Type-Dependence Analysis and Program Transformation for Symbolic Execution

Saswat Anand, Alessandro Orso, and Mary Jean Harrold

College of Computing, Georgia Institute of Technology
{saswat,orso,harrold}@cc.gatech.edu

Abstract. Symbolic execution can be problematic when applied to real applications. This paper addresses two of these problems: (1) the constraints generated during symbolic execution may be of a type not handled by the underlying decision procedure, and (2) some parts of the application may be unsuitable for symbolic execution (e.g., third-party libraries). The paper presents *type-dependence analysis*, which performs a context- and field-sensitive interprocedural static analysis to identify program entities that may store symbolic values at run-time. This information is used to identify the above two problematic cases and assist the user in addressing them. The paper also presents a technique to transform real applications for efficient symbolic execution. Instead of transforming the entire application, which can be inefficient and infeasible (mostly for pragmatic reasons), our technique leverages the results of type-dependence analysis to transform only parts of the program that may interact with symbolic values. Finally, the paper discusses the implementation of our analysis and transformation technique in a tool, STINGER, and an empirical evaluation performed on two real applications. The results of the evaluation show the effectiveness of our approach.

1 Introduction

Testing is one of the most commonly used techniques to gain confidence in the correct behavior of software. Because manual generation of test inputs is time consuming and usually results in inadequate test suites, researchers have proposed automated techniques for test-input generation. One of these techniques, symbolic execution, generates test-inputs by interpreting a program over symbolic values and solving constraints that lead to the execution of a specific program path. Although symbolic execution was first introduced in the mid 1970s [15], the dramatic growth in the computational power of the average machine and the availability of increasingly powerful decision procedures in recent years have renewed interest in using symbolic execution for test-input generation (e.g., [2,11,19,25,28]).

Despite the fact that symbolic execution is well understood, and performing symbolic execution on simple programs is straightforward, problems arise when attempting to symbolically execute real applications. In this paper, we address two such problems. The first problem concerns the capabilities of the underlying decision procedure used to check satisfiability and solve path conditions. If the

O. Grumberg and M. Huth (Eds.): TACAS 2007, LNCS 4424, pp. 117–133, 2007.

underlying decision procedure is incapable of (or inefficient in) handling the types of constraints produced during symbolic execution, users must rewrite parts of the program so that the offending constraints are not produced. However, this rewriting requires a user to identify those parts of the program that may generate problematic constraints, which is a difficult task. The second problem concerns the flow of symbolic values outside the boundaries of the software being symbolically executed. In these cases (e.g., when a symbolic value is passed as a parameter to an external library call), the execution must abort because external code cannot handle symbolic values. In real applications, there can be many instances of this problem, such as calls to native methods in Java, unmanaged code in the .NET framework, and third-party pre-compiled libraries. To address this issue, users must replace the calls to external components that may be reached by symbolic values with calls to stubs that model the components' behaviors. Like the first problem, performing this transformation requires manual intervention: the users must identify the external calls that may be problematic before actually performing symbolic execution.

In some studies on symbolic execution [25,28], the two problems do not arise because of the types of programs used (e.g., implementations of data structures). In other studies where these problems arise, researchers have taken various approaches to address them. Some researchers proposed approaches that replace symbolic values with concrete values whenever the symbolic values cannot be handled by the decision procedure or by an external component [10,19]; these approaches make the technique incomplete, in that it may fail to generate test inputs for a feasible program path. Other researchers proposed approaches based on "trial and error" [3]—every time the symbolic execution cannot continue because of a call to an external-library function with one or more symbolic parameters, the users are notified and must modify the code appropriately; although this solution may eventually lead to a successful execution, it can be inefficient when the interaction with the user is frequent. Yet other researchers proposed to use decision procedures for bit vectors that use boolean satisfiability (SAT) solvers [2]. Such decision procedures (e.g., STP [9]) can theoretically handle most types of constraints that may arise in a program under the assumption of finite representation of numbers. However, they are inefficient in handling linear integer arithmetic constraints, when compared to decision procedures specifically designed for this domain, such as Omega [17] and Yices [6].

To facilitate symbolic execution of real applications, where the previously described problems are frequently encountered, we present a new approach. Our approach is based on the insight that both of these problems are caused by the flow of symbolic values to *problematic variables*, such as parameters of library calls or operands of expressions that cannot be handled by the underlying decision procedure. Our approach is based on a novel static analysis, called type-dependence analysis, that identifies problematic variables before performing symbolic executions. Our type-dependence analysis formulates the problem of identifying variables that may assume symbolic values as a value-flow analysis problem. The analysis is context- and field-sensitive, which has the advantage

of providing fairly precise results. The benefit of our analysis is that it can automatically detect parts of the program that may be problematic for symbolic execution (e.g., a modulo operation that involves at least one symbolic operand). For any such part, the analysis reports to the users the identified problem, together with contextual information, to help them understand the issue and perform necessary program changes.

In this paper, we also present a technique that leverages the results of the analysis to transform applications and prepare them for symbolic execution. The basic idea behind the transformation is to replace concrete types with symbolic types and concrete operators with operators that work over symbolic values [14]. Naively applying such transformation to the entire application leads to two problems. First, in practice, execution engines such as the Java virtual machine make implicit assumptions about the internal structures of some components. Transforming such components is thus problematic. Second, symbolic operations are more expensive than their concrete counterparts, even when they operate on concrete values (the extra overhead is incurred in checking whether a value is symbolic or concrete). Therefore, transforming those components of the program that may not interact with symbolic values introduces inefficiencies. Because type-dependence analysis can identify which variables may be symbolic, our technique avoids transforming parts of the code that have no interactions with symbolic values, thus improving both applicability and efficiency of symbolic execution.

