Context-Sensitive Relevancy Analysis for Efficient Symbolic Execution ⋆

Xin Li[1], Daryl Shannon[2], Indradeep Ghosh[3]
Mizuhito Ogawa[1], Sreeranga P. Rajan[3], and Sarfraz Khurshid[2]

[1] School of Information Science,
Japan Advanced Institute of Science and Technology, Nomi, Japan
[2] Department of Electrical and Computer Engineering,
University of Texas at Austin, Austin, TX, USA
[3] Trusted Systems Innovation Group,
Fujitsu laboratory of America, Sunnyvale, CA, USA

Abstract. Symbolic execution is a flexible and powerful, but computationally expensive technique to detect dynamic behaviors of a program. In this paper, we present a *context-sensitive relevancy analysis* algorithm based on weighted pushdown model checking, which pinpoints memory locations in the program where symbolic values can flow into. This information is then utilized by a code instrumenter to transform only relevant parts of the program with symbolic constructs, to help improve the efficiency of symbolic execution of Java programs. Our technique is evaluated on a generalized symbolic execution engine that is developed upon Java Path Finder with checking safety properties of Java applications. Our experiments indicate that this technique can effectively improve the performance of the symbolic execution engine with respect to the approach that blindly instruments the whole program.

1 Introduction

A recent trend of model checking is to combine with the power of dynamic execution, such as simulation and constraint solving. Remarkable progress on both hardware and efficient decision procedures, such as Presburger arithmetic, satisfiability check on various logic, equality with uninterpreted function symbols, and various constraint solving, has made such a combination of model checking and off-the-shelf decision procedures more practical. For instance, symbolic execution [1], a classic technique for test-input generation, has been integrated into such model checking frameworks, including Bogor/Kiasan [2] and various extensions of Java Path Finder (JPF) [3, 4]. JPF has been combined with decision procedures such as first-order provers CVClite and Simplify, the SMT solver Yices, the Presburger arithmetic constraint solver OMEGA, and the constraint solver STP on bit-vectors and arrays, for correctness checking and automated

⋆ D.Shannon was an intern at Fujistu Labs. Sunnyvale during this work.

test-input generation. Symbolic execution interprets the program over symbolic values and allows model checking to reason variables with infinite data domain. Though such exhaustive checking is very powerful, it is computationally expensive. Further sophistication is needed to make it scale to industrial size software applications.

Fig. 1 shows an example of code instrumentation from the Java code fragment (the left-hand side) into its symbolic counterpart (the right-hand side) for symbolic execution. In the *Driver* class, the String *s* is designated as symbolic (by the assignment of *Symbolic.string()* to *s*). The methods of *IncL()*, *GetL()*, and the constructor *Limit(int x)* of the class *Limit* does not need a symbolic version as no symbolic values ever flow into these methods. However, in the symbolic execution of Java programs, most of existing approaches transform the entire program with regarding all program entities as symbolic. Blind instrumentation will incur unnecessary runtime overhead on symbolic execution along with the extra time required to instrument the entire program. Therefore, some program analysis is expected to help identify the part of program entities that are subject to symbolic execution at run-time.

```
public class Limit{
  int v = 0;
  Limit (int x){this.v = x;}
  int GetL(){return this.v;}
  int IncL(int t) {return this.v + t;}
  String CutExcess(String s){
  if(s.length() > v)
     return s.substring(0, v);
  else return s;
  }
}
public class Driver{
   public static void main(String[] args){
      String s = Symbolic.string();
      int i = 7;
      Limit limit = new Limit(i);
      limit.IncL(i);
      s = limit.CutExcess(s);
 }
}
```

```
public class Limit implements Symbolic{
  int v = 0;
  Limit (int x){this.v = x;}
  int GetL(){return this.v;}
  int IncL(int t) {return this.v + t;}
  String CutExcess(String s) {
  return ___CutExcess(StringExpr.
            _constant(s))._getValue();
  }
  StringExpr ___CutExcess(StringExpr s){
  if(Symbolic._GT(s.___length(), v))
  return s.___substring(Symbolic.IntConstant(0),
        Symbolic.IntConstant(v));
  else return s;
  }
}

public class Driver{
  public static void ___main(String args[]){
     StringExpr s = new Symbolic.string();
     int i = 7;
     Limit limit = new Limit(i);
     limit.IncL(i);
     s = limit.___CutExcess(s);
  }
  public static void main(String args[]){
     ___main(args);
  }
}
```

Fig. 1. A Java Example for Code Instrumentation of Symbolic Execution

This paper makes the following primary contributions:

– We present an interprocedural *relevancy analysis* (RA), formalized and implmented as weighted pushdown model checking [5] with PER-based abstraction [6]. Our RA is context-sensitive, field-sensitive, and flow-insensitive, and conservatively detects the set of memory locations (i.e., program variables of various kinds) where symbolic values can flow into. Then the instrumenter can use this information to instrument only the relevant parts of the program with symbolic constructs, thereby improving the performance of symbolic execution and code instrumentation itself.
– We perform experiments on the generalized symbolic execution engine [3], which is developed upon JPF, for checking safety properties on three Java applications. Relevancy analysis is used as the preprocessing step to detect program variables that may store symbolic values at run-time. Only these portions of the applications are later transformed using a code instrumenter. Experimental results indicate that our technique can effectively improve the performance of the symbolic execution engine with respect to the approach that blindly instruments the whole program.

The rest of the paper is organized as follows. In Section 2 the relevancy analysis based on weighted pushdown model checking techniques is presented. In Section 3 we describe in detail how Java programs are abstracted and modelled for relevancy analysis. Experimental results are presented and discussed in Section 4, and related work is surveyed in Section 5. Section 6 concludes the paper with a description of our future work.

