Heap Cloning: Enabling Dynamic Symbolic Execution of Java Programs

Saswat Anand
Georgia Institute of Technology
Atlanta, U.S.A.
Email: saswat@cc.gatech.edu

Mary Jean Harrold
Georgia Institute of Technology
Atlanta, U.S.A.
Email: harrold@cc.gatech.edu

***Abstract*—The dynamic symbolic-execution technique can automatically perform symbolic execution of programs that use problematic features of Java, such as native methods. However, to compute precise symbolic execution, the technique requires manual effort to specify models for problematic code. Furthermore, existing approaches to perform symbolic execution either cannot be extended to perform dynamic symbolic execution or incur significant imprecision. In this paper, we present a novel program-transformation technique called heap cloning. Heap cloning transforms a program in such a way that dynamic symbolic execution of the transformed program results in the same path constraints as dynamic symbolic execution of the original program. However, symbolic execution of the transformed program produces feedback on where imprecision is introduced, and that feedback can reduce the manual effort required to build models. Furthermore, such transformation can enable existing approaches to perform symbolic execution systems to overcome their limitations. In this paper, we also present a system, called Cinger, that leverages heap cloning, and that we used to perform an empirical evaluation. The empirical evaluation shows that Cinger can compute precise path constraints, and requires little (if any) manual effort for a set of large real-world programs.**

I. Introduction

Symbolic execution [15] is a program-analysis technique that interprets a program using symbolic inputs along a path, and computes a constraint, called a *path constraint*, for that path over those symbolic inputs. If the path constraint is satisfiable, any solution of the constraint represents a program input that exercises the path. Although in theory, computing path constraints is well-understood, in practice, it is still challenging to automatically compute precise[1] path constraints for real-world programs.

To be able to automatically compute path constraints for any program written in a particular programming language, a symbolic-execution system must support every feature of that language. If a symbolic-execution system does not support some language features, manual effort is required to specify models for those parts of a program that use those features. For Java, native methods and features such as reflection that are dependent on the internals of a Java virtual machine (JVM) complicate the implementation of a symbolic-execution system. However, these features are widely-used in real-world Java programs. One study [3] shows that the number of times native methods are called by programs in the SPEC JVM98 benchmark ranges from 45 thousand to 5 million. Native methods are written in languages such as C, and thus, they do not have Java bytecode representations. One approach to implementing a symbolic-execution system that supports those features extends a standard JVM (e.g., Oracle's JVM) that already supports all features of Java. However, none of the existing symbolic-execution systems for Java are based on this approach. This approach is difficult to implement, and also difficult to maintain because the addition of new language features to the Java language and changes or additions to Java's standard library may require modifications to the JVM and the symbolic-execution system.

To address the difficulty of implementing a symbolic-execution system, researchers have developed the dynamic symbolic-execution (DSE) technique, also known as concolic execution [9], [22]. Under the DSE technique, the problematic code of a program, which cannot be symbolically executed, is executed only concretely,[2] and the rest of the program is executed both symbolically and concretely. For Java, problematic code, such as native methods and code that uses reflection, can be executed concretely on a standard JVM that already supports those features.

However, two problems arise in DSE. The first problem arises because native methods are executed only concretely. This concrete execution leads to the computation of imprecise path constraints because the effects of those methods on the path constraint and the program state are ignored. Whenever a native method updates the value of a memory location, failure to update the symbolic value corresponding to that memory location can introduce imprecision. In general, manually-specified models, which model effects of native methods, are necessary for eliminating such imprecision. However, an automatic method for identifying native methods whose side-effects may introduce imprecision, and thus, need to be modeled, can significantly reduce the required manual effort. Existing approaches for implementing DSE [9], [22] do not address this problem.

The second problem arises from limitations of the two current approaches that are used to implement (dynamic)

[1] A path constraint of a path p is *precise* if any solution of the path constraint represents program inputs that cause the program to take path p.

[2] *Concrete execution* refers to normal execution of a program with non-symbolic values.

symbolic-execution systems for Java. These approaches either cannot incorporate the DSE technique or incur imprecision. The first approach uses a custom interpreter that interprets a program according symbolic-execution semantics. Under this approach, a standard JVM cannot execute the problematic code of a program because the program's state, which is managed by the interpreter, is unfamiliar to the JVM. The second approach instruments a program, and executes the instrumented code on a standard JVM such that the instrumentation code computes the path constraints. Under this approach, although the program can be executed concretely by the JVM, computed path-constraints can be imprecise because of problems in instrumenting Java's standard library classes. These problems are illustrated in Section II.

To address the two above-mentioned problems, we developed, and present in this paper, a novel program-transformation technique, which we call *heap cloning*. The heap-cloning technique transforms the subject program P so that the transformed program P' has two properties. First, P' uses copies of the original classes produced by the heap-cloning technique. Second, during execution of P', the heap consists of two partitions: a "concrete heap" consisting of objects of the original classes, and a "symbolic heap" consisting of objects of the copied classes. Native methods of the original classes operate on the "concrete heap" and Java methods of the copied classes operate on the "symbolic heap."

One important benefit of heap cloning is that it can determine the side-effects of native methods by comparing the value of a memory location in the "symbolic heap" with the value of the corresponding location in the "concrete heap." Thus, a system that leverages heap cloning can reduce the manual effort required to specify models of native methods by automatically identifying the subset of methods that must be modeled. Another important benefit of heap cloning is that it addresses the problems that arise in implementing DSE using custom-interpreter and instrumentation approaches. Under the custom-interpreter approach, heap cloning enables execution of native methods on a standard JVM using the concrete heap. In the instrumentation approach, heap cloning eliminates the imprecision that arises due to problems with instrumenting Java's standard library classes because the transformed program use copies of library classes.

In this paper, we also present CINGER (Concolic INput GEneratoR), a system we developed that implements heap cloning, along with the results of empirical studies we performed using CINGER to evaluate the effectiveness of the heap-cloning technique. CINGER uses the program-instrumentation based approach to symbolic execution, and leverages another technique, which we developed in our prior work [1], that aims to improve the efficiency of computing path constraints. We performed our studies on five real-world example programs that extensively use Java's standard library classes and other language features, such as reflection and native methods. The first study shows that the imprecision that arises in the instrumentation approach can be eliminated by leveraging heap cloning. The second study shows that heap cloning can reduce the manual effort required to specify models for native methods that possibly introduce imprecision and that a symbolic-execution system that uses the custom-interpreter approach can be applied automatically to a larger set of programs by leveraging heap cloning.