To evaluate our type-dependence analysis and transformation technique, we implemented them in a tool, called STINGER, that works on Java and is integrated in Java Pathfinder [13], and used the tool to perform an empirical evaluation on two real programs. To the best of our knowledge, the programs that we used are considerably larger than those used in previous studies on symbolic execution. The results of the studies show that our analysis can be effective in (1) statically identifying areas of the code that would be problematic for symbolic execution, (2) providing useful feedback to the users to guide them in the resolution of the problems, and (3) limiting the transformation necessary for symbolic execution.

The main contributions of the paper are

- A context- and field-sensitive static flow analysis that can identify the variables in a program that may hold symbolic values, given a set of symbolic inputs. The analysis results enable static identification of program segments that are potentially problematic for symbolic execution and can guide users in transforming the program to eliminate the problems.
- A general transformation technique that leverages the type-dependence analysis to transform programs into "symbolic programs" (i.e., programs whose execution essentially performs symbolic execution of the original program).
- A tool, STINGER, that implements our approach for Java and is integrated in Java Pathfinder.
- A set of empirical studies, performed on two real programs, whose results show the usefulness of our approach.

2 Type-Dependence Analysis

This section presents our type-dependence analysis, which computes the set of program entities that may store symbolic values when a program is symbolically executed. We target a typical scenario in which the user selects a set of variables to hold symbolic input values for a program and then symbolically executes the program. In this context, the type of the selected variables and of other variables that can hold values derived from these selected variables must be symbolic.

```
public class Object{
    public static native Object clone();
}
public class M extends Object{
    int m;
    M(int x){ this.m = x; }
    int getM(){ return this.m; }
    static native boolean isPrime(int x);
    public static void main(String[] arg){
        int s = Symbolic.integer();
        M a = new M(s); M b = new M(4);
        int p = a.getM(); int q = b.getM();
        if(isPrime(p) && q % 3 == 0)
            M c = (M) a.clone();
  }
}
```

Fig. 1. Motivating example

Before discussing the details of our analysis, we introduce a motivating example that illustrates some of the issues that the analysis can help to address. Suppose that we want to symbolically execute the Java program shown in Fig. 1, and that `s` represents the (symbolic) input to the program (as shown by the assignment of `Symbolic.integer()` to `s`). On initial inspection, the program contains three potentially-problematic cases: the use of the modulo (`%`) operation, which is not supported by many decision procedures; the invocation of native method `clone`; and the invocation of native method `isPrime`. However, a more careful inspection reveals that the first two cases are not problematic: the modulo operator never operates on symbolic values and native method `clone` can access only fields of class `Object`,[1] none of which may store symbolic values. As for the third potentially-problematic case, a symbolic value is passed as an argument to native method `isPrime` and is likely to be problematic because the method expects a concrete value. Our type-dependence analysis can discover that the first two cases are not problematic but the third case is. For this third case, the analysis can provide context information to help the user understand the problem and address it.

We call our analysis type-dependence analysis because it identifies type dependence between variables. We define type dependence as follows: For a given

[1] This conclusion is based on the common assumption that native methods do not use dynamic type discovery and thus access only fields of declared types of their parameters [23].

(assignment)	`p = x` $\Longrightarrow p \leftarrow x$
(binop)	`p = x` $\oplus$ `y` $\Longrightarrow p \leftarrow x, p \leftarrow y$
(load)	`p = o.f` $\Longrightarrow p \xleftarrow{get[f]} o$
(store)	`o.f = x` $\Longrightarrow o \xleftarrow{put[f]} x$
(return)	`return x` $\Longrightarrow R_m \leftarrow x$, where m is the concerned method
(array-new)	`a = new t[size]` $\Longrightarrow a \xleftarrow{put[length]} size$
(array-assign1)	`a[i] = x` $\Longrightarrow a \xleftarrow{put[elem]} x$
(array-assign2)	`p = a[i]` $\Longrightarrow p \xleftarrow{get[elem]} a$
(array-length)	`p = a.length` $\Longrightarrow p \xleftarrow{get[length]} a$
(invocation)	`x = a.foo(`$a_1, \ldots, a_n$`)` $\Longrightarrow x \leftarrow R_{foo}, P^1_{foo} \leftarrow a_1, \ldots, P^n_{foo} \leftarrow a_n$

Fig. 2. Rules for building the type-dependence graph

type-correct program, an entity x is *type dependent* on an entity y iff x's type may need to be changed as a consequence of a change in y's type to maintain type correctness. Our type-dependence analysis computes a conservative approximation of the type-dependence relation between a given set of entities and the other entities in the program. The type-dependence analysis is an instance of the more general value-flow analysis, which identifies whether the value of an entity x can flow to an another entity y in the program. In the definition of our analysis, we leverage techniques for demand-driven interprocedural analysis [12] and cloning-based interprocedural analysis [27], and techniques that use binary decision diagrams for scaling interprocedural analysis [1,27].

Type-dependence analysis consists of two phases. The first phase builds a *Type-Dependence Graph (TDG)* for the program, which encodes direct type-dependence information between program entities. The second phase performs Context-Free Language (CFL) reachability [18] on the TDG to identify transitive type dependences.

2.1 Building the TDG

In the first phase, the analysis builds the Type-Dependence Graph (TDG), a directed graph (N, E). N is a set of nodes, each of which represents one of several entities: a static field, a local variable of a method, a field of primitive type, a parameter of a method, or the return value of a method. E is a set of directed edges. An edge $x \leftarrow y$ in E indicates that there is a direct type dependence between the entity represented by y and the entity represented by x (i.e., x is directly type-dependent on y).

