Domain Partitioning for Open Reactive Systems

Scott D. Stoller*
Computer Science Dept., State University of New York at Stony Brook
Stony Brook, NY 11794-4400 USA

ABSTRACT

Testing or model-checking an open reactive system often requires generating a model of the environment. We describe a static analysis for Java that computes a partition of a system's inputs: inputs in the same equivalence class lead to identical behavior. The partition provides a basis for generation of code for a most general environment of the system, i.e., one that exercises all possible behaviors of the system. The partition also helps the generated environment avoid exercising the same behavior multiple times. Many distributed systems with security requirements can be regarded as open reactive systems whose environment is an adversary-controlled network. We illustrate our approach by applying it to a fault-tolerant and intrusion-tolerant distributed voting system and model-checking the system together with the generated environment.

1. INTRODUCTION

Testing or model-checking (*i.e.*, verification by systematic state-space exploration) an open reactive system often requires generating a model of the environment, for various reasons, *e.g.*, if the actual environment is not available (perhaps does not even exist yet), or cannot easily be controlled to induce the behaviors of interest, or is restricted in some ways compared to the most general environment in which the system is designed to operate. We assume the system interacts with its environment through an interface that offers a set of methods that the environment may invoke using local or remote method invocation, and that each method has a specified type signature. It is straightforward to generate an environment that, whenever the system is ready for an input, non-deterministically generates an arbitrary type-correct input. However, testing with such an environment is typically inefficient: many equivalent inputs—*i.e.*, inputs that lead to equivalent behaviors of the system, in the sense defined below—may be explored. Explicit-state model checking with such an environment is typically intractable: the number of type-correct inputs is too large. More sophisticated environment models are needed, which explore a proper subset of the inputs and preferably still provide some guarantee about coverage. Such models can be constructed based on how the system uses its inputs.

Manual construction of such models can be time consuming (especially if the tester is not the programmer and hence first needs to understand the code) and error prone. For example, it is very easy to overlook inputs that cause the system to exhibit exceptional behavior; in Java, this is especially true for exceptions that extend `RunTimeException` and hence may be thrown by statements other than `throw` and are not necessarily listed in `throws` clauses of methods. The primary goal of this work is to reduce the effort needed to construct such models and reduce the number of errors.

Our approach is based on domain partitioning. We regard a system as a function f from inputs to outputs. If the system was non-deterministic, we assume the non-determinism has been "factored out" and hence is controlled by non-deterministically selected values supplied by the environment (this reflects the desire to control all non-determinism during testing or model-checking). We describe a static program analysis that partitions the inputs into equivalence classes, symbolically represented by predicates, such that inputs x and y in the same equivalence class lead to the same output, *i.e.*, $f(x) = f(y)$. Let π denote the resulting partition. Exploring at least one input from each equivalence class in π provides 100% output coverage, *i.e.*, every possible output is exercised. Output coverage is incomparable in strength to branch or path coverage: it is sometimes stronger (because it distinguishes different return values) and sometimes weaker (*e.g.*, if there are multiple branches that throw the same exception, for different reasons, complete output coverage can be achieved by exploring one of them).

This paper focuses on reactive systems. We regard a reactive system as a function whose input $\langle s, args\rangle$ contains the current system state s and an input *args* (method arguments) from the environment, and whose output contains the updated system state and the return value of the invocation. Output coverage, and hence our partitioning, is less relevant to non-reactive systems, because correctness requirements for non-reactive systems are typically expressed as input-output relations, so covering the possible outputs without regard to the associated inputs is not of much sig-

*The author gratefully acknowledges the support of NSF under Grant CCR-9876058 and the support of ONR under Grants N00014-01-1-0109 and N00014-02-1-0363. Email: stoller@cs.sunysb.edu Web: http://www.cs.sunysb.edu/˜stoller/

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 2001 ACM X-XXXXX-XX-X/XX/XX ...$5.00.

nificance. In contrast, correctness requirements for reactive systems are often expressed as invariants on system states, or as temporal predicates on sequences of states. In this case, exploring multiple inputs that lead to the same output is unnecessary. We assume that the specification directly constrains only (1) states between processing of inputs and (2) return values; we call such a state, together with the return value of the preceding invocation, a *quiescent state*.

For testing with manually produced or systematically generated test suites, the analysis result can be used to avoid generating redundant scenarios or to eliminate redundant scenarios from existing test suites. Using the above model in which an input to a reactive system is a pair $\langle \mathit{state}, \mathit{args} \rangle$, let $\mathit{partnForState}(\pi, s)$ denote the set of equivalence classes E in π such that s is the first component of some pair in E. For random testing or model checking, one can synthesize an environment that, whenever the system is ready for an input in state s, randomly or non-deterministically selects an equivalence class from $\mathit{partnForState}(\pi, s)$ and uses a single representative (with first component s) thereof. Model checking with this environment explores all feasible sequences of quiescent states. The main issues in the synthesis are how to identify empty equivalence classes (which correspond to infeasible program paths) and how to choose a representative of each non-empty equivalence class. These important issues can be handled with a combination of decision procedures, automated theorem provers, and manual effort.

For flexibility, the analysis is parameterized by manually constructed abstractions for selected classes. Typically, such "custom" abstractions are used for classes in the Java API that are relevant to the application domain (e.g., classes in `java.security` for secure systems). It appears that the same abstractions will be effective for many programs in a given application domain; further experience is needed to confirm this. The analysis aims to discover how the program uses its inputs, so the appropriate abstractions of values and operations usually embody a form of symbolic evaluation. Automatically generated default abstractions are used for all other classes. Given the custom abstractions, our analysis is fully automatic. We are starting to implement it. Semi-automatic synthesis of an environment from the partition, as sketched briefly in Section 9, is future work.

For some systems, the partition may contain undesirably many equivalence classes, either because exploring all feasible sequences of quiescent states is inherently expensive for that system, or because the analysis was conservative and produced more equivalence classes than the coverage criterion requires. The latter cannot always be avoided, because determining whether two inputs lead to the same quiescent state is undecidable. The effectiveness of the analysis depends crucially on the use of appropriate abstractions.

If the number of equivalence classes is undesirably large, an additional step is needed to decide which equivalence classes to use in testing or model checking, based on other coverage criteria, heuristics, or user guidance (*e.g.*, the user supplies a temporal-logic formula, and sequences of inputs not satisfying the formula are not considered). Separation of rigorous domain partitioning from heuristics and *ad hoc* constraints is desirable to highlight where and how completeness is sacrificed. In the distributed voting application in Section 9, the partition analysis is effective in the sense that it produces a partition with a finite and tractable number of equivalence classes, so this additional step is not needed.

The analysis can also be useful for program understanding. In effect, it extracts from code a detailed declarative description of the structures of well-formed input messages. Including this information in a system's interface specification is often useful. If such a description was prepared manually (perhaps before implementation), our analysis can be used to help check its consistency with the implementation.

For secure systems, inputs to the system from the environment (*i.e.*, from the adversary-controlled network) include arguments to remote methods and return values from remote methods. Our analysis can also be used for domain partitioning of the latter by augmenting the calling method with fresh parameters representing return values of remote method invocations, replacing uses of those return values with uses of those parameters (turning the remote method invocations into dead code), computing an input partition for (the arguments of) the augmented method, and projecting the resulting partition onto each of the new parameters, by existential quantification over the other parameters.

Colby *et al.*'s work on automatically closing open reactive programs [1] effectively provides an environment. Application of a coarse data abstraction to the system's inputs from the environment is an integral part of their approach. Thus, their approach can be used to check only properties that are independent of the input values. Our approach does not incorporate such an abstraction (although such abstractions can be applied separately before our analysis) and hence can be used to check a larger class of properties, with correspondingly larger computational cost.

It is tempting to try to regard our analysis as producing a program-specific abstract interpretation, in which abstract values correspond to equivalence classes. However, two values that are equivalent as inputs are generally not equivalent as outputs. It is unclear how to reflect this asymmetry in standard formulations of abstract interpretation.

A direction for future work is to consider equivalence classes of sequences of inputs; this paper considers one input at a time.

2. OVERVIEW

Information from a method's inputs may escape from an invocation through storage flow or data (value) flow. An example of the former is a method that returns an object contained in one of its arguments, possibly without accessing the data in that object. An example of the latter is a method that copies values from an object in its arguments into a new object and then returns the new object. Correspondingly, our analysis has three steps:

1. Points-to escape (PTE) analysis [9] is used to analyze the flow of storage locations. The result of this step is a points-to escape graph (PTE graph) at each program point. PTE analysis determines which references (hence which storage locations) possibly escape from invocations of the analyzed method.

2. Data-flow analysis is used to analyze the flow of values of variables and objects. The result of this step is an (abstract) environment at each program point.

3. An input partition is constructed based on how parameters and global storage are used in branch conditions, return statements, and updates to global stor-

age. *Global storage* is static fields and objects reachable from them.

Our data-flow analysis, like the PTE analysis in [9], is expressed as a set of constraints that are solved by a worklist algorithm.

Step 3 is the most expensive. The cost is proportional to the sum of the numbers of simple paths in the control-flow graphs of the methods in the system's interface. Each of these methods is analyzed separately, so the total cost is linear in the overall size of the system, if the size of each method is fixed. Although the analysis can be expensive, its cost is typically dominated by the cost of testing or model checking. Step 3 can be interleaved with testing or model checking, by generating some equivalence classes, using representatives of them, generating more equivalence classes, and so on.

Input partition analysis must deal with exceptions explicitly to achieve sufficient accuracy. Our abstraction for a method distinguishes the conditions under which the method throws each type of exception (or terminates normally), and it characterizes separately the resulting environment in each case. The underlying heuristic is that different types of thrown exceptions typically correspond to qualitatively different behaviors. The PTE analysis in [9] assumes exceptions are replaced with simpler constructs during pre-processing. We modify it to treat exceptions explicitly.

The main contributions of this paper are: (1) the overall framework of using multiple program analyses to help efficiently achieve output coverage in testing of open reactive systems; (2) the abstractions of classes and methods used in Step 2; (3) the construction of the partition in Step 3; and (4) the explicit handling of exceptions in Step 1.

The construction in Step 3 is similar to the construction of the implementation partition in [6], although the target notion of coverage is different. The focus of [6] is on programs that manipulate numbers and arrays, while this paper focuses on object-oriented programs in which object creation, field access, and method invocation are essential operations. This forces us to incorporate PTE analysis and to use a more flexible (parameterized) data-flow analysis.

Due to space constraints, we omit the treatment of static fields and global storage. They cause no difficulties. For the example in Section 9, they are analyzed as in [8].

3. PROGRAM REPRESENTATION

The program representation is based on [9]. It is similar to Java bytecode, except that it uses temporary variables instead of an operand stack, and it makes safety checks (*e.g.*, check for null pointer before dereference) explicit as separate operations. For example, a getfield bytecode corresponds to a statement like: if l == null then throw NullPointerException else get *l.field*. Consequently, all exceptions originate at throw statements or invocation statements.

Each method in the interface between the system and its environment is analyzed separately. Fix the method m being analyzed. *Stmt* is the set of statements in m. L is the set of local variables of m. *Param* is the set of parameters of m. *Class* is the set of classes used in m. *Field* is the set of field names of classes in *Class*. Let *st*, l, p, *cl*, f, and v range over *Stmt*, *Local*, *Param*, *Class*, *Field*, and $Local \cup Param \cup Class$, respectively. *PrimTy* is the set of primitive (i.e., non-reference) types. τ ranges over all types, *i.e.*, $\tau \in Class \cup PrimTy$. c ranges over constants (literals). For each class $cl \in Class$, the static fields of *cl* are modeled as instance fields of a unique object associated with the class. The class name *cl* is treated as a read-only variable that points to that unique object. Each variable v has a type, denoted $type(v)$. A variable v *has reference type* if $type(v) \in Class$

The body of the method is represented by a control-flow graph (CFG) whose nodes are labeled with statements. The CFG for m starts with an enter node $\texttt{enter}_m$ and ends with an exit node $\texttt{exit}_m$. We assume the program has been pre-processed so that all statements relevant to the analysis are in one of the following forms:

- constant: $l = c$
- copy: $l = v$
- store: $l_2.f = l_1$
- load: $l_1 = l_2.f$
- array store: $l_2[l_3] = l_1$
- array load: $l_1 = l_2[l_3]$
- object creation: $l = \texttt{new}\ cl$
- return: $\texttt{return}\ l_1$
- method invocation: $l = l_0.m(l_1, \dots, l_k)$,
  m has the form *class.method*. l_0 is omitted if m is static.
- throw: $\texttt{throw}\ l_1$
- branch: $\texttt{if}\ l_1\ rel\ l_2$ or $\texttt{if}\ l_1\ rel\ c$
  rel is an equality or inequality test, or `instanceof`.
- primitive: $l = l_1\ op\ l_2$,
  op is an arithmetic or boolean operation.

An invocation statement invokes the method of the indicated class. Virtual method lookup is eliminated during pre-processing, by introducing branches with `instanceof` conditions. For $st \in Stmt$, pred(st) and succ(st) are the sets of statements that may execute immediately before and after *st*, respectively. The program points immediately before and after *st* are denoted $\bullet st$ and $st\bullet$, respectively. A branch statement *st* has a true successor and a false successor, corresponding to control flow when the condition is true and false, respectively.

Exceptions. Let $Stmt_{thrower}$ be the set of statements that can throw exceptions, namely, throw statements and invocation statements. For each $st \in Stmt_{thrower}$, there are edges in the CFG from *st* to each exception handler that can catch an exception thrown by *st*, and there is an edge from *st* to $\texttt{exit}_m$ if *st* might throw an exception not caught within m. For each such edge $\langle st, st' \rangle$, $excns(st, st')$ is the set of classes of exceptions that may be thrown at *st* and cause control to flow to st'. An invocation statement *st* has an additional outedge to the statement st' executed next if the invocation returns normally; for this edge, we take $excns(st, st') = \{\text{normal}\}$. For example, suppose *st* invokes a method that throws E_1, E_2, and E_3. Suppose st' is the target of a catch block that encloses *st* and catches exceptions of type E_4, and there are no smaller catch blocks enclosing *st*. Suppose E_1 and E_2 extend E_4. Then $excns(st, st') = \{E_1, E_2\}$.

Each method has a variable l_{exc} used for temporary storage of exceptions. When an exception is thrown (by a throw or invocation statement), l_{exc} is implicitly updated to point to the exception object. Exception handlers generally start with $l = l_{exc}$ for some local variable l and thereafter access the caught exception through l, because method calls in the handler that throw exceptions clobber l_{exc}.

