable(method, assignable_info)
conditional_blocks ← get_blocks_merge_at(instruction)
sort conditional_blocks by starting instruction descendingly

skip_to ← instruction // by default, no skipping at all
for each cond_block ∈ conditional_blocks:
  if !is_contributive(assignable_info.get(cond_block), $\phi$):
    skip_to ← cond_block.start - 1
  else break
Output skip_to
```

Fig. 4. Pseudo-code for skipping non-contributive conditional blocks

are then sorted by their starting line numbers in descending order. For the same merging location, a conditional block with a larger starting line number is covered by the ones with smaller starting line numbers. By checking the conditional blocks in descending order, we can quickly find out the first contributive conditional block which should not be skipped. For each sorted conditional block, the *is_contributive* procedure is called to check if this conditional block is contributive to the current formula ϕ according to the three *contributive criteria*. Whenever a contributive conditional block is found, or STAR has checked all conditional blocks merging at *instruction*, *skip_conditional* returns the first preceding instruction of the last non-contributive conditional block. The backward symbolic execution will thus skip all non-contributive conditional blocks and continue the execution from the first contributive conditional block found. In Figures 2 crash (a), the conditional block from line 03 to 04 is skipped as it is non-contributive. Moreover, the loop from line 02 to line 05 can be skipped as well.

The static path pruning approach might be unsafe if there are *throw* instructions in the conditional block but no *catch* clauses to catch them before the crash location. Therefore, STAR does not skip conditional blocks containing *throw* instructions. Another place where STAR might be unsafe is implicit exceptions which can be thrown without a *throw* instruction. For example, an implicit *NullPointerException* could be thrown by the statement *foo.str = “”* without the guarding condition: *foo ≠ null*. STAR currently does not avoid path pruning for implicit exceptions. However, in our experiments, implicit exceptions did not actually cause any incorrectness because guarding conditions were always added by other statements that had not been skipped.

3.2.2 Guided Backtracking

Another major challenge for typical worklist style symbolic execution algorithms is that their backtracking mechanism could be inefficient. For example, when we symbolically execute the code snippet in Figure 2 crash (b), after executing a path (line 20, line 17 - line 10) backward and finding it unsatisfiable, a typical symbolic execution algorithm will backtrack to the most recent branching point (i.e., line 12) and execute a new path (line 14, line 12 - line 10) from there. However, this kind of non-guided backtracking is inefficient as the conditional branch (line 12), which the algorithm backtracks to, is actually irrelevant to the target crash and the new path is definitely unsatisfiable as well.

To address this challenge, we introduce a new backtracking heuristic guided by the unsatisfiable core (unsat core) of previous unsatisfiable formula input to the SMT solver. The unsat core of an unsatisfiable formula is a subset of the formula constraints, which is still unsatisfiable by itself. Most modern SMT solvers support the extraction of the unsat core from an unsatisfiable formula. For example, for Figure 2 crash (b), after traversing the symbolic execution path (line 20, line 17 - line 10), constraint *{new Color(colorID) == null}* is the unsat core. The *guided backtracking* mechanism leverages the unsat core information to make backtracking more efficient.

The guided backtracking mechanism works as follows: when a formula for a path has been proven unsatisfiable by the SMT solver, STAR immediately extracts the unsat core of this formula. The SMT solver [28] used in STAR always returns a minimal unsat core for the unsatisfiable formula. STAR then backtracks to the closest point where symbols of the constraints in the unsat core could be assigned to different values, or any of the unsat core constraints has not yet been added to the formula. For the case of Figure 2 crash (b), after traversing the symbolic execution path (line 20, line 17 - line 10), STAR immediately backtracks to line 19 since *createdColor* could be assigned a different value in the *else* branch.

3.2.3 Intermediate Contradiction Recognition

Finally, during backward symbolic execution, inner contradictions could be developed in the current precondition formula ϕ_{pre}. For example, if a definition statement *v1 = null* is executed when a condition *{v1 ≠ null}* has been previously added to ϕ_{pre}, a contradiction is developed in ϕ_{pre} as condition *{v1 ≠ null}* cannot be satisfied anymore. Similarly, when a condition *{v1 ≠ v2}* is added to ϕ_{pre}, it may also cause an inner contradiction if a condition *{v1 == v2}* already exists in ϕ_{pre}. Therefore, recognizing inner contradictions and dropping contradicted paths as early as possible is important for improving the efficiency of symbolic execution.

However, even though the performance of SMT solvers has been improved significantly in recent years, it is still too expensive to perform satisfiable checks to the current precondition formula at every execution step. To stop contradicted executions early with a reasonable number of SMT checks, STAR performs SMT checks only at the merging points of conditional blocks. During backward symbolic executions, multiple paths can be pushed into the worklist only at the merging points. Therefore, detecting contradiction at merging points can prevent contradicting paths from being pushed into the worklist. In this way, already contradicted executions would not cause an exponential growth of potential paths in the worklist.

For Figure 2 crash (c), after traversing the path (line 32 - line 30, line 27), three conditions are added to the current precondition formula: 1) *{comp2 == null}*, which is the initial crash condition inferred at line 32, 2) *{comp1 ≠ null}*, which is the condition added at line 31, and 3) *{comp1 == comp2}*, which is the condition for taking the branch from line 30 to line 27. Since these three conditions can never hold at the same time, an inner contradiction has been developed in the current precondition formula. The intermediate contradiction recognition process quickly recognizes this contradiction and immediately drops the corresponding path.

To verify the effectiveness of the proposed optimization approaches in STAR, we present a detailed comparison study of STAR's performance with and without optimizations in Section 5.2.4.

4 TEST CASE GENERATION

After computing the crash triggering preconditions, we need to generate test inputs that satisfy these preconditions. If the necessary test inputs are primitive types, it is trivial to generate them. However, if the necessary test inputs are objects, generating objects satisfying the crash triggering preconditions could be a non-trivial task [21], [54], [60]. The major challenge is that non-public fields in objects (classes) are not directly modifiable. By using *Reflection*, it is possible to modify non-public fields, but this easily breaks class invariants [17] and may produce invalid objects. Such invalid objects may be able to trigger the desired crashes, but they are not useful to reveal the actual bugs. Even if objects are successfully created by Reflection using the class invariant information obtained from human or through sophisticated analysis, they may be less helpful for fixing the crashes, since developers still do not know how these crash triggering objects are created in non-Reflection manners. Therefore, test inputs should be created and mutated through a legitimate sequence of method calls.

In this section, we present a novel demand-driven test case generation approach, TESTGEN. TESTGEN can compose the necessary method sequences to create and mutate test inputs satisfying the target crash triggering preconditions.

4.1 Architectural Design

Figure 5 presents the architectural design of TESTGEN. For a given subject program, TESTGEN first computes the *intra-procedural* method summary information for the subject program (Section 4.3). Then, this intra-procedural summary information is used to compute the precise summary of each program method (Section 4.4). Finally, using the computed method semantic information, TESTGEN composes legitimate method sequences for generating test inputs that satisfy the desired crash triggering preconditions (Section 4.5.1).

4.2 Summary Overview

The summary of a method is the semantics of the method represented in the propositional logic. Since a method can have multiple paths, its summary can be defined as the disjunction of the summaries of its individual paths. The summary of each individual path can be further defined as $\phi_{path} = \phi_{pre} \wedge \phi_{post}$, where ϕ_{pre} is the path conditions represented as a conjunction of constraints over the method inputs (i.e., any

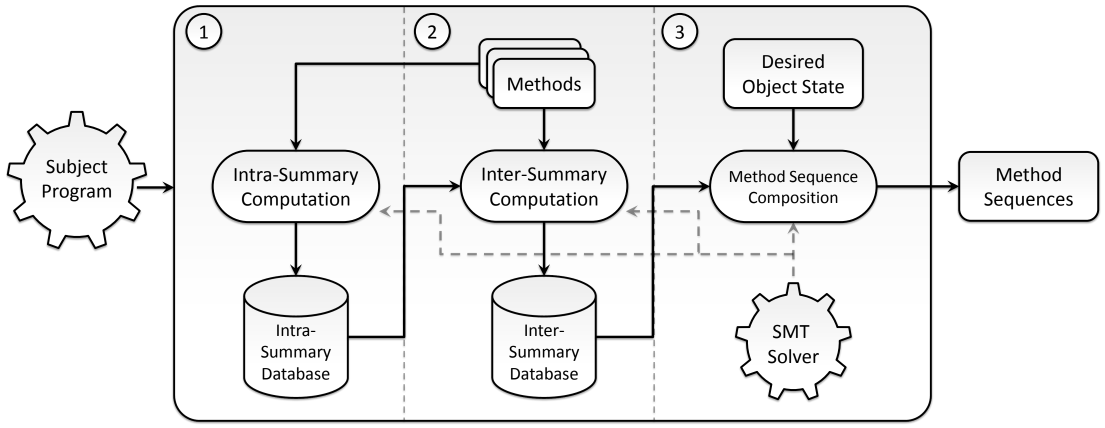

Fig. 5. The architecture of TestGen. TestGen consists of three main phases: intra-summary computation, inter-summary computation, and method sequence composition using the computed method semantic information. The SMT solver module is used in all three phases.

memory address read by the method). ϕ_{post} is the postcondition of the path represented as a conjunction of constraints over the method outputs (i.e., any memory address written by the method). We refer ϕ_{pre} and ϕ_{post} of a method path as the *path condition* and *effect* of this path from now on.

The method summary provides us the precise semantic meaning of this method. This semantic information is necessary for composing the method sequences for generating target objects. Therefore, the method summary needs to be computed first.

Computing the summary of a method can be expensive when this method involves numerous calls to other methods which in turn can make further calls. This is because the number of potential paths needed to be explored grows exponentially to the number of method calls. TESTGEN adapts the idea of *compositional* computation introduced in dynamic test generation [10], [31] to help reduce the number of paths needed to be explored. However, instead of performing a top-down style computation like the original compositional approaches, TESTGEN performs a bottom-up style computation. TESTGEN computes method summaries compositionally in two steps. First, it computes the intra-procedural method summaries (in short, *intra-summaries*) through a forward symbolic execution. Then, it computes the inter-procedural method summaries (in short, *inter-summaries*) in a bottom-up manner by merging the intra-summaries together.

4.3 Intra-procedural Summaries

4.3.1 Overview

The first phase of TESTGEN is intra-summary computation. In this phase, TESTGEN applies an *intra-procedural* path sensitive forward symbolic execution algorithm to all the methods whose summaries we want to compute. During the symbolic execution of a method path, TESTGEN collects and propagates the symbolic value of each variable along the path into a *symbolic memory*. To compute the summary of an executing path, ϕ_{path}, TESTGEN adds a constraint (expressed over the symbolic values of method inputs) to the path condition ϕ_{pre} whenever it meets a conditional branch during the execution. At the exit block of the path, TESTGEN constructs the effect ϕ_{post} of this path by collecting the symbolic values of each method outputs from the current symbolic memory. The summary of a method is computed by taking the disjunction of the summaries of its individual paths.

Since the symbolic execution algorithm applied in this phase is intra-procedural, effects introduced by callee methods along a path are not considered. TESTGEN uses skolem constants [20] to represent the possible effects induced by these skipped method calls. Section 4.3.4 discusses this issue in detail.

4.3.2 Forward Symbolic Execution

Figure 6 presents the simplified pseudo-code for the intra-procedural summary computation. In general, it is a worklist-based algorithm [44] which traverses every method path in preorder. For a target method, *method*, this algorithm computes the summaries of its execution paths. The initial element in the *worklist* stack is the method entry instruction along with a *TRUE* value for ϕ_{pre}, and an initial symbolic memory S where each method input is assigned with an initial symbolic value. Method instructions are traversed in preorder until every method path has been executed or some predefined thresholds have been satisfied (e.g., maximum time limit). At every method instruction, the symbolic values in the current symbolic memory S are propagated according to the type of *instruction* and the propagation rules listed in Table 1. Constraints may be added to the current path condition ϕ_{pre} according to the *instruction* type, as specified in Table 1. For conditional instructions, the symbolic expressions of their branch conditions are also added to ϕ_{pre} according to the execution control flow.

When the symbolic execution reaches to an exit node of the current method, the execution for this path finishes. The algorithm then leverages an SMT solver [28] to evaluate the satisfiability of ϕ_{pre}. If ϕ_{pre} is satisfiable, the effect of this path (i.e. ϕ_{post}) is further constructed by collecting the symbolic values of all method outputs from the current symbolic memory S. Note that the outputs of a method are not just

TABLE 1
Forward propagation rules for some of the instruction types

INSTRUCTION TYPE	PROPAGATION RULE	CONSTRAINTS ADDING TO ϕ_{pre}
v1 = *v2*	put_val(*v1*, get_val(*v2*))	$\emptyset$
v1 = *v2* op *v3*	put_val(*v1*, get_val(*v2*) op get_val(*v3*))	$\emptyset$
v1 = *v2.f*	put_val(*v1*, read(*f*, *v2*))	$v2 \neq null$
v1 = *v2[i]*	put_val(*v1*, read(*array*, (*v2*, *i*))	$v2 \neq null$, $i \geq 0$, $i < v2.length$
v1.f = *v2*	update(*f*, *v1*, get_val(*v2*))	$v1 \neq null$
v1[i] = *v2*	update(*array*, (*v1*, *i*), get_val(*v2*))	$v1 \neq null$, $i \geq 0$, $i < v2.length$
v1 = new *T*	put_val(*v1*, *instanceof(T)*)	$\emptyset$
v1 = *v2*.func(*v3*, ...)	put_val(*v1*, σ_{ret} (*func*)) for any member or static *field* reachable by *func* in the current symbolic memory: σ_{update} (*field*, *func*) for any member or static *array* reachable by *func* in the current symbolic memory: σ_{update} (*array*, *func*)	$v2 \neq null$