To build the TDG, our analysis processes each program statement once and adds an edge to the graph for each relevant statement, according to the rules shown in Fig. 2. Note that the rules apply only to non-constant right-hand side values—the analysis does not add nodes or corresponding edges to the TDG for constant entities. In the figure, $o.f$ represents field f of object o; P^i_m represents the i^{th} parameter of method m; R_m represents the return value of method m.

$$\frac{p \leftarrow x,\ x \in Sym}{p \in Sym}$$

$$\frac{p \xleftarrow{get[f_1]} q,\ y \xleftarrow{put[f_2]} x,\ f_1 = f_2,\ alias(y,q),\ x \in Sym}{p \in Sym}$$

Fig. 3. Context-insensitive inference rules for type dependence analysis.

For the definition of the rules, the analysis treats arrays as objects with two fields, *elem* and *length*, that represent all array elements and the length of the array, respectively. For space reasons, rules for statements involving static field references, unary operations, and casting are not shown; they are analogous to the assignment rule.

2.2 Performing CFL-Reachability on the TDG

In the second phase, the analysis performs CFL-reachability [18] on the TDG with a user-specified set of variables selected to be symbolic, Sym_0, and computes set Sym, which contains all local variables, static fields, formal parameters, and return values of scalar types that are type-dependent on variables in Sym_0. Instance fields and entities of array-types that are type-dependent on variables in Sym_0 are then computed from Sym; due to space constraint, this is described in Appendix A. The analysis initializes Sym to Sym_0 and applies a set of inference rules until a fix point on Sym is reached. For clarity, we first present a context-insensitive version of our analysis and then describe how it can be extended to be context-sensitive.

The context-insensitive version of our analysis is represented by the two inference rules in Fig. 3. The first rule states that an entity p is added to the Sym set if there is another entity x in Sym on which p is directly type dependent. The second rule captures transitive type dependence through heap aliases. It states that entity p must be added to Sym if there is another entity x in Sym and two object references y and q such that (1) p is directly type dependent on a field f of q, (2) the same field f of y is directly type dependent on x, and (3) y and q may point to the same object (expressed using the notation $alias(y, q)$). Without loss of generality, our analysis assumes that may-alias information is computed on demand by some points-to analysis (e.g., [22]).

Our analysis is *field-sensitive*—in the second rule, the labels $get[f_1]$ and $put[f_2]$ must refer to the same fields. This is in contrast to a *field-based* analysis, which does not distinguish between different fields of an object. Field sensitivity cannot be achieved through simple reachability. It requires our analysis to perform CFL reachability by matching $get[]$ and $put[]$ labels (two matching labels must refer to the same field), while identifying all nodes reachable from the initial set Sym_0,

The context-insensitive analysis described above may compute unnecessarily-large Sym sets. In the example in Fig. 1, for instance, the analysis would not

distinguish between the two calls to the `getM` method and, thus, would not be able to detect that variable `q` is not type-dependent on variable `s`. To improve the precision of the analysis, we define a context-sensitive version of the TDG using an approach similar to method cloning [27]. First, we create multiple nodes for each entity—one for each calling context of the method that contains the entity. The only exceptions are entities that correspond to global variables (e.g., static fields in Java) that are represented with a single node in the context-sensitive TDG. Second, we create copies of the TDG's edges so that if an edge exists between two nodes, there is an edge between corresponding (context-specific) copies of the nodes. Note that each copy of an invocation edge is an inter-context edge—an edge that connects nodes that belong in different contexts.

Because cloning-based approaches can lead to an exponential explosion in the size of the graphs, we use Binary Decision Diagrams (BDDs) to represent context-sensitive TDGs [16,27]. In addition, we adopt the k-CFA approach [21], which limits the context of a call to the top k elements of the call stack.

After building the context-sensitive TDG, our analysis uses a context-sensitive version of the inference rules described in Fig. 3 to compute Sym. We obtain the context-sensitive inference rules by modifying the context-insensitive rules: we identify each entity in the rule with respect to a specific context c. The context-sensitive version of the first rule in Fig. 3, for instance, is

$$\frac{p^c \leftarrow x^c,\ x^c \in Sym}{p^c \in Sym}$$

3 Program Transformation

One common way to perform symbolic execution of a program is to first transform the program so that it can operate on both symbolic and concrete values, and then execute it.[2] A naive program transformation technique would change the types of all program entities to symbolic types, and change all operations over concrete values to operations over symbolic values. In practice, this approach is not feasible for two reasons. First, execution engines typically make implicit assumptions about the types of some entities (e.g., fields of certain classes), and these assumptions would be violated by the transformation. Second, treating all variables in a program as symbolic can be inefficient (compared to having only a small subset of symbolic variables and executing parts of the programs not affected by those variables normally). In this section, we present a program-transformation technique that leverages the results of type-dependence analysis to transform, in an automated way, only a subset of the program. By doing this, our technique mitigates (when it does not completely eliminate) the two problems mentioned above.

