

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 environ-

*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.

ment 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-

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 $\langle \text{state}, \text{args} \rangle$, let $\text{partnForState}(\pi, s)$ 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 $\text{partnForState}(\pi, s)$ 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.

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.

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 storage. *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.

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.

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.

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 $Local \cup Param$, 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). 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 $enter_m$ and ends with an exit node $exit_m$. We assume the program has been pre-processed so that all statements relevant to the analysis are in one of the following forms:

- constant: $l = c$
- array store: $l_2[l_3] = l_1$
- copy: $l = v$
- array load: $l_1 = l_2[l_3]$
- store: $l_2.f = l_1$
- object creation: $l = \mathbf{new} \ c$
- load: $l_1 = l_2.f$
- return: $\mathbf{return} \ l_1$
- 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: $\mathbf{throw} \ l_1$
- branch: $\mathbf{if} \ l_1 \ rel \ l_2$ or $\mathbf{if} \ l_1 \ rel \ c$
 rel is an equality or inequality test, or $\mathbf{instanceof}$.
- 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 $\mathbf{instanceof}$

conditions. For $st \in Stmt$, $\text{pred}(st)$ and $\text{succ}(st)$ 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.

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 \mathbf{exit}_m if st might throw an exception not caught within m . For each such edge $\langle st, st' \rangle$, $\text{excns}(st, st')$ 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 $\text{excns}(st, st') = \{\text{normal}\}$.

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 .

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. $\text{excns}(st, st')$ 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.

4. CORRECTNESS REQUIREMENT

This section describes the correctness requirements for the analysis. For $\tau \in Class \cup PrimTy$, let the concrete domain Con_τ be the set of all possible values of type τ . For $\tau \in Class$, 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 $Bind(S)$ of *bindings* for S contains partial functions b from $S \cup Ref$ to Con such that $\text{domain}(b) = S \cup \text{refs}(b)$, where $\text{refs}(b)$ is the set of references that appear in b . The universe of concrete values is $Con = \bigcup_{\tau \in Class \cup PrimTy} Con_\tau$.

```

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;
}}

```

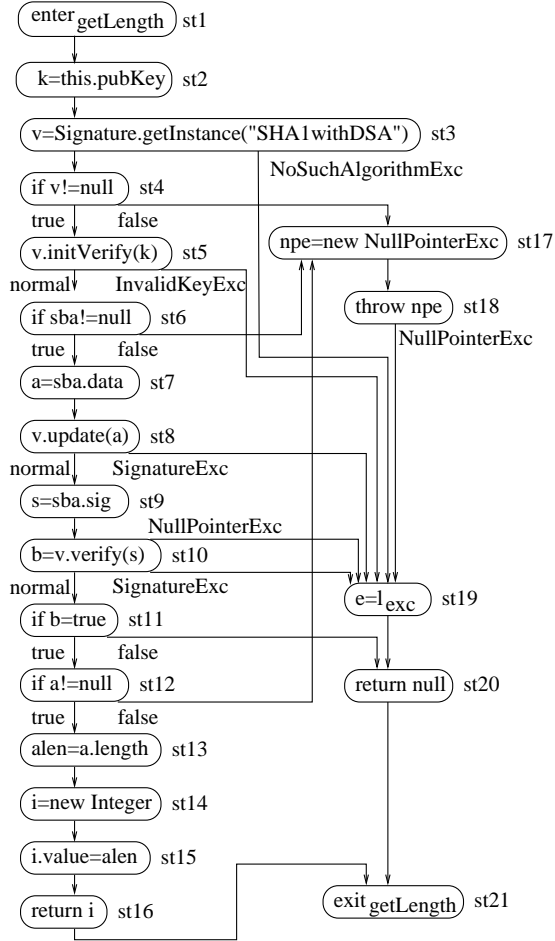


Figure 1: Code and CFG for getLength.

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 $R \subset Ref$, *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 $RetVal$ denote possible return values of m ; these are elements of Con tagged to indicate whether the method terminated normally or abruptly. For $args \in Bind(Param)$ and $r \in RetVal$, define the transition relation \xrightarrow{m} by: $args \xrightarrow{m} r$ iff execution of $m(args)$ can terminate with result r .

An *input partition* π for m is a partition of $Bind(Param)$ (*i.e.*, each equivalence class in π is a set of parameter bindings) such that: for all equivalence classes $\phi \in \pi$, for all $args_1, args_2 \in \phi$, for all $r_1, r_2 \in RetVal$, if $args_1 \xrightarrow{m} r_1$ and $args_2 \xrightarrow{m} r_2$, then r_1 and r_2 are structurally equal exact in $refs(args_1) \cup refs(args_2)$. We do not consider static variables in this paper, so a quiescent state is described just by the return value of the previous method invocation. With static variables, this requirement on return values is conjoined with a similar requirement on the updated global storage [8].

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.

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 (we ignore static

fields and hence class nodes). An object may be represented by multiple nodes.

Example. A PTE graph for `getLength` contains (among other nodes) an allocation node n_{st14} and a load node n_{st2} .

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 satisfies: $e(n)$ is non-empty if objects represented by n might escape from the method invocation in some way other than through the return value, specifically, through a parameter, static field, `Runnable` object, or invocation statement.

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_{st}^{exc} , 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]. For details, see [8].