```
Input the method to compute.
Global a worklist to hold the next instructions

Initialize a summaries set.
Initialize a Symbolic Memory $S$.
worklist.push(< entry_inst, TRUE, S >)
while worklist $\neq \emptyset$: // preorder traversal
    < instruction, $\phi_{pre}$, S > $\leftarrow$ worklist.pop()
    $S \leftarrow$ propagate $S$ according to instruction.
    $\phi_{pre} \leftarrow$ add constraints to $\phi_{pre}$ according to instruction.
    successors $\leftarrow$ findSuccessors(instruction)
    if successors $\neq \emptyset$:
        for each successor in successors:
            if instruction is a conditional branch instruction:
                $\phi_{pre}$' $\leftarrow$ add branch condition to $\phi_{pre}$
                worklist.push(< successor, $\phi_{pre}$', S >)
            else:
                worklist.push(< successor, $\phi_{pre}$, S >)
    else if solver.check($\phi_{pre}$) == SATISFIABLE:
        $\phi_{post} \leftarrow \bigwedge_{v \in outputs} S(v)$
        summaries.add($\phi_{pre} \wedge \phi_{post}$)
Output summaries.
```

Fig. 6. Pseudo-code for intra-procedural summary computation

its return value, but also other memory addresses that can be written by this method such as parameter object fields, static fields, and heap locations. Finally, the summary for this path (i.e., $\phi_{pre} \wedge \phi_{post}$) is added to *summaries*.

If a branch condition contains expressions that do not belong to the set of theories supported by the SMT solver (e.g., non-linear arithmetics), the execution will skip paths involving this branch.

In the presence of loops, the number of possible execution paths is infinite, causing the algorithm to run forever. To address this issue, the algorithm automatically unrolls loops for a certain number of iterations specified by users.

4.3.3 Propagation Rules

Table 1 lists some propagation rules for the forward symbolic execution. In this table, function *get_val* reads the current symbolic value of a variable from the current symbolic memory. Function *put_val* writes a symbolic value to a variable in the current symbolic memory. Functions *read* and *update*, as discussed in Section 3.1, correspond to the *read* and *update* functions in the theory of arrays.

4.3.4 Representing Uninterpreted Invocations

Since TESTGEN does not interpret method invocations during the intra-procedural summary computation, it uses skolem constants [20] to represent the possible effects induced by the uninterpreted method invocations as shown in Table 1. When the forward symbolic execution algorithm meets a method invocation, two types of skolem constants may be generated, namely σ_{ret} and σ_{update}. A σ_{ret} constant represents the possible symbolic value returned by this invocation. σ_{update} represents the possible uninterpreted functional updates induced by this invocation to a function representing a field or an array. However, the set of fields or arrays updated by this invocation is unknown during the current intra-procedural analysis. Therefore, all member and static fields or arrays, which are potentially reachable by the invocation and referenced in the current symbolic memory, will be updated.

With the help of the skolem constants, the forward symbolic execution algorithm can propagate the current symbolic memory without actually interpreting any method invocations. All skolem constants will be interpreted to compute the inter-procedural summaries in the next phase.

4.4 Inter-procedural Summaries

Intra-summaries do not contain the effects of method invocations. So they are not precise enough to be directly used to compose method sequences for generating objects. In this phase, TESTGEN computes the inter-procedural summaries (in short, summaries) for all program methods from their corresponding intra-summaries previously computed.

4.4.1 Applying Callee Summaries

TESTGEN is different from compositional test generations [10], [31] in that it computes method summaries in a bottom-up manner leveraging a method dependency graph computed for the subject program. This allows TESTGEN to compute the summary of a method before any of its dependent (i.e., caller) methods. The reason for using a bottom-up manner computation is that we want to retrieve the path summaries of a method as completely as possible with few redundant computations.

In general, the path summary of a (caller) method can be computed by applying the summaries of its callee methods one by one for all call sites along this path. Since one callee method can have multiple path summaries, each of these summaries will be applied to the current caller path summary separately. Therefore, the application of a callee method can generate a set of summaries corresponding to taking different paths within the callee method.

Applying a callee path ($\phi_{callee\_path} = \phi_{callee\_pre} \wedge \phi_{callee\_post}$) to a caller path ($\phi_{path} = \phi_{pre} \wedge \phi_{post}$) works as follows. First, the path condition of the applied summary (denoted as $\phi_{applied\_pre}$) is constructed by taking the conjunction of the path conditions of the two (caller and callee) paths. So, $\phi_{applied\_pre} = \phi_{pre} \wedge \phi_{callee\_pre}$. Formal variable names in the callee path are converted to actual names in the caller path before the application. Second, all skolem constants in ϕ_{path}, which were introduced at the current call site during intra-procedural computation, are substituted with actual values or effects from the callee. More specifically, all instances of σ_{ret} in ϕ_{path} are substituted with the actual symbolic return value of ϕ_{callee_path}. All instances of σ_{update} for a field f are substituted with the actual functional updates to field f found in ϕ_{callee_path}. This means that we are applying the effect of the callee path on field f to the caller path. Finally, the SMT solver checks for the satisfiability of the path condition of the applied summary (i.e. $\phi_{applied\_pre}$). Summaries with unsatisfiable path conditions are discarded immediately before applying the effect of other callee methods from the rest of the call sites.

To avoid infinite computation in presence of recursive or indirect recursive calls, TESTGEN sets a maximum depth for applying recursive method summaries. Thus, recursive method summaries are applied to the current ϕ_{path} for limited depth.

4.4.2 Other Details

Due to the presence of virtual calls, one call site may have multiple possible targets. TESTGEN determines the concrete targets for all virtual calls in the subject program with a pre-computed call graph using an Andersen-style analysis [11]. If multiple targets are possible at a call site during the process of applying callee summaries, TESTGEN applies each possible target separately to the current caller path.

For methods with infinite or large number of paths, TESTGEN may not retrieve the complete set of summaries corresponding to every possible method path. Therefore, STAR may not reproduce some crashes because of missing the path summary information. However, the potential incompleteness of the path summary information does not affect the correctness of any generated crash reproducible method sequences.

Figure 7 presents an example of a path summary for the method *ComponentImpl.createColor(int)* from Figure 2. This summary corresponds to the method path: (Lines 11 - 12, 14 - 17, 20 - 22). Variables *v1*, *v2* and *RET* in the example represent the method callee, the parameter *colorID* and the return object respectively. The effect of this method path is returning a *Color* object with its *m_colorID* field and *m_initialized* field set to *v2* and *true* respectively.

```
Path Conditions:
v1 != null
v1.m_doLogging == false
v2 > 0

Effect:
RET instanceof Color
RET.m_colorID     = v2
RET.m_initialized = true
```

Fig. 7. An example of a path summary

4.5 Method Sequence Composition

After computing the summary for each program method, TESTGEN can now use these method semantic information to compose method sequences to satisfy the crash triggering preconditions. A crash triggering precondition can consist of conditions involving several test inputs. All test inputs should satisfy their corresponding desired object states for the whole crash triggering precondition to be satisfied. Therefore, to create a crash reproducible test case, TESTGEN needs to compose method sequences that can generate all test inputs satisfying their corresponding object states.

4.5.1 Example

We first use a simple example in Figure 8 adapted from the Apache Commons Collections library to illustrate the general method sequence composition process[1]. For the purpose of illustration only, we omit some in-depth details such as method selection, path selection, and validating formula construction during the illustration. These details will be presented in Sections 4.5.3 to 4.5.5. In Figure 8, the solid rectangles represent some desired object states which we want to satisfy. They are numbered at its upper right corner. The ovals with dashed borders are candidate methods which have been selected by TESTGEN to compose the method sequence.

Initially, a desired object state S1-1 is input. This is the final object state we want to satisfy by TESTGEN. Through a method selection process, TESTGEN finds a method *MKMap.MKMap(LMap)* which can generate objects satisfying this object state. However, to achieve this object state by method *MKMap.MKMap(LMap)*, its input parameter (i.e., *v2* in the figure) should satisfy another object state S2-1. More precisely, object state S2-1 is the precondition for method *MKMap.MKMap(LMap)* to generate objects satisfying object state S1-1. Thus, the satisfaction problem for object state S1-1 has been deduced into the satisfaction problem for object state S2-1.

Following a similar method selection process, TESTGEN finds another method *LMap.addMapping(Object)* which can modify a *LMap* object to object state S2-1 if precondition object states S2-2 and S3-1 have been both satisfied. Then again, the satisfaction problem for object state S2-1 has been deduced into the satisfaction problems for object state S2-2 and object state S3-1 for parameters *v2* and *v3* respectively.

For object state S2-2, TESTGEN finds that by calling method *LMap.LMap(int, int)* with integer parameters *1* and *2*, a *LMap* object satisfying this object state can be generated. Similarly, by calling method *Object.Object()*, parameter object *v3* which satisfies object state S3-1 can be generated. After that, both object states S2-2 and S3-1 are deduced to *TRUE*, meaning

1. Class names and method names have been modified for better readability. MKMap is short for MultiKeyMap and LMap is short for LRUMap.

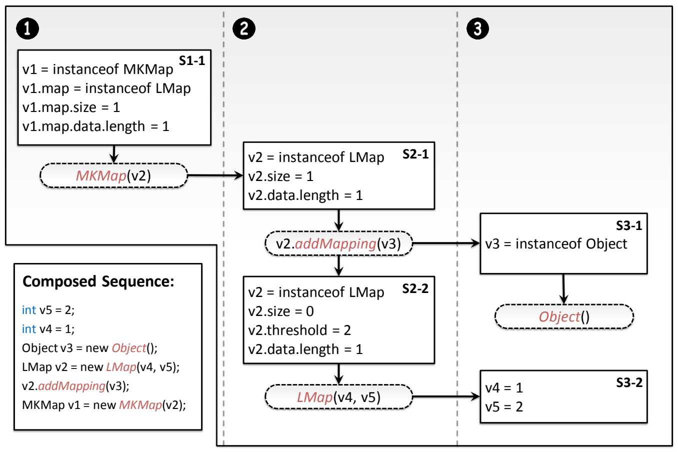

Fig. 8. A method sequence composition example adapted from Apache Commons Collections

that we do not need to satisfy additional precondition object states anymore.

Finally, by concatenating all selected candidate methods backward, TESTGEN composes a method sequence as shown in the lower left corner of the figure. By executing this method sequence, an object satisfying the input object state S1-1 can be generated.

4.5.2 Method Sequence Deduction

Figure 9 presents the pseudo-code for the sequence composition approach. This approach composes a method sequence to satisfy a desired object state in a deductive process. Given a desired object state, ϕ_{state}, the approach first selects a set of candidate methods which may produce an object satisfying this state. Details for the method selection process are presented in Section 4.5.3.

For each candidate method *m*, its path summaries are loaded. Then, TESTGEN performs a path summary selection as discussed in Section 4.5.4 to reduce the number of paths needed to be examined. For each selected path, TESTGEN checks if this path can produce the desired object state, by leveraging an SMT solver [28]. More precisely, TESTGEN examines whether for a path with path summary ϕ_{path}, $\phi_{path} \wedge \phi_{state}$ is satisfiable. Details about the construction of the validating formula from ϕ_{path} and ϕ_{state} are presented in Section 4.5.5.

If the SMT solver returns *satisfiable* for a method path, the approach further queries an input assignment (i.e. a model) which satisfies $\phi_{path} \wedge \phi_{state}$, from the solver. The current candidate method *m* can produce the desired object state when its inputs satisfy this assignment. Then, this assignment is separated into several states (*deduced_states*) corresponding to different inputs (e.g. callee, parameters, static fields, etc.). In this way, the satisfaction problem for ϕ_{state} is deduced into the satisfaction problems for the *deduced_states*.

If a *deduced_state* corresponds to a primitive type, it can be easily satisfied by directly assigning the desired value. If a *deduced_state* corresponds to an object type, the sequence composition approach is recursively invoked to compose the method sequence for generating this deduced object state.