The main contributions of the paper are

- A novel program-transformation technique, heap cloning, that reduces the manual effort required to write models for native methods, and addresses the problem that arise in implementing DSE using the two existing approaches.
- A practical DSE system, CINGER that leverages the heap-cloning technique and an optimized instrumentation approach developed in our prior work [1].
- An empirical evaluation of the technique that demonstrates that it is possible to efficiently compute precise path constraints for programs, which may use native methods and Java's standard library classes, with little (if any) manual effort.

```
1  class L{
2    int f;
3    L(int v){ this.f = v; }
4    native void complex();
5  }
6  class A{
7    L l;
8    A(int v){ this.l = new L(val); }
9    public static void main(String[] args){
10     init();
11     int i = intInput();
12     A a = new A(i);
13     L x = a.l;
14     int j = x.f + 3;
15     if(j < 0)
16       x.complex();
17     if(x.f < 0) error();
18   }
19 }
```

Fig. 1. Example Java program.

II. Background and Motivation

In this section, we first introduce the example that we use throughout the paper. We then illustrate the problems that arise in DSE using the example.

Example. Figure 1 gives an example Java program P, which consists of two classes: `A` and `L`. P inputs an integer (line 11), and performs several simple operations. The native method `complex`, which is invoked in line 16 of Figure 1, reads the current value of the field `f`, and stores its corresponding absolute value back into field `f`. For the purpose of discussion, suppose that the following Java code is equivalent to the code of the `complex` method.

```
void complex() {
  int i = this.f;
  if(i < 0) i = -i; this.f = i; }
```

Effects of native methods on precision. Suppose the program is executed with input `-10`. The intraprocedural path through the `main` method that is executed with this input is the sequence of all statements in that method. In DSE , suppose we introduce a symbol I to represent the symbolic program input, and symbolically execute along the path. As the concrete input value (i.e., `-10`) propagates through the program variables, the symbolic value is propagated similarly. For example, because the concrete input value is stored in the field `f` at line 13, the memory location corresponding to the field is mapped to symbolic value I.

$X + 3 < 0 \wedge X < 0 \wedge -X >= 0$ is the correct path constraint for this path. The first and third conjuncts[3] correspond to the branching statements of lines 15 and 17, respectively. The second conjunct corresponds the branching statement inside the `complex` method. Note that the third conjunct accounts for the side-effects of the native method `complex`, which on the current path reads the value (i.e., `-10`) of the field `f` at the time of its invocation, and then stores the absolute value (i.e., `10`) into the field `f`.

However, it is difficult to automatically compute the correct path constraint because symbolic execution of native methods is challenging, if not impossible. As a result, a DSE engine would ignore the update to field `f` that is made inside `complex`, and incorrectly compute the path constraint $X + 3 < 0 \wedge X >= 0$. This path constraint is incorrect because (1) it is missing the conjunct corresponding to the branch in `complex` method, and (2) the second conjunct in this path constraint is different from the third conjunct of the correct path constraint, even if both conjuncts are added to the respective path constraints as a result of execution of the statement in line 17.

To eliminate this imprecision, the user must provide a model for the `complex` method. Although, writing such models cannot be automated because it requires manual analysis of native methods, in many cases, it is possible to automatically identify methods whose side-effects introduce imprecision in computed path constraints. In this example, an automatic technique such as heap cloning can determine that ignoring the update to field `f` inside `complex` method can introduce imprecision because the memory location corresponding to the field is mapped to some symbolic value.

Custom interpreter approach for DSE. This approach, which is used in JFuzz [12], Symbolic Pathfinder [17], Kiasan [6], and symbolic-execution systems used in References [20], [24], uses a custom interpreter that executes the program according to the semantics of symbolic execution. It is problematic to use this approach to implement DSE because native methods cannot be concretely executed by a standard JVM . The interpreter manages the program's state and thus, a JVM cannot operate on such an external representation of program's state. Although recent work [18] demonstrated how this approach can concretely execute side-effect free native methods, handling native methods with side-effects still remains problematic.

[3]A *conjunct* is an atomic constraint that is generated as a result of different program statements that update the path constraint, such as a branch whose outcome depends on symbolic values.

For example, consider the native method `complex` in the program shown Figure 1. Recall that `complex` reads and updates fields of the receiver object on which the method is invoked. Because the method reads and writes to the program heap, it would not be possible to perform DSE using a custom-interpreter approach.

One potential solution to this problem is to translate the program state before an invocation of a native method to a representation that is familiar to a standard JVM, and after the invocation, translate the program state to the representation that the custom-interpreter uses. However, this solution may not scale well in general because, even "well-behaved" native methods can access not only the program heap that is reachable from their arguments, but also parts of heap that are reachable from any static fields. As a result, for this solution to work in general, significantly large portions of program's heap must be translated before and after calls to native methods.

Imprecision in instrumentation approach. Under this approach, a program is instrumented so that the instrumentation code maintains the symbolic program state and computes path constraints. Because the instrumented code can be executed on a standard JVM, this approach facilitates DSE. Two existing techniques are based on this approach. The first technique [14] is used in existing symbolic-execution systems such as JPF-SE [2], STINGER [1], Juzi [8], and the system used in Reference [16]. The second technique is used in existing systems such as JCUTE [21], LCT [13], and TaDa [10]. The main difference between the two techniques is the type of instrumentation they perform. However, each suffers from the imprecision problem for the same reasons. Under this approach, the instrumentation code tracks the symbolic values as they flow through the program. Thus, when symbolic values flow into Java's standard library classes, the approach requires those classes to be instrumented like the user-defined classes.

However, two problems make it difficult to instrument those library classes, and without such instrumentation, the path constraints can be imprecise. First, standard JVMs make implicit assumptions about the internal structures of the core library classes, such as `java.lang.String`. Thus, any instrumentation that violates those assumptions can cause the virtual machine to crash. For example, Sun's virtual machine will crash when fields of user-defined types are added to `java.lang.String` or when types of any fields of the core classes are changed. Second, if library classes are transformed to use some user-defined classes, those user-defined classes must not use library classes. Otherwise, the result may be non-terminating recursive calls. For example, suppose `L` is a library class, and the transformed `L` uses a non-library class `Expression`. If `Expression` internally uses class `L`, it will lead to non-terminating recursive calls. Although in theory, classes such as `Expression` can be written without using any library classes, it is cumbersome to do in practice.