Our technique supports two operators that enable selective program transformations: box and unbox. The *box* operator converts a concrete value to a corresponding symbolic value. The *unbox* operator converts a symbolic value

[2] There are also other approaches not based on transformation (e.g., [5,8]).

```
public class M {
    int m;
    Expression m_JPF_;
    M(int x) { this(Symbolic.makeSymbolic(x));}
    int getM() { return Symbolic.makeConcrete_int(getM_JPF_());}
    static native boolean isPrime(int i);
    M(Expression x) { this.m_JPF_ = x; }
    Expression getM_JPF_() { return this.m_JPF_; }
    static native boolean isPrime_JPF_(Expression expression);
    public static void main(String[] strings) {
        M a = new M(Symbolic.symbolic_int());
        M b = new M(4);
        Expression p = a.getM_JPF_();
        int q = b.getM();
        if (isPrime_JPF_(p) && q % 3 == 0)
           M c = (M) a.clone();
    }
}
```

Fig. 4. Transformed version of the example from Fig. 1.

created by the box operator to the corresponding concrete value. These operators are needed to handle program entities that must be of symbolic types for type correctness, but may store either symbolic and concrete values depending on contexts. The operators let these entities store (boxed) concrete values whenever necessary. The technique automatically adds to the program appropriate boxing and unboxing operators to enable assignments between entities of symbolic and concrete types. Note that unboxing a symbolic value (i.e., a symbolic value that is not the result of a boxing operation) would cause a run-time error. However, the transformation technique guarantees that such a situation will never occur due to its use of the results of the conservative type-dependence analysis.

Before presenting the formal definition of the transformation, we illustrate some features of our approach by showing, in Fig. 4, the transformed version of the example program from Fig. 1. In the code, `Expression` represents the type of symbolic expressions, and methods `makeSymbolic` and `makeConcrete_int` represent box and unbox operators, respectively. For each field that may store a symbolic value, such as `m`, the transformation adds a new field of symbolic type. Similarly, for each method that may operate on symbolic values, a new method is added that may take symbolic values as arguments and/or return symbolic values. Note that because the analysis determines that variable `q` can never store a symbolic value at runtime, `q`'s type is unchanged, and the `%` operation is not replaced by its corresponding symbolic operation. In contrast, `p`'s type is changed to `Expression` because the analysis determines that it may store a symbolic value. When a symbolic version of a method is created, only those calls that may pass and/or receive symbolic values are changed to invoke the new method. In the example, for instance, `getM_JPF_()` is called on `a` because a symbolic value may be returned by the method at that callsite. Conversely, the call to `getM()` on `b` is unchanged, as only concrete values are returned at the corresponding callsite.

Source language

$l \in \text{Local},\ f \in \text{Field},\ r \in \text{RefType}$
$n \in \text{NumType}\ n ::= \text{int} \mid \text{short} \mid \text{char} \mid \text{long} \mid \text{byte} \mid \text{float} \mid \text{double}$
$\tau \in \text{Type} \quad \tau ::= n \mid \text{boolean} \mid r \mid \tau[\,]$
$i \in \text{Immediate}\ i ::= l \mid \text{const}$
$e \in \text{Expr} \quad e ::= i \mid i_1\ \text{binop}\ i_2 \mid \text{unop}\ i \mid l.f^{(\tau)} \mid (\tau)\ i \mid l[i]^{(\tau)} \mid l.\text{length}^{(\tau)} \mid \text{new}\ \tau[i]$
$s \in \text{Stmt}) \quad s ::= l = e \mid l.f = i \mid l[i_1]^{(\tau)} = i_2$

$\text{binop} \in \{+, -, *, /, \%, =, >, \geq, <, \leq, \neq\}$
$\text{unop} \in \{-, !\}$

Extension for symbolic execution

$\tilde{l} \in \text{SymLocal},\ \tilde{f} \in \text{SymField}$
$\tilde{\tau} \in \text{SymType} \quad \tilde{\tau} ::= \textsc{Expr} \mid \textsc{ExprArray} \mid \textsc{BoolArray} \mid \textsc{RefArray}$
$\tilde{i} \in \text{SymImmediate}\ \tilde{i} ::= \tilde{l}$
$\tilde{e} \in \text{SymExpr} \quad \tilde{e} ::= \text{box}^{\tilde{\tau}}(e) \mid \tilde{i} \mid \text{symbinop}(\tilde{e_1}, \tilde{e_2}) \mid \text{symunop}\ \tilde{i} \mid l.\tilde{f} \mid \text{cast}^{\tau}(\tilde{e}) \mid \text{array\_get}^{\tilde{\tau}}(\tilde{l}, \tilde{e}) \mid \text{array\_len}^{\tilde{\tau}}(\tilde{l}) \mid \text{new\_array}^{\tilde{\tau}}(\tilde{e})$
$\tilde{s} \in \text{SymStmt} \quad \tilde{s} ::= \tilde{l} = \tilde{e} \mid l = \text{unbox}^{\tilde{\tau}}(\tilde{e}) \mid a.\tilde{f} = \tilde{e} \mid \text{array\_set}^{\tilde{\tau}}(\tilde{l}, \tilde{e_1}, \tilde{e_2})$

$\text{symbinop} \in \{\text{\_plus,\_minus,\_mul,\_div,\_mod,\_eq, \_gt, \_ge, \_lt, \_le, \_ne}\}$
$\text{symunop} \in \{\text{\_neg,\_not}\}$

Fig. 5. Source language and its extensions for symbolic execution

3.1 Source and Target Languages

For the sake of clarity, we define our transformation on a subset of Java, referred to as *source* language, that contains only those Java features relevant to the transformation. The transformation of a program in *source* language produces a program in *target* language. Fig. 5 presents the source language and its extensions for symbolic execution. The target language is the union of the source language and its extensions.

Both the source and the target languages are statically and explicitly typed according to Java's type rules. Types in the source language include all types supported by Java. The target language supports four symbolic types, namely Expr, ExprArray, BoolArray, and RefArray, that represent types of symbolic expressions, arrays of symbolic expressions, arrays of boolean values, and arrays of references, respectively. Each of the symbolic array types can also have symbolic length. The correspondence between concrete and symbolic types (for concrete types that have a corresponding symbolic type) is defined by function $stype : \text{Type} \rightarrow \text{SymType}$.