```
Input desired object state $\phi_{state}$ to satisfy
Input the target class $clazz$

Find $candidate\_methods$ from $clazz$.
for each method $m$ in $candidate\_methods$:
  $full\_sequence$ ← $EMPTY$
  $summaries$ ← load $m$'s path summaries
  $summaries$ ← select candidate path summaries
  for each $summary$ in $summaries$:
    $formula$ ← construct validate formula for $summary$
    if solver.check($formula$) == $SATISFIABLE$:
      $assignment$ ← solver.getModel($formula$)
      $deduced\_states$ ← separate $assignment$ by inputs.
      $deduced\_sequences$ ← generate for $deduced\_states$
      if $deduced\_sequences$ have all been generated:
        $full\_sequence$.add($deduced\_sequences$)
        $full\_sequence$.add(new InvokeStatement($m$))
        break
  if $full\_sequence$ != $EMPTY$:
    break
Output $full\_sequence$
```

Fig. 9. Pseudo-code for method sequence composition

If all *deduced_states* have been satisfied, a *full_sequence* is composed by adding the *deduced_states*' generation sequences (*deduced_sequences*) and an invocation statement to the current method *m* with the generated inputs. If the generation sequence for any of the *deduced_states* cannot be composed, the sequence composition approach goes on to examine the next path summary of *m*. During the composition process, TESTGEN caches all composed sequences and their corresponding *deduced_states*. For a new *deduced_state* that is subsumed by a previously satisfied *deduced_state*, TESTGEN will reuse the previously generated input or the cached sequence.

4.5.3 Candidate Method Selection

In the target class *clazz*, there are many methods with various effects. To reduce the number of methods needed to be examined during the sequence composition process, TESTGEN only selects a subset of these methods as the candidate set to be examined.

The criterion for selecting a candidate method is that the selected method should be a non-private method or constructor that can modify any of the fields referenced in ϕ_{state}. For example, for a desired object state: *v1.size == 1*, only methods or constructors which can modify the *size* field of the class of *v1* are selected as the candidate methods. For a desired object state which does not reference any fields, such as: *v1 == instanceof Foo*, the non-private constructors or factory methods of class *Foo* are selected as the candidate methods. In the future versions of TESTGEN, we will also try to select methods from other classes to generate the desired object states.

The selected candidate methods are also prioritized based on their number of path summaries, the number of non-primitive type parameters and how directly do they modify the target fields (i.e. how many subsequent calls they need to go through to modify the target fields). Methods with less path summaries, less non-primitive type parameters and which can modify the target fields more directly are assigned higher priorities to be examined.

4.5.4 Path Summary Selection

Not every loaded path summary is useful for the sequence composition. For a target object state, we want to examine as less path summaries as possible before finding the satisfiable one. Therefore, similar to the candidate method selection, TESTGEN also performs a path summary selection process to reduce the number of path summaries needed to be examined.

For the path summary selection, TESTGEN introduces a lattice-based summary selection approach. To construct the lattice, we define a partial order $\leq$ over the set of path summaries as follows. For two path summaries, *S1* and *S2*, we have $S1 \leq S2$ if the satisfaction of the *deduced_state* of *S2* subsumes the satisfaction of the *deduced_state* of *S1*. In other words, if the *deduced_state* of *S2* can be satisfied, the *deduced_state* of *S1* can always be satisfied as well.

Using this partial order definition, TESTGEN constructs a lattice $L = (S, \leq)$ where S is the set of path summaries, and $\leq$ is the summary partial order previously defined. During the method sequence composition process, TESTGEN only selects path summaries that are not greater than any element whose *deduced_state* cannot be satisfied. This is because, if the *deduced_state* of an element *S1* is not satisfiable, the *deduced_states* of all elements greater than *S1* in L are not satisfiable as well by definition.

For example, consider two path summaries *S1* and *S2* with their *deduced_states*: *S1: {foo.bar ≠ null}* and *S2: {foo.bar ≠ null && foo.size == 1}*, $S1 \leq S2$ holds in L according to the partial order definition. If the *deduced_state* of *S1* cannot be satisfied, the *deduced_state* of *S2* cannot be satisfied as well. Thus, TESTGEN discards *S2* and any other path summaries greater than *S1* during the method sequence composition process.

4.5.5 Validating Formula Construction

To examine whether a path summary ϕ_{path} can produce ϕ_{state}, TESTGEN translates this problem into the satisfaction problem of a validating formula F which is represented as a set of SMT statements acceptable by Yices. To construct F, all method inputs which have been referenced in either ϕ_{path} or ϕ_{state}, are defined as variables of F. In particular, for method inputs which are not related to field or array accesses, they are modeled as basic variables of F. For method inputs which are fields or arrays, TESTGEN models them as uninterpreted functions (a special kind of variable) of F. Any value assignments in the summary effect are modeled as functional *update* statements in F. Similarly, any value dereferences in ϕ_{path} or ϕ_{state} are modeled as functional *read* statements in F.

TESTGEN constructs F following a specific order. First, the summary path condition is modeled as a set of *assertion* statements over the method inputs. Then, the summary effect is introduced as a set of functional *update* statements to the method outputs represented as uninterpreted functions in F. Finally, the desired object state, ϕ_{state}, is encoded as *assertion* statements over the method outputs as well.

After the translation, the satisfaction problem of ϕ_{path} in producing ϕ_{state} is equivalent to the satisfaction problem of the constructed validating formula F. Moreover, any satisfiable assignment to F is also a satisfiable assignment to the inputs of the current method for producing ϕ_{state}. TESTGEN then uses the SMT solver to check the satisfiability of the constructed formula F.

Figure 10 presents an example of a validating formula. This formula validates the feasibility of method *MKMap.MKMap(LMap)* in Figure 8 to produce the desired object state S1-1. Lines 01 - 06 define the method inputs in the form of basic variables and uninterpreted functions of the formula. Line 07 models the method path condition as an *assertion* statement over the method input *v1*. Line 08 models the summary effect with a functional *update* statement. It means that the effect of this method path is to assign *v2* to *v1.map*. Finally, Lines 09 - 11 model the desired object state S1-1 using a set of *assertion* statements over the method outputs.