2 Context-sensitive Relevancy analysis

2.1 Interprocedural Program Analysis by Weighted Pushdown Model Checking

Definition 1. *A* ***pushdown system*** $P = (Q, \mathit{\Gamma}, \mathit{\Delta}, q_0, w_0)$ *is a pushdown automaton regardless of input, where* Q *is a finite set of states called control locations, and* $\mathit{\Gamma}$ *is a finite set of stack alphabet, and* $\mathit{\Delta} \subseteq Q \times \mathit{\Gamma} \times Q \times \mathit{\Gamma}^*$ *is a finite set of transition rules, and* $q_0 \in Q$ *and* $w_0 \in \mathit{\Gamma}^*$ *are the initial control location and stack contents respectively. We denote the transition rule* $((q_1, w_1), (q_2, w_2)) \in \mathit{\Delta}$ *by* $\langle q_1, w_1 \rangle \hookrightarrow \langle q_2, w_2 \rangle$*. A* ***configuration*** *of* P *is a pair* $\langle q, w \rangle$*, where* $q \in Q$ *and* $w \in \mathit{\Gamma}^*$*.* $\mathit{\Delta}$ *defines the transition relation* $\Rightarrow$ *between pushdown configurations such that if* $\langle p, \gamma \rangle \hookrightarrow \langle q, \omega \rangle$*, then* $\langle p, \gamma\omega' \rangle \Rightarrow \langle q, \omega\omega' \rangle$*, for all* $\omega' \in \mathit{\Gamma}^*$*.*

A pushdown system is a transition system with a finite set of control states and an unbounded stack. A weighted pushdown system extends a pushdown system by associating a weight to each transition rule. The weights come from a bounded idempotent semiring.

Definition 2. *A* ***bounded idempotent semiring*** $S = (D, \oplus, \otimes, 0, 1)$ *consists of a set* D $(0, 1 \in D)$ *and two binary operations* $\oplus$ *and* $\otimes$ *on* D *such that*

1. *$(D, \oplus)$ is a commutative monoid with 0 as its neutral element, and $\oplus$ is idempotent, i.e., $a \oplus a = a$ for $a \in D$;*
2. *$(D, \otimes)$ is a monoid with 1 as the neutral element;*
3. *$\otimes$ distributes over $\oplus$. That is, $\forall a, b, c \in D$, $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$ and $(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$;*
4. *$\forall a \in D, a \otimes 0 = 0 \otimes a = 0$;*
5. *The partial ordering $\sqsubseteq$ is defined on D such that $\forall a, b \in D, a \sqsubseteq b$ iff $a \oplus b = a$, and there are no infinite descending chains on D wrt $\sqsubseteq$.*

Remark 1. As stated in Section 4.4 in [5], the distributivity of $\oplus$ can be loosened to $a \otimes (b \oplus c) \sqsubseteq (a \otimes b) \oplus (a \otimes c)$ and $(a \oplus b) \otimes c \sqsubseteq (a \otimes c) \oplus (b \otimes c)$.

Definition 3. *A **weighted pushdown system** is a triple $W = (P, S, f)$, where $P = (Q, \Gamma, \Delta, q_0, w_0)$ is a pushdown system, $S = (D, \oplus, \otimes, 0, 1)$ is a bounded idempotent semiring, and $f : \Delta \rightarrow D$ is a function that assigns a value from D to each rule of P.*

Definition 4. *Consider a weighted pushdown system $W = (P, S, f)$, where $P = (Q, \Gamma, \Delta, q_0, w_0)$ is a pushdown system, and $S = (D, \oplus, \otimes, 0, 1)$ is a bounded idempotent semiring. Assume $\sigma = [r_0, ..., r_k]$ to be a sequence of pushdown transition rules, where $r_i \in \Delta (0 \leq i \leq k)$, and $v(\sigma) = f(r_0) \otimes ... \otimes f(r_k)$. Let path(c,c$'$) be the set of all rule sequences that transform configurations from c into c$'$. Let $C \subseteq Q \times \Gamma^*$ be a set of regular configurations. The **generalized pushdown reachability problem**(GPR) is to find for each $c \in Q \times \Gamma^*$:*

$$\delta(c) = \bigoplus \{ v(\sigma) | \sigma \in path(c, c'), c' \in C \}$$

Efficient algorithms for solving GPR are developed based on the property that the regular set of pushdown configurations is closed under forward and backward reachability [5]. There are two off-the-shelf implementations of weighted pushdown model checking algorithms, Weighted PDS Library [4], and WPDS+ [5]. We apply the former as the back-end analysis engine for relevancy analysis.

The GPR can be easily extended to answer the"meet-over-all-valid-paths" program analysis problem MOVP(EntryPoints, TargetPoints), which intends to conservatively approximate properties of memory configurations at given program execution points (represented as TargetPoints) induced by all possible execution paths leading from program entry points (represented as EntryPoints). A *valid* path here satisfies the requirement that a procedure always exactly returns to the most recent call site in the analysis. The encoding of a program into a weighted PDS in a (flow-sensitive) program analysis [5] typically models program variables as control locations and program execution points (equivalently, line numbers) as stack alphabet. The weighted domain is designed as follows:

[4] http://www.fmi.uni-stuttgart.de/szs/tools/wpds/
[5] http://www.cs.wisc.edu/wpis/wpds++/

- A weight function represents the data flow changes for each program execution step, such as transfer functions;
- $f \oplus g$ represents the merging of data flow at the meet of two control flows;
- $f \otimes g$ represents the composition of two sequential control flows;
- $\mathbf{1}$ implies that an execution step does not change each datum; and
- $\mathbf{0}$ implies that the program execution is interrupted by an error.

Moreover, assume a Galois connection (L, α, γ, M) between the concrete domain L and the abstract domain M, and a monotone function space $\mathcal{F}_l : L \to L$. Taking weight functions from the monotone function space $\mathcal{F}_m : M \to M$ defined as $\mathcal{F}_m = \{f_m \mid f_m \supseteq \alpha \circ f_l \circ \gamma,\ f_l \in \mathcal{F}_l\}$, a sound analysis based on weighted pushdown systems can be ensured according to abstract interpretation.

2.2 PER-based Abstraction and Relevancy Analysis Infrastructure

For program verification at source code level, it is a well understood methodology that *program analysis can be regarded as model checking of abstract interpretation* [7] on an *intermediate representation* (IR) of the target program. Our RA is designed and implemented as weighted pushdown model checking following this methodology.

The infrastructure of our RA is shown in Figure 2, with a soot [6] compiler as the front-end preprocessor and the Weighted PDS Library as the back-end model checking engine. The analysis starts off preprocessing by soot from Java to Jimple [8], which is a typed three-address intermediate representation of Java. In the meantime, points-to Analysis (PTA) is performed and thus a call graph is constructed. After preprocessing, Jimple codes are abstracted and modeled into a weighted PDS, and the generated model is finally checked by calling the Weighted PDS Library. The set of symbolic variables is detected and output into an XML file for later use by the code instrumenter.