Example. The program in Figure 1 serves as a running example. Method `getLength` takes an instance `sba` of `SBA` (mnemonic for "SignedByteArray") as argument and checks whether `sba.sig` is a valid signature of `sba.data`. If so, it returns an `Integer` containing `sba.data.length`; otherwise, it returns `null`. `getLength` uses the following methods of `java.security.Signature`. Static method `getInstance(`*alg*`)` returns a signature object that implements the signature algorithm *alg*. A signature object o can be used to verify that a byte array *sig* is a valid signature of a byte array *data* with respect to public key *key*, as follows: o`.initVerify(`*key*`)` supplies the key; o`.update(`*data*`)` supplies the data; o`.verify(`*sig*`)` returns a boolean, indicating whether the signature is valid. A similar sequence of steps is used to generate signatures, using the following methods of `Signature`: `initSign(`*key*`)`, `update(`*data*`)`, and `sign()`. `sign` returns a byte array containing the desired signature.

The CFG for `getLength` appears in Figure 1. $excns(st, st')$ is shown as edge labels. Exception names are abbreviated. The constructors for `Integer` and `NullPointerException` have been inlined (because they are not in M_{cust}, introduced in Section 6). For brevity, in all of the examples in this paper, we ignore the `detail` field in exception classes, because the detail message is not used in a significant way in these examples. As a result, the `NullPointerException` constructor does not contribute any nodes to the CFG.

```
import java.security.*;
class SBA { byte[] data; byte[] sig; }
class C {
final PublicKey pubKey=...; // initializer elided
Integer getLength(SBA sba) {
  PublicKey k = this.pubKey;
  try {
    Signature v =
      Signature.getInstance("SHA1withDSA");
    v.initVerify(k);
    byte[] a = sba.data;
    v.update(a);
    byte[] s = sba.sig;
    boolean b = v.verify(s);
    if (b) {
      Integer i = new Integer(a.length);
      return i; }
  } catch (Exception e) {}
  return null;
}}
```

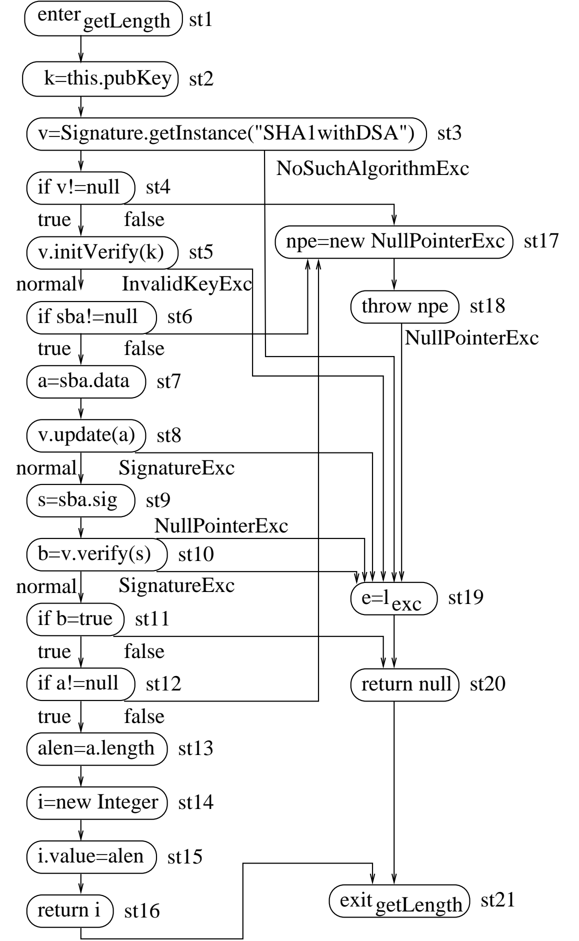

Figure 1: Code and CFG for `getLength`.

4. CORRECTNESS REQUIREMENT

This section describes the correctness requirements for the analysis. For $\tau \in \mathit{Class} \cup \mathit{PrimTy}$, let the concrete domain Con_τ be the set of all possible values of type τ. For $\tau \in \mathit{Class}$, Con_τ contains records whose fields contain primitive values and references (*i.e.*, object identifiers). Let Ref be the set of references. For any set S, the set $\mathit{Bind}(S)$ of *bindings* for S contains partial functions b from $S \cup \mathit{Ref}$ to Con such that $\mathrm{domain}(b) = S \cup \mathit{refs}(b)$, where $\mathit{refs}(b)$ is the set of refrences that appear in b. The universe of concrete values is $\mathit{Con} = \bigcup_{\tau \in \mathit{Class} \cup \mathit{PrimTy}} \mathit{Con}_\tau$.

Structural equality is an equivalence relation on Con, defined by: primitive values are structurally equal iff they are equal; objects o_1 and o_2 are structurally equal iff the graphs of objects reachable from them are isomorphic, *i.e.*, they have the same shape and contain the same primitive values in the same places but may contain different references. Structurally equal values have the same serialized form. For $R \subset \mathit{Ref}$, *structural equality exact in* R is defined like structural equality with the additional condition that references in S match exactly in the two graphs. We extend structural equality to tuples and bindings in the usual homomorphic way.

Fix a method m. Let Var_{static} be the set of static variables used by m. Let RetVal denote possible return values of m; these are elements of Con tagged to indicate whether the method terminated normally or abruptly. For bindings $\mathit{args} \in \mathit{Bind}(\mathit{Param})$ and $\mathit{glbl}, \mathit{glbl}' \in \mathit{Bind}(\mathit{Var}_{static})$, and a return value $r \in \mathit{RetVal}$, define the transition relation $\xrightarrow{m}$ by: $\mathit{glbl}, \mathit{args} \xrightarrow{m} \mathit{glbl}', r$ iff execution of $m(\mathit{args})$ starting with static variable binding glbl can terminate with result r and with static variable binding glbl'.

Functions f_1 and f_2 are *compatible* if $(\forall x \in \mathrm{domain}(f_1) \cap \mathrm{domain}(f_2) : f_1(x) = f_2(x))$. For compatible functions f_1 and f_2, let $f_1 \oplus f_2$ denote their "union". An *input partition* π for m is a partition of $\mathit{Bind}(\mathit{Param} \cup \mathit{Var}_{static})$ (*i.e.*, each equivalence class in π is a set of bindings for parameters and static variables) such that: for all $\phi \in \pi$, for all $\mathit{args}_1, \mathit{args}_2 \in \mathit{Bind}(\mathit{Param})$, for all $\mathit{glbl}, \mathit{glbl}'_1, \mathit{glbl}'_2 \in \mathit{Bind}(\mathit{Var}_{static})$ such that glbl is compatible with args_1 and args_2, for all $r_1, r_2 \in \mathit{RetVal}$, if $\mathit{args}_1 \oplus \mathit{glbl} \in \phi$ and $\mathit{args}_2 \oplus \mathit{glbl} \in \phi$ and $\mathit{glbl}, \mathit{args}_1 \xrightarrow{m} \mathit{glbl}'_1, r_1$ and $\mathit{glbl}, \mathit{args}_2 \xrightarrow{m} \mathit{glbl}'_2, r_2$,

then the quiescent states $\langle glbl'_1, r_1\rangle$ and $\langle glbl'_2, r_2\rangle$ are structurally equal exact in $refs(args_1) \cup refs(args_2) \cup refs(glbl)$.

In comparing the final states, we use structural equality for objects created during execution of the method. This is appropriate because the semantics of object creation does not specify which fresh object reference is returned by `new`, so the output of well-behaved programs does not depend on specific heap addresses. In other words, we assume that programs do not use, *e.g.*, `Object.hashCode` in pathological ways. Similarly, if the system's interface is based on RMI (not local invocation), then it is appropriate to adopt a weaker correctness requirement that uses structural equality for objects in *args* other than `this`, because arguments of a remote method on the server side contain freshly created objects, so exact object references in the arguments are not significant.

The correctness requirement and the analysis may be parameterized by a set of state components to ignore when comparing quiescent states. This is an example of improving the analysis by considering how (if at all) information that escapes an invocation is subsequently used. For example, in the examples in this paper, we ignore detail messages in exceptions thrown by m, because those messages are not used by the rest of the program in any significant way. Another common example is to ignore output to a log file that is never read by the program.

The correctness requirement may also be parameterized by an equivalence relation (coarser than structural equality) on specified classes to use when comparing quiescent states. The same equivalence relation is then used in the analysis to define $struct(cl)$ (introduced in Section 8) for those classes. For example, an instance of the class `SignedObject` described in Section 9 encapsulates a serializable object, called the *payload*, and a signature of that object. Our equivalence relation for this class equates instances containing valid signatures that differ only in the initialization vector used to generate the signature.

5. STEP 1: PTE ANALYSIS

The points-to escape (PTE) analysis in [9] produces a PTE graph $\alpha(x) = \langle N, O, I, e, r\rangle$ at each program point x of a method m. N is the set of nodes. Each node represents a set of objects. O and I are sets of edges that characterize the objects to which variables and fields with reference type might point. The *return set* r indicates which objects might escape from (an invocation of) m through its return value. The *escape function* e describes the other ways in which objects might escape from m. The rest of this section describes PTE graphs in more detail.

Five kinds of nodes are used in [9], corresponding to the ways in which a program can obtain references to objects. There is an *allocation node* n_{st} for each object creation statement st in m; n_{st} represents objects allocated at st. There is a *parameter node* n_p for each parameter p of m with reference type; n_p represents the argument bound to p. For each load statement st of the form $l_1 = l_2.f$ in m such that l_1 has reference type, there is a *load node* n_{st} which represents objects o such that $l_2.f$ might point to o and the reference to o was stored in $l_2.f$ by code outside m. There is a *return node* n_{st} for each invocation statement st in m such that the method invoked by st returns a reference; n_{st} represents objects returned by invocations at st. There is a *class node* n_{cl} for each class cl; it represents an imaginary object whose fields are the static fields of cl. An object may be represented by multiple nodes, *e.g.*, if a reference is loaded by multiple load statements.

Example. A PTE graph for `getLength` contains (among other nodes) an allocation node $n_{\mathrm{st}14}$ and a load node $n_{\mathrm{st}2}$.

O and I are sets of *outside* edges and *inside* edges, respectively. An edge may connect a variable v to a node, or a node to a node. An edge from a variable v to a node n represents the possibility that v points to an object represented by n. An edge from a node n_1 to a node n_2 is labeled with a field name f and represents the possibility that the f field of some object represented by n_1 points to some object represented by n_2. Outside edges represent references created before the analyzed method was invoked or by concurrently executing threads; inside edges represent references created by the analyzed method.

The return set $r \subseteq N$ satisfies: if an object represented by a node n possibly appears in the return value of m (hence escapes to the caller), then $n \in r$. The escape function e describes the ways objects represented by a node might escape from the method invocation other than through the return value. $e(n)$ contains a parameter p if some objects represented by n might be reachable from p. $e(n)$ contains a class name cl if some objects represented by n might be reachable from a static field of cl. $e(n)$ contains an allocation node n_{st} if some objects represented by n might be reachable from (instance fields of) a runnable object allocated at st. $e(n)$ contains an invocation statement st if some objects represented by n might be passed as parameters to or appear in the return value of an invocation at st.

Exceptions. To analyze exceptions explicitly, we introduce a sixth kind of node. For each invocation statement st that can throw an exception, the PTE graph contains an *exc-return node* n^{exc}_{st}, which represents exceptions thrown there. The transfer function for `throw` l updates the PTE graph the same way as the copy $l_{exc} = l$. The transfer function for a skipped invocation statement st of the form $l = l_0.m(l_1, \dots, l_k)$ is similar to the transfer function for skipped invocation sites in [9, Section 6.1]. Let st_n be the normal successor of st (*i.e.*, $excns(st, st_n) = \{\text{normal}\}$). Let st_e be an exceptional successor of st (*i.e.*, any successor other than st_n). The PTE graph at $\bullet st_n$ is computed exactly as in [9, Section 6.1]. The PTE graph at $\bullet st_e$ is computed similarly, except (1) updating outedges of l_{exc} to point to n^e_{st}, instead of updating outedges of l to point to n_{st}, and (2) requiring $st \in e(n^e_{st})$ instead of $st \in e(n_{st})$.

Example. The PTE graph for `C.getLength` at `•return i` appears in Figure 2. Parameter nodes are identified by printing the name of the parameter inside. Other nodes in the PTE graph are identified by the label of the corresponding node in the CFG in Figure 1. For example, the load node labeled "st2" (also known as $n_{\mathrm{st}2}$) corresponds to the load statement in node 2 of the CFG

6. PARAMETERS OF THE ANALYSIS AND SIMPLIFYING ASSUMPTIONS

The analysis is parameterized by the following sets.

M_{fun} is a set of functional methods. A method m_1 is *functional* if (i) an invocation of m_1 does not update stor-

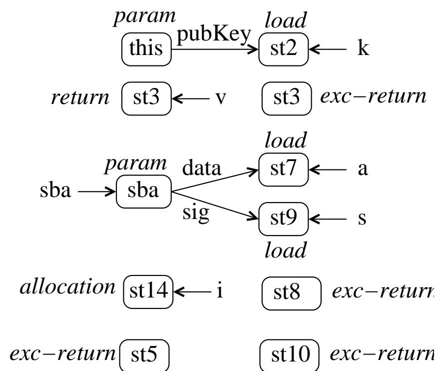

Figure 2: PTE graph for `C.getLength` at •`return i`. Italic labels indicate the kind of each node. The return set is $r = \{n_{\text{st14}}\}$. The escape function e is: $e(n_{\text{this}}) = \text{this}$, $e(n_{\text{sba}}) = \{\text{sba}\}$, $e(n_{\text{st2}}) = \{\text{this}, \text{st5}\}$, $e(n_{\text{st3}}) = \{\text{st3}\}$, $e(n_{\text{st5}}) = \{\text{sba}\}$, $e(n_{\text{st7}}) = \{\text{sba}\}$, and $e(n^e_{st}) = \emptyset$ for all invocation statements st.

age locations that escape from m_1, and (*ii*) every storage location it reads from either is captured in m_1, appears in its arguments, or is read-only. Note that this definition depends on which storage locations are read-only and hence depends on $Class_{rdOnly}$ and $Param_{rdOnly}$, defined below.

M_{cust} is a set of methods for which the user supplies custom abstractions of the kind described in Section 7.2. All other methods are inlined before Step 1; we assume those methods are not recursive. During PTE analysis, invocations of methods in M_{cust} are treated as skipped invocation sites. M_{cust} is the only analysis parameter that affects Step 1. We require $M_{cust} \supseteq M_{fun}$. For convenience, we assume that all exceptions thrown by methods in M_{cust} are caught by the caller; this can always be satisfied by modifying the program to catch and re-throw the exceptions. This assumption ensures that all exceptions thrown by the method being analyzed are thrown explicitly. We also assume that information does not escape through methods in M_{cust}. This assumption could easily be eliminated by supplying, for each method in M_{cust}, a characterization of which parameters' values might escape, and generalizing the definitions of $Stmt_{esc}$ and $esc(st)$ to reflect that values may also escape via method invocations.