Object class

$$[\![\mathtt{class\ Object}\ \{\bullet\ \mathtt{K};\ \overline{\mathtt{M}}\}]\!] = \mathtt{interface\ Object}^{\mathtt{I}}_{\mathtt{HC}} \tag{1}$$

$$= \mathtt{class\ Object_{HC}\ implements\ Object}^{\mathtt{I}}_{\mathtt{HC}}\{\mathtt{Object\ shadow;}\ \ [\![\mathtt{K}]\!]\mathtt{;}\ \ \overline{[\![\mathtt{M}]\!]}\} \tag{2}$$

Other classes

$$\left[\!\!\left[\begin{array}{l}\mathtt{class\ C\ extends\ D}\\ \quad\mathtt{implements}\ \overline{\mathtt{I}}\{\\ \ \overline{\mathtt{E}}\ \overline{\mathtt{f}};\\ \ \overline{\mathtt{K}};\\ \ \overline{\mathtt{M}}\\ \}\end{array}\right]\!\!\right] = \begin{array}{ll}\mathtt{class\ C_{HC}\ extends\ D_{HC}\ implements}\ \overline{\mathtt{I_{HC}}}\ \{ & (3)\\ \ \overline{\mathtt{E_{HC}}}\ \overline{\mathtt{f_{HC}}}; & (4)\\ \ \overline{[\![\mathtt{K}]\!]},\mathtt{C_{HC}}()\{\mathtt{r = new\ C;\ r.C(null);\ this.shadow = r;}\} & (5)\\ \ \overline{},\mathtt{C_{HC}(C\ s)}\{\mathtt{this.shadow = s;}\}; & (6)\\ \ \overline{[\![\mathtt{M}]\!]} & (7)\\ \} & (8)\end{array}$$

$$= \begin{array}{ll}\mathtt{class\ C\ extends\ D\ implements}\ \overline{\mathtt{I}},\mathtt{Wrapper;}\{ & (9)\\ \ \overline{\mathtt{E}}\ \overline{\mathtt{f}}; & (10)\\ \ \overline{\mathtt{K}},\mathtt{C(Dummy_{HC}\ d)}\{\mathtt{super(d);}\}; & (11)\\ \ \overline{\mathtt{M}},\mathtt{C_{HC}\ wrap()}\{\mathtt{r = new\ C_{HC};\ r.C_{HC}(this);\ return\ r;}\} & (12)\\ \} & (13)\end{array}$$

Constructor

$$[\![\mathtt{C}(\overline{\mathtt{D}}\ \overline{\mathtt{x}})\{\overline{\mathtt{S}}\}]\!] = \mathtt{void\ C}^{\mathtt{K}}_{\mathtt{HC}}(\mathtt{C_{HC}\ y},\ \overline{\mathtt{D_{HC}}}\ \overline{\mathtt{x}})\{\overline{[\![\mathtt{S}]\!]}\} \tag{14}$$

Non-native method

$$[\![\mathtt{D\ m}(\overline{\mathtt{C}}\ \overline{\mathtt{x}})\{\overline{\mathtt{S}}\}]\!] = \mathtt{D_{HC}\ m_{HC}}(\overline{\mathtt{C_{HC}}}\ \overline{\mathtt{x}})\{\overline{[\![\mathtt{S}]\!]}\} \tag{15}$$

Native method

$$[\![\mathtt{native\ D\ m}(\overline{\mathtt{C}}\ \overline{\mathtt{x}});]\!] = \begin{array}{ll}\mathtt{D_{HC}\ m_{HC}}(\overline{\mathtt{C_{HC}}}\ \overline{\mathtt{x}})\{ & (16)\\ \ \ \mathtt{r = m}(\overline{\mathtt{shadow(x)}}); & (17)\\ \ \ \mathtt{return\ r.wrap();}\} & (18)\end{array}$$

Allocation

$$[\![\mathtt{v = new\ C}]\!] = \mathtt{v = new\ C_{HC}} \tag{19}$$

Constructor invocation

$$[\![\mathtt{v.C}(\overline{\mathtt{D}}\ \overline{\mathtt{x}})]\!] = \mathtt{C}^{\mathtt{K}}_{\mathtt{HC}}(\mathtt{C_{HC}\ v},\overline{\mathtt{D_{HC}}}\ \overline{\mathtt{x}}) \tag{20}$$

Non-static method invocation

$$[\![\mathtt{v.m}(\overline{\mathtt{D}}\ \overline{\mathtt{x}})]\!] = \mathtt{v.m_{HC}}(\overline{\mathtt{D_{HC}}}\ \overline{\mathtt{x}}) \tag{21}$$

Static method invocation

$$[\![\mathtt{C.m}(\overline{\mathtt{D}}\ \overline{\mathtt{x}})]\!] = \mathtt{C_{HC}.m_{HC}}(\overline{\mathtt{D_{HC}}}\ \overline{\mathtt{x}}) \tag{22}$$

Store

$$[\![\mathtt{v_1.f = v_2}]\!] = \begin{array}{ll}\mathtt{v_1.f_{HC} = v_2;} & (23)\\ \sigma(\mathtt{v_1})\mathtt{.f} = \ \pi(\mathtt{v_2}); & (24)\end{array}$$

Load

$$[\![\mathtt{v_1 = v_2.f}]\!] = \begin{array}{ll}\mathtt{v = v_2.f_{HC};} & (25)\\ \mathtt{v_s} = \sigma(\mathtt{v_2})\mathtt{.f;} & (26)\\ \mathtt{if(v == v_s)\ goto\ l;} & (27)\\ \mathtt{v_2.f_{HC} = v_s.wrap();} & (28)\\ \mathtt{l : v_1 = v_2.f_{HC};} & (29)\end{array}$$

Assignment

$$[\![\mathtt{v_1 = v_2}]\!] = \mathtt{v_1 = v_2} \tag{30}$$

$$[\![\mathtt{v = null}]\!] = \mathtt{v = null} \tag{31}$$

Reference equality

$$\left[\!\!\left[\begin{array}{l}\mathtt{if(v_1 == v_2)}\\ \ \ \mathtt{goto\ l}\end{array}\right]\!\!\right] = \begin{array}{ll}\mathtt{if}(\pi(\mathtt{v_1}) == \pi(\mathtt{v_2})) & (32)\\ \ \ \mathtt{goto\ l} & (33)\end{array}$$

Cast

$$[\![\mathtt{v_1 = (C)\ v_2}]\!] = \mathtt{v_1 = (C_{HC})\ v_2} \tag{34}$$

Auxiliary functions:

$$\sigma(\mathtt{v}) = \begin{cases}\mathtt{null} & \mathtt{v == null}\\ \mathtt{v} & \mathtt{otherwise}\end{cases} \qquad \pi(\mathtt{v}) = \begin{cases}\sigma(\mathtt{v}) & \mathtt{v\ is\ of\ reference\ type}\\ \mathtt{v} & \mathtt{otherwise}\end{cases}$$

Fig. 2. Heap-cloning transformation rules. The numbers in the parentheses shown next to the rules are used to refer to corresponding lines.

III. Heap-cloning Technique

In this section, we first present the notation that we use, and then we present the heap-cloning technique. Throughout the section, we use P to represent the original program and P' to represent the transformed program produced by applying heap cloning to P.

Figure 2 presents the transformation rules of the heap-cloning technique. Our notation is inspired by that used in the specification of Featherweight Java [11]. Because of space constraints, we (1) present rules for only a representative subset of the Java language that is relevant to heap cloning and (2) do not present the syntax of the subset language.