We choose Jimple as our target language since its language constructs are much simpler than those of either Java source code or Java Bytecode. Although the choice of PTA is independent of RA, the precision of RA depends on the precision of PTA, in that (i) call graph construction and PTA are mutually dependent due to dynamic method dispatch; and (ii) a precise modelling on instance fields (a.k.a., field-sensitivity [9]), array references, and containers (Section 4.3) depends on PTA to cast aliasing.

The objective of our relevancy analysis is to compute the set of program variables of interested type that are *relevant* to any designated variables that are meant to be symbolic. We mark a variable as *relevant* if it can store symbolic values at run-time. Our relevancy analysis is leveraged from an interprocedural irrelevant code elimination [10]. The idea is that, if the change of a value does not affect the value of outputs, we regard it as irrelevant, and relevant otherwise.

[6] http://www.sable.mcgill.ca/soot/

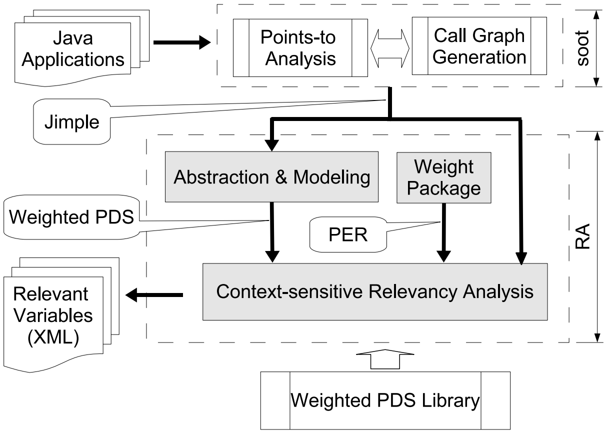

Fig. 2. The Analysis Infrastructure.

The weighted domain for this analysis is constructed on a 2-point abstract domain L (Definition 5) based on a partial equivalence relation (PER). A PER on a set S is a transitive and symmetric relation $S \times S$. It is easy to see that $\gamma\ l$ is a PER for all $l \in L$. Our relevancy analysis works with an interpretation on L as follows: ANY is interpreted as any values, and ID is interpreted as fixed values. Designating a seed variable x to be ANY in a program, a variable y is relevant to x if its value can be ANY at run-time.

Definition 5. *Define the 2-point abstract domain L as $L = \{\text{ANY}, \text{ID}\}$, with the ordering* ANY $\supset$ ID*. Taking the concrete domain D as integers or other data sets of interest, the concretization γ of L is defined as $\gamma\ \text{ANY} = \{(x,y) \mid x, y \in D\}, \quad \gamma\ \text{ID} = \{(x,x) \mid x \in D\}$.*

Definition 6. *Define a set of transfer functions $\mathcal{F}: \ L \to L$ as*

$$\mathcal{F} = \{\lambda x.x, \lambda x.\text{ANY}, \lambda x.\text{ID} \ \mid x \in L\}$$

Let $f_0 = \lambda x.\text{ANY}$, $f_1 = \lambda x.x$, and $f_2 = \lambda x.\text{ID}$. We have $\forall x \in L,\ f_0\ x \supset f_1\ x$ and $f_1\ x \supset f_2\ x$. Thus, $\mathcal{F}$ is a monotone function space with the ordering $f_0 \sqsupseteq f_1 \sqsupseteq f_2$, where $f \sqsupseteq f'$ iff $\forall x \in L, fx \supset f'x$. The weighted domain in our analysis thus consists of $\mathcal{F}$ plus **0** element, and binary operations over weights are correspondingly induced by the ordering $\sqsupseteq$.

3 Modelling Java Programs

3.1 Building the Weighted Dependence Graph

Provided with a Java points-to analyzer, the analysis first builds a *weighted dependence graph* (WDG), a directed and labelled graph $G = (N, L, \rightsquigarrow)$. The

WDG is then encoded as the underlying weighted pushdown system for model checking. Let $\texttt{Var}$ denote the set of program variables of interested type which consist of local variables and field or array references, and let $\texttt{ProS}$ denote the set of *method identifiers* which is identified as a pair of class names and method signatures. $N \subseteq \texttt{Var} \times \texttt{ProS}$ is a set of nodes and each of them represents a program variable and the method where it resides. $L \subseteq \mathcal{F}$ is a set of labels, and $\rightsquigarrow \subseteq N \times L \times N$ is a set of directed and labelled edges that represent some dependence among variables regarding the changes of data flow. By $v_1 \overset{l}{\rightsquigarrow} v_2$, we mean that there is a data flow from v_1 to v_2 represented by a weight l. A WDG can be regarded as an instance of the exploded supergraph [11].

Let $\texttt{Stmt}$ be the set of Jimple statements and let $\mathcal{P}$ be the powerset operator. An evaluation function $\mathcal{A}[\![\_]\!] : \texttt{Stmt} \to \mathcal{P}(\rightsquigarrow)$, which models Jimple statements (from a method $f \in \texttt{ProS}$) into edges in G, is given in Table 1, where

- $\texttt{GlobVar}$ ($\subseteq \texttt{Var}$) denotes the set of static fields and instance fields, as well as array references, in the analysis after casting aliasing;
- $\texttt{env}$ ($\in \texttt{GlobVar}$) denotes the program environment that allocates new memories; $\texttt{SymVal}$ and $\texttt{ConstVal}$ denotes symbolic values and program constants respectively; $\texttt{binop}$ denotes binary operators;
- $\texttt{pta}(r, \texttt{cc})$ ($cc \in \texttt{CallingContexts}(f)$) denotes points-to analysis on a reference variable $\texttt{r}$ with respect to calling contexts $\texttt{cc}$, and $\texttt{CallingContexts}(f)$ represents the calling contexts of a method f where r resides;
- $[\![o]\!]$ ($o \in \texttt{pta}(r)$) denotes the unique representative of array members $r[i]$ after calling points-to analysis on the base variable r;
- $\texttt{rp}(\in \texttt{RetP})$ is a return point associated with a method invocation. Return points denoted by $\texttt{RetP}$ are introduced in addition to *method identifiers*, so that each method invocation is assigned with a unique return point;
- $f'_{\texttt{ret}}$ (resp. $f'_{\texttt{arg}_i}$) is a variable that denotes a return value (resp. the i-th parameter) of the method $f' \in \texttt{ProS}$.