$stype(\tau) = \textsc{Expr} \quad \tau \in \text{NumType} \qquad stype(\text{[]boolean}) = \textsc{BoolArray}$
$stype(\text{[]}\tau) = \textsc{ExprArray}\ \tau \in \text{NumType} \qquad stype(\text{[]}r) = \textsc{RefArray}$

Expressions include local variables, constants, unary and binary operations, field references, casts, array references, array length and array allocation

expressions. In the source language, τ represents the element type of array l in terms $l[i]^{(\tau)}$ and $l.\texttt{length}^{(\tau)}$, and the type of field f in term $l.f^{(\tau)}$.

In the target language, there is one syntactic category for each category in the source language, represented by the same symbol with a tilde on the top. In addition, for each unary, binary, and comparison operators in the source language, the target language provides a corresponding operator that operates on symbolic values. In the definition of the language extensions, we use the following terminology (where $\tilde{\tau}$ denotes the type of array element): $\text{array\_get}^{\tilde{\tau}}(\tilde{l}, \tilde{e})$ is an operation that returns the $\tilde{e}^{th}$ element of symbolic array $\tilde{l}$; $\text{array\_len}^{\tilde{\tau}}(\tilde{l})$ returns the length of $\tilde{l}$; $\text{new\_array}^{\tilde{\tau}}(\tilde{e})$ allocates a symbolic array of size $\tilde{e}$; and $\text{array\_set}^{\tilde{\tau}}(\tilde{l}, \tilde{e}_1, \tilde{e}_2)$ stores symbolic expression $\tilde{e}_2$ as the element at index $\tilde{e}_1$ of array $\tilde{l}$. The box operator is represented by $box^{\tilde{\tau}}(e)$, which transforms the concrete value e into the corresponding symbolic value of type $\tilde{\tau}$. Analogously, $unbox^{\tilde{\tau}}(\tilde{e})$ indicates the transformation of the symbolic value contained in $\tilde{e}$, of type $\tilde{\tau}$, into its original concrete value and type.

3.2 Transformation

The transformation is performed in two steps. In the first step, new fields, methods, and local variables of symbolic types are added to the program. For each field that may store symbolic values, the transformation adds a new field with corresponding symbolic type. For each method m that may operate on symbolic values, the transformation adds a new method m_s, which may potentially have parameters and return value of symbolic types. Also, for each of m's local variables v, if v may store symbolic values, a local variable of corresponding symbolic type is added to m_s; otherwise, the original v is added to m_s. Finally, all statements of m are moved to m_s, and m is transformed into a proxy that invokes m_s and performs boxing and unboxing of parameters and/or return values as needed. Note that, even if the analysis is context-sensitive, it generates at most one variant of each method because the results of the analysis are unified over all contexts.

In the second step of the transformation, statements in the newly-added methods are transformed according to the rules provided in Fig. 6. Note that Fig. 6 does not include transformation rules that involve arrays, which are provided in Fig. 8 (see Appendix A). Each rule defines how a specific statement in the source language is transformed and is applicable only if the respective guard is satisfied. The rules use the following notations:

- For a given local variable or field x that may store symbolic values, $\overline{x}$ represents the corresponding entity of symbolic type added by the transformation in the first step. If x cannot store a symbolic value, then $\overline{x}$ simply represents the original entity. In particular, if x is a constant, $\overline{x}$ always represents x.
- $\overline{\tau}$, represent the symbolic type corresponding to a concrete type τ, as defined by function *stype*.
- For an expression e of concrete type τ, $< e >$ represents $\texttt{box}^{\overline{\tau}}(e)$ (i.e., e boxed as a value of its corresponding symbolic type). For an expression $\tilde{e}$ of a symbolic type, $< \tilde{e} >$ represents $\tilde{e}$ itself.

Original statement	Transformed statement	Guard	
$[l = i]$	$[\overline{l} = < \overline{i} >]$	$\overline{l} \neq l$	(1)
$[l = i_1 \texttt{ binop } i_2]$	$[\overline{l} = \texttt{box}^{\text{EXPR}}(i_1 \texttt{ binop } i_2)]$	$\overline{l} \neq l, \overline{i_1} = i_1, \overline{i_2} = i_2$	(2)
	$[\overline{l} = \texttt{symbinop}(< \overline{i_1} >, < \overline{i_2} >)]$	$\overline{i_1} \neq i_1$ or $\overline{i_2} \neq i_2$	(3)
$[l = \texttt{unop } i]$	$[\overline{l} = \texttt{box}^{\text{EXPR}}(\texttt{unop } i)]$	$\overline{l} \neq l, \overline{i} = i$	(4)
	$[\overline{l} = \texttt{symunop}(\overline{i})]$	$\overline{i} \neq i$	(5)
$[l.f = i]$	$[l.\overline{f} = < \overline{i} >]$	$\overline{f} \neq f$	(6)
$[l_1 = l_2.f^{(\tau)}]$	$[\overline{l_1} = < l_2.\overline{f} >]$	$\overline{l_1} \neq l_1$	(7)
	$[l_1 = \texttt{unbox}^{\overline{\tau}}(l_2.\overline{f})]$	$\overline{l_1} = l_1, \overline{f} \neq f$	(8)
$[l_1 = (\tau)\ l_2]$	$[\overline{l_1} = \texttt{box}^{\overline{\tau}}((\tau)\ l_2)]$	$\overline{l_1} \neq l_1, \overline{l_2} = l_2$	(9)
	$[\overline{l_1} = \texttt{cast}^{\tau}(\overline{l_2})]$	$\overline{l_1} \neq l_1, \overline{l_2} \neq l_2$	(10)

Fig. 6. Transformation rules for program statements

For space reasons, we discuss transformation rules for only two types of statements: assignments of a local variable or constant to a local variable and assignments of a field to a local variable. According to Rule 1, assignment statements of the form $l = i$ are transformed only if l may store a symbolic value. If so, a local variable of symbolic type that corresponds to l, $\overline{l}$, is added and becomes the l-value of the transformed statement. If i is a non-constant local variable and has a corresponding local variable of symbolic type, $\overline{i}$, $\overline{i}$ becomes the r-value of the transformed statement. Otherwise, if i is either a constant or a local variable without a corresponding local of symbolic type, i's value is boxed and assigned to $\overline{l}$.