```
01 (define v1::MKMap)
02 (define v2::LMap)
03 (define map::(-> MKMap LMap))
04 (define size::(-> LMap int))
05 (define data::(-> LMap Object[]))
06 (define length::(-> Object[] int))
07 (assert (/= v1 null))
08 (define map@2::(-> MKMap LMap) /* effect */
   (update map v1 v2))
09 (assert (= typeOf(map@2 v1) LMap))
10 (assert (= size (map@2 v1) 1))
11 (assert (= length (data (map@2 v1)) 1))
```

Fig. 10. A validating formula example

5 EXPERIMENTAL EVALUATION

5.1 Experimental Setup

The objective of the evaluation is to investigate the following research questions:

RQ1 Exploitability: How many crashes can STAR exploit based on crash stack traces?

RQ2 Usefulness: How many of the exploitation from RQ1 are useful crash reproductions for debugging?

5.1.1 Subjects

We used three open source projects as the subjects in this evaluation: apache-commons-collections [2] (ACC), apache-ant [1] (ANT) and apache-log4j [3] (LOG). ACC is a data container library that implements additional data structures over JDK. ANT is a Java build tool that supports a number of built-in and extension tasks such as compile, test and run Java applications. LOG is a library that improves runtime logging for Java programs. All three subjects are real world medium size projects with 25,000, 100,000 and 20,000 lines of code respectively.

ACC, ANT and LOG were chosen because their bug tracking systems (i.e., Bugzilla and JIRA) were well maintained and many bug reports contained crash stack traces along with detailed bug descriptions. In addition, these subjects have also been commonly used in the literature [22], [25], [32], [46].

All experiments were run on an Intel Corei7 machine running Windows 7 and Java 6 with 1.5GB of maximum heap space.

5.1.2 Bug Report Collection and Processing

We extracted bug report[1] information, such as the ids of the bugs, the affected program versions, and their corresponding crash stack traces.

In our experiments, we used all bug reports except the following: (1) not fixed bug reports because we could not determine whether they were real bugs; (2) bug reports which did not contain any crash stack traces or provide sufficient information (e.g. crash triggering inputs) for manual reproduction because STAR requires stack traces to reproduce crashes; (3) bug reports with invalid crash stack traces. To examine the validity, we extracted and checked if the methods in the crash stack traces actually exist in the specified lines in the program. The main cause of the occurrences of invalid crash stack trace is the incorrect version numbers reported by the users; (4) finally, STAR currently accepts crashes caused by three types of exceptions: a) *Explicitly thrown exception*, b) *NullPointerException* and c) *ArrayIndexOutOfBoundsException*. We chose these three types of exceptions because they are the most common types of exceptions [41] in Java.

We have collected 12 bug reports from ACC, 21 from ANT and 19 from LOG for the experiments. Table 2 presents statistics of the collected bug reports. In the table, columns 'Versions', 'Avg. Fix Time' and 'Bug Report Period' list the versions, the average bug fixing time and the bug creation dates for the collected bug reports. Columns 'Method Size' and 'Method Size (All)' present the average size of the crashed methods and the average size of all methods in the crash stack traces for the bug reports. The full list of bug reports used in this paper is available at https://sites.google.com/site/starcrashstack/.

TABLE 2
Bug reports collected from ACC, ANT and LOG

Name	Verions	Method Size	Method Size (All)	Avg. Fix Time	Bug Report Period
ACC	2.0 - 4.0	14.9	13.6	42 days	Oct 03 - Jun 12
ANT	1.6.1 - 1.8.3	30.4	36.9	25 days	Apr 04 - Aug 12
LOG	1.0.0 - 1.2.16	16.7	14.0	77 days	Jan 01 - Oct 09

We applied STAR to these collected crashes and tried to generate test cases to reproduce them. During our experiments, we set the maximum loop unrollment to 1, and the maximum depth of invocation to 10 for the crash precondition computation process. We also set the maximum computation time for this process to 100 seconds. In our experiments, further increasing these maximum values did not bring noticeable improvement to the current experiment result. As we will further discuss in Section 6.1, the experiment setting is not the major cause of irreproducible crashes.

1. bug reports for ACC: https://issues.apache.org/jira/browse/collections/, and bug reports for ANT and LOG: https://issues.apache.org/bugzilla/

5.2 Crash Exploitability

5.2.1 Criterion of Exploitability and Results

The first research question we investigated is crash exploitability. We used the following criterion [12] to determine whether a crash can be exploited by STAR:

Criterion 1. *A crash is considered exploitable if the generated test case by* STAR *can trigger the same type of exception at the same crash line.*

TABLE 3
Overall Exploitation Result

Subject	# of Crashes	# Exploitable Crashes	Exploitation Rate
ACC	12	8	66.7%
ANT	21	12	57.1%
LOG	19	11	57.9%
Total	52	31	59.6%

Table 3 presents the overall crash exploitation result of STAR. For the three subjects, STAR successfully exploited 8 crashes out of 12 for ACC, 12 out of 21 for ANT, and 11 out of 19 for LOG. Overall, STAR exploited 31 (59.6%) out of the 52 crashes.

5.2.2 Exploitation Level

Original crash stack traces collected from bug reports may have multiple stack frames as shown in Figure 11. Reproducing (exploiting) a crash from higher frame levels may increase the chance of revealing the bug. This is because if a test case can exploit the crash from frame levels higher than the frame level which the bug lies in, developers can follow its execution in a debugger and reveal how buggy behaviors are introduced. However, a crash may not always be exploited up to the highest stack frame level because of the object creation challenge [60]. In this section, we investigate the exploitation levels of crashes.

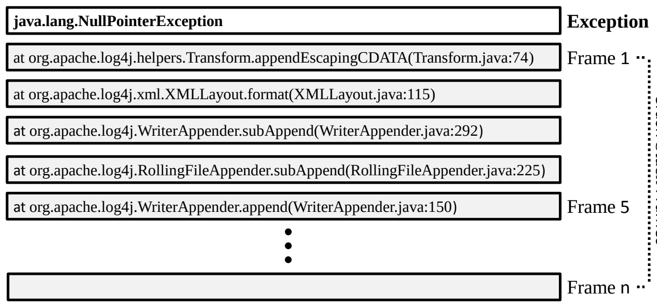
```
java.lang.NullPointerException
at org.apache.log4j.helpers.Transform.appendEscapingCDATA(Transform.java:74)
at org.apache.log4j.xml.XMLLayout.format(XMLLayout.java:115)
at org.apache.log4j.WriterAppender.subAppend(WriterAppender.java:292)
at org.apache.log4j.RollingFileAppender.subAppend(RollingFileAppender.java:225)
at org.apache.log4j.WriterAppender.append(WriterAppender.java:150)
```

Fig. 11. An example of crash stack frames

Table 4 lists all exploitable crashes with their detailed exploitation results. Column 'Prec / Exploit / Total' in the table shows 1) the highest stack frame level for which STAR successfully computed crash triggering preconditions (i.e., *Prec*), 2) the highest stack frame level for which STAR successfully generated a crash exploiting test case (i.e., *Exploit*), and 3) the total number of stack frames in the original crash stack trace (i.e., *Total*). The 'Time' column shows the total time for computing crash triggering preconditions and generating crash exploiting test cases. For example, the total number of

TABLE 4
Detailed Crash Exploitation Results

Bug id	Prec / Exploit / Total	Buggy	Usefulness	Time (s)
ACC-4	Frame 1 / 1 / 1	Frame 1	Useful	3.4
ACC-28	Frame 1 / 1 / 1	Frame 1	Useful	2.0
ACC-35	Frame 3 / 3 / 3	Frame 2	Useful	1.6
ACC-48	Frame 6 / 6 / 6	Frame 5	Useful	2.3
ACC-53	Frame 1 / 1 / 1	Frame 1	Useful	4.2
ACC-77	Frame 2 / 2 / 2	Frame 2	Not Useful	17.2
ACC-104	Frame 1 / 1 / 1	Frame 1	Useful	4.6
ACC-411	Frame 3 / 3 / 3	Frame 3	Useful	13.8
ANT-33446	Frame 3 / 1 / 9	Frame 4	Not Useful	2.6
ANT-36733	Frame 2 / 2 / 6	Frame 4	Not Useful	1.4
ANT-38458	Frame 2 / 1 / 2	Frame 1	Useful	10.9
ANT-38622	Frame 4 / 1 / 5	Frame 1	Useful	17.3
ANT-41422	Frame 2 / 2 / 2	Frame 2	Useful	20.2
ANT-43292	Frame 2 / 2 / 2	Frame 1	Useful	48.5
ANT-44689	Frame 7 / 7 / 14	N/A	Not Useful	34.0
ANT-44790	Frame 3 / 3 / 4	Frame 1	Useful	11.6
ANT-49137	Frame 2 / 1 / 4	Frame 4	Not Useful	14.4
ANT-49755	Frame 3 / 3 / 4	Frame 2	Useful	14.9
ANT-49803	Frame 4 / 4 / 4	Frame 4	Useful	13.3
ANT-50894	Frame 6 / 4 / 14	N/A	Not Useful	21.9
LOG-29	Frame 2 / 2 / 2	Frame 1	Useful	5.1
LOG-11570	Frame 10 / 10 / 10	Frame 1	Useful	9.4
LOG-31003	Frame 1 / 1 / 1	Frame 1	Useful	3.6
LOG-40159	Frame 2 / 2 / 2	Frame 1	Useful	2.3
LOG-40212	Frame 2 / 2 / 2	Frame 1	Not Useful	1.9
LOG-41186	Frame 12 / 12 / 13	Frame 3	Useful	22.3
LOG-45335	Frame 1 / 1 / 1	Frame 1	Not Useful	1.4
LOG-46271	Frame 6 / 6 / 6	Frame 1	Useful	20.5
LOG-47547	Frame 1 / 1 / 1	Frame 1	Useful	9.3
LOG-47912	Frame 3 / 3 / 5	Frame 4	Not Useful	0.7
LOG-47957	Frame 6 / 6 / 6	Frame 1	Useful	41.5

crash stack frames extracted from bug report *ANT-50894* is 14. STAR successfully computed crash triggering preconditions for this crash for up to stack frame level 6. STAR exploited this crash from stack frame level 4. The total time spent on computing the crash triggering precondition and generating the crash exploiting test case was 21.9 seconds.

Table 4 shows that 20 (64.5%) out of the 31 crashes were fully exploited (i.e. *Exploit* = *Total*). For the 11 crashes that were not fully exploited, STAR was able to exploit 7 crashes to multiple frame levels. Overall, 27 (87.1%) of the 31 exploited crashes were fully or multiple frame-level exploitation.

In terms of time spent, STAR could exploit crashes within a reasonable time. For 16 out of the 31 exploited crashes, STAR was able to exploit them in less than 10 seconds. On average, each crash was exploited in 12.2 seconds.

5.2.3 Test Case Generation

As shown in Table 4, STAR's test case generation approach is very effective for generating crash exploiting test cases from crash triggering preconditions. For 26 (83.9%) out of the 31 exploited cases, STAR was able to generate crash exploiting test cases for up to the same levels as the crash triggering preconditions (i.e., *Prec* = *Exploit*). This means that once a crash triggering precondition is computed, STAR's test case generation approach can effectively compose the necessary method sequences for generating test inputs that exploit the crash.

Table 5 lists further detailed statistics for the test case generation process. Column 'Avg. Objects' shows the average number of input objects required to exploit crashes for the three subjects. Column 'Avg. Candidates' shows the average number of method candidates considered during the test case generation. Column 'Avg. (Min - Max) Sequence' shows the average, minimum and maximum lengths of the generated crash exploiting sequences. Overall, for each crash, it required an average of 1.5 input objects to be created. During the method sequence generation process, an average of 21.4 method candidates had been considered. Finally, the average length for the generated method sequences was 7.7 lines (irrelevant lines such as method headers and import declarations, etc. were all excluded).

TABLE 5
Statistics of Test Case Generation

Subject	Ave. Objects	Avg. Candidates	Ave. (Min - Max) Sequence
ACC	1.5	35.5	9.4 (2 - 19)
ANT	1.4	11.7	6.2 (2 - 14)
LOG	1.5	21.8	8.1 (2 - 17)
Total	1.5	21.4	7.7 (2 - 19)

5.2.4 Optimizations

To investigate the effectiveness of STAR's optimization approaches (Section 3.2), we compared the crash exploitability of STAR with and without the optimizations.

Table 6 presents the detailed results. Columns 'Lv' in the table show the highest stack frame levels for which STAR successfully generated crash exploiting test cases. In total, STAR's optimizations improved the highest crash exploiting stack frame levels for 16 cases compared to STAR without optimizations. For cases such as *LOG-41186* and *LOG-46271*, the optimizations significantly improved the highest crash exploiting stack frame levels by 5 and 6 levels respectively.

Columns 'Time' present the crash precondition computation time (in second) for the highest stack frame levels. For example, for *ANT-44689*, STAR with optimizations took 4.6 seconds to compute the crash triggering preconditions for stack frame level 7. However, without optimizations, STAR was not able to compute any crash triggering precondition for level 7 within the maximum time limit (denoted as *100* in Table 6). For each crash, we compared the computation time for the same (highest) stack frame level only because comparing the computation time for different stack frame levels is meaningless. In total, STAR's optimizations reduced the crash precondition computation time for 27 cases. Among them, 22 cases that are highlighted in bold have achieved significant improvements.

After comparing the time spent, we further studied three important numbers during the computation: the number of paths considered, the number of SMT solver checks performed, and the number of conditional blocks encountered. Since these three numbers are closely related to the scalability of the symbolic execution, they are useful indicators of the effectiveness of STAR's optimization approaches. Columns 'Path', 'Check' and 'Block' list the three numbers with and without optimizations. In total, for 25 out of the 31 cases, at least one of the three numbers (for 19 cases, all three numbers) have been significantly reduced (highlighted in bold in the table) after applying STAR's optimizations. On average, the number of paths considered, the number of SMT solver checks

TABLE 6
STAR with / without Optimizations

	Without Optimizations								**With Optimizations**										
Bug id	Lv	Path	Check	Block	Field	Array	Call	Time	Lv	Path	Check	Block	Field	Array	Call	Prune	Back	Recog	Time
ACC-4	1	32	32	43	356	8	222	3.6	1	**10**	**16**	**6**	**63**	**3**	**39**	3.3	3.0	2.3	**2.0**
ACC-28	1	12	12	8	55	3	12	0.7	1	10	13	6	45	3	10	0.6	0.5	0.6	0.6
ACC-35	3	2	2	0	6	0	6	0.4	3	2	2	0	6	0	6	0.4	0.4	0.4	0.3
ACC-48	6	2	2	0	4	0	10	0.7	6	2	2	0	4	0	10	0.7	0.7	0.7	0.7
ACC-53	1	18	18	18	230	52	0	1.2	1	**11**	18	**7**	**119**	31	0	1.3	0.8	1.3	1.0
ACC-77	2	2654	2654	2171	9448	216	5762	39.7	2	**10**	**20**	**10**	**54**	**0**	**36**	**2.2**	**20.6**	**3.4**	**2.2**
ACC-104	1	18	18	16	230	34	88	1.8	1	**11**	18	**6**	**119**	20	49	1.8	1.2	1.6	1.3
ACC-411	2	533	533	1007	11609	508	7241	100	3	**45**	**104**	**93**	**550**	**63**	**347**	**84.4**	100	100	**8.4**
ANT-33446	1	544	544	1067	4500	2383	3789	100	3	**20**	**37**	**28**	**44**	**2**	**97**	100	100	100	**2.4**
ANT-36733	2	2	2	0	1	0	4	0.3	2	2	2	0	1	0	4	0.3	0.3	0.3	0.3
ANT-38458	1	848	848	586	11659	1316	11511	100	2	**11**	**40**	**21**	**46**	**0**	**61**	**5.9**	100	100	**5.9**
ANT-38622	2	1793	1793	1472	9399	2891	4358	100	4	**3**	**11**	**18**	**77**	**2**	**61**	100	**5.0**	100	**6.2**
ANT-41422	2	648	648	654	5576	24	5712	84.2	2	**2**	**7**	**5**	**16**	**0**	**23**	**2.8**	73.8	**41.0**	**2.8**
ANT-43292	0	689	689	525	7943	322	7381	100	2	**11**	**33**	**18**	**22**	**0**	**25**	100	100	100	**4.3**
ANT-44689	3	468	468	524	4894	22	6765	100	7	**9**	**26**	**21**	**149**	**5**	**115**	100	100	100	**4.6**
ANT-44790	2	1306	1306	1046	13185	196	4906	100	3	**5**	**18**	**12**	**40**	**0**	**20**	100	100	100	**1.7**
ANT-49137	1	1292	1292	207	15078	11661	4786	100	2	**211**	**226**	**278**	**2843**	**1667**	**1217**	100	**18.9**	100	**13.3**
ANT-49755	1	318	318	639	4217	1027	3125	100	3	**14**	**39**	**41**	**115**	**4**	**64**	100	100	100	**3.0**
ANT-49803	3	4242	4242	2394	20707	2177	19461	100	4	**15**	**23**	**13**	**69**	**3**	**71**	**1.3**	100	100	**1.3**
ANT-50894	4	2	2	705	9337	851	6424	100	6	28	52	**40**	**191**	**73**	**121**	100	100	100	**13.2**
LOG-29	1	27	27	4275	19487	957	21857	100	2	**11**	**18**	**13**	**58**	**0**	**25**	**1.3**	100	100	**1.0**
LOG-11570	5	2032	2032	2697	10382	68	5065	100	10	**5**	**18**	**14**	**63**	**1**	**58**	100	100	100	**2.2**
LOG-31003	1	13	13	1	20	0	27	2.4	1	**5**	**5**	0	**3**	0	**7**	**0.6**	2.5	2.5	**0.6**
LOG-40159	2	1	1	2	21	1	17	0.4	2	1	2	2	21	1	17	0.4	0.4	0.5	0.4
LOG-40212	2	2	2	778	21227	686	14313	100	2	2	2	**0**	**2**	**0**	**4**	**0.6**	100	100	**0.6**
LOG-41186	7	693	693	1618	9577	724	5497	100	12	**28**	**61**	**37**	**179**	**20**	**120**	100	100	**25.8**	**4.5**
LOG-45335	1	1	1	6	30	2	27	0.4	1	1	2	**0**	**1**	0	**3**	0.3	0.4	0.5	0.3
LOG-46271	0	1329	1329	1221	5155	48	2450	100	6	**37**	**86**	**56**	**202**	**3**	**110**	**6.0**	**20.6**	100	**5.7**
LOG-47547	1	36	36	15	204	0	252	2.2	1	**14**	**18**	**6**	**74**	0	**86**	2.1	2.1	**0.9**	**1.0**
LOG-47912	3	2	2	0	6	0	7	0.3	3	2	2	0	5	0	7	0.3	0.3	0.3	0.3
LOG-47957	0	1939	1939	2566	11309	73	5478	100	6	**39**	**94**	**63**	**236**	**7**	**136**	100	100	100	**9.5**
Average	2.0	693.5	693.5	847.1	6640.4	846.8	4727.5	59.3	3.4	**18.6**	**32.7**	**26.3**	**174.7**	**61.5**	**95.1**	39.2	50.0	54.3	**3.3**

performed and the number of conditional blocks encountered have been significantly reduced by 674.9 (97.3%), 660.8 (95.3%) and 820.8 (96.9%), respectively.

We also present in columns 'Field', 'Array' and 'Call': the number of field accesses, array accesses and method invocations during symbolic execution with and without STAR's optimizations. The results in these columns also show similar significant improvements after applying the optimizations.

To investigate the improvement coming from each individual optimization, we further studied the performance of crash precondition computation by using only one optimization at a time. Columns 'Prune', 'Back' and 'Recog' list the detailed results. The numbers in these three columns correspond to the time spent (in second) by allowing only *static path pruning*, *guided backtracking* or *intermediate contradiction recognition* respectively. In general, static path pruning was the most effective optimization, followed by guided backtracking and intermediate contradiction recognition. The three optimizations achieved significant performance improvements (highlighted in bold in the table) for 9, 4 and 4 cases (13 distinct cases) respectively. However, the combination of the three optimizations (i.e. the 'Time' column) achieved much higher improvements (22 distinct cases) than any of them individually.

Overall, Table 6 demonstrates the effectiveness of our optimizations to compute crash preconditions and create crash exploiting test cases in higher-level stack frames.

5.3 Usefulness

5.3.1 Criterion of Useful Reproduction

Not every crash exploitation by STAR might be considered a useful reproduction of the original crash for debugging. For example, suppose a crashed method was purposely designed not to take any *null* value as parameters. In such a case, exploiting the crash by passing a *null* value to this method is not useful to reveal the underlying crash triggering bug. The more important question is, where did the crash triggering *null* value come from? Therefore, a crash exploitation would only be considered a useful reproduction if it can reveal the origin of the *null* value passed to this method.

Therefore, we study how many crash exploitations by STAR are useful reproductions of the original crashes for debugging using the following criterion:

Criterion 2. *A crash exploitation is considered to be a useful reproduction if it can reveal the actual bug that causes the original crash.*

We manually examined each crash exploitation from Section 5.2 to investigate if it can reveal the crash triggering bug. To reveal a bug, the exploited crash stack trace should include the buggy frame (i.e., the stack trace frame which the buggy method lies in). To determine the buggy frame level for each crash, we carefully inspected the actual patch linked to the corresponding bug report and identified the bug fixing locations. For example, the buggy frame level for *ACC-35* is 2, and STAR exploited this crash from stack frame level 3.

In addition, to be a useful reproduction of the original crash, the crash exploiting test case by STAR should construct test inputs that trigger buggy behaviors (e.g., crashes immediately or generates corrupt values that eventually cause a crash) in the buggy location. Otherwise, the exploitation by this test case would not be considered a useful reproduction. For example, for *LOG-40212*, although STAR could exploit the crash from stack frame level 2, which includes the buggy frame, it was not considered a useful reproduction because it did not demonstrate how the incorrect *null* value could be constructed for the buggy method.

5.3.2 Result

Table 4 columns 'Buggy' and 'Usefulness' present the investigation results. Column 'Buggy' in the table shows the stack frame levels which the corresponding buggy methods lie in. Column 'Usefulness' presents our manual investigation results. We manually inspected and marked each crash exploitation as *Useful* or *Not Useful* based on Criterion 2.

In total, 22 (42.3%) out of the 31 crash exploitations were evaluated as *Useful* reproductions, since they could reveal the bugs that caused the original crashes. For *ANT-44689* and *ANT-50894*, their bug locations were not in any of the methods in their crash stacks, so they were marked as *N/A* in the 'Buggy' column and their exploitations were identified as *Not Useful*.

Different crashes might be caused by the same bug. Therefore, we also investigated how the useful crash reproductions by STAR were distributed across distinct bugs. We manually examined all useful crash reproductions and their corresponding bug fixes. Overall, 21 out of the 22 useful crash reproductions were caused by distinct bugs. *LOG-47957* was the only exception as it shared the same bug fix with *LOG-46271*. However, since *LOG-47957* and *LOG-46271* were reported from different program versions, their crash reproducing test cases were still different from each other.

5.3.3 Reproduction Examples

To demonstrate the capability of STAR in generating useful crash reproductions, we present several examples:

Example 1 (ACC-411) Figure 12 (a) shows the buggy method *ListOrderedMap.putAll* in *ACC-411*[2]. The source code has been slightly modified for better readability[3]. This method is buggy because the *index* variable at line 246 is always increased after the invocation to the *put* method. However, by design, *index* should not be increased if the *put* method had only *replaced* an existing entry in the map instead of *adding* a new entry. The incorrect increment to *index* can cause an undesired *IndexOutOfBoundsException* in certain situations as reported in *ACC-411*.

To fix this bug, the project developers applied a fix as shown in Figure 12 (b). Essentially, this fix tries to find out if method *put* has replaced an existing entry by checking whether its return value is *null*. Unfortunately, this fix is actually *not correct* as it did not consider the case where the value of the replaced entry could also be *null*. According to the Java specification, the *put* method of any *Map* class should return the value of the replaced entry or *null* if no entry had been replaced. Therefore, in the case where an entry with a *null* value is replaced, the *put* method will still return *null*, causing *index* to be incorrectly increased in the fixed version.

By applying STAR, a test case was automatically generated as shown in Figure 12 (c). The generated test case can trigger an undesired *IndexOutOfBoundsException* in both the original and the fixed program. We reported[4] this bug to the developers along with the test case generated by STAR. The developers quickly confirmed and fixed the reported bug with the help of the submitted test case. Figure 12 (d) presents the final fixed code which uses the *containsKey* method to check the existence of any entry having the same key as the input entry before performing *put*. To help test the future releases of the project, the crash reproducible test case generated by STAR was added[5] to the official test suite of the project by the developers.

This example demonstrates that, STAR can not only help the debugging process of the reported software crashes, but also help discover incomplete or incorrect bug fixes by the developers.