We write $\overline{\mathtt{C}}$ as shorthand for a possibly empty sequence $\mathtt{C}_1, \ldots, \mathtt{C}_n$, and similarly for $\overline{\mathtt{F}}$, $\overline{\mathtt{x}}$, $\overline{\mathtt{S}}$, etc. We write the empty sequence as $\bullet$ and denote concatenation of sequences using a comma. We abbreviate operations on pairs of sequences by writing "$\overline{\mathtt{C}}\ \overline{\mathtt{f}}$" for "$\mathtt{C}_1\mathtt{f}_1, \ldots, \mathtt{C}_n\mathtt{f}_n$". The meta-variables `C` and `D` range over class names; `f` ranges over field names; `m` ranges over method names; `x` and `v` ranges over variables.

The class declaration "`class C extends D implements I`$_1$`,I`$_2$ $\{\overline{\mathtt{D}}\ \overline{\mathtt{f}};\ \mathtt{K};\ \overline{\mathtt{M}}\}$" represents a class named `C`. `D` is `C`'s superclass, and `C` implements interfaces $\mathtt{I}_1$ and $\mathtt{I}_2$. The class has fields $\overline{\mathtt{f}}$ with types $\overline{\mathtt{D}}$, a single constructor `K`, and a suite of methods $\overline{\mathtt{M}}$. The method declaration "`D m(`$\overline{\mathtt{C}}\ \overline{\mathtt{x}}$`){ return e; }`" introduces a method named `m` with result type `D` and parameters $\overline{\mathtt{x}}$ of types $\overline{\mathtt{C}}$. The body of the method is the single statement `return e;`. `this` represents a special variable that store the reference to the receiver of a non-static method call.

In rest of this section, we refer to the transformation rules in Figure 2 by the numbers shown in parentheses next to the

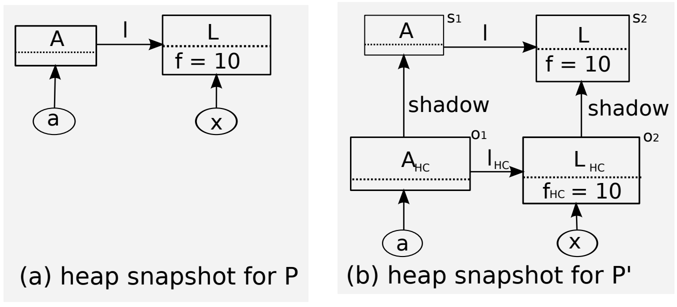


Fig. 3. Snapshots of the heaps at line 15 of (a) the original program P (Figure 1) and (b) the transformed program P', when the programs are executed with input `10`.

rules. Those numbers are shown as superscripts in parentheses (e.g., text$^{(100)}$) in the text describing the rule.

A. Creating New Classes

The heap-cloning technique creates a new class (interface) $\mathtt{A}_{\mathtt{HC}}$, which we call the *Heap Cloning* (abbreviated *HC*) class, corresponding to every class (interface) `A` of the original program $P^{(3-8)}$. Like other regular classes, the heap-cloning technique also creates HC classes corresponding to Java's built-in array classes. For example, a regular class $\mathtt{char[]}_{\mathtt{HC}}$ is created corresponding to a one-dimensional array of characters. In rest of the paper, we use the following convention: names of HC classes have HC as subscripts, and the HC class corresponding to an original class `A` is named $\mathtt{A}_{\mathtt{HC}}$.

Heap cloning aims to minimize the dissimilarity between the class hierarchy consisting of original classes and interfaces of P and the hierarchy consisting of HC classes and interfaces. The similarity between the two hierarchies facilitates type-safe transformation of the program. The inheritance relationship of HC classes and interfaces is based on the two rules$^{(3)}$: (1) if a class `A` is a subclass of `B`, $\mathtt{A}_{\mathtt{HC}}$ is a subclass of $\mathtt{A}_{\mathtt{HC}}$; (2) for each interface `I` that a class `A` implements, $\mathtt{A}_{\mathtt{HC}}$ implements the interface $\mathtt{I}_{\mathtt{HC}}$.

Figure 4 shows an example of the way in which the heap-cloning technique transforms the class hierarchy of a program. Figure 4(a) shows the original class hierarchy, consisting of a class `A`, the built-in character array class `char[]`, and the class `Object`. Figure 4(b) shows the class hierarchy consisting of the generated HC classes. A solid arrow from a class `A` to a class `B` means that class `A` extends class `B`. A dotted arrow from a class or interface `A` to an interface `B` means that `A` implements `B`. $\mathtt{Object}^{\mathtt{I}}_{\mathtt{HC}}$ is a special interface introduced by heap cloning. Every HC interface and the $\mathtt{Object}_{\mathtt{HC}}$ implement this special interface$^{(2)}$.

For every field, `f`, of an original class `A`, heap cloning adds a field $\mathtt{f}_{\mathtt{HC}}$ to $\mathtt{A}_{\mathtt{HC}}{}^{(4)}$. If the type of `f` corresponds to class `X`, then $\mathtt{f}_{\mathtt{HC}}$ has type of class $\mathtt{X}_{\mathtt{HC}}{}^{(4)}$. For each method `m` of an original class `A`, the heap-cloning technique adds a method $\mathtt{m}_{\mathtt{HC}}$ to $\mathtt{A}_{\mathtt{HC}}{}^{(14,15,16)}$. If `m` has a method body, then `m`'s body is copied into $\mathtt{m}_{\mathtt{HC}}{}^{(15)}$. If `m` is a native method, then $\mathtt{m}_{\mathtt{HC}}$ delegates the calls to $\mathtt{m}^{(16-18)}$. We discuss this delegation in the next section.

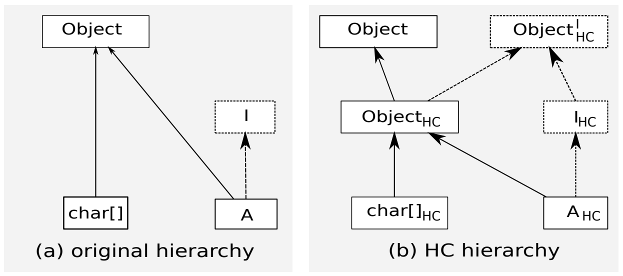


Fig. 4. Class hierarchy before and after the heap-cloning transformation.

Heap cloning transforms every invocation statement in the HC classes so that it will invoke methods of the HC classes. Thus, when the transformed program is executed, Java methods of the HC classes, and only native methods and static class initializers of the original classes, are executed.

B. Concrete and Symbolic Heap Partitions

For every object of a class `A` contained in the heap of P at some program point during the execution of P, the heap of P' contains one object o_s of class `A`. The heap of P' also contains at least one object o of class $\mathtt{A}_{\mathtt{HC}}$ at the corresponding program point in P''s execution. We refer to o_s as the *shadow object* of o. We call the partition of the heap during P''s execution that consists of objects of the original class (i.e., shadow objects) the *concrete heap partition*. We call the partition of the heap that consists of objects of HC classes the *symbolic heap partition*.