$Class_{cust}$ and $PrimTy_{cust}$ are sets of classes and primitive types, respectively, for which the user supplies custom abstractions of the kind described in Section 7.1.

$Class_{rdOnly}$ is a set of read-only classes. A class cl is *read-only* if the system never updates objects reachable from static fields of cl, except during initialization, which is assumed to complete before processing of inputs from the environment.

$Param_{rdOnly}$ is a set of read-only parameters. A parameter p of method m is *read-only* if m never updates objects reachable from p. For systems with RMI-based interfaces, typically $Param_{rdOnly} = Param \setminus \{\texttt{this}\}$. The analysis tracks values retrieved from read-only parameters and (static fields of) read-only classes but (for simplicity) not from other parameters and static fields. To eliminate this limitation, we would need to consider possible aliasing relationships among those objects (because, in the presence of aliasing, an update to, say, $p_1.f_1$ could also update $p_2.f_1$).

As mentioned in Section 1, we assume non-determinism in the system has been factored out.

7. STEP 2: DATA-FLOW ANALYSIS

Section 7.1 introduces the abstract domains. We define an abstract domain D_τ for each type τ and then combine these domains into a single overall abstract domain D. An element of D_{cl} represents a set of states that an instance of class cl can be in. By default, D_{cl} is a tuple of abstract values, each indicating the possible values of one field of the instance. For selected classes, manually constructed abstractions may be used instead of this default abstraction.

Section 7.2 describes the data-flow analysis algorithm.

7.1 Domains and Environments

The data-flow analysis aims to discover how the program uses its inputs, so each domain D_τ should be able to represent values retrieved from the inputs. Let $Rtrvd_\tau$ be a set of expressions that represent values of type τ retrieved from read-only parameters and read-only static variables. Specifically, these expressions are built recursively from read-only parameters and read-only static variables and two kinds of retrieval operations, namely, field accesses and invocations of functional methods. For example, if $p_1, p_2 \in Param$ and $f_1, f_2 \in Field$ and $m_1 \in M_{fun}$, then $Rtrvd_\tau$ contains the expression $p_1.f_1.m_1(p_2.f_2)$, assuming it type-checks and has type τ. Note that $Rtrvd_{cl}$ includes expressions whose type is a subclass of cl. To achieve the above desideratum, we require $D_\tau \supseteq Rtrvd_\tau$.

For each class cl, the analysis uses: (1) an abstract domain D_{cl}, which comes with a join (least upper bound) operation $\sqcup_{cl}$ (equivalently, it could come with a partial ordering, and we could infer the join operation); (2) a *load function* such that, for s in D_{cl}, $\text{load}_{cl}(s, f)$ represents the possible values of field f in an instance of cl in a state represented by s; (3) a *store function* such that for s in D_{cl} and a field f of cl and a value d, $\text{store}_{cl}(s, f, d)$ is in D_{cl} and represents the possible states of an instance of cl obtained by starting in a state represented by s and then storing d in field f.

Default Abstractions. The following default abstraction is used for classes not in $Class_{cust}$. D_{cl} is the union of $Rtrvd_{cl}$, a few special values (discussed next), and the cross product of the abstract domains for the fields of cl. A special value, null, is included to represent that a reference might be null. Let $\tau_1, \ldots, \tau_n$ denote the types of the instance fields of cl, and let $cl(x_1, \ldots, x_n)$ denote a record (tuple) with the indicated fields (components). The default abstract domain for cl is

$$D_{cl} = \{cl(d_1, \ldots, d_n) \mid d_1 \in D_{\tau_1}, \ldots, d_n \in D_{\tau_n}\} \cup Rtrvd_{cl} \cup \{\text{null}\}.$$

When applied to a tuple in D_{cl}, the default load and store functions for cl select or update the specified field of the record. When applied to null, the load and store functions return $\top$, a special abstract value that represents "all information" (or "arbitrary information"). When applied to an element of $Rtrvd_{cl}$, the load function performs the symbolic retrieval in the obvious way (*e.g.*, $\text{load}_{cl}(p_1.f_1, f_2) = p_1.f_1.f_2$), and the store function simply returns $\top$. More accurate store functions are possible but have not been necessary.

The default abstract domain for an array class has two fields: `length` and `elements`. It does not distinguish values of individual elements: the value of the `elements` field represents possible values of all elements.

For a primitive type T, the analysis uses an abstract domain D_T. The default abstract domain for a primitive type T is $D_T = Con_T \cup Rtrvd_T$, and the join operation returns $\top$ whenever its operands are not equal and are both not $\bot$. One could easily parameterize the analysis with respect to abstractions for operations on primitive types. For simplicity, very coarse abstractions of these operations are hard-wired into the transfer functions in Section 7.2. This suffices for our current applications.

Custom Abstractions. For each class cl in $Class_{cust}$, the abstract domain and the load and store functions are manually written. Custom domains are also required to satisfy $D_\tau \supseteq Rtrvd_\tau$.

The custom abstractions needed for our current applications are designed in a straightforward way, by introducing symbolic representations of (*i*) return values of methods of classes in $Class_{cust}$ and (*ii*) return values of methods of other classes whose results are typically used as arguments to methods of classes in $Class_{cust}$ and whose values affect the qualitative behavior (*e.g.*, whether an exception is thrown, and if so, which type of exception) of those methods. Our naming convention is that the symbolic representation of the return value of a method `foo` is $foo(args)$, where *args* denotes abstractions of the arguments of `foo`, including `this`.

Overall Domain. The type-specific domains D_τ are enhanced to include bottom and top elements and to allow sets of possible values. The special abstract values $\bot$ and $\top$ represent "no information" and "all information", respectively. The enhanced type-specific domains $PwrD_\tau$ are combined into an overall domain D.

$$PwrD_\tau = \{\bot, \top\} \cup \text{PowerSet}(D_\tau) \tag{1}$$

$$x \sqcup_\tau y = \begin{cases} y & \text{if } x = \bot \\ x & \text{if } y = \bot \\ \top & \text{if } x = \top \vee y = \top \\ x \cup y & \text{otherwise} \end{cases}$$

$$D = \bigcup_{\tau \in Class \cup PrimTy} PwrD_\tau$$

Example. Analysis of programs that use cryptography relies on custom (*i.e.*, manually written but typically application-independent) abstractions for some classes in `java.security` and related classes and primitive types. We describe an abstraction for `java.security.Signature`, which is used for the running example and the voting system in Section 9. For brevity, we usually omit package names; for example, `Signature` refers to `java.security.Signature`. A similar abstraction for `MessageDigest` is used in the analysis of the voting system in Section 9.

The `Signature` API requires that the information to be signed be marshalled into a byte array. The most common approach to marshalling uses `ObjectOutputStream` and `ByteArrayOutputStream`, like the code in Figure 5. Another approach is to form a string and then use `String.getBytes`, which converts a string into a byte array. Thus, custom abstractions are also used for `String`, `StringBuffer`, `ByteArrayOutputStream` and `ObjectOutputStream`. The custom abstract domains defined below use the following abstract values. For $x \in D_{\texttt{ByteArrayOutputStream}}$, $toByteArray(x)$ represents the return value of x`.toByteArray()`. $getBytes(x)$ and $toString(x)$ are analogous to $toByteArray(x)$. $writeObj(o)$ represents the data appended to an `ObjectOutputStream` by `writeObject(`o`)`.$writeInt$ is analogous to $writeObj$. $sign(alg, key, data)$ represents the return value of an invocation of `Signature.sign()` on an instance of `Signature` with signature algorithm alg (for brevity, we omit the algorithm parameters), public key key, and data $data$. $Signature(alg, mode, key, data)$ represents the state of an instance of `Signature`, where: alg represents the signature algorithm; $mode$ is "uninitialized", "signing", or "verifying" (depending on whether `initSign` or `initVerify` was invoked more recently); key represents the cryptographic key; and $data$ represents the data for which a signature will be created or verified when `sign()` or `verify(byte[] signature)` is invoked. For a set S, S^* denotes the set of finite sequences of elements of S.

$$D_{\texttt{String}} = Rtrvd_{\texttt{String}} \cup (\texttt{char}^* \cup \{toString(x) \mid x \in Rtrvd_{\texttt{Object}}\})^*$$
$$D_{\texttt{StringBuffer}} = D_{\texttt{String}}$$
$$D_{\texttt{ObjectOutputStream}} = (\{writeObj(x) \mid x \in Rtrvd_{\texttt{Object}}\} \cup \{writeInt(x) \mid x \in D_{\texttt{int}}\})^* \cup Rtrvd_{\texttt{ObjectOutputStream}}$$
$$D_{\texttt{ByteArrayOutputStream}} = D_{\texttt{ObjectOutputStream}} \cup Rtrvd_{\texttt{ByteArrayOutputStream}}$$
$$D_{\texttt{byte[]}} = \{getBytes(x) \mid x \in D_{\texttt{String}}\} \cup \{toByteArray(x) \mid x \in D_{\texttt{ByteArrayOutputStream}}\} \cup \{sign(alg, key, data) \mid alg \in Rtrvd_{\texttt{String}} \wedge key \in D_{\texttt{PublicKey}} \wedge data \in D_{\texttt{byte[]}}\} \cup Rtrvd_{\texttt{byte[]}}$$
$$D_{\texttt{Signature}} = \{Signature(alg, mode, key, data) \mid alg \in Rtrvd_{\texttt{String}} \wedge mode \in \{\text{uninit, signing, verifying}\} \wedge key \in D_{\texttt{Key}} \wedge data \in D_{\texttt{byte[]}}\} \cup Rtrvd_{\texttt{signature}}$$

This definition of $PwrD_{\texttt{ByteArrayOutputStream}}$ suffices for applications in which every `ByteArrayOutputStream` is fed from an `ObjectOutputStream`. We use the default abstract domain for `int`, `PublicKey`, and `Key`. Note that `PublicKey` extends `Key`.

These classes have no public instance fields, so for the custom load and store functions, it suffices to use functions that always return $\top$.

A custom abstraction is used for `boolean`. Its design follows the general pattern described above. Specifically, $PwrD_{\texttt{boolean}}$ contains the following kinds of elements related to `Signature`: $verify(alg, key, data, sig)$, which represents a return value of `Signature.verify`, holds iff sig is a valid signature of $data$ using key key and signature algorithm alg (this is false if, *e.g.*, sig is null, or key is invalid); $availableSigAlg(alg)$ holds iff signature algorithm alg is available in the run-time environment; and $compatible(keyAlg, sigAlg)$ holds if the specified algorithms are compatible (*e.g.*, *compatible*(`"DSA"`, `"SHA1withDSA"`) holds; *compatible*(`"DSA"`, `"SHA1withRSA"`) does not). To see why, for example, elements of the form $verify(\cdots)$ are useful for analysis of `getLength`, note that the output of `getLength` depends on the return value of the invocation of `verify`, so, to achieve accurate input partitioning, the analysis needs to determine

how that return value depends on the inputs to the method; such elements of $PwrD_{\texttt{boolean}}$ support this.

Environments. An environment maps local variables with primitive types, parameters with primitive types, and nodes in the PTE graph to D. (The PTE graph contains everything we need to know about values of local variables and parameters with reference types.) Let Lcl_{prim} and $Param_{prim}$ be the sets of local variables and parameters, respectively, with primitive types. Let N be the set of nodes of the PTE graph. Then $Env = (Lcl_{prim} \cup Param_{prim} \cup N) \to D$. Let ρ range over Env.

Example. The environment ρ at the program point before node st16 of the CFG for `C.getLength` appears below. $Integer(d)$ is an element of the default abstraction for `Integer`. [ ] denotes a byte array of length zero. n_{lbl} denotes the node labeled with lbl in Figure 2.

$$\begin{aligned}
\rho(n_{\text{this}}) &= \{\text{this}\} \\
\rho(n_{\text{sba}}) &= \{\text{sba}\} \\
\rho(n_{\text{st2}}) &= \{\text{this.pubKey}\} \\
\rho(n_{\text{st3}}) &= \{Signature(\text{"}\texttt{SHA1withDSA}\text{"}, \text{verifying}, \\
&\qquad \text{this.pubKey}, [\,])\} \\
\rho(n^e_{\text{st3}}) &= \bot \\
\rho(n^e_{\text{st4}}) &= \bot \\
\rho(n_{\text{st5}}) &= \{\text{sba.data}\} \\
\rho(n_{\text{st7}}) &= \{\text{sba.sig}\} \\
\rho(n^e_{\text{st8}}) &= \bot \\
\rho(n_{\text{st14}}) &= \{Integer(\text{sba.data.length})\} \\
\rho(b) &= \{verify(\text{"}\texttt{SHA1withDSA}\text{"}, \text{this.pubKey}, \\
&\qquad \text{sba.data}, \text{sba.sig})\}
\end{aligned}$$

In $\rho(n_{\text{st3}})$, the *data* component of the *Signature* is a byte array of length zero, because `Signature.verify` resets that component. All the exc-return nodes n have $\rho(n) = \bot$ and $e(n) = \emptyset$ at this program point, because the corresponding exceptions are not thrown on any path to this point.

7.2 Data-Flow Analysis Algorithm

This subsection describes how to calculate an environment at each program point. Readers not interested in the details of this calculation may jump to Section 8.

When analyzing a method m in the system's interface, all methods not in M_{cust} are inlined. For each method m in M_{cust}, a *method abstraction* $[\![m]\!]$ describing the behavior of m must be supplied. The data-flow analysis is expressed as a set of constraints that are solved by a worklist algorithm. Each constraint relates the state before a statement st with the state after execution of st; the constraint is expressed in terms of a *transfer function* $[\![st]\!]$ that captures the relevant semantics of st. The transfer function for an invocation statement uses the method abstraction for the invoked method. The next three subsections describe method abstractions, transfer functions, and the constraint-based analysis algorithm, respectively.

7.2.1 Abstractions of Methods in M_{cust}

Let thrownExc(m) denote the set of exception types that method m can throw (including subclasses of `RunTimeException`) plus a special element "normal" representing normal termination. For an invocation statement st that invokes m, let thrownExc(st) = thrownExc(m).

For each method m in M_{cust}, an abstract version $[\![m]\!]$ is supplied. Consider an invocation statement st of the form $l = l_0.m(l_1, \ldots, l_k)$. If m returns a reference and can throw an exception, then the data-flow analysis determines the effect of st on the environment by calling $[\![m]\!]_{\alpha(\bullet st)}(l, l_0, l_1, \ldots, l_k, n_{st}, n^{exc}_{st}, \rho)$. Recall that α is the result of PTE analysis; the PTE graph $\alpha(\bullet st)$ is an argument to $[\![m]\!]$. If m does not return a reference or cannot throw an exception, then the next-to-next-to-last or next-to-last argument to $[\![m]\!]$, respectively, is a dummy value.