```
243  public void putAll(int index, Map map) {
244    for (Map.Entry entry : map.entrySet()) {
245      put(index, entry.getKey(), entry.getValue());
246      ++index; // buggy increment
247    }
248  }
```

(a) a buggy method in ACC 4.0-r1351903: the *index* variable at line 246 might be incorrectly increased which could lead to an *IndexOutOfBoundsException*.

```
243  public void putAll(int index, Map map) {
244    for (Map.Entry entry : map.entrySet()) {
245      V old =put(index,entry.getKey(),entry.getValue());
246      if (old == null) {
247        // if no key was replaced, increment the index
248        index++;
249      } else {
250        // otherwise put the next item after the ...
251        index = indexOf(entry.getKey()) + 1;
252      }
253    }
254  }
```

(b) The first (incorrect) fix applied by the project developers.

```
01  public void test0() throws Throwable {
02    Object key1 = new Object();
03    Object key2 = new Object();
04    HashMap map = new HashMap();
05    map.put(key1, null);
06    map.put(key2, null);
07    ListOrderedMap listMap = new ListOrderedMap();
08    listMap.put(key1, null);
09    listMap.put(key2, null);
10    listMap.putAll(2, map);
11  }
```

(c) automatically generated test case by STAR which can trigger undesired crashes in both the original and the first fix version of the code

```
244  public void putAll(int index, Map map) {
245    for (Map.Entry entry : map.entrySet()) {
246      K key = entry.getKey();
247      boolean contains = containsKey(key);
250      put(index, entry.getKey(), entry.getValue());
251      if (!contains) {
252        // if no key was replaced, increment the index
253        index++;
254      } else {
255        // otherwise put the next item after the ...
256        index = indexOf(entry.getKey()) + 1;
257      }
258    }
259  }
```

(d) The final fix applied by the project developers.

Fig. 12. A buggy method from ACC and a test case generated by STAR.

2. https://issues.apache.org/jira/browse/collections-411
3. The complete source code for *ListOrderedMap* is available at http://www.cse.ust.hk/~ning/star/ACC411-ListOrderedMap.java.
4. https://issues.apache.org/jira/browse/collections-474
5. http://svn.apache.org/r1496168

```
 92  public UnboundedFifoBuffer(int initialSize) {
 96    buffer = new Object[initialSize + 1];
 97    head = tail = 0;
 99  }

168  public boolean add(Object obj) {
195    if (++tail >= buffer.length)
196      tail = 0;
199  }

221  public Object remove() {
231    head += 1;
238  }

273  public Iterator iterator() {
274  return new Iterator() {
276    private int index = head;
277    private int lastReturnedIndex = -1;