$\lambda x.\textsc{any}$ models that $\texttt{env}$ assigns symbolic values to seed variables; $\lambda x.\textsc{id}$ models that $\texttt{env}$ assigns variables with program constants; $\lambda x.x$ models that a data flow is kept unchanged. For readability, the label $\lambda x.x$ (on $\rightsquigarrow$) is omitted in the table.

Our analysis is *context-sensitive* by encoding the program as a pushdown system, and thus calling contexts that can be infinite are approximated as regular pushdown configurations. A WDG G is encoded into a Weighted PDS as follows,

- The set of control locations is the first projection of N ($\subseteq \texttt{Var}$);
- The stack alphabet is $\texttt{ProS} \cup \texttt{RetP}$;
- The weighted domain is the set of labels L;
- Let $(v_1, f_1) \overset{l}{\rightsquigarrow} (v_2, f_2) \in E$ for $l \in L$ such that
  - $\langle v_1, f_1 \rangle \hookrightarrow \langle v_2, f_2 \rangle$ if $f_1 = f_2$,
  - $\langle v_1, f_1 \rangle \hookrightarrow \langle v_2, f_2 f_r \rangle$ if the method f_1 calls f_2 with f_r designated as the return point, and
  - $\langle v_1, f_1 \rangle \hookrightarrow \langle v_2, \epsilon \rangle$ if $f_2 \in \texttt{RetP}$.

Table 1. Rules for Building the Weighted Dependence Graph

$$
\begin{array}{l}
\mathcal{A}\llbracket x = \mathtt{SymVal} \rrbracket = \{(\mathtt{env}, f) \overset{\lambda.\text{ANY}}{\rightsquigarrow} (x, f)\} \\
\mathcal{A}\llbracket x = \mathtt{ConstVal} \rrbracket = \{(\mathtt{env}, f) \overset{\lambda.\text{ID}}{\rightsquigarrow} (x, f)\} \\
\mathcal{A}\llbracket x = y \rrbracket = \{(y, f) \rightsquigarrow (x, f)\} \\
\mathcal{A}\llbracket z = x \ \mathtt{binop} \ y \rrbracket = \{(x, f) \rightsquigarrow (z, f), (y, f) \rightsquigarrow (z, f)\} \\
\mathcal{A}\llbracket x = r[n] \rrbracket = \{(\llbracket o \rrbracket, f) \rightsquigarrow (x, f) \mid o \in \mathtt{pta}(r, \mathtt{cc})\} \\
\mathcal{A}\llbracket r[n] = x \rrbracket = \{(x, f) \rightsquigarrow (\llbracket o \rrbracket, f) \mid o \in \mathtt{pta}(r, \mathtt{cc})\} \\
\mathcal{A}\llbracket x = r.g \rrbracket = \{(o.g, f) \rightsquigarrow (x, f) \mid o \in \mathtt{pta}(r, \mathtt{cc})\} \\
\mathcal{A}\llbracket r.g = x \rrbracket = \{(x, f) \rightsquigarrow (o.g, f) \mid o \in \mathtt{pta}(r, \mathtt{cc})\} \\
\mathcal{A}\llbracket x = \text{lengthof } r \rrbracket = \{(o.len, f) \rightsquigarrow (x, f) \mid o \in \mathtt{pta}(r, \mathtt{cc})\} \\
\mathcal{A}\llbracket r = \text{newarray RefType } [x] \rrbracket = \{(x, f) \rightsquigarrow (o.len, f) \mid o \in \mathtt{pta}(r, \mathtt{cc})\} \\
\mathcal{A}\llbracket \text{return } x \rrbracket = \{(x, f) \rightsquigarrow (f_{\mathtt{ret}}, f)\} \\
\mathcal{A}\llbracket x := \text{@parameter}_k : \ \text{Type} \rrbracket = \{(f_{\mathtt{arg}_k}, f) \rightsquigarrow (x, f)\} \\
\mathcal{A}\llbracket x := (Type)y \rrbracket = \{(y, f) \rightsquigarrow (x, f)\} \\
\mathcal{A}\llbracket z = x.f'(m_1, ..., m_l, m_{l+1}, ...m_n) \rrbracket = \\
\qquad \{(m_i, f) \rightsquigarrow (f'_{\mathtt{arg}_i}, f') \mid 1 \leq i \leq l\} \cup \{(f'_{\mathtt{ret}}, f') \rightsquigarrow (f'_{\mathtt{ret}}, \mathtt{rp})\} \\
\qquad \cup \{(f'_{\mathtt{ret}}, \mathtt{rp}) \rightsquigarrow (z, f)\} \cup \{(v, f) \rightsquigarrow (v, f') \mid v \in \mathtt{GlobVar}\} \\
\qquad \cup \{(v, f') \rightsquigarrow (v, \mathtt{rp}) \mid v \in \mathtt{GlobVar}\} \cup \{(v, \mathtt{rp}) \rightsquigarrow (v, f) \mid v \in \mathtt{GlobVar}\}
\end{array}
$$

where $m_i (1 \leq i \leq l)$ are variables of interested type, and $\mathtt{cc} \in \mathtt{CallingContexts}(f)$.

Our analysis is also *field-sensitive*, so that not only different instance fields of an object are distinguished (otherwise called field-based analysis), but also are instance fields that belong to different objects (otherwise called field-insensitive analysis). Note that array references are treated similarly as instance fields. The abstraction we take is to ignore the indices of arrays, such that members of an array are not distinguished. Both field and array references can be nested, and we choose to avoid tracking such a nesting in the analysis by cast aliasing on their base variables with calling a points-to analysis.