The call to $[\![m]\!]$ returns a pair of functions $\langle f_{\text{env}}, f_{\text{guard}}\rangle$, both with domain thrownExc(m). $f_{\text{env}}(e)$ is the environment obtained by updating ρ to reflect the effect of execution of m when m terminates in the manner described by e. $f_{\text{guard}}(e)$ is (an over-approximation of) the pre-condition for m to terminate in the manner described by e. $f_{\text{guard}}(e)$ is in $D_{\texttt{boolean}}$. If m returns a reference or primitive value, the return value is reflected in $f_{\text{env}}(\text{normal})$ by an updated binding for n_{st} or l, respectively. For $e \neq$ normal, $f_{\text{env}}(e)$ contains an updated binding for n^{exc}_{st}. We represent f_{env} and f_{guard} as sets of pairs or as λ-terms. Recall that $(\lambda v.\ expr)$ is a function with parameter v that returns the value of *expr*.

The program checks whether the target l_0 is null before the method invocation, so abstractions of methods can assume l_0 is non-null.

The environment $\rho[x \mapsto y]$ is the same as ρ except that x is mapped to y. Similarly, $\rho[\forall x \in S : x \mapsto y]$ is an environment with an updated binding for each x in S. Let g range over PTE graphs. I_g is the set of inside edges of g, and $I_g(v) = \{n \mid \langle v, n\rangle \in I_g\}$.

Two auxiliary functions, getVal and setVal, are useful for expressing many method abstractions. $\text{getVal}_g(\rho, v)$ looks up the value of a variable v at a program point with PTE graph g and environment ρ. If v has reference type, getVal follows edges in g to determine the nodes to which v might point, uses the environment ρ to obtain the values associated with those nodes, and returns the join of those values. The join operation on D is induced by the join operations on the type-specific domains:

$$x \sqcup_D y = \begin{cases} x \sqcup_\tau y & \text{if } x \in D_\tau \wedge y \in D_\tau \\ & \quad \text{for some } \tau \\ \top & \text{otherwise} \end{cases}$$

Since $\sqcup_D$ is associative and commutative, we generalize it to apply to any subset of D, and take $\sqcup_D \emptyset = \bot$.

$$\text{getVal}_g(\rho, v) = \text{if } type(v) \in Class \text{ then } \sqcup_D \{\rho(n) \mid n \in I_g(v)\} \\ \text{else } \rho(v)$$

$\text{setVal}_g(\rho, l, d)$ returns an updated environment that reflects the effect of an assignment $l = d$ executed from a program point with PTE graph g and environment ρ. Suppose $type(l) \in Class$. If l has outedges to exactly one node n, and n represents at most one object (it is easy to give sufficient conditions for this), then g determines a single storage location to which l definitely points, and we say that l is *singular* in g, denoted $\text{singular}_g(l)$. In this case, $\text{setVal}_g(\rho, l, d)$ returns an environment in which that storage location is mapped to d. This is called *strong update*. Otherwise, each storage location to which g possibly points is mapped to the join of its current value and d, because we do not know which location will actually be updated by the assignment. This is called *weak update*. If $type(l) \in PrimTy$, strong update is

always used.

$$\begin{aligned}\mathrm{setVal}_g(\rho, l, d) = \text{if } &\mathit{type}(l) \in \mathit{Class} \text{ then}\\ &\quad \text{if singular}_g(l) \text{ then } \rho[\forall n \in I_g(l) : n \mapsto d]\\ &\quad \text{else } \rho[\forall n \in I_g(l) : n \mapsto \rho(n) \sqcup_D d]\\ &\text{else } \rho[l \mapsto d]\end{aligned}$$

singular is defined as follows. A node n is singular in a PTE graph g if it represents at most one object; this is true if (1) either n is an inside node, and the object creation statement st corresponding to n is not contained in a cycle in the CFG (hence st executes at most once), or n is a parameter node; and (2) n does not escape except possibly through the parameters or return value (*i.e.*, $e_g(n) \subseteq \mathit{Param}$). $\mathrm{singular}_g(l)$ holds if $I_g(l)$ contains exactly one node n, and n is singular in g.

Example. Our abstractions for methods of `ByteArrayOutputStream`, `ObjectOutputStream`, and `Signature` perform straightforward manipulations of the abstract values defined in Section 7.1. We give an abstraction for `Signature.verify` here; a few other method abstractions appear in Appendix A.

The behavior of `verify` is sketched in Section 3. An abstraction for it appears in Figure 3. The notation is based on Standard ML. The method declarator `Signature.verify([B)Z` specifies the method's name and type signature using notation from Java bytecode. Recall that `verify` has two parameters: `this` of type `Signature`, and `signature` of type `byte[]`. getVal is used to obtain the values *this* and *sig* of the parameters. If either argument to our abstraction for `verify` is an abstract value containing multiple possible values, our abstraction "gives up" (*i.e.*, returns a very approximate answer). *getMode* is a selector for $D_{\texttt{Signature}}$, defined by: $\mathit{getMode}(\mathit{Signature}(\mathit{alg}, \mathit{mode}, \mathit{key}, \mathit{data})) = \mathit{mode}$. If $\mathit{getMode}(\mathit{this}) \neq$ verifying, `SignatureException` is thrown, *i.e.*, the environment is updated to bind the exc-return node n^e to a `SignatureException`. Recall that $cl()$ is a tuple constructor for the default abstraction for a class cl with no instance fields. We use a zero-ary constructor for `SignatureException` and other exception classes, because we ignore the sole instance field of these classes, namely the detail message, as mentioned in Section 3. This is safe because, when using the default cryptography provider in Sun JDK 1.3, the detail messages do not contain significant information from the parameters. If $\mathit{getMode}(\mathit{this}) =$ verifying, the environment is updated in two steps: first, setVal is used to update the values bound to nodes corresponding to `this` (specifically, nodes that l_0 points to); second, the return value is bound to l (the return value is bound to a variable, not a node, because it has primitive type). The return value of the call to `sign` is represented by $\mathit{verify}(\mathit{alg}, \mathit{key}, \mathit{data}, \mathit{sig})$, which is described in Section 7.1. It is easy to improve this abstraction so that it sometimes returns a specific answer (true or false) when the argument is an element of $\mathit{PwrD}_{\texttt{byte[]}}$ of the form $\{\mathit{sign}(\mathit{alg}, \mathit{key}, \mathit{data})\}$.

$$\begin{aligned}&[\![\texttt{Signature.verify([B)Z}]\!]_g(l, l_0, l_1, n, n^{exc}, \rho) =\\ &\quad \text{let } \mathit{this}_p = \mathrm{getVal}_g(\rho, l_0)\\ &\quad \text{and } \mathit{signature}_p = \mathrm{getVal}(\rho, l_1) \text{ in}\\ &\quad \text{if } \mathit{this}_p = \{\mathit{this}\} \text{ and } \mathit{signature}_p = \{\mathit{sig}\}\\ &\quad \text{for some } \mathit{this} \in D_{\texttt{Signature}} \text{ and } \mathit{sig} \in D_{\texttt{byte[]}} \text{ then}\\ &\qquad \text{let } \mathit{exns} = \{\texttt{SignatureException},\\ &\qquad\qquad \texttt{NullPointerException}\}\\ &\qquad \text{and } \langle \mathit{alg}, \mathit{mode}, \mathit{key}, \mathit{data}\rangle =\\ &\qquad\quad \langle \mathit{getAlg}(\mathit{this}), \mathit{getMode}(\mathit{this}),\\ &\qquad\quad \mathit{getKey}(\mathit{this}), \mathit{getData}(\mathit{this})\rangle \text{ in}\\ &\qquad \text{let } \mathit{result} = \mathit{verify}(\mathit{alg}, \mathit{key}, \mathit{data}, \mathit{sig})\\ &\qquad \text{and } \mathit{this}' = \{\mathit{Signature}(\mathit{alg}, \mathit{mode}, \mathit{key}, [\,])\} \text{ in}\\ &\qquad \text{let } f_{\mathrm{env}} = \{\langle \text{normal}, \mathrm{setVal}_g(\rho, l_0, \mathit{this}')[l \mapsto \mathit{result}]\rangle\}\\ &\qquad\qquad \cup \textstyle\bigcup_{cl \in \mathit{exns}} \{\langle cl, \rho[n^{exc} \mapsto cl()]\rangle\}\\ &\qquad \text{and } f_{\mathrm{guard}} =\\ &\qquad\quad \{\langle \text{normal}, \mathit{getMode}(\mathit{this}) = \text{verifying} \wedge \mathit{sig} \neq \text{null}\rangle,\\ &\qquad\quad \langle \texttt{SignatureException}, \mathit{getMode}(\mathit{this}) \neq \text{verifying}\rangle,\\ &\qquad\quad \langle \texttt{NullPointerException}, \mathit{getMode}(\mathit{this}) = \text{verifying}\\ &\qquad\qquad\qquad \wedge \mathit{sig} = \text{null}\rangle\} \text{ in}\\ &\qquad \langle f_{\mathrm{env}}, f_{\mathrm{guard}}\rangle\\ &\quad \text{else } \langle (\lambda\, \mathit{exc}.\top), (\lambda\, \mathit{exc}.\text{true})\rangle\end{aligned}$$

Figure 3: Abstraction for `Signature.verify`.

7.2.2 Transfer Functions

For a statement st, the transfer function $[\![st]\!]$ characterizes the effect of execution of st on the environment (a more formal requirement appears in Section 10). Stmt_{invoc} is the set of invocation statements. If $st \notin \mathit{Stmt}_{invoc}$, then $[\![st]\!](\rho)$ is the environment resulting from execution of st in environment ρ. If $st \in \mathit{Stmt}_{invoc}$, then for $x \in \mathrm{thrownExc}(st)$, $[\![st]\!](\rho)(e)$ is the environment resulting from execution of st in environment ρ if x terminates in the manner described by e. If st executed in ρ cannot terminate in the manner described by e, then $[\![st]\!](\rho)(e)$ is unconstrained (although returning $\perp$ makes the analysis more accurate).

The transfer functions are implicitly parameterized by the result α of PTE analysis; this allows, *e.g.*, the transfer function for a store statement st to use the PTE graph $\alpha(\bullet st)$ to determine which nodes represent objects that might be updated by st. Let e_g denote the escape function of PTE graph g. For $d \in D$, let

$$\mathrm{load}(d, f) = \begin{cases} d & \text{if } d \in \{\perp, \top\}\\ \{\mathrm{load}_{cl}(d, f) \mid d \in d\} & \text{if } d \subseteq D_{cl} \text{ for some}\\ & \ cl \in \mathit{Class} \text{ with a field } f\\ \top & \text{otherwise}\end{cases}$$

The transfer functions in Figure 4 are reasonably straightforward, except perhaps for load and store statements. Consider a load statement st_{load}. If $\mathit{type}(l_1) \in \mathit{PrimTy}$, the transfer function for st_{load} updates the environment in a straightforward way. If $\mathit{type}(l_1) \in \mathit{Class}$, the transfer function for st_{load} updates $\rho(n_{st_{\mathrm{load}}})$ to represent values that may be loaded into l_1 from field f of an object o such that a reference to o was stored in $l_2.f$ by code outside m; if S_E is empty, no such objects exist, and $\rho(n_{st_{\mathrm{load}}})$ is set to $\perp$. The transfer function for store statements, like the definition of setVal, uses strong update or weak update, as appropriate. A field f of class cl is singular, denoted $\mathrm{singularFld}(cl.f)$, if it represents a single location in an instance of cl; this is false only for the `elements` field of array classes.

Our data-flow analysis, like the PTE analysis in [9], does not attempt to exclude infeasible execution paths; thus, the transfer function for branch statements is simply the identity function, and the guard function returned by $[\![m]\!]$ is ignored in $[\![st_{\mathrm{call}}]\!]$. Currently, branch conditions and guards are used only in Step 3. They could also be used to increase the accuracy of data-flow analysis.

$[\![l = c]\!](\rho) = \text{if } type(l) \in Class \text{ then } \rho \text{ else } \rho[l \mapsto \{c\}]$

$[\![l = v]\!](\rho) = \text{if } type(l) \in Class \text{ then } \rho \text{ else } \rho[l \mapsto \rho(v)]$

$$
\begin{array}{l}
[\![st_{\text{load}}]\!](\rho) = \\
\quad \text{let } g = \alpha(\bullet st_{\text{load}}) \text{ in} \\
\quad \text{if } type(l_1) \in Class \text{ then} \\
\qquad \text{let } S_E = \{n \in I_g(l_2) \mid e_g(n) \neq \emptyset\} \text{ in} \\
\qquad \rho[n_{st_{\text{load}}} \mapsto \sqcup_D \{\text{load}(\rho(n), f) \mid n \in S_E\}] \\
\quad \text{else } \rho[l_1 \mapsto \sqcup_D \{\text{load}(\rho(n), f) \mid n \in I_g(l_2)\}]
\end{array}
$$

$$
\begin{array}{l}
[\![st_{\text{store}}]\!](\rho) = \\
\quad \text{let } \tau = type(l_2) \\
\quad \text{and } g = \alpha(\bullet st_{\text{store}}) \\
\quad \text{and } d = \text{getVal}_g(\rho, l_1) \text{ in} \\
\quad \text{if singular}_g(l_2) \wedge \text{singularFld}(\tau.f) \text{ then} \\
\qquad \rho[(\forall n \in I_g(l_2) : n \mapsto \text{store}_\tau(\rho(n), f, d))] \\
\quad \text{else } \rho[(\forall n \in I_g(l_2) : n \mapsto \text{store}_\tau(\rho(n), f, \\
\qquad\qquad d \sqcup_D \text{load}(\rho(n), f)))]
\end{array}
$$

$[\![l_1 = l_2[l_3]]\!](\rho) = [\![l_1 = l_2.\texttt{elements}]\!](\rho)$
$[\![l_2[l_3] = l_1]\!](\rho) = [\![l_2.\texttt{elements} = l_1]\!](\rho)$

$[\![st_{\text{id}}]\!](\rho) = \rho$

$$
\begin{array}{l}
[\![st_{\text{call}}]\!](\rho) = \\
\quad \text{let } \langle f_{\text{env}}, _\rangle = [\![m]\!]_{\alpha(\bullet st_{\text{call}})}(l, l_0, l_1, \ldots, l_k, n_{st_{\text{call}}}, n^{exc}_{st_{\text{call}}}, \rho) \\
\quad \text{in } f_{\text{env}}
\end{array}
$$

$[\![l := l_1\ op\ l_2]\!](\rho) = \rho[l \mapsto \top]$

Figure 4: Transfer functions. m denotes an instance method in M_{cust}; for a static method, omit l_0. st_{load} has the form $l_1 = l_2.f$. st_{store} has the form $l_2.f = l_1$. st_{id} is a return, throw, object creation, or branch statement. st_{call} has the form $l = l_0.m(l_1, \ldots, l_k)$.