During P''s execution, for an object o and its field $\mathtt{f}_{\mathtt{HC}}$, the value of the heap location $o.\mathtt{f}_{\mathtt{HC}}$ remains consistent with the value of the heap location $o_s.\mathtt{f}$, where o_s is the shadow object of o and `f` is the shadow field of $\mathtt{f}_{\mathtt{HC}}$. The value, v of $o.\mathtt{f}_{\mathtt{HC}}$ is *consistent* with the value v_s of $o_s.\mathtt{f}$ if (1) `f` is of primitive type and $v = v_s$, or (2) `f` is of non-primitive type and v_s is shadow object corresponding to v.

Recall that during P''s execution, some code, such as native methods and class initializers, of the original classes of P get executed. Because the two heap partitions are kept consistent, that code can read from and write to the concrete heap partition. Heap cloning handles native methods as follows. If `m` is a native method, then $\mathtt{m}_{\mathtt{HC}}$ delegates the calls to `m` with arguments that are the shadow objects corresponding to the HC objects that it receives as parameters. Also, if `m` returns an heap object, $\mathtt{m}_{\mathtt{HC}}$ wraps it as an object of corresponding HC class, and returns the wrapped object. Wrapping operation is discussed later in this section. In contrast to native methods, handling class initializers do not require any method delegation because those methods are implicitly invoked by a JVM.

Heap cloning adds a special field `shadow` to $\mathtt{Object}_{\mathtt{HC}}$ that stores the shadow object corresponding to an object of a HC class$^{(2)}$. The object o stores a reference to its shadow object o_s in a designated field `shadow`. The program is transformed such that when an object o of a HC class $\mathtt{A}_{\mathtt{HC}}$ is allocated, a shadow object o_s of the corresponding original class `A` is also allocated, and is stored in the special field `shadow` of $o^{(5)}$. To allocate the shadow object, a new constructor is added to the original class $\texttt{A}^{(11)}$.

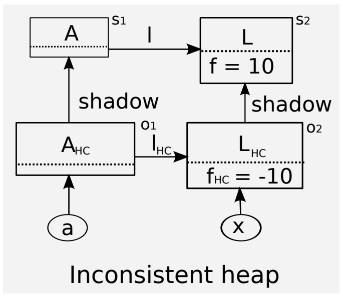
To illustrate, consider program P shown in Figure 1. P allocates an object of class `L` (line 8) and an object of class `A` (line 12). Figure 3(a) shows the snapshot of P's heap just before execution of line 15, when P is executed with input `10`. In Figure 3, each rectangle (shown with a dotted divider) represents a heap object. The label (if there is one) shown near the upper right corner of each rectangle refers to the object that it represents. Ovals represent program variables, and arrows emanating from ovals represent values of reference-type variables. For an object, the name of the class of the object is shown above the divider, and values of the primitive-type fields (if any) of the object are shown below the divider. The value of a reference-type field of an object is represented by an arrow labeled with the field name and originating from the rectangle representing the object.

Figure 3(b) shows the snapshot of P''s heap just before execution of line 15, when P' is executed with input `10`. For every object that exists in P's heap, P''s heap contains two objects: one object o of a HC class and the other object of the original class that is the shadow object of o. For example, objects s_1 and s_2 are the shadow objects of o_1 and o_2, respectively. Note that the value of the heap location $o_2.\texttt{f}_{\texttt{HC}}$ (i.e., `10`) is consistent with the value of $s_2.\texttt{f}$ (i.e., `10`) because the field `f` is of `int` type, and the two values are identical. Similarly, the value (i.e., the reference to o_2) of $o_1.\texttt{l}_{\texttt{HC}}$ is consistent with the value (i.e., the reference to s_2) of $s_1.\texttt{l}$ because the field `l` is of reference type and s_2 is the shadow object of o_2.

C. Handling Side-effects of Uninstrumented Code

Side-effects of uninstrumented code (e.g., native methods) manifest as inconsistencies between the concrete and symbolic heap partitions. In heap cloning, two values–one in concrete heap and another in symbolic heap–are stored corresponding to every heap location. Instrumented Java methods update values corresponding to a heap location in both partitions. In contrast, uninstrumented code update the value corresponding to a heap location only in the concrete partition. As a result, any inconsistency in values of a heap location in the two partitions is an indication that the heap location was updated inside uninstrumented code. Such inconsistency is detected when the value of the concerned heap location is later read after execution of the problematic code. If such an inconsistency is detected and the concerned heap location is mapped to a symbolic value at the time of read, a symbolic execution system that leverages heap cloning can notify the user that an imprecision might have been introduced.

To illustrate, recall that the native method `complex`, which is invoked in line 16 of Figure 1, reads the current value of the field `f`, and stores its corresponding absolute value back into the field `f`. Suppose the transformed program is executed with input `-10`. The in-lined figure shows the potential heap snapshot of the transformed program at line 17 in this execution. In the figure, note that the value of $o_2.\texttt{f}_{\texttt{HC}}$ is not consistent with the value of $s_2.\texttt{f}$ because, $s_2.\texttt{f}$ is updated inside the `complex` native method.



If consistency between two heap partitions is not maintained, the transformed program produced by the heap-cloning technique may behave differently from the original program for some inputs. For example, in this program, method `error` will never be called because at line 17, the value of the heap location `x.f` is always non-negative. However, in the transformed program produced by heap cloning, if the above-mentioned inconsistency is not eliminated, the `error` method will be called because at line 17, the value of the heap location $o_2.\texttt{f}_{\texttt{HC}}$ (i.e., -10) will be used to decide which branch is taken. The heap-cloning technique eliminates the potential inconsistency in this example, by copying the value of $s_2.\texttt{f}$ (i.e., 10) in the heap location $o_2.\texttt{f}_{\texttt{HC}}$, before the branch decision is made. Following paragraphs present the transformation rules concerning load and store statements. Those rules ensure consistency between the two heap partitions.

In P', an update of a non-static field $\texttt{f}_{\texttt{HC}}$ of an object o to a new value v, is accompanied by an update to heap location $o_s.\texttt{f}$, where o_s is the shadow object corresponding to $o^{(23,24)}$. If the field `f` is of primitive type, values of both heap locations, $o.\texttt{f}_{\texttt{HC}}$ and $o_s.\texttt{f}$ are updated to the new value v. If `f` is of reference type, then the values of $o.\texttt{f}_{\texttt{HC}}$ and $o_s.\texttt{f}$ are updated to v and v_s, respectively, where v_s is shadow object corresponding to v.

The transformation rule for a load statement handles cases in which a field value in the concrete heap partition is changed outside the methods of HC classes (e.g., native methods). In P', a load from field $\texttt{f}_{\texttt{HC}}$ of an object o involves three steps:

1) Read the current values v and v_s of both heap locations $o.\texttt{f}_{\texttt{HC}}$ and $o_s.\texttt{f}$, respectively$^{(25,26)}$.
2) If v and v_s are consistent, return $v^{(27)}$.
3) If v and v_s are inconsistent, update the value of the heap location, $o.\texttt{f}_{\texttt{HC}}$ to a value v_{new} that is consistent with ${v_s}^{(28)}$, and return v_{new}. For primitive type values, v_{new} is v_s. For reference type values, v_{new} is obtained from wrapping v_s.