Considering efficiency, we perform a *flow-insensitive* analysis, i.e., each method is regarded as a set of instructions by ignoring their execution order. Note that soot compiles Jimple in a SSA-like (Static Single Assignment) form. When a program is in the SSA-like form, a flow-insensitive analysis on it is expected to enjoy a similar precision of that of a flow-sensitive analysis [9], except that, in a flow-insensitive analysis, the return points of call sites also shrink to the nodes (i.e., methods) of the call graph. Thus, calling contexts of a method that is multiply invoked from one method are indistinguishable due to sharing the same return points. We remedy such a precession loss by associating a unique return point with each invocation site.

Definition 7. *Assume that the program under investigation starts with the entry point* $\mathtt{ep} \in \mathtt{ProS}$*. Our relevancy analysis on a variable* $v \in \mathtt{Var}$ *that resides in the method* $s \in \mathtt{ProS}$ *computes* $\mathtt{ra}(v, m) = \mathtt{MOVP}(\mathtt{S}, \mathtt{T})$*, where* $\mathtt{S} = \langle \mathit{env}, \mathtt{ep} \rangle$ *and* $\mathtt{T} = \langle v, m.(\mathtt{RetP})^* \rangle$*.* v *is marked as relevant if and only if* $\mathtt{ra}(v, m)$ *returns* $\lambda x.\text{ANY}$.

Remark 2. To compute `MOVP(S, T)`, our analysis calls the Weighted PDS Library to (i) first construct a weighted automaton that recognizes all pushdown configurations reachable from `S`; and (ii) then read out weights associated with pushdown configurations from `T` with respect to the variables of interest. The latter phase seems not to be a dominant factor in practice, and the time complexity of the former is $O(m\ n^2)$, where m is the number of variables and n is the program size (Lemma 1 in [12]).

Remark 3. We are interested in variables of primite type, strings, and the classes explicitly modelled (Appendix A) in the analysis. An array is regarded as symbolic if its unique representative is detected as symbolic by the analysis. Since our analysis is field-sensitive, an instance field f of a class is regarded as symbolic, if it is detected as symbolic when belonging to any instance o of this class, i.e., when $o.f$ is detected as symbolic.

3.2 Precision Enhancement by Refined Modelling on Globals

A typical approach to perform context-sensitive analysis is based on *context-cloning*. In such methods, program entities, such as methods and local variables, typically have a separate copy for different calling contexts. Since possible calling contexts can be infinite due to recursions, this infinity is often bounded by limiting the call depth within which the precision is preserved (like k-CFA analysis [13]) or by performing context-insensitive analysis on all the procedure calls involved in any recursions [14]. In contrast, our approach to context-sensitivity is based on *context-stacking*. That is, the infinite program control structures are modelled by the pushdown stack with no limit on recursions and procedure calls. The context-stacking-based approach has an advantage over the context-cloning-based approach when there are deep procedure calls, or when a large number of procedures is involved in various recursions in the program. However, in some cases, it can be less precise than the context-cloning-based analysis.

Example 1. Suppose the int s is designated as symbolic (by the assignment of $Symbolic.int()$) in the Java code fragment in Fig. 3 that uses class Limit in Fig. 1. For the driver, variables c and p are symbolic, but variables d and q are not.

Assume the heap objects allocated at lines 3 and 4 are respectively O_1 and O_2. Fig. 4 shows part of the WDG corresponding to lines 3-4 and 5-6. Each dotted circle demarcates a method. There are two kinds of nodes identified by variable names: circles for local variables and rectangles for global variables, such as instance fields and array references. Dashed edges are induced by the method invocation at line 6, which is to be distinguished from the method invocation at line 5. Return points for method invocations at lines 5 and 6 are represented as triangles, and named r_1 and r_2 respectively. The variable ret represents the return value of $IncL()$. Our analysis can precisely distinguish that c and d are returned from two invocations on the same method, and only c is relevant to s.

```
0.public class Driver {
1.  public static void main (String[] args){
2.      int s = Symbolic.int();
3.      Limit a = new Limit(s);
4.      Limit b = new Limit(5);
5.      int c = b.IncL(s);
6.      int d = b.IncL(5);
7.      int p = a.GetL();
8.      int q = b.GetL();
9.  }
10.}
```

Fig. 3. A Java Code Fragment

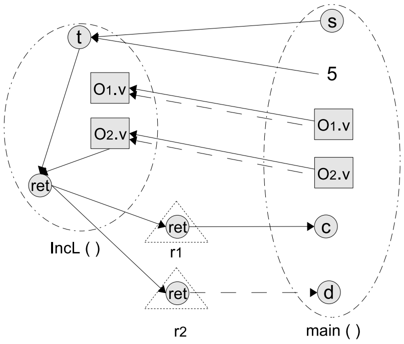

Fig. 4. The WDG for lines 3-4, 5-6

However, the analysis cannot correctly conclude that q can only store concrete values at run-time, whereas this case can be handled by the 1-CFA context-sensitive approach based on context-cloning. Fig. 5 shows part of the WDG corresponding to lines 3-4 and 7-8, where dashed edges are induced by the method invocation at line 6, and for readability, return points for line 7 and 8 are omitted in the figure. *GetL*() are invoked on objects O_1 and O_2 respectively from line 7 and 8, and instance fields of both $O_1.v$ and $O_2.v$ can flow to *ret* under two calling contexts. However, since the pushdown transition only depends on the control location (i.e., variable) and the topmost stack symbol (i.e., the method where the variable presently resides), the pushdown transitions are incapable of distinguishing under which calling contexts a global variable ($\in$ `GlobVar`) flows into a method. Therefore, pushdown transitions that model edges e_1 and e_2 cannot distinguish invocations from lines 7 and 8.

To remedy a precision, our choice is to refine modelling on global variables from `GlobVar` to avoid an invalid data flow. Such an extension is obtained by modifying rules from Table 1 in which global variables are involved. Table 2 shows some of the extensions on array references and method invocations. Assume that the calling context of main() is C_0, the calling contexts of GetL() are C_1 and C_2, and that the calling contexts of Limit(int x) are C_3 and C_4. Fig. 6 shows the refined version of the WDG shown in Fig. 5. Note that the precision of refined modelling closely depends on that of the underlying context-sensitive points-to analysis.