We discuss rules for statements of type $l_1 = l_2.f^{(\tau)}$ because they make use of the unbox operator. There are two rules that involve these statements. In the first case (Rule 7), l_1 has a corresponding local of symbolic type, $\overline{l}_1$, which becomes the l-value of the transformed statement. If field f has a corresponding field of symbolic type, $\overline{f}$, the value of $\overline{f}$ of l_2 is assigned to $\overline{l_1}$; otherwise, the value of field f of l_2 is boxed and assigned to $\overline{l_1}$. In the second case (Rule 8), where l_1 does not have a corresponding local of symbolic type, but f has a corresponding field of symbolic type, $l_2.f$'s value is unboxed and assigned to l_1.

4 Empirical Studies

To assess the effectiveness of our approach, we implemented our type-dependence analysis and automatic transformation technique in a tool named STINGER (Symbolic-execution based Test INput GenEratoR), and used STINGER to perform a set of empirical studies. STINGER works on Java bytecode, leverages the SOOT framework [24], and is integrated with Java Pathfinder [13]. The type-dependence analysis is implemented using Jedd [16], a Java language extension that supports use of binary decision diagrams to store and manipulate relations. STINGER inputs a program in Java bytecode, the initial set of program entities

specified to be symbolic (called Sym_0 in Section 2), and a specification of the capabilities of the decision procedure used by the symbolic executor (in terms of supported operators). Given these inputs, STINGER performs two tasks: (1) it performs type-dependence analysis and identifies and reports the two kinds of problematic cases considered (i.e., constraints that cannot be handled by the decision procedure and symbolic values that may flow outside the scope of the software being symbolically executed); (2) it performs an automated translation of the program and generates skeleton stubs for the problematic cases identified, which the user is expected to complete with appropriate code.

We used STINGER to investigate three research questions:

RQ1: How effective is our technique in identifying parts of the code responsible for constraints that cannot be handled by the decision procedure in use?

RQ2: How often do symbolic values flow outside the boundaries of the program being symbolically executed? When that happens, can our analysis correctly identify and report problematic cases beforehand?

RQ3: To what extent can the use of our analysis reduce the transformation needed to perform symbolic execution?

Empirical Setup. As subjects for our studies, we used two freely-available Java programs: NANOXML and ANTLR. NANOXML (http://nanoxml.cyberelf.be/) is an XML-parsing library that consists of approximately 6KLOC. We selected NANOXML because it is small yet not trivial, and lets us evaluate our technique and inspect our results in detail. ANTLR (http://www.antlr.org/) is a widely-used language-independent lexer and parser generator that consists of 46KLOC. ANTLR was selected because it is a relatively large and complex software that can provide more confidence in the generality of our results. NANOXML inputs a file containing an XML document, and ANTLR inputs a file containing the grammar of a language. We changed both applications so that they input an array of symbolic characters *arr* instead of reading from a file. We then ran STINGER and specified *arr* as the only element in the initial set of symbolic entities. STINGER produced, for each program, a report and a transformed version of the program.

4.1 Results and Discussion

To address our research questions, we ran STINGER on the subjects, and measured several statistics as shown in Fig. 7. In the figure, the number of methods includes both methods of the application and methods in the Java standard library, which may also need to be transformed when symbolically executing a program. We first discuss the results for each research question independently, and then discuss the precision of the analysis.

RQ1. STINGER finds 48 (for NANOXML) and 82 (for ANTLR) cases that would be problematic for our decision procedure of choice [6]. In this context, the problematic cases are those that involve bit-wise and modulo operations over symbolic values. These problematic cases reside in 10 and 23 methods of NANOXML and ANTLR, respectively. These cases would be reported to the user, who would then

	Statistics	NanoXML	Antlr
RQ1	No. of problematic operations	48	82
	No. of methods with problematic operations	10	23
RQ2	No. of native calls that may be reached by symbolic values	3	8
	Total no. of native calls	27	48
RQ3	No. of methods transformed	89	253
	No. of reachable methods	438	1176
	No. of statements transformed	1253	4052
	No. of statements in all transformed methods	2642	8547

Fig. 7. Empirical results

need to modify the methods (or replace them with stubs) to eliminate the problem. After inspecting stinger's report, we found that many of these problematic constraints arise because of the use of modulo operators in classes `HashMap` and `HashTable`. Replacing these classes with another implementation of a map, such as `TreeMap`, eliminates the problem. The remaining problematic methods were methods operating on characters (e.g., to change a character from upper to lower case). We were able to rewrite these methods and eliminate the use of bit-wise operators in them by assuming that the input characters are ASCII characters.

RQ2. For the two subject programs, the only instances of symbolic values that may flow outside the boundaries of the program consist of calls to native methods. stinger determines that for 3 of the 27 (for NanoXML) and for 8 of the 48 (for Antlr) calls to native methods, a symbolic value may actually be passed as a parameter, either through a primitive value or as a field of an object. Based on these results we first observe that, for the two (real) applications considered, symbolic values may indeed cross the program boundaries and create problems for symbolic execution. We also observe that our technique is successful in identifying such problematic cases and in identifying methods that, although potentially problematic, are guaranteed to never be actually reached by symbolic values. For NanoXML and Antlr, our analysis lets users focus their attention on only 15% of the potentially-problematic calls.