284    public Object next() {
288      lastReturnedIndex = index++;
291    }

293    public void remove() {...
294      if (lastReturnedIndex == -1) {...}
299      if (lastReturnedIndex == head) {...}
306      int i = lastReturnedIndex + 1;
307      while (i != tail) {
308        if (i >= buffer.length) {
309          buffer[i - 1] = buffer[0];
310          i = 0;
311        } else {
312          buffer[i - 1] = buffer[i]; // AIOBE
313          i++;
314        }
315      } ...
321    }
323  };
324  }
```

(a) buggy methods in ACC 3.1: the array access operation at line 312 could raise an *ArrayIndexOutOfBoundsException* when *i* is 0

```
01  public void test0() throws Throwable {
02    UnboundedFifoBuffer v1 = new UnboundedFifoBuffer(3);
03    v1.add(new java.lang.Object());
04    v1.add(new java.lang.Object());
05    v1.add(new java.lang.Object());
06    v1.remove();
07    v1.add(new java.lang.Object());
08    v1.remove();
09    v1.add(new java.lang.Object());
10    java.util.Iterator v2 = v1.iterator();
11    v2.next();
12    v2.next();
13    v2.remove(); // crashed method
14  }
```

(b) automatically generated test case by STAR

Fig. 13. Buggy methods from ACC and a test case generated by STAR.

Example 2 (ACC-53) Figure 13 (a) shows the methods related to bug *ACC-53*[6]. The source code has been slightly modified to show only the code snippets that are related to this bug[7]. The *while* loop from line 307 to 314 in method *UnboundedFifoBuffer.iterator.remove()* is buggy, and could cause an *ArrayIndexOutOfBoundsException* at line 312 when *i* is 0. For example, after executing line 308 to 310, variable *i* is set to 0. At this time, if field *tail* is not 0 and *buffer.length* is greater than 0, the *while* loop is continued, and line 312 is executed. Because *i* has already been set to 0 in the last loop execution, an *ArrayIndexOutOfBoundsException* is raised at this line.

Figure 13 (b) presents the test case generated by STAR, which exploits the crash in (a). The crash exploitation by this test case can be considered a useful reproduction of the original crash for debugging because it contains the detailed method invocation steps to re-create the object state that causes the program to crash. After executing line 12 of the test case, the object state for *v2* is: *{head == 2, tail == 1, buffer.length == 4, lastReturnedIndex == 3}*. This object state satisfies the precondition for triggering the *ArrayIndexOutOfBoundsException* in the *remove* method at line 13. By following this method sequence, developers can easily find out that the program code from line 307 to 314 is buggy, as it did not consider the case of entering the loop after *i* had been reset to 0. Since the generated test case clearly demonstrates this crashing scenario, developers can quickly determine the bug. After understanding the cause of this bug, it is easy to fix it.

Even though the total number of stack frame levels for this crash is only one, it is not easy to reproduce this crash as it only happens in certain corner situations. In fact, the generated test case by STAR is the minimum test case that can reproduce the crash and reveal the crash triggering bug.

To further confirm the usefulness of this test case, we sent it to the developer who fixed this bug. The developer confirmed the usefulness of this test case with the reply, *"The auto-generated test case would reproduce the bug. …I think that having such a test case would have been useful."*

Example 3 (ANT-49755) Figure 14 (a) shows the buggy method *FileUtils.createTempFile* in *ANT-49755*[8]. This method is buggy because it does not consider the case where parameter *prefix* is null. After invoking the *File.createTempFile* method with a null *prefix* in line 888, a *NullPointerException* is raised.

```
881  public File createTempFile(String prefix, ...) {...
886    if (createFile)
887     try {
888       result = File.createTempFile(prefix, ...); // NPE
      ...
906  }
```

(a) buggy method in ANT 1.8.1: *File.createTempFile* invocation at line 888 could raise a *NullPointerException* if parameter *prefix* is *null*

```
01  public void test0() throws Throwable {
02    java.lang.String v1 = new java.lang.String();
03    java.io.File v2 = new java.io.File(null, v1);
04    java.lang.String v3 = String.valueOf(new char[1]);
05    TempFile v4 = new TempFile();
06    v4.setProperty(v3);
07    v4.setDestDir(v2);
08    v4.setCreateFile(true);
09    v4.execute();
10  }
```

(b) automatically generated test case by STAR

Fig. 14. A buggy method from ANT and a test case generated by STAR.

Figure 14 (b) presents the test case generated by STAR, which exploits the crash in (a). This test case first prepares a *TempFile* object. Then, the crashing method *TempFile.execute* is invoked, which in turn, calls the buggy *FileUtils.createTempFile* method. The *TempFile* object is prepared from Lines 02 to 08, such that the execution with this object can reach the buggy *FileUtils.createTempFile* method with the desired *null*

6. https://issues.apache.org/jira/browse/collections-53
7. The complete source code for *UnboundedFifoBuffer* is available at http://www.cse.ust.hk/~ning/star/ACC53-UnboundedFifoBuffer.java
8. https://issues.apache.org/bugzilla/show_bug.cgi?id=49755

```
172  public String encode(String value) {
173    StringBuffer sb = new StringBuffer();
174    int len = value.length(); // NPE
       ...
207  }
```

(a) a crashed method in ANT 1.6.2: the dereference at line 174 could raise a *NullPointerException* if parameter *value* is *null*