4 Evaluations

4.1 Configuration of the Evaluation Steps

Our evaluation of checking safety properties of Java web applications through symbolic execution generally consists of the following steps:

Table 2. Refined Modelling on Globals

$$\mathcal{A}[\![x = r[n]]\!] = \{(([\![o]\!], \mathtt{cc}), f) \rightsquigarrow (x, f) \mid o \in \mathtt{pta}(r, \mathtt{cc}), \mathtt{cc} \in \mathtt{CallingContexts}(f)\}$$

$$\begin{aligned}
&\mathcal{A}[\![z = x.f'(m_1, ..., m_l, m_{l+1}, ...m_n)]\!] = \\
&\quad \{(m_i, f) \rightsquigarrow (f'_{\mathrm{arg}_i}, f') \mid 1 \le i \le l\} \cup \{(f'_{\mathtt{ret}}, f') \rightsquigarrow (f'_{\mathtt{ret}}, \mathtt{rp})\} \cup \{(f'_{\mathtt{ret}}, \mathtt{rp}) \rightsquigarrow (z, f)\} \\
&\quad \cup \{((v, \mathtt{cc}), f) \rightsquigarrow ((v, \mathtt{cc}'), f') \mid v \in \mathtt{GlobVar},\ \mathtt{cc} \in \mathtt{CallingContexts}(f), \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathtt{cc}' \in \mathtt{CallingContexts}(f')\} \\
&\quad \cup \{((v, \mathtt{cc}'), f') \rightsquigarrow ((v, \mathtt{rp}), \mathtt{rp}) \mid v \in \mathtt{GlobVar}, \mathtt{cc}' \in \mathtt{CallingContexts}(f')\} \\
&\quad \cup \{((v, \mathtt{rp}), \mathtt{rp}) \rightsquigarrow ((v, \mathtt{cc}), f) \mid v \in \mathtt{GlobVar}, \mathtt{cc} \in \mathtt{CallingContexts}(f)\}
\end{aligned}$$

where $m_i (1 \le i \le l)$ are variables of concerned type.

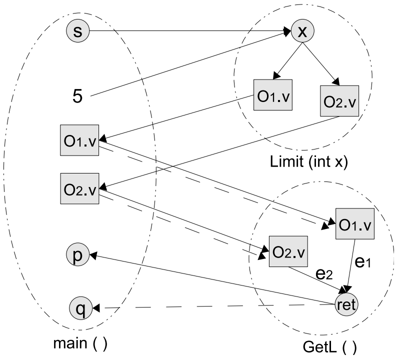

Fig. 5. The WDG for lines 2-4, 7-8

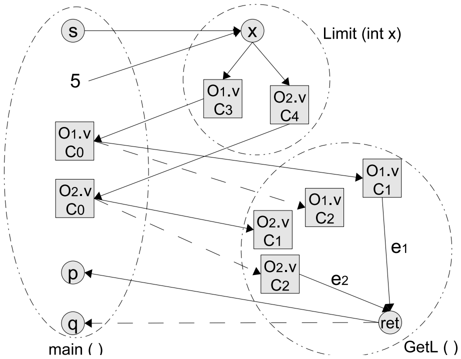

Fig. 6. Fig.5 with Refined Modeling

- **Environment Generation:** Since model checking techniques require a closed system to run on, the first step is to convert a heterogeneous web application that uses various components into a closed Java program. This process is known as *environment generation* [15], which decomposes web applications into the *module* that is typically the middle tier of web applications, which comprises the business logic and the *environment* with which the module interacts with. The environment is further abstracted into *drivers* from Java classes that hold a thread of control and *stubs* from the rest of Java classes and components of the application. After this step, all the applications consist of a driver file that provides all input values to the application. Typically, these values are provided by a user using forms in a webpage. The back-end database is abstracted as a series of stubs that use a two-dimentional table structure, and also to store the input data if needed.
- **Property Specification:** Once the model is generated, some specific input values in the drivers are made to be symbolic quantities, such as values of integer, float, Boolean, or String. For example, if the requirement is that the shopping cart total must be the product of the item price and the item quantity specified by the user, then the item price, quantity, and cart-total

are made to be symbolic entities. Sometimes this can be achieved using the input variables in the driver only. However, sometimes this also means changing the database stubs for inputs that were stored in the database (such as item price). The requirement is further inserted as an assertion comprising the symbolic entities and placed in an appropriate location in the program.

- **Code instrumentation:** The program is thus instrumented using a code instrumenter that replaces java codes that use concrete values with the counterparts that can handle symbolic values. For this purpose, extensive libraries have been developed that can handle symbolic integer, symbolic float, symbolic Boolean, *etc.*. The instrumenter uses the relevancy analysis to pinpoint the portions of the program that are required to tackle symbolic entities. The results of the relevancy analysis are conveyed as series of classes, methods, parameters, and variables to the instrumenter in an XML file.
- **Symbolic program model checking:** Finally, once the instrumentation phase is completed, a state-based model checker is used to check safety properties of interest. The symbolic libraries create a system of equations with the symbolic variables, whenever those variables are manipulated. At each control point that consists of symbolic variables, an off-the-shelf decision procedure is invoked to check whether the system of equations is satisfiable. If not, the exploration is terminated along that path. An assertion containing symbolic variables is inserted into the program at an appropriate point to check for the negation of the property being checked. When the assertion is hit, a solution to the equations points to the existence of a counterexample or bug. If there are no solutions, then the requirement holds.

4.2 Experimental Results

Our experimental platform is built upon JPF at the University of Texas at Austin. The targets of our experiments are Java applications as shown in Table 3. Note that these numbers reflect the size of the generated Java model. The original application is usually much larger since it is heterogeneous and consists of HTML pages, JSP code, some database code, *etc.*. It is extremely difficult to estimate the exact size of the original application.