RQ3. Our analysis discovers that symbolic values are confined within approximately one fifth of the total number of methods for both subjects. Furthermore, within methods that may handle symbolic values, less than half of the statements are actually affected by these values. Our translator is therefore able to transform the program so that half of the statements can be executed without incurring any overhead due to symbolic execution.

Precision. Our analysis is conservative and can be imprecise in some cases (i.e., it may conclude that a variable may store symbolic values even if it never does so in reality). Although context-sensitivity increases the precision significantly, the underlying points-to analysis does not scale beyond 2-cfa for our subjects, and stinger can thus produces imprecise results. For example, for NanoXML, we found that many standard library classes are unnecessarily transformed because

of the imprecision of the analysis. We believe that this imprecision could be reduced by using a demand-driven, highly-precise points-to analysis.

5 Related Work

Our work is related to approaches that provide tool support for abstraction in model checking (e.g., [4,7]). In [4], type inference is used to identify a set of variables that can be removed from a program when building a model for model checking. In [7], a framework for type inference and subsequent program transformation is proposed. In both approaches, the type-inference algorithm used is not as precise as our type-dependence analysis. Precision is crucial for our goal of reducing manual intervention and reducing the transformations that must be performed. However, unlike our work, where only one kind of abstraction (concrete to symbolic) is supported, the framework in [7] allows multiple user-defined abstractions.

Our approach to symbolic execution (i.e., execution of a transformed program) is also used in several other approaches (e.g., [2,11,19]). These approaches, however, transform the entire program, whereas our technique leverages type-dependence analysis to transform only the parts of the program actually affected by the symbolic execution. In this way, our technique reduces both the manual intervention and the amount program transformation needed. Also related to ours is the technique presented in [5], which is based on executing the program symbolically. The technique differs from our approach because it does not transform the program, but executes it using a virtual machine with a special semantics that support symbolic values.

Finally, being our type-dependence analysis a specific instance of flow analysis, it bears similarity to other approaches based on flow analysis, such as taint analysis [20] and information-flow analysis [26]. Our demand-driven formulation of type-dependence analysis is similar to the formulation of points-to analysis in [22], and our cloning-based approach to interprocedural analysis and use of binary decision diagrams to make context-sensitive analysis scale were studied in [27] and [1], respectively.

6 Conclusion

In this paper, we address two problems that hinder the application of symbolic execution to real software: (1) the generation of constraints that the decision procedure in use cannot handle and (2) the flow of symbolic values outside the program boundary. We present type-dependence analysis, which automatically and accurately identifies places in the program where these two problems occur, and a technique that uses the analysis results to help users address the identified problems. We also present a program-transformation technique that leverages the analysis results to selectively transform applications into applications that can be symbolically executed. We have implemented the analysis and transformation techniques in a tool, STINGER, that is integrated with Java

Pathfinder's symbolic execution engine. In our empirical evaluation, we applied STINGER to two Java applications. The results show that the problems that we target do occur in real applications, at least for the subjects considered, and that our analysis can identify these problems automatically and help users to address them. Moreover, we show that our analysis is precise enough to allow for transforming only the part of the code actually affected by symbolic values at runtime.

In future work, we plan to use STINGER for generating test inputs for real software and investigate techniques for guiding symbolic execution to exercise new program behaviors (e.g., coverage of specific program states). In this paper, we consider program boundaries defined by pragmatic reasons, such as interfaces with external libraries. In the future, we will investigate the application of our approach to cases where the boundaries are defined by the user (e.g., to exclude part of the system and thus reduce the state space to explore).

References

1. M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using BDDs. In *PLDI*, pages 103–114, 2003.
2. C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: Automatically generating inputs of death. In *CCS*, pages 322–335, 2006.
3. C. Cadar, P. Twohey, V. Ganesh, and D. R. Engler. EXE: A system for automatically generating inputs of death using symbolic execution. Technical Report CSTR 2006-01, Stanford University., 2006.
4. D. Dams, W. Hesse, and G. J. Holzmann. Abstracting C with abC. In *CAV*, pages 515–520, 2002.
5. 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.
6. B. Dutertre and L. de Moura. A Fast Linear-Arithmetic Solver for DPLL(T). In *CAV*, pages 81–94, 2006.
7. M. B. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. S. Pasareanu, Robby, H. Zheng, and W. Visser. Tool-supported program abstraction for finite-state verification. In *ICSE*, pages 177–187, 2001.
8. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI*, pages 234–245, 2002.
9. V. Ganesh and D. Dill. System Description of STP. http://www.csl.sri.com/users/demoura/smt-comp/descriptions/stp.ps.
10. P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *PLDI*, pages 213–223, 2005.
11. W. Grieskamp, N. Tillmann, and W. Schulte. XRT–exploring runtime for .NET architecture and applications. *Electr. Notes Theor. Comp. Sci.*, 144(3):3–26, 2006.
12. S. Horwitz, T. W. Reps, and S. Sagiv. Demand interprocedural dataflow analysis. In *FSE*, pages 104–115, 1995.
13. Java PathFinder. http://javapathfinder.sourceforge.net.
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. Symbolic execution and program testing. *CACM*, 19(7):385–394, 1976.