D. Wrapping Concrete Objects

In two cases, it is necessary to wrap objects of original classes as objects of HC classes. The first case concerns values returned from native methods. For example, a native method `m` of a class `X` may return an object of a class `A`. The method $\texttt{m}_{\texttt{HC}}$ in $\texttt{X}_{\texttt{HC}}$ that corresponds to `m`, delegates the call to `m`, and thus, must wrap the object that `m` returns as an object of $\texttt{A}_{\texttt{HC}}$. The second case concerns situations where a field value of a heap object is updated outside the methods of the HC classes (e.g., native methods).

To support the wrapping operation, the technique performs two transformations.

- Every original class, except `Object`, is transformed to implement an interface `Wrapper`, which defines only one method `wrap`$^{(9)}$. The implementation of `wrap` in an original class `A`, wraps the receiver object as an object of the corresponding HC class $\texttt{A}_{\texttt{HC}}{}^{(12)}$.
- A special constructor is added to every HC class $\texttt{A}_{\texttt{HC}}$ that is used in the `wrap` implementation of the original class $\texttt{A}^{(6)}$.

Note that, as a result of wrapping an object of the original class as described above, one object of an original class `A` can be the shadow object of multiple objects of the HC class $\texttt{A}_{\texttt{HC}}$. This may cause the transformed program to behave differently from the original program. Specifically, the difference in behavior may arise because of program operations, such as reference equality check, use of the `hashcode` method of the `Object` class, and use of monitors for thread synchronization that explicitly or implicitly compares identities of two objects. However, the heap-cloning technique eliminates this potential problem. In P', shadow objects as operands to these problematic operations, instead of objects of HC classes$^{(32)}$.

E. Eliminating Imprecision in the Instrumentation Approach

In the instrumentation approach, the inability to flexibly instrument Java's standard library classes leads to imprecision. However, this inability to instrument library classes does not lead to any imprecision while symbolically executing the transformed program P' produced by heap cloning because P' does not use the original library classes. Instead, with a few exceptions, P' uses the HC classes corresponding to the library classes. As a result, symbolic values cannot flow into original library classes in the transformed program, and thus, it is not necessary to instrument library classes for symbolic execution. In the exceptional cases, native methods and static class initializers of the library classes are executed, and thus, can potentially lead to computation of imprecise path constraints. This issue of precision is further explained in Section III-G.

However, heap cloning instruments the original library classes to maintain concrete heap partition. There are several types of minor instrumentation that are applied to these classes: (1) addition of getter and setter methods to read and update fields of those classes, (2) addition of a constructor$^{(11)}$, (3) making each of those classes (except `Object`) implement the interface Wrapper$^{(9)}$ and the addition of the method `wrap()`$^{(11)}$. However, such instrumentation is not fundamental to heap cloning—it can be avoided using features, such as reflection or low-level API classes (e.g., `sun.misc.Unsafe`). However, compared to those alternatives, instrumentation provides efficiency and portability across JVM's, and at the same time, in our experience, such instrumentations have not caused any problem in the JVM's operation.

F. Enabling DSE in Custom-interpreter Approach

In the custom-interpreter approach, DSE cannot be performed: a standard JVM cannot execute native methods because the program's state is managed by the interpreter, and the JVM cannot operate on an external representation of program's state. By leveraging heap cloning, a custom-interpreter based symbolic-execution system can perform DSE on the transformed program P' by maintaining the concrete heap partition internal to a standard JVM as the custom interpreter interprets P'. The interpreter maintains the symbolic heap partition. When a native method is invoked during interpretation, because the heap-cloning technique guarantees that at that point concrete heap partition is consistent with symbolic heap partition, the native method can be safely executed concretely on the JVM. During its execution, the native method reads from and writes to the concrete heap partition. After the native method returns, the interpreter continues interpreting the rest of P'. If the native method updates a heap location in the concrete heap partition, as described in Section III-B heap cloning ensures that the update is reflected onto the symbolic heap partition when the value of that location is later read.

The remaining issue is how the interpreter can maintain the concrete heap partition on a standard JVM . For a concise, yet concrete description of the idea, we will describe how this can be done in a custom-interpreter based symbolic-execution system that uses Java Pathfinder (JPF).[4] We chose JPF because two existing symbolic-execution systems, Symbolic Pathfinder [17] and JFuzz [12] are implemented using JPF. We leverage following three characteristics of JPF.

- JPF itself is written in Java, and thus, executes on top of a standard JVM, which we call the host JVM.
- JPF supports Model Java Interface (MJI) API that can be used to intercept invocations of methods of the subject program, and then execute arbitrary code on the host JVM in place of the invoked method. The interception code can also update the subject program's heap that JPF manages.
- JPF supports data annotation feature [17] that allows attaching metadata to every piece of data that the subject program operates on.

Using JPF, the concrete heap can be maintained internal to JPF's host JVM as follows.

- Two types of operations of the transformed program P' are executed on the host JVM using MJI: (1) operations that read or update field values of objects in the concrete heap partition, (2) allocation of a shadow object in the concrete heap partition.
- The mapping from a HC object to its corresponding shadow object is maintained using JPF's data annotation feature.
- The feature that MJI code can allocate objects in the subject program's heap is leveraged to allocate objects of HC classes. The need for this arises to wrap an object of an original class as an object of the corresponding HC class.

[4]http://babelfish.arc.nasa.gov/trac/jpf

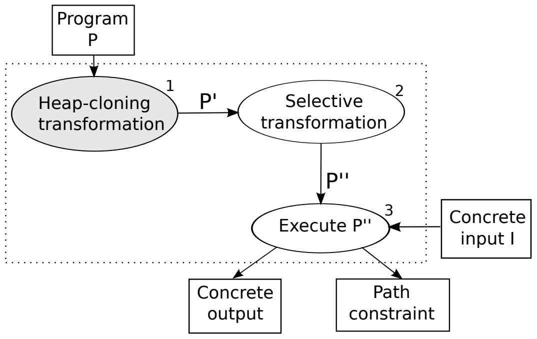


Fig. 5. The CINGER dynamic symbolic-execution system.

G. *Improving Precision of Path Constraints*

There are two important issues related to the precision of path constraints. The first issue is whether heap cloning can always automatically compute precise path constraints. Heap cloning can automatically compute precise path constraints of a path only if the precision of the path constraint is not affected by side-effects of methods (e.g., native methods) that are only concretely executed. If imprecision is introduced because some of those methods update memory locations that have symbolic values corresponding to them, heap cloning detects, and notifies to the user of such updates. Such notification can help the user to identify the problematic methods. To ensure computation of precise path constraints, the user can then provide models of those methods such that those models update the symbolic values of updated memory locations.