Table 3. Benchmark Statistics

Application	Description	#classes	#lines
WebStore	Simple Web E-store	6	410
DB-Merge	Database Application	26	706
Petstore	SUN's J2EE Sample App.	752	23,701

We now discuss the efficiency issues of this whole exercise. Table 4 shows the CPU times (in seconds) for various parts of the process by comparing symbolic execution with blind instrumentation and symbolic execution with RA-based instrumentation. The underlying points-to analysis of the relevancy analysis is provided by soot, which is 1-CFA context-sensitive analysis with call graph constructed on-the-fly. Since symbolic execution is computationally expensive, it was impractical to check all symbolic inputs in one pass for large-scale applications. As a result various symbolic execution instances of the same application were created based on the requirement that was being checked. As shown in the Table, multiple properties over symbolic variables are checked in separate passes. Therefore, although the soot PTA phase is one of the dominant factors in the execution time, it has been reused across all these requirements for a particular application as only some different set of variables are marked as symbolic in the program in each instance. Thus this analysis comprises of an one-time cost and is amortized across all the requirements that are checked. Typically hundreds of requirements need to be checked for a medium size application. Hence, this time has not been included in the overall CPU runtimes. Moreover, note that the Soot PTA time is relatively large even for small examples like *WebStore* and does not increase that much for the larger example. This is due to its analysis of many Java library classes used by the example which dominate the runtime. These library analysis results can be cached and reused not only across different requirements in the same application but across different applications that use the same libraries. This will reduce the PTA overhead even further. We can observe that there is an average gain in overall runtime as well as reduction in instrumented code size due to the static analysis phase. The CPU time improvement can be as high as 61% for larger examples. This can only grow as the number and influence of symbolic inputs become smaller compared to the overall application size. The CPU times are for a 1.8Ghz dual core Opteron machine running the Redhat Linux operating system and having a memory of 4GB.

5 Related Work

Symbolic execution for model checking of Java programs has been proposed in [3]. However, it is well known that symbolic execution is computationally expensive and efforts have been made to reduce its complexity by abstracting out library classes [16]. A framework for type inference and subsequent program transformation for symbolic execution is proposed in [17] which allows multiple user-defined abstractions. Execution of a transformed program for symbolic execution has been used in several approaches. In most of those approaches, the whole program is transformed akin to our blind instrumentation technique [18], [19]. In [20], the performance of symbolic execution is enhanced by randomly concretizing some symbolic variables at the cost of coverage. Instead of transforming the source code, an enhanced Java virtual machine is used to symbolically execute code in [2].

The approach in [21] is closely related to this work. However, the focus of that paper is precise instrumentation through static analysis whereas the focus

Table 4. Performance of symbolic execution with instrumenter using static analysis

App. Program	Prop.	*Blind Instrumentation*			*RA based Instrumentation*					RT	CR
		Instr.	SECK	Total	PTA	RA	Instr.	SECK	Total		
WebStore	Prop.1	6.2	1.9	8.1	509	0.8	2.0	1.9	4.7	42%	43%
	Prop.2	6.2	2.9	9.1		0.9	2.0	2.8	5.7	37%	41%
DB-Merge	Prop.1	3.1	10.5	13.6	523	0.6	2.1	9.4	12.1	11%	19%
Petstore	Prop.1	36.9	259.2	296.1	575	1.2	4.8	109.4	115.4	61%	16%
	Prop.2	38.1	593.6	631.7		1.1	5.1	319.9	326.1	48%	16%
	Prop.3	39.2	2566.2	2605.4		1.2	5.4	1053.6	1060.2	59%	17%

Prop.: property being checked   Instr.: time for code instrumentation
PTA: time for points-to analysis   RA: time for relevancy analysis
Total: time for both Instr. and SECK   RT: percentage of runtime improvement
SECK: time for symbolic execution of requirement checking
CR: percentage of the reduction on the instrumented code size
All time above are measured in seconds

of this work is on performance enhancements through the static analysis phase. Approaches adopted in [21] and this work consider two streams of performing context-sensitive program analysis, i.e., context-cloning vs. context-stacking. The analysis in [21] borrows ideas and algorithms from points-to analysis, e.g., the match of field read and write operations are formalized as a CFL (Context Free Language)-reachability problem. Their approach to context-sensitivity is based on context-cloning following [14] and the k-CFA approach to handle procedural calls, whereas our analysis based on pushdown system complemented with the refined modelling on globals can reach a higher precision due to the absence of restriction on recursions and call depth. Note that we cannot compare our work with [21] as neither the tool described in that work or the versions of the example programs used there are in the public domain. No performance statistics are mentioned in that paper.

6 Conclusions

We formalized a context-sensitive, field-sensitive and flow-insensitive relevancy analysis as weighted pushdown model checking, to help the symbolic execution technique scale to realistic Java applications. Our analysis was used as a preprocessing step of symbolic execution, which helps in identifying relevant sections of a program where symbolic values can flow into.

We evaluated the methodology on the generalized symbolic execution platform, built upon JPF at the Univiersity of Texas at Austin. Though the dominant overhead of this methodology is the PTA phase, the results of PTA can be cached and reused. Experimental results indicate that, as the program size increases,, the performance gains obtained from the symbolic execution phase far outweigh the overhead of analysis and thus produce a significant gain in overall performance. Moreover, the symbolic programs thus obtained are much smaller than the ones obtained by blind transformation, which avoids running out of memory during symbolic execution.

The precision and scalability of the relevancy analysis is closely related to that of the underlying points-to analysis. Currently, we performed a 1-CFA context-sensitive on-the-fly points-to analysis provided by Soot. In this work, we limit our focus to the performance improvement of symbolic execution. We are planning to apply a context-sensitive points-to analysis based on weighted pushdown model checking, to see the room of precision enhancement. Points-to analysis on Java web applications is expensive since, even for a small web application, the libraries of the web applications are huge and easily reach millions of lines. Thus, more sophisticated treatments for analyzing libraries are expected. Another interesting direction is that our relevancy analysis can be regarded as an instance of the traditional *taint-style analysis*, thus it is applicable to other application scenarios such as security vulnerability check on Java web applications [22].

References

1. J.C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.
2. X. Deng, J. Lee, and Robby. Bogor/Kiasan: A k-bounded symbolic execution for checking strong heap properties of open systems. In *the 21st IEEE International Conference on Automated Software Engineering (ASE 2006)*, pages 157–166, 2006.
3. S. Khurshid, C. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2003)*, pages 553–568, 2003.
4. S. Anand, C.S. Pasareanu, and W. Visser. JPF-SE: A symbolic execution extension to Java PathFinder. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2007)*, pages 134–138, 2007. Springer LNCS 4424.
5. T.W. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Sci. Comput. Program.*, 58(1-2):206–263, 2005.
6. S. Hunt. PERs generalize projections for strictness analysis. In *Functional Programming: Proc. 1990 Glasgow Workshop*, pages 114–125. Springer-Verlag, 1990.
7. D.A. Schmidt. Data flow analysis is model checking of abstract interpretations. In *the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 1998)*, pages 38–48, 1998.
8. R. Vallée-Rai, P. Co, E. Gagnon, L.J. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON 1999)*, 1999.
9. R. Hasti and S. Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. In *ACM SIGPLAN conference on Programming language design and implementation (PLDI '98)*, pages 97–105, 1998.