```
01  public void test0() throws Throwable {
02    DOMElementWriter v1 = new DOMElementWriter();
03    v1.encode(null);
04  }
```

(b) automatically generated test case by STAR

Fig. 15. An example of not useful test case generated by STAR.

value for parameter *prefix*. Although STAR exploits this crash only up to stack frame level 3 (the full stack has four frame levels), the crash exploitation by this test case is adequate to reveal the bug because it constructs a concrete scenario that demonstrates the incorrect behavior (i.e., passing a null *prefix* to *File.createTempFile*) in *FileUtils.createTempFile*. Therefore, it is considered a useful reproduction of the original crash.

Example 4 (ANT-33446) Figure 15 is a crash exploiting test case generated by STAR for crash *ANT-33446*[9]. Even though this test case triggers the same exception at the same line as the original crash, it is actually a typical example of an illegal method usage. The *DOMElementWriter.encode* method by design contract, should not be passed with a *null* parameter. Therefore, triggering a *NullPointerException* in this method by simply passing a *null* parameter does not reveal the actual bug. In fact, the actual bug that causes the original crash lies in *XmlLogger.buildFinished* method in stack frame level four. Therefore, according to *Criterion 2*, we consider this and other similar illegal method usage cases *Not Useful* because they cannot reveal the actual bugs that cause the original crashes.

5.3.4 Developer Survey

Since our usefulness evaluation may be subjective, we consulted developers to evaluate the usefulness of STAR. For each exploited crash, we sent out a survey email to developers who had fixed the corresponding bug. The survey email included the crash exploiting test case along with information about the original bug.

We have received 6 responses out of 31 (19% response rate). While the number of responses was not large enough to be conclusive, among the six responses, three test cases were confirmed as useful by the developers. For the other three cases, the developers showed interest in our system and gave us valuable feedback. For example, one developer confirmed that the test case by STAR for *ACC-35* is accurate, but the original crash can be easily debugged by just reading the exception message. In this case, a crash exploiting test case does not provide additional help. For *ANT-41422*, the developer responded that test cases for their project should generally be XML script files rather than JUnit style test cases. However, the developer confirmed that the generated test case by STAR might provide a hint to construct their XML script files for debugging. Overall, we received promising responses from developers. The complete developer responses are available at http://www.cse.ust.hk/~ning/star/.

5.4 Comparison

To further evaluate the effectiveness of STAR, we conducted a comparison study between STAR and two different frameworks, Randoop [45] and BugRedux [33].

Randoop is a state-of-the-art feedback-directed random test input generation framework. It can generate thousands of random method sequences to achieve high structural coverage and reveal bugs. We applied Randoop to generate method sequences to reproduce our 52 subject crashes. Randoop was executed with its default settings except for the *timeout* option and the *classlist* option. To increase the probability for Randoop to generate crash reproducible sequences, we set its *timeout* option to 1000 seconds while the default value is only 100 seconds. Further increasing the *timeout* option value does not bring improvements to the results. Furthermore, we reduced the search scope of Randoop by providing the set of classes that is relevant to the target crash using the *classlist* option. Only the classes which appear in the crash stack trace are considered relevant and provided to Randoop. In this way, Randoop can focus on generating method sequences for only the crash related classes. Essentially, our setting favors Randoop over STAR.

The second framework used in our comparison study is BugRedux [33]. Similar to STAR, BugRedux supports failure reproductions leveraging symbolic execution and constraint solving techniques. BugRedux has several different variants which use different levels of crash information (i.e. the crash location, the crash stack trace, the call sequence and the complete execution trace) to reproduce crashes. Since the call sequence information and the complete execution trace were not available without performing runtime monitoring of the original program executions, we chose the BugRedux variant[10] that makes use of the crash stack trace information for our comparison study. BugRedux was applied to generate test cases for each level of the crash stack frames using the same settings as STAR.

Figure 16 presents the crash reproduction results achieved by Randoop, BugRedux and STAR for the 52 crashes. The *Precondition* category shows the number of crashes whose crash triggering preconditions have been computed. Since Randoop does not support crash precondition computation, its value is denoted as *N/A*. The *Exploitation* category shows the number of crashes successfully exploited by the three frameworks according to *Criterion 1*. The *Usefulness* category shows the number of crash exploitations which were identified as useful reproductions according to *Criterion 2*.

Overall, STAR outperformed both frameworks in all categories. STAR outperformed Randoop in both the crash *exploitation* and *usefulness* categories by 19 and 14 more crashes respectively. The underlying reason is that, Randoop uses a randomized approach to construct method sequences. Due to the large search space of real world programs, the probability for a randomized approach to generate non-trivial

9. https://issues.apache.org/bugzilla/show_bug.cgi?id=33446

10. We re-implemented it using Java as it was originally written in C/C++.

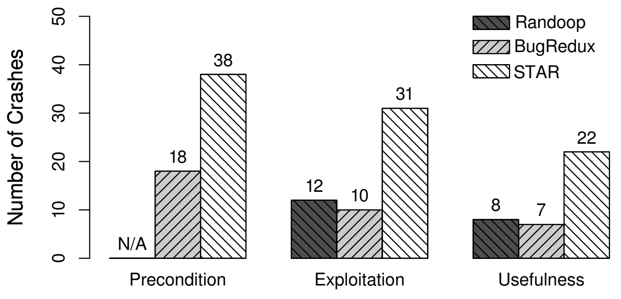

Fig. 16. Crash reproduction results achieved by Randoop, BugRedux and STAR

inputs satisfying desired crash triggering object states is low. However, it is worth noting that, although Randoop is capable of reproducing crashes, it is not a framework that is specially designed for crash reproduction. Instead, Randoop is a well-engineered test generation framework that can scale to very large programs and has been widely used in practice.

Similar to Randoop, STAR also outperformed BugRedux in the three categories by 20, 21 and 15 more crashes respectively. STAR was able to achieve better results than BugRedux for two main reasons. First, STAR leverages several effective optimizations to improve the efficiency of the crash precondition computation process. Therefore, STAR could successfully compute crash preconditions which were too complex for BugRedux. Second, STAR introduces a novel test input generation approach that can compose complex method sequences for generating crash reproducible input objects. As a result, crashes which required non-trivial input objects to trigger could be reproduced by STAR.

Overall, the comparison study demonstrates that STAR can effectively outperform Randoop and BugRedux in crash reproduction using stack traces.

6 DISCUSSION

6.1 Major Challenges

STAR successfully exploited 31 crashes, among which, 22 were considered useful for revealing the crash triggering bugs. However, there were still 30 crashes that were either not exploited or their exploitations were not considered useful. To investigate the major challenges for reproducing these crashes, we manually inspected each crash.

Table 7 presents the major challenges which we have identified for the not reproduced crashes. In total, seven major kinds of challenges have been identified. The numbers of occurrences of each challenge for the three subject programs are also listed in the table.

Environment dependency The most common challenge we identified is the dependency of the crashes on the external environment, such as file or network inputs. Crashes that depend on the external environment are difficult to reproduce because STAR currently does not support the modification of the external environment. This challenge accounts for 11 (36.7%) out of the 30 crashes that could not be reproduced.

TABLE 7
Major Challenges for Crash Reproduction

Challenge	ACC	ANT	LOG	Total
Environment Dependency	0	8	3	11 (36.7%)
SMT Solver Limitations	2	2	3	7 (23.3%)
Concurrency	1	0	4	5 (16.7%)
Path Explosion	2	0	0	2 (6.7%)
Reflection	0	1	0	1 (3.3%)
Exception Handling	0	1	0	1 (3.3%)
Other	0	2	1	3 (10.0%)

The environment dependency challenge has also been identified in related studies [55] as the primary reason for low test coverage.

SMT solver limitations The second most common challenge we identified is the limitations of SMT solver. During both the crash precondition computation and the test case generation processes, the SMT solver is used to validate the satisfactions of different formulae. However, current state-of-the-art SMT solvers such as Yices [28] and Z3 [26] have only limited support for theories like non-linear arithmetic and floating-point arithmetic. Therefore, crashes related to these arithmetics may not be reproduced because of the limitations of the SMT solver. Moreover, crashes related to string regular expressions are also difficult to reproduce as they usually introduce complex string constraints which cannot be handled by the SMT solver. Recent study by Erete et al. [29] also demonstrates weaknesses in SMT solvers when used in symbolic execution. In total, 7 (23.3%) not reproduced crashes fall within this category.

Concurrency and non-determinism STAR currently is designed to reproduce single-threaded deterministic crashes. However, there are also many crashes which are only reproducible under concurrent or non-deterministic executions. Therefore, the extension of STAR to support concurrent and non-deterministic crashes is an important future work. In total, 5 (16.7%) not reproduced crashes fall within this category.

Path explosion Both the crash precondition computation process and the test case generation process may suffer from the path explosion problem. Although STAR leverages different optimization heuristics to improve the efficiency of both processes, there are still crashes too complex to reproduce. The path explosion problem may be alleviated by adding more computation resources or introducing additional optimization heuristics, but it is difficult to solve completely. In total, 2 (6.7%) not reproduced crashes fall within this category.

Other challenges Other than the previous common challenges, there are also challenges which are less common but worth noting. The first one is Reflection. Crashes related to the Reflection mechanism are usually difficult to reproduce because the specific classes or methods triggering these crashes can only be determined during runtime. The second challenge worth noting is exception handling. It means that a crash can only be triggered if a particular exception has been previously triggered.

6.2 Future Work

To improve the effectiveness of STAR, we identify several important future work directions.

The first potential future work for STAR is to add the ability to modify the external environment. Currently, many crashes are not reproduced because they have dependencies to the external environment such as the content of a file or the state of a network connection. To reproduce these crashes, STAR needs to be able to modify the external environment according to the crash triggering preconditions.

Another potential future work for STAR is to integrate more specialized constraint solvers to overcome some of the weaknesses of Yices. For example, for crashes involving non-trivial string regular expressions or complex string operations, Yices may not be able to compute a feasible input model for reproducing the crashes. Therefore, the integration of a more specialized string constraint solver such as HAMPI [34] or Z3-str [64] may greatly improve the reproducibility of STAR for this kind of crashes.

Currently, STAR only supports three major types of exceptions. To extend STAR's applicability, we plan to add support for more exception types in the future versions. In general, it is harder to support exceptions whose initial crash conditions are difficult to infer, such as *ClassNotFoundException* and *InterruptedException*. To support these exceptions, STAR needs to have more information on the global program state and the runtime environment at the time of the crash. On contrast, for exceptions like *ClassCastException*, whose initial crash conditions are relatively easy to infer, they can be supported more easily.

Finally, as concurrent programs are becoming more popular, it is also desired that we extend STAR to support concurrent crashes.

6.3 Threats to Validity

We identify the following threats to validity of our approach and the evaluation study.

Implementation might have faults. Considering the complexity of the Java semantics, we could have introduced bugs in our implementation. We have carefully reviewed the implementation and tried to eliminate any bugs found in the code. We also welcome users to report any bugs found in STAR.

The subject systems might not be representative. In our evaluation study, we used three open source projects, ACC, ANT and LOG as subjects, because their bug reporting systems have high quality bug reports containing crash stack traces. We might have a subject selection bias. Also, since they are open source projects, they may not be representative of closed-source projects.

To mitigate the subject selection bias, we have selected three subjects with very different functionalities: data containers, build system and runtime logging. Despite the differences in functionality, all three subjects have a high quality code base and are widely used by industry. In our future studies, we may also evaluate STAR on closed-source projects.

Usefulness evaluation could be subjective. We carefully evaluated the usefulness of STAR by inspecting the generated test cases along with the actual bug patches and consulting developers. However, the evaluation could still be subjective. Therefore, we put all generated test cases online for examine.

7 Related Work

Many record-replay approaches [12], [23], [43], [50], [52] have been proposed for reproducing crashes. However, record-replay approaches yield high performance overhead because they need to monitor program executions. Therefore, these approaches try to reduce the performance overhead by reducing the amount of data needed to record during executions. For example, ReCrash [12] performs a light-weight recording of the program state during executions. Record-replay approaches for reproducing concurrency failures [9], [37], [47] try to reduce the overhead by recording only partial of the execution trace and thread scheduling. To be able to replay concurrency failures with only partial execution data, they rely on more intelligent execution replayers which employ various techniques such as data race inference [9] or probabilistic replay [47] to guide their execution replay processes. Techniques [61], [62] have also been proposed to use clues from existing application logs instead of execution traces to help diagnose crashes.

In contrast to record-replay approaches, STAR is a pure static approach. Since it does not monitor software executions, no performance overhead is incurred. Besides performance overhead, record-replay approaches need to modify the deployment environment to collect runtime information from client site to reproduce crashes. However, STAR does not require any changes to the existing deployment environment. Due to these fundamental differences, we do not compare STAR with record-replay approaches in our experiments.

Other than the record-replay approaches, post-failure-process approaches [20], [33], [38], [49], [63] have also been proposed. Post-failure-process approaches conduct analysis on crashes only after they have occurred. Since they do not record runtime information during executions, they do not incur performance overhead.

Manevich et al. proposed PSE [38], a typical post-failure-process approach which performs postmortem data-flow analysis to explain program failures with minimal information. STAR is different from PSE in two major aspects. First, STAR is able to compute the crash triggering precondition using a backward symbolic execution. PSE, however, only identifies the trace of the crash triggering value using a data-flow analysis. Second, PSE only tries to explain a failure by identifying possible error traces, while STAR aims at both identifying the complete crash path and constructing real test cases that can actually reproduce the crash.

Besides explaining failures, approaches have also been proposed to actually reproduce the crashes using various post-mortem program data. Rossler et al. proposed RECORE [49], an execution synthesis approach which analyzes the core dump (i.e. a snapshot of the stack and heap memory) extracted at the time of the crash to generate crash reproducible test cases. RECORE uses an evolutionary search-based technique to generate test cases that can reproduce the original crashes. Weeratunge et al. also proposed an approach [58] that achieves the reproduction of concurrency failures using core dumps. Leitner et al. proposed Cdd [35], [36], an approach that uses both the core dump and the developer written contracts in the program code to facilitate auto-generation of crash reproducing test cases. STAR is different from all these approaches as it does not rely on core dumps to generate crash reproducible test cases. In reality, only a limited number of applications support the generation and submission of the core dump at the time of a crash. Therefore, approaches relying on core dumps may not be widely applicable. However, in the case where both the core dump and the crash stack trace are available, these approaches and STAR may be applied together to have a better chance of reproducing the crash.