The second issue is whether the increase in precision is beneficial. The benefits of increased precision depends on the specific application of symbolic execution. One example where precision of path constraints is important is a recent technique [5] that anonymizes user inputs that lead to software failures in the field using symbolic execution. Given one set of inputs I, the technique uses symbolic execution to generate another set of inputs I_a that are different from I, but that cause the program to take the same path as I. The software developers then use I_a to reproduce the failure. In this application of symbolic execution, if the computed path constraint is imprecise, then a new set of inputs I_a obtained from solving the path constraints may not take the same path as I. As a result, the technique may not be able to reproduce the failure.

IV. EMPIRICAL EVALUATION

In this section, we first discuss our implementation of CINGER, then we describe our subject programs, and finally, we present the results of our two studies.

A. *Cinger Dynamic Symbolic-Execution System*

CINGER contains approximately 10,000 lines of Java code, and uses the Soot framework [25]. To compute path constraints, CINGER transforms all types of Java constructs and operators, such as bitwise operations, array accesses, arrays with symbolic length, and type conversion of primitive-type values.

TABLE I
SUBJECT PROGRAMS FOR THE STUDIES.

Subject	**Methods**		**Classes**		**Lines of Code**	
	User	**Library**	**User**	**Library**	**User**	**Library**
NanoXML	87	709	15	161	1,230	14,604
JLex	136	717	27	158	6,566	13,702
Sat4J-Dimacs	292	789	45	190	3,908	17,195
Sat4J-CSP	329	1,486	51	339	4,125	39,617
BCEL	112	614	45	149	2,321	12,659
Lucene	942	1,852	215	409	20,821	56,622

Figure 5 shows a dataflow diagram[5] of CINGER. CINGER inputs a program P and a concrete input (i.e., non-symbolic program input) I. CINGER transforms P to P'', and executes P'' with input I on a standard JVM . Execution of P'' produces (1) the same output as the output produced by executing the original program P with input I and (2) the path constraint of the path that P takes for I.

Transformation of P to P'' involves two program-transformation steps. In Step 1, the *Heap-cloning transformation* technique transforms P to P'. Recall that P' consists of original classes of P along with HC classes produced by the heap-cloning technique. In Step 2, only the HC classes of P' are further transformed to produce the final transformed program P''. Step 2 uses a variation of the program-transformation technique that we developed in our prior work [1], *Selective transformation*, which is a static analysis that identifies parts of the program that may operate on symbolic values, and transforms only those parts. Such selective transformation ensures that the overhead of symbolic execution is not incurred for program parts that do not operate on symbolic values.

B. *Subjects*

We used six publicly available Java programs as subjects for our studies: NanoXML, JLex, Sat4J (2 versions), BCEL, and Lucene. For each subject, we created a test suite consisting of test cases that we gathered from the web or created manually. Table I provides details about the subjects. The first column lists the name of the subject. For each subject, the second column shows the number of methods that were covered by the test suite. The third and fourth columns show the number of classes containing at least one covered method and the number of lines of code corresponding to all covered methods, respectively. Additionally, each column shows the numbers of corresponding entities belonging to user-defined (User) classes and Java's standard library classes (Library). For example, for JLex, 136 user-defined methods and 717 library methods are covered by the set of test cases. Our subjects are significantly larger than subjects used in prior works that applied symbolic execution to Java programs at the whole-program level.

[5]In a *dataflow diagram*, rectangles represent input data, ovals represent processing elements, and labels on edges represent the flow of data between elements.

TABLE II
RESULTS OF STUDY 1.

Subject	Average Number of Conjuncts	Average Percentage of of Conjuncts Generated Inside Library Classes
NanoXML	19,582	25.24%
JLex	65,068	17.47%
Sat4J-Dimacs	60,351	0.0%
Sat4J-CSP	629,078	66.90%
BCEL	34,161	89.76%
Lucene	47,248	0.0%

TABLE III
RESULTS OF STUDY 2.

Subject	Number of Covered Native Methods
NanoXML	4
JLex	6
Sat4J-Dimacs	0
Sat4J-CSP	9
BCEL	4
Lucene	33

C. *Study 1: Increase in Precision in Instrumentation Approach*

The goal of this study is to assess the increase in precision that a symbolic-execution system based on the instrumentation approach would achieve by leveraging the heap-cloning technique. Path constraints computed using this approach can be imprecise because computed path constraints may not contain conjuncts that arise from symbolic execution of library code that may not be instrumented due to problems described in Section II.

To do this assessment, we performed three steps.

1) We symbolically executed each subject program using CINGER to compute path constraints corresponding to paths that the programs took for each test case in the subject's test suite.
2) We computed the number of conjuncts in those path constraints, and computed the average over all these conjuncts.
3) We computed the percentage of conjuncts that are generated inside the library classes, and computed the average over these percentages.

Table II shows the results of the study. In the table, for each subject, the second column shows the average number of conjuncts in path constraints over different inputs. The third column shows the average percentage of all conjuncts of a path constraint generated inside methods of HC classes corresponding to Java's standard library classes.

The table shows that the percentage of conjuncts generated inside library classes varies widely across the subjects. For BCEL, approximately 90% of the conjuncts on a path constraint are generated inside library classes, whereas, for Sat4J-dimacs and Lucene, no conjuncts are generated inside library classes. The number of conjuncts generated inside library classes depends on whether the library classes operate on symbolic values. Lucene and Sat4J-Dimacs use their own input processing front-ends (e.g., scanner), and thus, symbolic input values do not flow into the library classes.

In summary, the results of this study show that a significant percentage of conjuncts on path constraints are indeed generated inside Java's standard library classes, and CINGER is able compute them precisely.

D. *Study 2: Reduction in Required Manual Effort to Specify Models*

The goal of this study is to assess whether heap cloning can reduce the manual effort required to specify models for native methods that introduce imprecision.

We performed this assessment in three parts. In the first part, we estimated the number of native methods that would need to be modeled if the user did not know which native methods could introduce imprecision. To do this, for each subject program we counted the number of unique native methods that are executed and have at least one reference-type parameter. We counted only native methods with reference-type parameters because performing DSE in the presence of native methods that take only primitive-type (e.g., `int`) parameters is straight-forward. The number of counted native methods is a lower limit on the number of methods for which the user may have to provide models because there could be other native methods that accessed program's heap through static fields.

Table III shows the results of this part. For each subject, the second column shows the number of such native methods that are executed for at least one input of the corresponding test suite. The data in the table show that each of our subjects, except Sat4J-Dimacs, calls at least one problematic native method. Thus, if the user did not know which native methods could introduce imprecision, then she would have to model each of those methods. Although the number of the problematic methods is small, it may still require significant manual effort to create models for those methods.