7.2.3 Calculation of Environments

The data-flow analysis produces a function β that maps program points to environments, except that, for each invocation statement st, β maps the program point $st\bullet$ to a function of type thrownExc(st) $\rightarrow$ *Env*. β is the least solution of the following constraints. It can be computed by a straightforward worklist algorithm that uses the transfer functions to propagate successively better approximations to the environment at each program point until a fixed-point is reached.

The initial environment is

$$
\rho_0(x) = \begin{cases} \{x\} & \text{if } x \in Param_{prim} \text{ or } x \text{ is a parameter node} \\ \{x\} & \text{if } x \text{ is } n_{cl} \in N \text{ for some } cl \in Class_{rdOnly} \\ \{\top\} & \text{if } x \text{ is } n_{cl} \in N \text{ for some } cl \notin Class_{rdOnly} \\ \bot & \text{otherwise} \end{cases}
$$

The join operation $\sqcup_{Env}$ for *Env* is the point-wise extension of $\sqcup_D$. Since $\sqcup_{Env}$ is associate and commutative, we generalize it to apply to any set of environments. Note that $\sqcup_{Env}\emptyset = \bot$. $\geq_{Env}$ is the partial order induced by the join operation $\sqcup_{Env}$. N_{esc} is the set of nodes that possibly escape the method before it returns (*i.e.*, that escape other than through the return value) and are not read-only. Recall from Section 6 that nodes reachable from $Class_{rdOnly} \cup Param_{rdOnly}$ are read-only, and methods in M_{cust} do not let objects escape, so N_{esc} is the set of nodes n such that $e_{\alpha(\texttt{exit}_m\bullet)}(n) \not\subseteq Class_{rdOnly} \cup Param_{rdOnly} \cup Stmt_{invoc}$. Constraints on the last line below use $\top$ to reflect the possibility that nodes in N_{esc} might be updated arbitrarily by concurrently executing threads.

$\beta(\texttt{enter}_m\bullet) = \rho_0$

$$
\begin{array}{l}
\beta(\bullet st) = (\bigsqcup_{Env}\{\beta(st'\bullet) \mid st' \in \text{pred}(st) \setminus Stmt_{thrower}\}) \\
\qquad \sqcup_{Env}(\bigsqcup_{Env}\{\beta(st'\bullet)(e) \mid st' \in \text{pred}(st) \cap Stmt_{thrower} \\
\qquad\qquad \wedge\ e \in excns(st', st)\})
\end{array}
$$

$\beta(st\bullet) \geq_{Env} [\![st]\!](\beta(\bullet st)) \quad$ for $st \neq \texttt{enter}_m$

$\beta(st\bullet)(n) = \top \quad$ for $st \in Stmt$ and $n \in N_{esc}$

8. STEP 3: CALCULATION OF PARTITION

We introduce three items that capture the relevant information from Steps 1 and 2. We then use these items to construct a partition. The partition reflects the information about inputs that can escape from the method. Information can escape either by being part of a value that escapes (*e.g.*, sba.data.length is part of the return value of `getLength`) or by being inferrable from a value that escapes (*e.g.*, *verify*("`SHA1withDSA`", this.pubKey, sba.data, sba.sig) is inferrable when the return value of `getLength` is an `Integer`). Accordingly, the items are: (1) a set $Stmt_{esc}$ of statements that might let values escape from invocations of m; (2) for each $st \in Stmt_{esc}$, an abstract value $esc(st)$ representing information that possibly escapes at st; and (3) for each $st \in Stmt$ and $st' \in \text{succ}(st)$, a predicate $guard(st, st')$ that is a necessary condition for control to flow from st to st'.

These three items are easily computed from the results α and β of PTE analysis and data-flow analysis, respectively. $Stmt_{esc}$ contains all return statements, throw statements that throw an exception that might not be caught in m, and store statements $l_2.f = l_1$ and array store statements $l_2[l_3] = l_1$ such that $I_{\alpha(\bullet st)}(l_2) \cap N_{esc} \neq \emptyset$, where N_{esc} is defined in the last paragraph of Section 7. For $st \in Stmt_{esc}$, $var(st)$ denotes the variable denoted by l_1 in Section 3. For a statement st in $Stmt_{esc}$, $esc(st) = \text{getVal}_{\alpha(\bullet st)}(\beta(\bullet st), var(st))$. For statements other than branches and method invocations, $guard(st, st')$ is the constant predicate "true". If st is an invocation statement and st' is a successor of st, then $guard(st, st')$ is determined by the conditions in f_{guard} (returned by the method abstraction) and $excns(st, st')$ in a straightforward way. If st is a branch statement, then the predicate is constructed in a straightforward way from st and $\beta(\bullet st)$. For example, suppose st is `if` l != 0, st' is the false successor of st, the type of l is `int`, and the default abstraction is used for `int`. If $\beta(\bullet st)(l) \in \{\bot, \top\}$, then $guard(st, st')$ is true (true is always a safe approximation); otherwise, $guard(st, st')$ is $\bigvee_{x \in \beta(\bullet st)(l)} x = 0$. If st is `if` $l ==$ null, and st' is the true successor of st, and $\beta(\bullet st)(l) \notin \{\bot, \top\}$, then $guard(st, st') = \bigvee_{x \in \beta(\bullet st)(l)} x = \text{null}$. Conditionals of the form `if` $l \neq$ null are handled similarly. For all other pointer comparisons, we take $guard(st, st_{\text{true}})$ and $guard(st, st_{\text{false}})$ to be true, although a more accurate analysis is possible based on the PTE graph. With this simple approximation, guards are unaffected by aliasing.

In general, guards are boolean formulas built from $\bigcup_\tau Rtrvd_\tau$, the binary relations corresponding to kinds of branch statements (namely, $=, \neq, >, <, \geq, \leq$, instanceof), and boolean operators.

We extend the notion of a guard from edges to paths. Let *Paths* contain all feasible (*i.e.*, executable for some input) edge-simple (*i.e.*, no repeated edges) paths from $\texttt{enter}_m$ to

$\texttt{exit}_m$. For $\sigma \in \mathit{Paths}$, let $\mathit{guard}(\sigma)$ be the conjunction of the guards of the edges in σ. Note that guards are expressed in terms of initial, not current, values of parameters and global storage, so they are not invalidated by assignments. If *Paths* also contains infeasible paths, the analysis result may contain some empty equivalence classes; this does not violate the correctness requirement but is undesirable noise.

For $\sigma \in \mathit{Paths}$, let $st_1, \ldots, st_n$ be the sequence of statements st in Stmt_{esc} that appear on σ and satisfy $\mathit{esc}(st) \notin \{\bot, \top\}$. The information that escapes along σ is represented abstractly (symbolically) by $\mathit{esc}(\sigma) = \mathit{esc}(st_1) \times \cdots \times \mathit{esc}(st_n)$. The set of distinct concrete data structures that might escape along σ is

$$\mathit{escStruct}(\sigma) = \mathit{struct}(\mathit{type}(\mathit{var}(st_1))) \times \cdots \times \mathit{struct}(\mathit{type}(\mathit{var}(st_n)))$$

where $\mathit{struct}(\tau)$ is Con_τ quotiented by structural equality; in other words, $\mathit{struct}(\tau)$ is a partition of Con_τ into equivalence classes based on structural equality.

Our partition aims for output coverage, so inputs that produce the same output may be placed in the same equivalence class. Accordingly, let *PPaths* be the partition of *Paths* under the equivalence relation: $\sigma \equiv \sigma'$ if $\mathit{esc}(\sigma) = \mathit{esc}(\sigma') \wedge |\mathit{esc}(\sigma)| = 1$. The condition $|\mathit{esc}(\sigma)| = 1$ reflects the fact that, if $|\mathit{esc}(\sigma)| > 1$, then different paths might be needed to produce different elements of $\mathit{esc}(\sigma)$. For $\hat{\sigma} \in \mathit{PPaths}$, let $\mathit{guard}(\hat{\sigma}) = \bigvee_{\sigma \in \hat{\sigma}} \mathit{guard}(\sigma)$ and $\mathit{escStruct}(\hat{\sigma}) = \bigcup_{\sigma \in \hat{\sigma}} \mathit{escStruct}(\sigma)$.

If there exists $st \in \mathit{Stmt}_{esc}$ such that $\mathit{esc}(st) = \top$, then the analysis result is undefined (the analysis could still provide some constraints on an input partition based on analysis results for paths that do not contain such statements, but we do not pursue this). Otherwise, the partition contains an equivalence class for each structurally distinct value s that can escape along some path in some equivalence class $\hat{\sigma}$ in *PPaths*. The partition also reflects that only inputs satisfying $\mathit{guard}(\hat{\sigma})$ can lead to output s along $\hat{\sigma}$.

$$\mathit{partn}(m) = \bigcup_{\substack{\hat{\sigma} \in \mathit{PPaths} \\ x \in \mathit{esc}(\hat{\sigma}) \\ s \in \mathit{escStruct}(\hat{\sigma})}} \{\{\mathit{parstat} \mid x \in s \ \wedge \ \mathit{guard}(\hat{\sigma})\}\} \tag{2}$$

where *parstat* is a tuple containing the parameters of m and classes whose static fields are used by m. Occurrences of parameters and static variables in x and $\mathit{guard}(\hat{\sigma})$ are captured (bound) by *parstat* in the set comprehension. The formula has two levels of curly braces, because the partition is a set of equivalence classes, and each equivalence class is a set (of bindings). For an abstract value x, the meaning of $x \in s$ is: $x \in s$ iff s contains some instance of cl represented by x. The set of instances represented by an abstract value is closed under structural equality, so s contains either all or none of the instances of cl represented by x.

We apply the following simplifications to obtain the final analysis result. Guards are simplified using standard boolean identities. If all fields of a class cl have primitive type, we replace the union over $\mathit{struct}(cl)$ with unions over the values of those fields; for example, replace

$$\bigcup_{s \in \mathit{struct}(\texttt{Integer})} \{\{\mathit{parstat} \mid x \in s \ \wedge \ \cdots\}\}$$

with $\bigcup_{i \in \texttt{int}} \{\{\mathit{parstat} \mid x.\text{value} = i \wedge \cdots\}\}$. If $x \in \mathit{esc}(\hat{\sigma})$ is a constant (*e.g.*, null), we simplify $\bigcup_{s \in \mathit{escStruct}(\hat{\sigma})} \{\{\mathit{parstat} \mid \cdots\}\}$ to $\{\{\mathit{parstat} \mid \mathit{guard}(\hat{\sigma})\}\}$.

Example. To illustrate the calculation of $\mathit{partn}(\texttt{getLength})$, consider the path $\sigma \in \mathit{Paths}(\texttt{getLength})$ that contains nodes 1-16 and 21. The only statement on σ and in Stmt_{esc} is st16. We have $\mathit{esc}(\text{st16}) = \{\mathit{Integer}(\text{sba.data.length})\}$. We take the constructor for tuples with 1 component to be the identity function, so $\mathit{esc}(\sigma) = \mathit{esc}(\text{st16})$ and

$$\mathit{escStruct}(\sigma) = \{\{o \in \mathit{Con}_{\texttt{Integer}} \mid o.\texttt{value} = \mathit{val}\} \mid \mathit{val} \in \texttt{int}\},$$

and $\mathit{guard}(\sigma) = \mathit{normalGetLength}$, where the macro *normalGetLength* is defined by

$$\begin{aligned} &\mathit{normalGetLength} = \\ &\quad \mathit{availableSigAlg}(\text{"}\texttt{SHA1withDSA}\text{"}) \\ &\wedge \ \text{sba} \neq \text{null} \ \wedge \ \text{this.pubKey} \neq \text{null} \\ &\wedge \ \mathit{compatible}(\text{this.pubKey.getAlgorithm}(), \text{"}\texttt{SHA1withDSA}\text{"}) \\ &\wedge \ \mathit{verify}(\text{"}\texttt{SHA1withDSA}\text{"}, \text{this.pubKey}, \text{sba.data}, \text{sba.sig}). \end{aligned}$$

In *PPaths*, σ is in a singleton equivalence class, which we denote by $\hat{\sigma}$, so $\mathit{guard}(\hat{\sigma}) = \mathit{guard}(\sigma)$ and $\mathit{escStruct}(\hat{\sigma}) = \mathit{escStruct}(\sigma)$. The contribution of $\hat{\sigma}$ to the partition is

$$\bigcup_{c \in \mathit{struct}(\texttt{Integer})} \{\{\langle \text{this}, \text{sba} \rangle \mid \mathit{Integer}(\text{sba.data.length}) \in c \ \wedge \mathit{normalGetLength}\}\}.$$

Applying the above simplifications (specifically, replacing the union of `Integer` with a union over `int`), we obtain the expression on lines 2 and 3 of equation (3) below. Equation (3) is the final analysis result. All incorrectly signed inputs are in one equivalence class. All correctly signed inputs with the same data length are in one equivalence class.

$$\begin{aligned} &\mathit{partn}(\texttt{C.getLength}) = \\ &\quad \{\{\langle \text{this}, \text{sba} \rangle \mid \neg \mathit{normalGetLength}\}\} \\ &\quad \cup \textstyle\bigcup_{i \in \texttt{int}} \{\{\langle \text{this}, \text{sba} \rangle \mid \text{sba.data.length} = i \\ &\qquad\qquad \wedge \ \mathit{normalGetLength}\}\} \end{aligned} \tag{3}$$

9. CASE STUDY: VOTING SYSTEM

We applied the analysis to our implementation of the fault-tolerant and intrusion-tolerant distributed voting system described in [5]. It has seven remote methods. We describe the analysis of equivalence classes for the arguments of one of them, namely, `PSI.contend`. `PSI` stands for PollingStationImplementation. The `contend` method is a challenge for the analysis, because its entire argument escapes in some cases. When a voter casts a ballot at a server ("polling station" and "server" are synonymous), that server remotely invokes `contend` on a quorum of servers. In the notation of [5], the argument of `contend` is a request of the form $\langle y_1, y_2 \rangle$ signed by some polling station u_i, where y_1 identifies the voter, and y_2 is evidence that the voter actually voted at u_i. `contend` checks that $\langle y_1, y_2 \rangle$ is correctly signed by u_i, and that the values of y_1 and y_2 are correct, based on data in the hashtable `PSI.accessTag`. If not, `contend` throws a `VotingExc`. Otherwise, the request is well-formed, and u_j believes that voter y_1 attempted to vote at u_i. A dishonest voter might try to vote multiple times, so `contend` next checks a hashtable to see whether voter y_1 already voted at some polling station u_j. If so, `contend` returns evidence that y_1 voted at u_j; if not, `contend` stores its argument in the hashtable as evidence that voter y_1 voted at u_i and then signs and returns its argument. For brevity, Figures 5 and 6

are based on a slightly simplified version of `contend` that, if the request is well-formed, simply returns its argument (*i.e.*, the check of whether voter y_1 already voted is omitted); this leads to almost the same equivalence classes.