Recently, approaches that do not rely on the core dump have also been proposed. Zamfir et. al. proposed ESD, an execution synthesis approach which automatically synthesizes failure executions using only the stack trace information (though, the stack trace information is still extracted from the core dump). Jin et al. proposed BugRedux [33], an approach which aims to reproduce crashes by leveraging different levels of the execution information, including the crash stack trace. STAR is similar to these approaches in that all of them try to reproduce crashes in-house with limited information (e.g. the crash stack trace) from bug reports.

STAR is different from ESD and BugRedux in two major aspects. First, both ESD and BugRedux rely on the forward symbolic execution to achieve execution synthesis. Since a forward symbolic execution is *non-demand-driven*, it has to explore many paths that are not relevant to the target crash. Instead, STAR uses a backward symbolic execution-based approach for crash precondition computation. Since the backward symbolic execution is *demand-driven*, STAR needs to explore only paths that are relevant to the crash. Moreover, a backward symbolic execution-based approach can be more effectively optimized by approaches such as *static path pruning*. This is because most crash related information (e.g. variables contributive to the crash) near the crash location is available soon after the backward execution starts. Because of these reasons, STAR's crash precondition computation is applicable to real world complex programs with tens or even hundreds of thousands of lines of code. The second major difference between STAR and these approaches is that, both ESD and BugRedux do not handle the object creation challenge [60] which is a major challenge for reproducing object-oriented crashes. STAR introduces a novel test input generation approach that can compose complex method sequences for generating crash triggering input objects. Therefore, crashes which required non-trivial input objects to trigger, could be reproduced by STAR. Our evaluation study (Section 5.4) shows that STAR can outperform BugRedux in both the ability to compute crash triggering preconditions and the ability to generate useful crash reproducible test cases.

Generating input objects to satisfy desired object states can be achieved either by directly assigning values to the member fields or by composing method sequences to create and mutate the target objects. Boyapati et al. [17] proposed Korat, an approach to create desired input objects through direct field assignments. However, this kind of approaches relies on specifications like the class invariant information written by developers, which are rarely available.

Instead, method sequence generation approaches are more commonly used. JCrasher [24] and Randoop [45] generate random method sequences to achieve high structural coverage. Such approaches can generate large numbers of method sequences. However, due to the large search space of real world programs, the probabilities for randomized approaches to generate non-trivial input objects satisfying desired object states are low.

Thummalapenta et al. proposed Seeker [55], a method sequence generation approach that uses the combination of dynamic symbolic execution and static analysis to synthesize method sequences for achieving target object states. The method sequence composition phase of STAR is similar to Seeker in that both approaches perform backward generations of method sequences from the goal object states to initial primitive values. However, Seeker relies on the use of dynamic symbolic execution [56] to identify feasible paths to cover the intermediate goals during its generation process. Essentially, whenever a candidate sequence is suggested by the static analyzer, Seeker has to perform dynamic symbolic executions on all paths within the suggested candidate sequence until a feasible path is found. In contrast, STAR does not rely on dynamic symbolic execution to achieve method sequence composition. Instead of executing all paths in the candidate sequence both symbolically and dynamically like Seeker, STAR makes use of the precise method summary information which only needs to be computed once. The evaluation of a method path by STAR can be achieved simply by performing an SMT check on a validating formula. In general, Seeker is designed and optimized to achieve higher code coverage while STAR focuses on generating test cases to cover a particular crashing path.

Various approaches [16], [48], [53], [57] have also been proposed to improve different aspects of (dynamic) symbolic execution. For example, Boonstoppel et al. proposed RWset [16], an optimization technique for reducing the number of paths traversed by forward symbolic execution frameworks such as [18], [19]. Their technique identifies and discards paths which have the same side-effects as previously traversed paths. STAR's *static path pruning* approach is similar to RWset in that it also identifies and prunes away paths from non-contributive conditional blocks. However, STAR's approach is different from RWset in several major aspects. First, it is designed to identify and discard non-contributive conditional blocks instead of just individual paths. Second, it leverages crash related information, such as the exception triggering variables to guide the non-contributive block identification and pruning process. Finally, STAR's approach has been designed to work with backward instead of forward symbolic execution. As previously stated, backward symbolic execution is potentially more suitable for the crash precondition computation. Similarly, Taneja et al. proposed eXpress [53], an approach for improving the efficiency of regression test generation for path-exploration-based test generation techniques. STAR's *static* *path pruning* approach is also quite different from eXpress. eXpress is designed to identify and prune conditional blocks that cannot reach or infect the newly added statements in regression testing. While STAR's *static path pruning* approach prunes conditional blocks whose effects do not contribute to the target crash.

For improving constraint solving during symbolic execution, Păsăreanu et al. proposed an approach [48] to split path conditions into solvable and complex non-linear constraints that cannot be handled by the SMT solvers. Their approach uses the concrete solutions from the solvable constraints to help the solution of the complex constraints. Visser et al. [57] proposed Green, a framework for storing and reusing previous constraint solutions to speedup the solution of new constraints. Different from these approaches which focus on improving the capability or efficiency of the constraint solving process, STAR's *guided backtracking* approach aims to improve the overall traversal of the symbolic execution by leveraging the solution information from the constraint solving process.

8 CONCLUSIONS

This paper has presented STAR, an automatic crash reproduction framework using only crash stack traces. Our experiments on 52 real world crashes from ACC, ANT and LOG showed that STAR successfully exploited 31 (59.6%) of the 52 crashes. Among these exploited crashes, 22 (42.3%) were useful reproductions of the original crashes for debugging as they could reveal the actual crash triggering bugs. A comparison study between STAR and two existing approaches also demonstrated that STAR effectively outperforms these approaches in crash reproduction using only stack traces.

STAR can be applied to existing bug reporting systems without incurring additional performance overhead. Automatically reproducing crashes significantly reduces the time and effort spent by developers on crash debugging.

The implementation of STAR and the experiment data are publicly available at http://www.cse.ust.hk/~ning/star/.

REFERENCES

[1] Apache Software Foundation. Apache Ant. http://ant.apache.org/.
[2] Apache Software Foundation. Apache Commons Collections. http://commons.apache.org/collections/.
[3] Apache Software Foundation. Apache log4j. http://logging.apache.org/log4j/.
[4] Atlassian Inc. JIRA. http://www.atlassian.com/software/jira/.
[5] Google Inc. Breakpad. http://code.google.com/p/google-breakpad/.
[6] Mozilla Foundation. Talkback. http://talkback.mozilla.org.
[7] Crashreporter. Technical Report TN2123, Apple Inc., Cupertino, CA, 2004.
[8] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proc. PLDI*, pages 246–256, New York, NY, USA, 1990. ACM.
[9] G. Altekar and I. Stoica. Odr: output-deterministic replay for multicore debugging. In *Proc. SOSP*, pages 193–206, New York, NY, USA, 2009. ACM.
[10] S. Anand, P. Godefroid, and N. Tillmann. Demand-driven compositional symbolic execution. In *Proc. TACAS*, pages 367–381, Berlin, Heidelberg, 2008.
[11] L. O. Andersen. Program Analysis and Specialization for the C Programming Language. PhD thesis, University of Copenhagen, DIKU, 1994.
[12] S. Artzi, S. Kim, and M. D. Ernst. ReCrash: Making Software Failures Reproducible by Preserving Object States. In *Proc. ECOOP*, pages 542–565, 2008.
[13] D. Babic and A. J. Hu. Calysto: Scalable and precise extended static checking. In *Proc. ICSE*, pages 211–220, New York, NY, USA, 2008. ACM.
[14] K. Bartz, J. W. Stokes, J. C. Platt, R. Kivett, D. Grant, S. Calinoiu, and G. Loihle. Finding similar failures using callstack similarity. In *Proc. SysML*, pages 1–1, Berkeley, CA, USA, 2008. USENIX Association.
[15] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann. What makes a good bug report? In *Proc. FSE*, pages 308–318, New York, NY, USA, 2008. ACM.
[16] P. Boonstoppel, C. Cadar, and D. R. Engler. Rwset: Attacking path explosion in constraint-based test generation. In *Proc. TACAS*, pages 351–366, 2008.
[17] C. Boyapati, S. Khurshid, and D. Marinov. Korat: automated testing based on Java predicates. In *Proc. ISSTA*, pages 123–133, 2002.
[18] C. Cadar, D. Dunbar, and D. Engler. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. OSDI*, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.
[19] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: automatically generating inputs of death. In *Proc. 13th ACM Conference on Computer and Communications Security*, pages 322–335, 2006.
[20] S. Chandra, S. J. Fink, and M. Sridharan. Snugglebug: A powerful approach to weakest preconditions. In *Proc. PLDI*, pages 363–374, New York, NY, USA, 2009. ACM.
[21] N. Chen and S. Kim. Puzzle-based automatic testing: Bringing humans into the loop by solving puzzles. In *Proc. ASE*, pages 140–149, New York, NY, USA, 2012. ACM.
[22] H. Cibulski and A. Yehudai. Regression test selection techniques for test-driven development. In *Proc. ICSTW*, pages 115 –124, march 2011.
[23] J. Clause and A. Orso. A Technique for Enabling and Supporting Debugging of Field Failures. In *Proc. ICSE*, pages 261–270, Minneapolis, Minnesota, May 2007.
[24] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience*, 34:1025–1050, 2004.
[25] B. Daniel and M. Boshernitsan. Predicting effectiveness of automatic testing tools. In *Proc. ASE*, pages 363–366, Washington, DC, USA, 2008.
[26] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. TACAS*, pages 337–340, 2008.
[27] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.
[28] B. Dutertre and L. de Moura. System description: Yices 1.0. In *Proc. SMT-COMP*, 2006.
[29] I. Erete and A. Orso. Optimizing constraint solving to better support symbolic execution. In *Proc. CSTVA*, pages 310 –315, March 2011.
[30] 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. SOSP*, pages 103–116, New York, NY, USA, 2009. ACM.
[31] P. Godefroid. Compositional dynamic test generation. In *Proc. POPL*, pages 47–54, 2007.
[32] H. Jaygarl, S. Kim, T. Xie, and C. K. Chang. OCAT: Object Capture-based Automated Testing. In *Proc. ISSTA*, July 2010.
[33] W. Jin and A. Orso. Bugredux: Reproducing field failures for in-house debugging. In *Proc. ICSE*, June 2012.
[34] A. Kiezun, V. Ganesh, P. J. Guo, P. Hooimeijer, and M. D. Ernst. Hampi: a solver for string constraints. In *Proc. ISSTA*, pages 105–116, New York, NY, USA, 2009. ACM.
[35] A. Leitner, I. Ciupa, M. Oriol, B. Meyer, and A. Fiva. Contract driven development = test driven development - writing test cases. In *Proc. FSE*, pages 425–434, New York, NY, USA, 2007. ACM.
[36] A. Leitner, A. Pretschner, S. Mori, B. Meyer, and M. Oriol. On the effectiveness of test extraction without overhead. In *Proc. ICST*, pages 416–425, Washington, DC, USA, 2009. IEEE Computer Society.
[37] Q. Luo, S. Zhang, J. Zhao, and M. Hu. A lightweight and portable approach to making concurrent failures reproducible. In *Proc. FASE*, pages 323–337, Berlin, Heidelberg, 2010. Springer-Verlag.
[38] R. Manevich, M. Sridharan, S. Adams, M. Das, and Z. Yang. Pse: explaining program failures via postmortem static analysis. In *Proc. FSE*, pages 63–72, New York, NY, USA, 2004. ACM.
[39] J. McCarthy. A basis for a mathematical theory of computation. Technical report, MIT, Cambridge, MA, USA, 1962.
[40] J. McCarthy. Towards a mathematical science of computation. In *Information Processing 62: Proceedings of IFIP Congress 1962*, pages 21–28, Amsterdam, 1963. North-Holland.

[41] J. Nam and N. Chen. Mining crash fix patterns. Technical report, The Hong Kong University of Science and Technology, Hong Kong, 2010. http://www.cse.ust.hk/~ning/star/mining-fix-patterns.pdf.

[42] M. G. Nanda and S. Sinha. Accurate interprocedural null-dereference analysis for Java. In *Proc. ICSE*, pages 133–143, Washington, DC, USA, 2009. IEEE Computer Society.

[43] S. Narayanasamy, G. Pokam, and B. Calder. Bugnet: Continuously recording program execution for deterministic replay debugging. *SIGARCH Comput. Archit. News*, 33(2):284–295, 2005.

[44] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[45] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Proc. ICSE*, pages 75–84, Minneapolis, MN, USA, May 23–25, 2007.

[46] C.-S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *Proc. FSE*, pages 135–145, New York, NY, USA, 2008. ACM.

[47] S. Park, Y. Zhou, W. Xiong, Z. Yin, R. Kaushik, K. H. Lee, and S. Lu. Pres: probabilistic replay with execution sketching on multiprocessors. In *Proc. SOSP*, pages 177–192, New York, NY, USA, 2009. ACM.

[48] C. S. Păsăreanu, N. Rungta, and W. Visser. Symbolic execution with mixed concrete-symbolic solving. In *Proc. ISSTA*, pages 34–44, New York, NY, USA, 2011. ACM.

[49] J. Rossler, A. Zeller, G. Fraser, C. Zamfir, and G. Candea. Reconstructing core dumps. In *Proc. ICST*, pages 114–123, 2013.

[50] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst. Automatic test factoring for Java. In *Proc. ASE*, pages 114–123, November 2005.

[51] N. Serrano and I. Ciordia. Bugzilla, itracker, and other bug trackers. *IEEE Software*, 22:11–13, 2005.

[52] J. Steven, P. Chandra, B. Fleck, and A. Podgurski. jRapture: A capture/replay tool for observation-based testing. In *Proc. ISSTA*, pages 158–167, 2000.

[53] K. Taneja, T. Xie, N. Tillmann, and J. de Halleux. express: guided path exploration for efficient regression test generation. In *Proc. ISSTA*, pages 1–11, New York, NY, USA, 2011. ACM.

[54] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and W. Schulte. Mseqgen: Object-oriented unit-test generation via mining source code. In *Proc. ESEC/FSE*, pages 193–202, New York, NY, USA, 2009. ACM.

[55] S. Thummalapenta, T. Xie, N. Tillmann, J. de Halleux, and Z. Su. Synthesizing method sequences for high-coverage testing. In *Proc. OOPSLA*, pages 189–206, New York, NY, USA, 2011. ACM.

[56] N. Tillmann and J. de Halleux. Pex-white box test generation for .NET. In *Proc. TAP*, pages 134–153, 2008.

[57] W. Visser, J. Geldenhuys, and M. B. Dwyer. Green: reducing, reusing and recycling constraints in program analysis. In *Proc. FSE*, pages 58:1–58:11, New York, NY, USA, 2012. ACM.

[58] D. Weeratunge, X. Zhang, and S. Jagannathan. Analyzing multicore dumps to facilitate concurrency bug reproduction. In *Proc. ASPLOS*, pages 155–166, New York, NY, USA, 2010. ACM.

[59] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, 1984.

[60] X. Xiao, T. Xie, N. Tillmann, and J. de Halleux. Precise identification of problems for structural test generation. In *Proc. ICSE*, pages 611–620, New York, NY, USA, 2011. ACM.

[61] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy. Sherlog: error diagnosis by connecting clues from run-time logs. In *Proc. ASPLOS*, pages 143–154, New York, NY, USA, 2010. ACM.

[62] D. Yuan, J. Zheng, S. Park, Y. Zhou, and S. Savage. Improving software diagnosability via log enhancement. In *Proc. ASPLOS*, pages 3–14, New York, NY, USA, 2011. ACM.

[63] C. Zamfir and G. Candea. Execution synthesis: a technique for automated software debugging. In *Proc. EuroSys*, pages 321–334, New York, NY, USA, 2010. ACM.

[64] Y. Zheng, X. Zhang, and V. Ganesh. Z3-str: A z3-based string solver for web application analysis. In *Proc. FSE*, pages 114–124, New York, NY, USA, 2013. ACM.