In the second part, we assessed whether the user must write models for one or more of the above-mentioned problematic methods if a DSE system that used JPF is used. We chose JPF because models for a large number of native methods and other problematic classes (e.g., `java.lang.Class`) are already available for JPF . We estimated whether the user would need to write new models in addition to those that are already available. To do this, we implemented a symbolic-execution system, JAZZ, on top of JPF. We applied JAZZ to our subjects to compute path constraints for each path that corresponds to a test case for the subjects.

JAZZ could not automatically handle all our selected subjects because of the lack of models for some native methods. For Sat4J and BCEL, we manually removed those statements of the program that called the problematic native methods—removal of those statements did not affect the functionality of the programs. As an example, for Sat4J-Dimacs and Sat4J-CSP, we removed the code that output a banner at the beginning of the application. For Lucene, we did not use this approach because of our unfamiliarity with the Lucene program and the problematic native methods (of `java.lang.management.ManagementFactory` class). Thus, we could not apply `Jazz` to Lucene.

In the third part, we enabled CINGER to generate a notification whenever it detected an update to a memory location such that (1) the update occurred inside uninstrumented code, and (2) that memory location was mapped to some symbolic value. Those notifications pointed to only one native method `arraycopy` in the class `java.lang.System` that performed such updates, and this meant that only this method needed to be modeled.

In summary, without knowing which methods introduced imprecision, the user must write models for each method that is counted in Table III. Even the set of models that is currently available for JPF is not sufficient for our subjects, and thus, the user must write models for additional methods. In contrast, if heap cloning identifies only one method that introduced imprecision, the user has to specify a model for that method.

V. RELATED WORK

The requirement to transform Java's standard library classes arises in a number of domains, such as distributed computing and software testing. Thus, a number of domain-specific techniques (e.g., [19], [23]) have been presented that address the problem by leveraging domain-specific knowledge, and thus, they are not suitable for symbolic execution. Prior work [4], [7] presents general techniques to address this problem. Our heap-cloning technique builds on the Twin Class Hierarchy (TWH) approach [7], in which copies of the classes are used instead of the original classes. However, for native methods, TWH requires manually-specified conversion routines that convert objects of original classes to objects of copy classes, and vice versa. In heap cloning, because a concrete-heap partition is kept consistent with the symbolic-heap partition, native methods can be executed on the concrete-heap partition without requiring any conversion routines. Compared to the technique presented in Reference [4], heap cloning is a more general solution (i.e., not specific to an API), and allows arbitrary instrumentation (e.g., adding fields) of library classes.

VI. CONCLUSION

In this paper, we presented a novel program-transformation technique—heap cloning—that can be used to automatically identify native methods whose side effects can introduce imprecision in path constraints. Thus, it can reduce the manual effort required to apply DSE to real-world programs. Heap cloning enables implementation of DSE using the custom-interpreter approach, and eliminates the imprecision of path constraints that is incurred in the instrumentation approach.

In this work, we assumed that only native methods cannot be symbolically executed. In future work, we plan to allow the user to annotate any method to be outside the scope of symbolic execution, and let the user provide models. Heap cloning can then detect whether such annotated methods cause side effects that introduce imprecision, and thus, reduce the burden of writing models. We believe that the path explosion problem from which symbolic execution suffers can be addressed by using models of library (e.g., Java's standard library) or framework (e.g., Google's Android) classes that are used by a large number of application. Currently, we are working to make the CINGER system available to the research community.

ACKNOWLEDGEMENT

This research was supported in part by NSF CCF-0725202 and CCF-0541048 and IBM Software Quality Innovation Award to Georgia Tech. The reviewers provided many helpful suggestions that improved the paper's presentation.

REFERENCES

[1] S. Anand, A. Orso, and M. J. Harrold. Type-dependence analysis and program transformation for symbolic execution. In *TACAS*, pages 117–133, 2007.

[2] S. Anand, C. S. Pasareanu, and W. Visser. JPF-SE: A symbolic execution extension to Java Pathfinder. In *TACAS*, pages 134–138, 2007.

[3] W. Binder, J. Hulaas, and P. Moret. A quantitative evaluation of the contribution of native code to Java workloads. In *IISWC*, pages 201–209, 2006.

[4] W. Binder, J. Hulaas, and P. Moret. Advanced Java bytecode instrumentation. In *PPPJ*, pages 135–144, 2007.

[5] J. A. Clause and A. Orso. Camouflage: automated anonymization of field data. In *ICSE*, pages 21–30, 2011.

[6] X. Deng, J. Lee, and Robby. Bogor/kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems. In *ASE*, pages 157–166, 2006.

[7] M. Factor, A. Schuster, and K. Shagin. Instrumentation of standard libraries in object-oriented languages: the twin class hierarchy approach. In *OOPSLA*, pages 288–300, 2004.

[8] I. García. Enabling symbolic execution of Java programs using bytecode instrumentation. Master's thesis, Univ. of Texas at Austin, 2005.

[9] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *PLDI*, pages 213–223, 2005.

[10] M. Grechanik, C. Csallner, C. Fu, and Q. Xie. Is data privacy always good for software testing? In *ISSRE*, pages 368–377, 2010.

[11] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *TOPLAS*, 23(3):396–450, 2001.

[12] K. Jayaraman, D. Harvison, V. Ganeshan, and A. Kiezun. A concolic whitebox fuzzer for Java. In *NFM*, pages 121–125, 2009.

[13] K. Kahkonen, T. Launiainen, O. Saarikivi, J. Kauttio, K. Heljanko, and I. Niemel. LCT: An open source concolic testing tool for Java programs. In *BYTECODE*, pages 75–80, 2011.

[14] S. Khurshid, C. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *TACAS*, pages 553–568, 2003.

[15] J. C. King. A new approach to program testing. In *Programming Methodology*, pages 278–290, 1974.

[16] X. Li, D. Shannon, I. Ghosh, M. Ogawa, S. P. Rajan, and S. Khurshid. Context-sensitive relevancy analysis for efficient symbolic execution. In *APLAS*, pages 36–52, 2008.

[17] C. S. Pasareanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. R. Lowry, S. Person, and M. Pape. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In *ISSTA*, pages 15–26, 2008.

[18] C. S. Pasareanu, N. Rungta, and W. Visser. Symbolic execution with mixed concrete-symbolic solving. In *ISSTA*, pages 34–44, 2011.

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

[20] R. A. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold. Test-suite augmentation for evolving software. In *ASE*, pages 218–227, 2008.

[21] K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *CAV*, pages 419–423, 2006.

[22] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *ESEC/SIGSOFT FSE*, pages 263–272, 2005.

[23] E. Tilevich and Y. Smaragdakis. Transparent program transformationsin the presence of opaque code. In *GPCE*, pages 89–94, 2006.

[24] A. Tomb, G. P. Brat, and W. Visser. Variably interprocedural program analysis for runtime error detection. In *ISSTA*, pages 97–107, 2007.

[25] R. Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. Soot - A Java optimization framework. In *CASCON*, pages 125–135, 1999.