In our implementation, the argument of `contend` is an instance of `SignedObject`, which is similar to `java.security.SignedObject`; it has fields `Serializable obj` (the payload), `int signer` (the ID of the server that signed the payload), and `byte[] sig` (the signature). Code for `contend` and `SignedObject.verify` appear in Figure 5. `contend` invokes static method `PSI.pubKey(int ps)`, which looks up `ps` in static variable `PSI.pubKeyArray`. `contend` directly accesses static variable `PSI.accessTag`. `SignedObject.verify` accesses static variable `PSI.sigAlg`.

The payload of the `SignedObject` argument to `contend` should be a `NetObject`. A `NetObject` has two fields, `y1` and `y2`, corresponding to y_1 and y_2 above. These fields have type `ByteArrayEquals`, which is like `byte[]` except that `equals` and `hashCode` are overridden with methods whose return values depend only on the contents of the array. `SignedObject.verify` and `ByteArrayEquals.equals` are not in M_{cust}, so they are inlined for the analysis. The latter method calls `java.util.Arrays.equals`, which is in M_{fun}.

The control-flow graph and PTE graph for `contend` appear in Appendix B.

Analysis results. Analysis of `contend` produces the partition in Figure 6. The three items in *partn*(`contend`) correspond to `contend` returning a `VotingExc`, a `NullPointerException`, and the argument `so`, respectively.

The analysis result uses elements of $D_{\texttt{boolean}}$ of the following forms: *serializable*(o), introduced by the abstraction for `ObjectOutputStream.writeObject`, holds if o is serializable; *containsKey*(h,k), introduced by the abstraction for `Hashtable.containsKey` (see Section 9.1), holds if hashtable h contains an entry with key k; and *arraysEquals*($a,a2$), introduced by the abstraction for `Arrays.equals(byte[] a, byte[] a2)`, holds if the two byte arrays have the same contents. In the voting system, signatures (*i.e.*, byte arrays returned by `Signature.sign`) are only passed around and verified, so it is safe to define *struct*(`SignedObject`) based on the equivalence relation that equates instances that differ only in the initialization vector used to generate the signature. The partition also uses abstract values of the form *element*(v), which represents all elements of the collection v; such abstract values are introduced by the abstraction for `Vector.elementAt`.

For any fixed values of read-only global storage, the partition *partn*(`PSI.contend`) contains a finite number of equivalence classes. The form in which the partition is expressed does not make this readily apparent. In future work, we plan to investigate automatic transformations that make this more evident by replacing predicates with unions. A predicate of the form *containsKey*(h,k) corresponds naturally to a union over the set of keys in h. Similarly, a predicate of the form *arraysEquals*(a, *element*($a2$)) corresponds to a union over the set of elements of $a2$. Introduction of unions is also a step towards generation of code for the environment; the next step is to translate unions into iterations (loops).

If incorrect values of static variables are considered infeasible in this application, then the second equivalence class

```
// remote method PSI.contend
public synchronized SignedObject
contend(SignedObject so) throws VotingExc {

 if (so==null || !so.verify(pubKey(so.signer)))
  // null ref. or invalid signature
  throw new VotingExc();

 if (!(so.obj instanceof NetObject))
  throw new VotingExc();

 NetObject n = (NetObject)so.obj;

 boolean aTcK = accessTag.containsKey(n.y1);
 if (!aTcK)
  throw new VotingExc();  // invalid voter ID y1

 Vector v = (Vector)accessTag.get(n.y1);

 // The description in [MalkiReiter98] suggests:
 // y2OK = v.contains(n.y2);
 // The correct check is:
 boolean y2OK = n.y2.equals(v.elementAt(so.signer));
 if (!y2OK)
  throw new VotingExc(); // invalid access tag y2

 // remainder of method has been simplified (see text)
 return so;
}

// method SignedObject.verify
public boolean verify(PublicKey k) {
 boolean verifies = false;
 if (obj != null && sig != null) {
  try {
   Signature dsa = Signature.getInstance(PSI.sigAlg);
   dsa.initVerify(k);
   ByteArrayOutputStream baos =
     new ByteArrayOutputStream();
   ObjectOutputStream oos =
     new ObjectOutputStream(baos);
   oos.writeObject(obj);
   dsa.update(baos.toByteArray());
   verifies = dsa.verify(sig);
  }
  catch (Exception exc) { }
 }
 return verifies;
}
```

Figure 5: Source code for `contend` and `verify`.

in *partn*(`PSI.contend`) is empty.

Input partition analysis of all the remote methods produces about 25 equivalence classes or families of equivalence classes for arguments or return values of remote methods.

Model checking. To generate code for the adversary based on the analysis result, we followed the approach in [7, 4]. Writing code that simulates voters is trivial. We tested the voting system together with the resulting environment in our implementation of a VeriSoft-like [2, 3] state-less model checker for distributed Java programs that communicate by RMI. It intercepts all invocations of and returns from remote methods and gives control to its own scheduler at those points. It provides an operation `int Random(int n)` that (like VeriSoft's VS_Toss) non-deterministically selects and returns a number between 0 and $\texttt{n}-1$; the scheduler

$$
\begin{aligned}
&\mathit{partn}(\texttt{PSI.contend}) = \\
&\quad \{\{\langle \text{this}, \text{so}, \text{PSI}\rangle \mid \neg \mathit{contendNormal}_1\}\} \\
&\quad \cup \{\{\langle \text{this}, \text{so}, \text{PSI}\rangle \mid \mathit{contendNormal}_1 \\
&\qquad\qquad \wedge (\text{PSI.pubKeyArray} = \text{null} \\
&\qquad\qquad\quad \vee\ \text{PSI.accessTag} = \text{null})\}\} \\
&\quad \cup \textstyle\bigcup_{c \in \mathit{struct}(\text{SignedObject})} \{\{\langle \text{this}, \text{so}, \text{PSI}\rangle \mid \mathit{contendNormal}_2 \\
&\qquad\qquad \wedge\ \text{so} \in c\}\} \\
&\mathit{contendNormal}_1 = \\
&\quad \text{so} \neq \text{null} \wedge \text{so.obj} \neq \text{null} \\
&\quad \wedge\ \text{so.signer} \in [0..\text{PSI.pubKeyArray.length} - 1] \\
&\quad \wedge\ \text{so.sig} \neq \text{null} \wedge \mathit{availableSigAlg}(\text{PSI.sigAlg}) \\
&\quad \wedge\ \text{PSI.pubKey(so.signer)} \neq \text{null} \\
&\quad \wedge\ \mathit{compatible}(\text{PSI.pubKey(so.signer).getAlgorithm()}, \\
&\qquad\qquad \text{PSI.sigAlg}) \\
&\quad \wedge\ \mathit{serializable}(\mathit{so.obj}) \\
&\quad \wedge\ \mathit{verify}(\text{PSI.sigAlg}, \\
&\qquad\qquad \text{PSI.pubKey(so.signer)}, \\
&\qquad\qquad \text{toByteArray}(\mathit{writeObj}(\text{so.obj})), \text{so.sig}) \\
&\quad \wedge\ \text{so.obj instanceof NetObject} \\
&\mathit{contendNormal}_2 = \\
&\quad \mathit{contendNormal}_1 \wedge \text{PSI.pubKeyArray} \neq \text{null} \\
&\quad \wedge\ \text{PSI.accessTag} \neq \text{null} \\
&\quad \wedge\ \mathit{containsKey}(\text{PSI.accessTag}, \text{so.obj.y1}) \\
&\quad \wedge\ \mathit{arraysEquals}(\text{so.obj.y2}, \\
&\qquad\qquad \mathit{element}(\text{PSI.accessTag.get(so.obj.y1)}))
\end{aligned}
$$

Figure 6: Partition for `contend`.

explores all of these values. This feature is used to nondeterministically select which remote method to invoke next and which equivalence class to use for the arguments or return value of an RMI. As in VeriSoft, the model checker backs up to a previous state s by restarting the system and re-executing a sequence of transitions that leads to s.

We model-checked the voting phase (not the tallying phase) of an election in a system configuration with 4 honest polling stations, 1 compromised polling station, 6 quorums, 1 voter, 1 candidate, and a stack depth limit (*i.e.*, bound on number of interactions between system and environment in an execution) of 4. This small configuration is useful for checking the property: if an uncompromised server u_j records (in its hash table) that a voter v voted at an uncompromised server u_i, then v voted at u_i. The model checker found a violation of this property in 2 seconds (on a 440 MHz Sun Ultra 10), due to an error in the description of the algorithm in [5], which says, "Each server, upon receiving $\langle y_1, y_2\rangle$ in u_i's signed request ..., finds the access tag $\langle y_1, S\rangle$ and verifies that $h(y_2) \in S$." In fact, the correct (and intended by the authors) version is to regard S as an array indexed by server ID and have the server verify that $h(y_2) = S[i]$. Figure 5 shows the original and corrected code. The model-checker checked the corrected system in 83 seconds, exploring 31,125 transitions (excluding 80,629 transitions executed during replay) and using 28 MB RAM. The number of distinct explored states is not easily determined, because the model-checker is state-less.

9.1 Analysis of Aggregates

We extend the above data-flow analysis to analyze aggregate data structures—such as arrays and classes that implement `java.util.Collection`—more accurately. Specifically, the goals are (1) to recognize when a method's behavior depends on whether a value is an element of a particular collection, and (2) to recognize when all elements of an aggregate satisfy a predicate (*e.g.*, all elements of `ballotArray` are signed with an appropriate key).

Analyzing Membership. A typical subproblem of membership analysis is, for a `Vector` v, to recognize membership in v for a return value of v.`elementAt` and for an object o when v.`contains`(o) returns true. A general approach to membership analysis is to introduce a map from control points to membership relationships and give transfer functions to compute it. We adopt a more concise but less expressive approach, by encoding this information in $\mathit{Rtrvd}_{\texttt{Object}}$ and $D_{\texttt{bool}}$ and using appropriate abstractions for methods that access collections. For our current applications, it suffices to keep track of membership in (read-only) collections referenced from read-only parameters and read-only static fields, so we extend $\mathit{Rtrvd}_{\texttt{Object}}$ with terms of the form $\mathit{element}(v)$, which represents an arbitrary element of the collection represented by v, where $v \in \mathit{Rtrvd}_{\texttt{Object}}$ (if v does not represent a collection, then $\mathit{element}(v)$ is not well-formed and will not arise in analysis of type-correct programs). Load and store functions for $D_{\texttt{Object}}$ are easily extended to accommodate these elements. We briefly describe abstractions for some methods of collections. Details of some of the abstractions appear in Figure 7.

`Hashtable.containsKey` is in M_{fun}. Our abstraction for it is similar in style to the abstraction for `verify`. Recall that h.`containsKey`(k) returns true iff hashtable h contains an entry with key k. The abstraction uses getVal to obtain both an element *this* of $\mathit{Rtrvd}_{\texttt{Hashtable}}$ representing the state of the target `Hashtable` and an element k of $\mathit{Rtrvd}_{\texttt{Object}}$ representing the key being looked up, and, if the key is not null, returns $\mathit{containsKey}(\mathit{this}, k)$. This requires that $\mathit{PwrD}_{\texttt{boolean}}$ contains $\mathit{containsKey}(h, k)$ for all $h \in \mathit{Rtrvd}_{\texttt{Hashtable}}$ and $k \in \mathit{Rtrvd}_{\texttt{Object}}$.

`Hashtable.get` is in M_{fun}. Our abstraction for it is like the abstraction for `containsKey` except that the environment associated with normal termination in the "true" branch is $\text{setVal}_g(\rho, l, \{m.\mathit{get}(k)\})$, not $\rho[l \mapsto \{\mathit{containsKey}(m, k)\}]$, where $m.\mathit{get}(k)$ is in $\mathit{Rtrvd}_{\texttt{Object}}$.

`Vector.contains` and `Vector.elementAt` are in M_{fun}. Our abstraction for `Vector.contains` is similar to our abstraction for `containsKey`. For `Vector.elementAt`, if the target object is a vector v and the method terminates normally (it can also throw `ArrayIndexOutOfBoundsException`), the abstraction returns $\mathit{element}(v)$, to indicate that the return value is an element of v.

Currently, we track membership only in read-only collections that are not updated after initialization, so our abstractions for update methods simply return $\top$. More accurate abstractions could easily be used.

`Arrays.equals` is in M_{fun}. Our abstraction for it is similar in style to our abstraction for `containsKey`. $\mathit{PwrD}_{\texttt{boolean}}$ contains $\mathit{arraysEquals}(a_1, a_2)$ for all array classes τ and all $a_1 \in \mathit{Rtrvd}_\tau$ and $a_2 \in \mathit{Rtrvd}_\tau$.

Properties of Elements of Aggregates. The second goal above requires extending the form of predicates to allow quantification over aggregates and recognizing code that iterates over aggregates. For analysis of many programs, it suffices to recognize cycles in the CFG with one or two sim-

$[\![\texttt{Hashtable.containsKey(LObject;)Z}]\!]_g(l, l_0, l_1, n, n^{exc}, \rho) =$
let $\mathit{this}_p = \mathrm{getVal}_g(\rho, l_0)$
and $\mathit{key}_p = \mathrm{getVal}_g(\rho, l_1)$ in
if $\mathit{this}_p = \{\mathit{this}\}$ and $\mathit{key}_p = \{\mathit{key}\}$
for some $\mathit{this} \in \mathit{Rtrvd}_{\texttt{Hashtable}}$ and $\mathit{key} \in \mathit{Rtrvd}_{\texttt{Object}}$ then
let $\mathit{exns} = \{\texttt{ClassCastException}, \texttt{NullPointerException}\}$ in
let $f_{\mathrm{env}} = \{\langle \mathrm{normal}, \rho[l \mapsto \{\mathit{containsKey}(\mathit{this}, \mathit{key})\}]\rangle\} \cup \bigcup_{cl \in exns}\{\langle cl, \rho[n^{exc} \mapsto cl()]\rangle\}$
and $f_{\mathrm{guard}} = \{\langle \mathrm{normal}, \mathit{key} \neq \mathrm{null}\rangle, \langle \texttt{NullPointerException}, \mathit{key} = \mathrm{null}\rangle, \langle \texttt{ClassCastException}, \mathrm{false}\rangle\}$ in
$\langle f_{\mathrm{env}}, f_{\mathrm{guard}}\rangle$
else $\langle(\lambda\, exc.\top), (\lambda\, exc.\mathrm{true})\rangle$

$[\![\texttt{Vector.elementAt(I)LObject;}]\!]_g(l, l_0, l_1, n, n^{exc}, \rho) =$
let $\mathit{this}_p = \mathrm{getVal}_g(\rho, l_0)$
and $\mathit{index}_p = \mathrm{getVal}_g(\rho, l_1)$ in
if $\mathit{this}_p = \{\mathit{this}\}$ and $\mathit{index}_p = \{\mathit{index}\}$
for some $\mathit{this} \in \mathit{Rtrvd}_{\texttt{Vector}}$ and $\mathit{index} \in \mathit{Rtrvd}_{\texttt{int}}$ then
let $\mathit{exns} = \{\texttt{ArrayIndexOutOfBoundsException}\}$ in
let $f_{\mathrm{env}} = \{\langle \mathrm{normal}, \mathrm{setVal}_g(\rho, l, \{\mathit{element}(\mathit{this})\})\rangle\} \cup \bigcup_{cl \in exns}\{\langle cl, \rho[n^{exc} \mapsto cl()]\rangle\}$
and $f_{\mathrm{guard}} = \{\langle \mathrm{normal}, \mathit{index} \in [0, \mathit{this}.\mathrm{size}() - 1]\rangle, \langle \texttt{ArrayIndexOutOfBoundsException}, \mathit{index} \notin [0, \mathit{this}.\mathrm{size}() - 1]\rangle\}$ in
$\langle f_{\mathrm{env}}, f_{\mathrm{guard}}\rangle$
else $\langle(\lambda\, exc.\top), (\lambda\, exc.\mathrm{true})\rangle$

Figure 7: Abstractions for some methods related to aggregates.

ple forms (*e.g.*, counter-based loops that iterate over arrays and vectors, and loops that use iterators over collections). If analysis of the loop body shows that some predicate ϕ holds for the "current" element and some other syntactic conditions hold, then the guard on the control flow edge traversed during normal termination of the loop may assert that ϕ holds for all elements of the aggregate.