10. X. Li and M. Ogawa. Interprocedural program analysis for java based on weighted pushdown model checking. In *the 5th International Workshop on Automated Verification of Infinite-State Systems (AVIS 2006)*. ETAPS, April 2006.
11. T.W. Reps. Program analysis via graph reachability. In *International Logic Programming Symposium (ILPS '97)*, pages 5–19. MIT Press, 1997.
12. T.W. Reps, A. Lal, and N. Kidd. Program analysis using weighted pushdown systems. In *the 27th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2007)*, pages 23–51, 2007. Springer LNCS 4855.
13. O. Shivers. Control flow analysis in scheme. In *ACM SIGPLAN conference on Programming Language design and Implementation (PLDI '88)*, pages 164–174, 1988.
14. J. Whaley and M.S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2004)*, pages 131–144, 2004.
15. O. Tkachuk, M. B. Dwyer, and C. Păsăreanu. Automated environment generation for software model checking. In *the 18th IEEE International Conference on Automated Software Engineering (ASE 2003)*, pages 116–129, 2003.
16. S. Khurshid and Y.L. Suen. Generalizing symbolic execution to library classes. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering (PASTE'05)*, pages 103–110, 2005.
17. M. 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 *the 23rd International Conference on Software Engineering (ICSE 2001)*, pages 177–187, 2001.
18. C. Cadar, V. Ganesh, P.M. Pawlowski, D.L. Dill, and D.R. Engler. EXE: Automatically generating inputs of death. In *ACM Conference on Computer and Communications Security 2006 (CCS 2006)*, pages 322–335, 2006.
19. W. Schulte W. Grieskamp, N. Tillmann. XRT-exploring runtime for .NET architecture and applications. *Electronic Notes in Theoretical Computer Science*, 144(3):3–26, 2006.
20. Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 263–272. ACM, 2005.
21. S. Anand, A. Orso, and M. J. Harrold. Type-dependence analysis and program transformation for symbolic execution. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 2007)*, pages 117–133, 2007. Springer LNCS 4424.
22. V.B. Livshits and M.S. Lam. Finding security vulnerabilities in Java applications with static analysis. In *the 14th conference on USENIX Security Symposium (SSYM'05)*, pages 18–18. USENIX Association, 2005.
23. A. Christensen, A. Møller, and M. Schwartzbach. Precise analysis of string expressions. In *the 10th International Symposium on Static Analysis (SAS 2003)*, pages 1–18, 2003. Springer LNCS 2694.

A Application-oriented Modelling for Efficiency

It is often intractable and unnecessary to explore the whole Java libraries. We hence propose application-oriented explicit modelling on some popular Java libraries, such as containers and strings for better efficiency.

Container, such as HashMap, vectors and trees, is widely used in Java web applications, to store and fetch event attributes. A precise analysis on containers is nontrivial, since the capacity (or the index space) of containers can be unbounded. Our treatment on containers is inspired by the treatment on instance fields, based on the insight that *keys of containers can be regarded as fields of class instances*. Compared with modelling on instance fields, the modelling on containers differs in that keys of containers can be either string constants or more often reference variables. Therefore, both containers and keys need to be cast back to heap objects by calling points-to analysis. Table 5 gives rules of modelling Map containers. As shown in Table 5, the `key` of a `map` is bound with its corresponding `value` by the *put* and *get* methods. The pair of containers and keys are treated as variables from `GlobVar` when building the WDG. In particular, a Map container is marked as symbolic if there is any symbolic value put into or any symbolic key taken from it.

Strings are also heavily used in Java web applications. For instance, the keys and values of containers are usually of type String. The space of string values is generally infinite, and to conduct a precise *string analysis* [23] will put too much overhead on the static analysis phase. However, we are only interested in the relevancy relationship among string variables. In our analysis, string constants that syntactically appear in the program (and are thus essentially bounded) are considered as distinguished string instances. Java library methods related to strings, i.e., `java.lang.String`, `java.lang.StringBuffer` are explicitly modelled. They fall into the following categories: (1) The receiver object is relevant to all arguments for a constructor; (2) The return value, if any, is relevant to all method arguments, as well as the receiver object, if any. Table 5 also shows a few of examples that require specific treatments.

Table 5. Application-oriented Modelling

Map Container:
$\mathcal{A}[\![\texttt{map}.\text{put}(\texttt{key},\ \texttt{value})]\!]=\{(\texttt{value}, f) \rightsquigarrow (o_m.o_k, f) \mid o_m \in \texttt{pta}(\texttt{map}), o_k \in \texttt{pta}(\texttt{key})\}$
$\mathcal{A}[\![\texttt{value} = \texttt{map}.\text{get}(\texttt{key})]\!]=\{(o_m.o_k, f) \rightsquigarrow (\texttt{value}, f) \mid o_m \in \texttt{pta}(\texttt{map}), o_k \in \texttt{pta}(\texttt{key})\}$
java.lang.String, *java.lang.StringBuffer*:
$\mathcal{A}[\![\texttt{str}.\text{getBytes}(m_0, m_1, m_2, m_3)]\!] = \mathcal{A}[\![\texttt{str}.\text{getChars}(m_0, m_1, m_2, m_3)]\!]$
$= \{(m_i, f) \rightsquigarrow (m_2, f) \mid 0 \leq i \leq 3 \text{ and } i \neq 2\} \cup \{(\texttt{str}, f) \rightsquigarrow (m_2, f)\}$
$\mathcal{A}[\![\texttt{strbuffer}.\text{getChars}(m_0, m_1, m_2, m_3)]\!]$
$= \{(m_i, f) \rightsquigarrow (m_2, f) \mid 0 \leq i \leq 3 \text{ and } i \neq 2\} \cup \{(\texttt{strbuffer}, f) \rightsquigarrow (m_2, f)\}$