16. O. Lhoták and L. J. Hendren. Jedd: a BDD-based relational extension of Java. In *PLDI*, pages 158–169, 2004.
17. W. Pugh. The Omega test: a fast and practical integer programming algorithm for dependence analysis. In *SC*, pages 4–13, 1991.
18. T. W. Reps. Program analysis via graph reachability. In *ILPS*, pages 5–19, 1997.
19. K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *FSE*, pages 263–272, 2005.
20. U. Shankar, K. Talwar, J. S. Foster, and D. Wagner. Detecting format-string vulnerabilities with type qualifiers. In *USENIX Security Symposium*, pages 201–218, 2001.
21. O. Shivers. Control-flow analysis in Scheme. In *PLDI*, pages 164–174, 1988.
22. M. Sridharan, D. Gopan, L. Shan, and R. Bodík. Demand-driven points-to analysis for Java. In *OOPSLA*, pages 59–76, 2005.
23. E. Tilevich and Y. Smaragdakis. Transparent program transformations in the presence of opaque code. In *GPCE*, pages 89–94, 2006.
24. 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.
25. W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In *ISSTA*, pages 97–107, 2004.
26. D. M. Volpano, C. E. Irvine, and G. Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996.
27. J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, pages 131–144, June 2004.
28. T. Xie, D. Marinov, W. Schulte, and D. Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *TACAS*, pages 365–381, 2005.

A Type-Dependence Analysis for Fields and Entities of Array-Types

Sym, computed by the fix-point algorithm described in section 2, contains only local variables, static fields, formal parameters, and return values of scalar types that are type-dependent on variables in Sym_0. In this section, we describe how type-dependent instance fields and entities of array-types are computed from Sym.

The type-dependent instance fields are represented by the set $\{f\ s.t.\ y \xleftarrow{put[f]} x, x \in Sym\}$. In other words, a field f is type-dependent on a variable in Sym_0 if the value of a local variable x that is type-dependent on Sym_0 is stored into field f of some reference variable y. To compute the type-dependent entities of array-types, the algorithm first computes a set of program statements that allocate arrays Arr_s, as follows:

$$Arr_s = \{s\ s.t.\ s \in pt(a), a \xleftarrow{put[elem]} x \text{ or } a \xleftarrow{put[length]} x \text{ in TDG}, x \in Sym\}$$

A statement that allocates an array a is in Arr_s if either (1) a may store a value that is not type-compatible with a's current element-type, or (2) *length* of a may not be of integer type as a result of change in the types of variables in

Sym_0. $pt(a)$ returns all of the statements that allocates arrays to which a local variable a of array-type may point-to at run-time. After computing Arr_s, the entities of array-type that are type-dependent on variables in Sym_0 are given by the set $\{v \ s.t. \ pt(v) \cap Arr_s \neq \Phi\}$. In other words, an entity of array-type is type-dependent on a variable in Sym_0 if it may store an array allocated by one of the statements in Arr_s.

B Transformation Rules

Fig. 8 shows the transformation rules for statements referencing arrays.

Original statement Transformed statement	Guard	
$[s : l = \texttt{new}\ \tau[i]]$		
$[\overline{l} = \texttt{new\_array}^{\overline{\tau}}(< \overline{i} >)]$	$s \in Arr_s$	(11)
$[l_1 = l_2.\texttt{length}^{(\tau)}]$		
$[\overline{l_1} = \texttt{box}^{\text{EXPR}}(l_2.\texttt{length})]$	$\overline{l_1} \neq l_1, \overline{l_2} = l_2$	(12)
$[l_1 = \texttt{unbox}^{\text{EXPR}}(\texttt{array\_len}^{\overline{\tau}}(\overline{l_2}))]$	$\overline{l_1} = l_1, \overline{l_2} \neq l_2$	(13)
$[\overline{l_1} = \texttt{array\_len}^{\overline{\tau}}(\overline{l_2})]$	$\overline{l_1} \neq l_1, \overline{l_2} \neq l_2$	(14)
$[l[i_1]^{(\tau)} = i_2]$		
$[\texttt{array\_set}^{\overline{\tau}}(< \overline{l} >, < \overline{i_1} >, < \overline{i_2} >)]$	$\tau \in$ NumType, $\overline{l} \neq l$ or $\overline{i_1} \neq i_1$	(15)
$[\texttt{array\_set}^{\overline{\tau}}(< \overline{l} >, < \overline{i_1} >, i_2)]$	$\tau \in$ RefType or $\tau =$ boolean, $\overline{l} \neq l$ or $\overline{i_1} \neq i_1$	(16)
$[l_1 = l_2[i]^{(\tau)}]$		
$[\overline{l_1} = \texttt{box}^{\tau}(l_2[i])]$	$\overline{l_1} \neq l_1, \overline{l_2} = l_2, \overline{i} = i$	(17)
$[\overline{l_1} = \texttt{array\_get}^{\overline{\tau}}(< \overline{l_2} > , < \overline{i} >)]$	$\tau \in$ NumType, $\overline{l_2} \neq l_2$ or $\overline{i} \neq i, \overline{l_1} \neq l_1$	(18)
$[l_1 = \texttt{unbox}^{\text{EXPR}}(\texttt{array\_get}^{\overline{\tau}}(< \overline{l_2} > , < \overline{i} >))]$	$\tau \in$ NumType, $\overline{l_2} \neq l_2$ or $\overline{i} \neq i, \overline{l_1} = l_1$	(19)
$[l_1 = \texttt{array\_get}^{\overline{\tau}}(< \overline{l_2} > , < \overline{i} >)]$	$\tau =$ boolean, $\overline{l_2} \neq l_2$ or $\overline{i} \neq i$	(20)
$[l_1 = (\tau)\ \texttt{array\_get}^{\overline{\tau}}(< \overline{l_2} > , < \overline{i} >)]$	$\tau \in$ RefType, $\overline{l_2} \neq l_2$ or $\overline{i} \neq i$	(21)

Fig. 8. Tranformation Rules (Continuation from Fig. 6.)