We extend predicates to contain quantifications over aggregate data structures. We discuss only quantifications over arrays; quantifications over collections can be handled similarly. Extend the formulas used to express guards to include variables x (bound by quantifiers) and quantifications of the form $(\forall x \in a : \phi)$, where a is in D_{cl} for some array class cl, and ϕ is a formula. Occurrences of elements of Rtrvd_τ (for any τ) in a predicate may be tagged with a local variable, as discussed below.

A statement st *accesses* a node n if st contains a variable v of reference type such that $n \in I_{\alpha(\bullet st)}(v)$. A statement st (possibly) *updates* a node n if (1) st is a store statement $l_2.f = l_1$ and $n \in I_{\alpha(\bullet st)}(l_2)$, or (2) st is an invocation statement $l = l_0.m(l_1, \ldots, l_n)$ and there exists ρ such that the return value of $[\![m]\!](l, l_0, \ldots, l_k, n_{st}, n_{st}^{exc}, \rho)$ contains an environment ρ_1 such that $\rho(n) \neq \rho_1(n)$ (this is easily determined by inspection of the definition of $[\![m]\!]$).

For an array class τ and $a \in \mathit{Rtrvd}_\tau$, an *array iteration* over a with iteration variable l is a subgraph of the CFG of the form in Figure 8 such that there exists $n_3 \in N$ (which represents the array a) such that

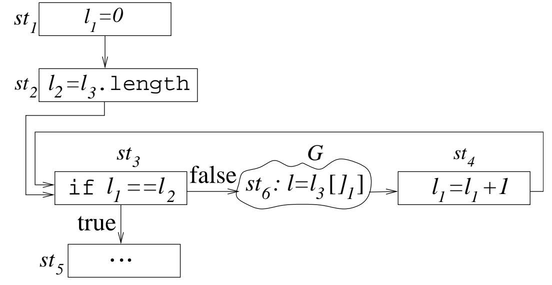

Figure 8: Subgraph recognized as iteration over an array.

1. subgraph G (the body of the loop) and all pictured nodes except st_1 have only the pictured inedges (they may have additional outedges, and there may be multiple edges from G to st_4)
2. $I_{\alpha(\bullet st_2)}(l_3) = \{n_3\} \;\wedge\; \beta(\bullet st_2)(n_3) = a$
3. G contains an array load statement st_6 of the form $l = l_3[l_1]$ that appears before all other statements in G that access l. G contains no other updates to l.
4. G does not update l_1 or l_2. G does not update n_3 (which represents the array) or load node n_{st_6} (which represents an element of the array).

st_5 is called the *final node* of the iteration.

$\mathit{guard}_A(st, st')$ is defined the same way as $\mathit{guard}(st, st')$ except as follows. If st' is in the body of an iteration over a with iteration variable l, and $st \in \mathrm{pred}(st')$, then $\mathit{guard}_A(st, st')$ is the same as $\mathit{guard}(st, st')$ except that uses of the value of l are tagged to indicate that they are from l. If st' is the final node of an iteration over a with iteration variable l, and st is the predecessor of st', then

$$\mathit{guard}_A(st, st') = \mathit{guard}(st, st') \;\wedge\; (\forall x \in a : \mathit{guard}(\bullet st)[v^l := x]) \tag{4}$$

where the expression in square brackets indicates substitution of x for all values v labeled with l. This replacement is safe because, in the transfer function for array load, the array index is not used on the right side of the equation, so $\mathit{guard}(st, st')$ is independent of l_1 and hence holds for all emements of the array, not only the last element. The analysis results are defined as in Section 8, except using guard_A instead of guard.

10. SEMANTICS OF TRANSFER FUNCTIONS

This section describes soundness for transfer functions.

A binding $b \in \mathit{Bind}(\mathit{Param} \cup \mathit{Var}_{\mathit{static}})$ can be extended to a function $[\![\cdot]\!]_b^{\mathrm{base}} \in (\bigcup_\tau \mathit{Rtrvd}_\tau) \to \mathrm{PowerSet}(\mathit{Con})$, defined by structural induction on elements of Rtrvd_τ. The use of a powerset here is needed only to accommodate values of the form $\mathit{element}(v)$, described in Section 9.1. We further extend this function to an interpretation

$$[\![\cdot]\!]_b^{D1} \in (\bigcup_{\tau \in \mathit{Class} \cup \mathit{PrimTy}} D_\tau) \to \mathrm{PowerSet}(\mathit{Con}).$$

For this purpose, we require that each custom abstraction D_τ is accompanied by an interprertation for elements of D_τ.

For elements $cl(d_1, \ldots, d_n)$ of default abstractions, we use the straightforward interpretation, namely, $[\![cl(d_1, \ldots, d_n)]\!]_b^{D1}$ is the set of instances of cl whose field values are consistent with $d_1, \ldots, d_n$, *i.e.*, the i'th field has a value in $[\![d_i]\!]_b^{D1}$. Finally, we extend this function to an interpretation $[\![\cdot]\!]_b^D \in D \to \mathrm{PowerSet}(Con)$, with the stipulation that $[\![\bot]\!]_b^D = Con$ and $[\![\top]\!]_b^D = Con$.

A *(concrete) state* σ contains values for operand stack, static variables, heap, *etc.* For a PTE graph g and an environment ρ, let $[\![g, \rho]\!]_b^{\mathrm{state}}$ be the set of states σ represented by g and ρ, *i.e.*, (1) g represents the heap in σ under some abstraction relation $r \subseteq Con \times N$, according to the semantics of PTE graphs [9], (2) for all $v \in Param_{prim} \cup Lcl_{prim}$, the value of v in σ is in $[\![\rho(v)]\!]_b^D$, and (3) for all $\langle o, n\rangle \in r$, $o \in [\![\rho(n)]\!]_b^D$. For a state σ, let $\mathrm{ps}(\sigma)$ denote the restriction of σ to bindings for parameters and static variables. Define the relation $\xrightarrow{st}$ by: $\sigma \xrightarrow{st} \sigma'$ iff execution of statement st in state σ can lead to state σ'.

Soundness of the environment transfer function $[\![st]\!]$ for a statement st in a method m requires the following. Let α be the result of PTE analysis of m. For all $\rho \in Env$, for all states σ and σ' such that $\sigma \in [\![\alpha(\bullet st), \rho]\!]_{\mathrm{ps}(\sigma)}^{\mathrm{state}}$ and $\sigma \xrightarrow{st} \sigma'$, (1) if $st \notin Stmt_{invoc}$, $\sigma' \in [\![\alpha(st\bullet), [\![st]\!](\rho)]\!]_{\mathrm{ps}(\sigma)}^{\mathrm{state}}$ and (2) if $st \in Stmt_{invoc}$, $\sigma' \in [\![\alpha(st\bullet), [\![st]\!](\rho)(e)]\!]_{\mathrm{ps}(\sigma)}^{\mathrm{state}}$, where $e \in \mathrm{thrownExc}(st)$ is the manner in which the invocation terminated (this can be determined from σ').

11. PARTITIONS FOR RETURN VALUES OF REMOTE METHODS

The preceding analysis is aimed at determining equivalence classes for arguments of remote methods. In secure systems, it is also necessary to determine equivalence classes for return values of remote methods, since these are also controlled by the adversary. We adapt the preceding analysis for this purpose by augmenting the caller with fresh parameters representing return values of remotely invoked methods and computing an input partition for the augmented method.

For each invocation statement st that invokes a remote method m (we assume such statements can be identified statically), (1) augment m with parameters $p_{st,x}$ for $x \in \mathrm{thrownExc}(m) \cup$ `RemoteException`, (2) add m to M_{cust} (hence the invocation is not inlined) with a custom abstraction for this call site (we use m_{st} as the method name to emphasize that this abstraction is for use only at st):

$$[\![m_{st}]\!]_g(l, l_0, l_1, \ldots, l_k, n, n^{exc}, \rho) = \{\langle x, \mathrm{setVal}_g(\rho, l, n_{p_{st,x}})\rangle \mid x \in (\mathrm{thrownExc}(m) \cup \texttt{RemoteException})\}. \quad (5)$$

and (3) add st to $Stmt_{esc}$, because the arguments escape from the caller. Note that we allow here an exception to our usual assumption that information does not escape through methods in M_{cust}.

Since $Stmt_{esc}$ now contains (remote) invocation statements, in order to use the partition construction in Section 8, we need to extend the definition of $var(st)$ to apply to invocation statements. The simplest approach is to treat an invocation statement of the form $l = l_0.m(l_1, \ldots, l_k)$ as k separate statements, where $var(st_i)$ returns l_i. (Generalizing var so it returns a set of variables is equivalent but notationally awkward.)

The analysis yields an input partition for the augmented method m. A partition for the return values (normal and exceptional) of each remote-method call site is obtained by projecting the input partition for m onto the parameters introduced for that call site, by existential quantification over the other parameters.

If a remote method m invokes remote methods, the input partition for arguments of m is obtained by projecting the input partition for the augmented version of m onto the original parameters. The remote method invocations are treated as skipped call sites in PTE analysis, so the results of PTE analysis reflect the fact that arguments to remote methods escape from m.

Acknowledgments. I thank the reviewers for their helpful comments.

12. REFERENCES

[1] Christopher Colby, Patrice Godefroid, and Lalita Jagadeesan. Automatically closing open reactive programs. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 345–357, 1998.

[2] Patrice Godefroid. Model checking for programming languages using VeriSoft. In *Proc. 24th ACM Symposium on Principles of Programming Languages (POPL)*, pages 174–186. ACM Press, 1997.

[3] Patrice Godefroid, Robert S. Hanmer, and Lalita Jagadeesan. Model checking without a model: An analysis of the heart-beat monitor of a telephone switch using VeriSoft. In *Proc. ACM International Symposium on Software Testing and Analysis (ISSTA)*, pages 124–133. ACM Press, 1998.

[4] Gavin Lowe. Casper: A compiler for the analysis of security protocols. In *Proc. 10th IEEE Computer Security Foundations Workshop (CSFW)*, 1997.

[5] Dahlia Malkhi and Michael Reiter. Secure and scalable replication in Phalanx. In *Proc. 17th IEEE Symposium on Reliable Distributed Systems*, pages 51–60, 1998.

[6] Debra J. Richardson and Lori A. Clarke. Partition analysis: A method combining testing and verification. *IEEE Transactions on Software Engineering*, 11(12):1477–1490, December 1985.

[7] A.W. Roscoe and M.H. Goldsmith. The perfect "spy" for model-checking cryptoprotocols. In *Proc. DIMACS Workshop on Design and Formal Verification of Security Protocols*, September 1997.

[8] Scott D. Stoller. Domain partitioning for open reactive systems. Technical Report DAR-02-6, SUNY at Stony Brook, Computer Science Dept., February 2002. Available at www.cs.sunysb.edu/~stoller/partn.html.

[9] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 187–206, October 1999.

APPENDIX

A. METHOD ABSTRACTIONS FOR `SIGNATURE`

Figures 9 and 10 contain abstractions of some methods of `Signature`. Abstractions of the other methods are similar. *getAlg* and *getData* are selectors for $D_{\texttt{Signature}}$, like *getMode* in Section 7.2. Each exception class has an instance field with type `String` that holds a detail message. With the default cryptography provider in Sun JDK 1.3, the detail message does not contain significant information from the parameters, so we omit it below; *e.g.*, we write `InvalidKeyException`() instead of `InvalidKeyException("Could not initialize for signing with the given key.")`. For brevity, we omit package names; *e.g.*, we write `C.m(LString;)V` instead of `C.m(Ljava.lang.String;)V`. In many applications, `update` is called only once between calls to `initSign` or `initVerify`; we use a simple abstraction for `update` that suffices for such applications. In the abstraction for `verify`, we use strong update for n, because if the method invocation completes normally, then it definitely updates the object represented by n; similarly, an incorrect mode causes an exception, which is not represented by the normal return value, so the latter can be determined independently of the mode. `Signature.verify` resets the working data to empty.

B. ANALYSIS DETAILS FOR `PSI.CONTEND`

The control-flow graph of `contend`, with `SignedObject.verify` inlined, appears in Figure 11. The classes of all the objects can easily be determined statically, so dynamic dispatch code is not introduced.

To help the figure fit on the page, some `instanceof` checks and corresponding throws of `ClassCastException` (namely, from `(Vector)accessTag.get(y1)` and `ByteArrayEquals.equals`) are elided, as are store statements that involve l_{exc}. Some load statements (e.g., for `so.sig` and `n.y1`) and the invocation statement for `elementAt` are folded into other statements in the figure but are really separate statements. The exception handler in `SignedObject.verify` is not shown explicitly. The unique exit node appears in multiple places in the figure.

The PTE graph for `contend` at •`return so` appears in Figure 12. In Figure 11, some load statements are folded into other statements to save space (*e.g.*, a load of `so.obj` is folded into `if so.obj!=null`), so the target variable of the load does not appear in Figure 11. In Figure 12, such variables are named as follows: the i'th statement that loads `so.obj` is $\texttt{soobj}_i$ = `so.obj`; and so on. (The CFG for `contend` is acyclic, so this is unambiguous.) The identity of each node in the PTE graph is clear from the annotations and the variable pointing to it; for example, baos points to the allocation node corresponding to `new ByteArrayOutputStream`. So, instead of putting identifying information inside each node n (as in Figure 2), we put $\rho(n)$ (we elide curly braces for singleton sets), where ρ is the environment at •`return so`. The environment for variables with primitive type is[1]

$$
\begin{aligned}
&\rho(\texttt{verifies}) = \{\mathit{verify}(\text{PSI.sigAlg}, \text{PSI.pubKey(so.signer)},\\
&\qquad \text{toByteArray}(\mathit{writeObj}(\text{so.obj})), \text{so.sig}),\\
&\qquad \text{false}\}\\
&\rho(\texttt{aTcK}) = \mathit{containsKey}(\text{PSI.accessTag}, \text{so.obj.y1})\\
&\rho(\texttt{y2OK}) = \mathit{arraysEquals}(\text{so.obj.y2},\\
&\qquad \mathit{element}(\text{PSI.accessTag.get(so.obj.y1)}))
\end{aligned}
$$

[1] The fact that `verifies` cannot be false at this point is not reflected in the environment; it is reflected in the guards, described in Section 8.

$$
\begin{aligned}
&[\![\texttt{Signature.initVerify(LPublicKey;)V}]\!]_g(l, l_0, l_1, n, n^{exc}, \rho) =\\
&\ \text{let } \mathit{this}_p = \text{getVal}_g(\rho, l_0) \text{ in}\\
&\ \text{let } \mathit{key}_p = \text{getVal}_g(\rho, l_1) \text{ in}\\
&\ \text{if } \mathit{this}_p = \{\mathit{this}\} \text{ and } \mathit{key}_p = \{\mathit{key}\} \text{ for some } \mathit{this} \in D_{\texttt{Signature}}\\
&\ \text{and } \mathit{key} \in D_{\texttt{Key}} \text{ then}\\
&\quad \text{let } \mathit{exns} = \{\texttt{InvalidKeyException}\}\\
&\quad \text{and } \mathit{result} = \{\mathit{Signature}(\mathit{getAlg}(\mathit{this}), \text{verifying}, \mathit{key}, [\,])\} \text{ in}\\
&\quad \text{let } f_{\text{env}} = \{\langle \text{normal}, \text{setVal}_g(\rho, l_0, \mathit{result})\rangle\}\\
&\qquad\qquad \cup \textstyle\bigcup_{cl \in exns} \{\langle cl, \rho[n^{exc} \mapsto cl()]\rangle\}\\
&\quad \text{and } f_{\text{guard}} =\\
&\qquad \{\langle \text{normal}, \mathit{key} \neq \text{null} \wedge \mathit{compatible}(\mathit{key}.\text{getAlgorithm}(),\\
&\qquad\qquad \mathit{getAlg}(\mathit{this}))\rangle,\\
&\qquad \langle \texttt{InvalidKeyException},\\
&\qquad\quad \mathit{key} = \text{null} \vee \neg \mathit{compatible}(\mathit{key}.\text{getAlgorithm}(),\\
&\qquad\qquad \mathit{getAlg}(\mathit{this}))\rangle\} \text{ in}\\
&\quad \langle f_{\text{env}}, f_{\text{guard}}\rangle\\
&\ \text{else } \langle (\lambda\, \mathit{exc}.\top), (\lambda\, \mathit{exc}.\text{true})\rangle
\end{aligned}
$$

$$
\begin{aligned}
&[\![\texttt{Signature.getInstance(LString;)LSignature;}]\!]_g(l, l_1, n, n^{exc}, \rho) =\\
&\ \text{let } \mathit{alg}_p = \text{getVal}_g(\rho, l_1) \text{ in}\\
&\ \text{if } \mathit{alg}_p = \{\mathit{alg}\} \text{ for some } \mathit{alg} \in D_{\texttt{String}} \text{ then}\\
&\quad \text{let } \mathit{exns} = \{\texttt{NoSuchAlgorithmException}\}\\
&\quad \text{and } \mathit{result} = \{\mathit{Signature}(\mathit{alg}, \text{uninit}, \text{null}, \text{null})\} \text{ in}\\
&\quad \text{let } f_{\text{env}} = \{\langle \text{normal}, \text{setVal}_g(\rho, l, \mathit{result})\rangle\}\\
&\qquad\qquad \cup \textstyle\bigcup_{cl \in exns} \{\langle cl, \rho[n^{exc} \mapsto cl()]\rangle\}\\
&\quad \text{and } f_{\text{guard}} = \{\langle \text{normal}, \mathit{alg} \neq \text{null} \wedge \mathit{availableSigAlg}(\mathit{alg})\rangle,\\
&\qquad\qquad \langle \texttt{NoSuchAlgorithmException},\\
&\qquad\qquad\ \mathit{alg} = \text{null} \vee \neg \mathit{availableSigAlg}(\mathit{alg})\rangle\} \text{ in}\\
&\quad \langle f_{\text{env}}, f_{\text{guard}}\rangle\\
&\ \text{else } \langle (\lambda\, \mathit{exc}.\top), (\lambda\, \mathit{exc}.\text{true})\rangle
\end{aligned}
$$

$$
\begin{aligned}
&[\![\texttt{Signature.update([B)V}]\!]_g(l, l_0, l_1, n, n^{exc}, \rho) =\\
&\ \text{let } \mathit{this}_p = \text{getVal}_g(\rho, l_0)\\
&\ \text{and } \mathit{data}_p = \text{getVal}_g(\rho, l_1) \text{ in}\\
&\ \text{if } \mathit{this}_p = \{\mathit{this}\} \text{ and } \mathit{data}_p = \{\mathit{data}\} \text{ and } \mathit{getData}(\mathit{this}) = [\,]\\
&\ \text{for some } \mathit{this} \in D_{\texttt{Signature}} \text{ and } \mathit{data} \in D_{\texttt{byte[]}} \text{ then}\\
&\quad \text{let } \mathit{exns} = \{\texttt{SignatureException}\}\\
&\quad \text{and } \langle \mathit{alg}, \mathit{mode}, \mathit{key}, [\,]\rangle =\\
&\qquad \langle \mathit{getAlg}(\mathit{this}), \mathit{getMode}(\mathit{this}),\\
&\qquad\ \mathit{getKey}(\mathit{this}), \mathit{getData}(\mathit{this})\rangle \text{ in}\\
&\quad \text{and } \mathit{result} = \{\mathit{Signature}(\mathit{alg}, \mathit{mode}, \mathit{key}, \mathit{data})\} \text{ in}\\
&\quad \text{let } f_{\text{env}} = \{\langle \text{normal}, \text{setVal}_g(\rho, l_0, \mathit{result})\rangle\}\\
&\qquad\qquad \cup \textstyle\bigcup_{cl \in exns} \{\langle cl, \rho[n^{exc} \mapsto cl()]\rangle\}\\
&\quad \text{and } f_{\text{guard}} = \{\langle \text{normal}, \mathit{getMode}(\mathit{this}) \neq \text{uninit}\rangle,\\
&\qquad\qquad \langle \texttt{SignatureException},\\
&\qquad\qquad\ \mathit{getMode}(\mathit{this}) = \text{uninit}\rangle\} \text{ in}\\
&\quad \langle f_{\text{env}}, f_{\text{guard}}\rangle\\
&\ \text{else } \langle (\lambda\, \mathit{exc}.\top), (\lambda\, \mathit{exc}.\text{true})\rangle
\end{aligned}
$$

Figure 9: Abstractions of some methods of `Signature`.

$$
\begin{array}{l}
[\![\texttt{Signature.sign}()[\mathrm{B}]]\!]_g(l, l_0, n, n^{exc}, \rho) = \\
\quad \text{let } \mathit{this}_p = \mathrm{getVal}_g(\rho, l_0) \text{ in} \\
\quad \text{if } \mathit{this}_p = \{\mathit{this}\} \text{ for some } \mathit{this} \in D_{\texttt{Signature}} \text{ then} \\
\qquad \text{let } \mathit{exns} = \{\texttt{SignatureException}\} \\
\qquad \text{and } \langle \mathit{alg}, \mathit{mode}, \mathit{key}, \mathit{data} \rangle = \\
\qquad\quad \langle \mathit{getAlg}(\mathit{this}), \mathit{getMode}(\mathit{this}), \\
\qquad\quad\ \mathit{getKey}(\mathit{this}), \mathit{getData}(\mathit{this}) \rangle \text{ in} \\
\qquad \text{let } \mathit{result} = \mathit{sign}(\mathit{alg}, \mathit{mode}, \mathit{key}, \mathit{data}) \\
\qquad \text{and } \mathit{this}' = \{\mathit{Signature}(\mathit{alg}, \mathit{mode}, \mathit{key}, [\,])\} \text{ in} \\
\qquad \text{let } f_{\mathrm{env}} = \{\langle \mathrm{normal}, \mathrm{setVal}_g(\rho, l_0, \mathit{this}')[n \mapsto \mathit{result}] \rangle\} \\
\qquad\qquad\qquad \cup \bigcup_{cl \in \mathit{exns}} \{\langle cl, \rho[n^{exc} \mapsto cl()] \rangle\} \\
\qquad \text{and } f_{\mathrm{guard}} = \{\langle \mathrm{normal}, \mathit{getMode}(\mathit{this}) = \mathrm{signing} \rangle, \\
\qquad\qquad\qquad \langle \texttt{SignatureException}, \\
\qquad\qquad\qquad\ \mathit{getMode}(\mathit{this}) \neq \mathrm{signing} \rangle\} \text{ in} \\
\qquad \langle f_{\mathrm{env}}, f_{\mathrm{guard}} \rangle \\
\quad \text{else } \langle (\lambda\, \mathit{exc}.\top), (\lambda\, \mathit{exc}.\mathrm{true}) \rangle
\end{array}
$$

Figure 10: Abstractions of some methods of `Signature`, continued.

The return set r of the PTE graph contains $n_{\texttt{so}}$, all load nodes reachable from $n_{\texttt{so}}$, all allocation nodes for `VotingExc` and `NullPointerException`, and the exc-return nodes for the invocation statements that call `PSI.pubKey` and `v.elementAt` (these two nodes represent `ArrayIndexOutOfBoundsException`). Regarding the escape function e, $e(n)$ is non-empty for all nodes except the exc-return nodes and the allocation nodes for `ObjectOutputStream`, `VotingExc`, and `NullPointerException`.

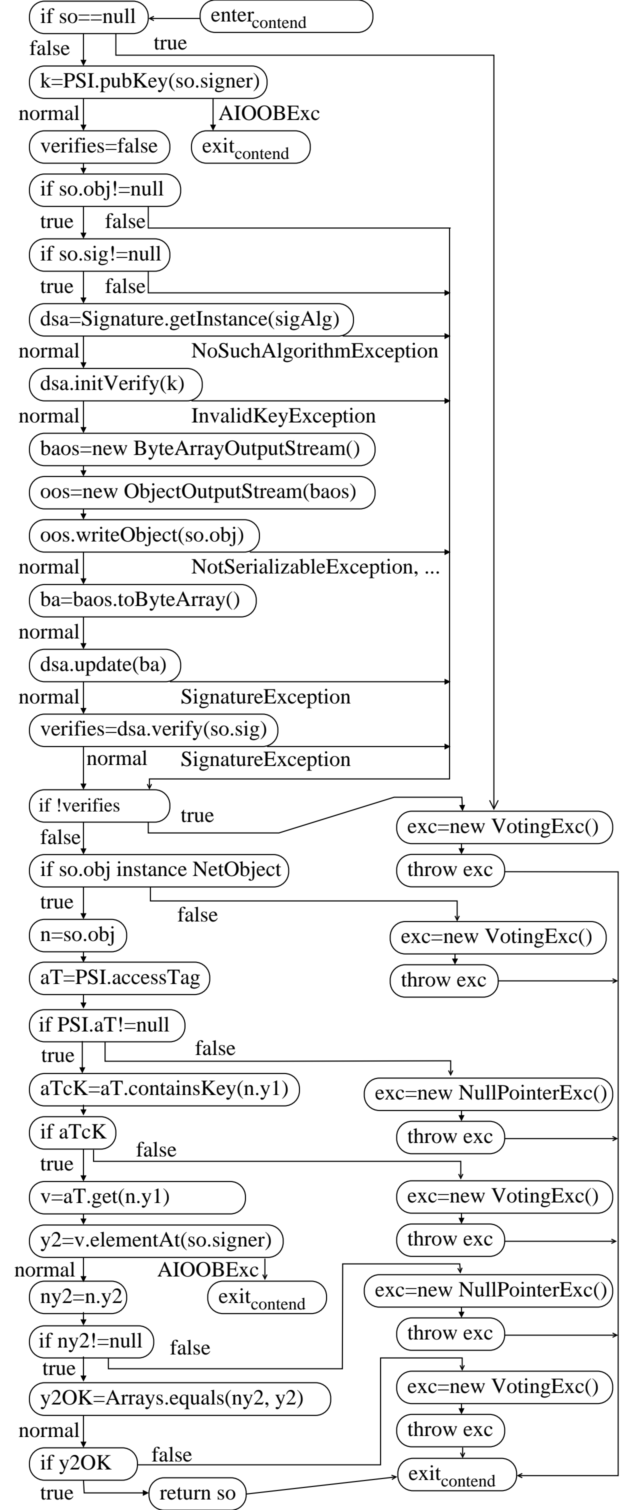

Figure 11: Control-flow graph for `contend`. AIOOBExc abbreviates ArrayIndexOutOfBoundsException.

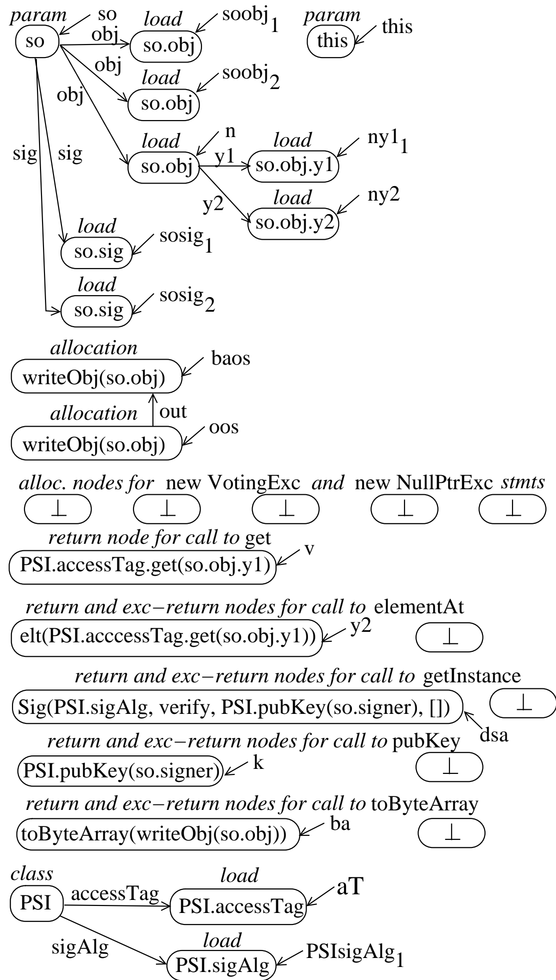

Figure 12: PTE graph for `contend` at •`return so`.