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_{st2}) corresponds to the load statement in node 2 of the CFG

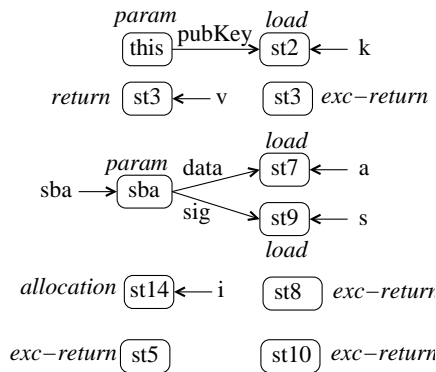


Figure 2: PTE graph for `C.getLength` at `return i`. Italic labels indicate the kind of each node. The return set is $r = \{n_{st14}\}$. $e(n)$ is non-empty for nodes n_{this} , n_{sba} , n_{st2} , n_{st3} , n_{st5} , and n_{st7} .

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 is *functional* if there is no invocation of it that updates storage locations that escape from the invocation.

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. We also assume that information does not escape through methods in M_{cust} . This assumption could easily be eliminated.

$Class_{cust}$ 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.

$Param_{rdOnly}$ is a set of read-only parameters. A parameter p of method m is *read-only* if m never updates objects reachable from p . The analysis tracks values retrieved from read-only parameters but (for simplicity) not from other parameters. 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. As mentioned in Section 2, static fields are treated in [8] but not in this paper.

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. Specifically, these expressions are built recursively from read-only parameters 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 $R\text{trvd}_{cl}$, 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. Special abstract values \perp and \top are included to represent “no information” and “all information” (or “arbitrary information”), respectively. Let τ_1, \dots, τ_n denote the types of the instance fields of cl , and let $cl(x_1, \dots, x_n)$ denote a record (tuple) with the indicated fields (components). The default abstract domain for cl is

$$D_{cl} = \{cl(d_1, \dots, d_n) \mid d_1 \in D_{\tau_1}, \dots, d_n \in D_{\tau_n}\} \cup R\text{trvd}_{cl} \cup \{\text{null}, \perp, \top\}.$$

$$x \sqcup_{\tau} y = \begin{cases} y & \text{if } x = \perp \text{ or } x = y \\ x & \text{if } y = \perp \\ cl(z_1, \dots, z_n) & \text{if } x = cl(x_1, \dots, x_n) \text{ and } \\ & y = cl(y_1, \dots, y_n) \text{ and } \\ & z_i = x_i \sqcup_{\tau_i} y_i \\ \top & \text{otherwise.} \end{cases}$$

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 . When applied to an element of $R\text{trvd}_{cl}$, 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 .

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 $D_T = \text{Con}_T \cup R\text{trvd}_T$, and the join operation returns \top whenever its operands are not equal and are both not \perp . One could easily parameterize the analysis with respect to abstractions for operations on primitive types. For simplicity, very coarse abstractions of these operations are hardwired 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 $D_{\tau} \supseteq R\text{trvd}_{\tau}$.

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. We combine the type-specific domains D_{τ} into an overall domain D . In [8], we apply a powerset construction to D_{τ} before forming D ; this can improve the accuracy of the analysis. For brevity, the powerset construction is omitted from this paper, so $D = \bigcup_{\tau \in \text{Class} \cup \text{PrimTy}} D_{\tau}$.

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`.

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. Thus, custom abstractions are also used for these classes. The custom abstract domains defined below use the following abstract values. For $x \in D_{\text{ByteArrayOutputStream}}$, `toByteArray(x)` represents the return value of `x.toByteArray()`. `writeObj(o)` represents the data appended to an `ObjectOutputStream` by `writeObject(o)`. `sign(alg, key, data)` represents the return value of an invocation of `Signature.sign()` on an instance of `Signature` with signature algorithm `alg`, 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 .

$$D_{\text{ObjectOutputStream}} = \{\text{writeObj}(x) \mid x \in R\text{trvd}_{\text{Object}}\}^* \cup R\text{trvd}_{\text{ObjectOutputStream}}$$

$$D_{\text{ByteArrayOutputStream}} = D_{\text{ObjectOutputStream}} \cup R\text{trvd}_{\text{ByteArrayOutputStream}}$$

$$D_{\text{byte}[]} = \cup \{\text{toByteArray}(x) \mid x \in D_{\text{ByteArrayOutputStream}}\} \cup \{\text{sign}(alg, key, data) \mid alg \in R\text{trvd}_{\text{String}} \wedge key \in D_{\text{PublicKey}} \wedge data \in D_{\text{byte}[]}\} \cup R\text{trvd}_{\text{byte}[]}$$

$$D_{\text{Signature}} = \{\text{Signature}(alg, mode, key, data) \mid alg \in R\text{trvd}_{\text{String}} \wedge mode \in \{\text{uninit}, \text{signing}, \text{verifying}\} \wedge key \in D_{\text{Key}} \wedge data \in D_{\text{byte}[]}\} \cup R\text{trvd}_{\text{signature}}$$

This definition of $D_{\text{ByteArrayOutputStream}}$ suffices for applications in which every `ByteArrayOutputStream` is fed from an `ObjectOutputStream`. We use the default abstract domain for `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, D_{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 ; $availableSigAlg(alg)$ 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 $verify(\dots)$ 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 $D_{boolean}$ 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 $Env = (Lcl_{prim} \cup Param_{prim} \cup N) \rightarrow D$. Let ρ range over Env .

Example. A sampling of values in 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.

$$\begin{aligned} \rho(n_{this}) &= this \\ \rho(n_{sba}) &= sba \\ \rho(n_{st2}) &= this.pubKey \\ \rho(n_{st3}) &= Signature("SHA1withDSA", verifying, \\ &\quad this.pubKey, []) \\ \rho(n_{st5}) &= sba.data \\ \rho(n_{st14}) &= Integer(sba.data.length) \\ \rho(b) &= verify("SHA1withDSA", this.pubKey, \\ &\quad sba.data, sba.sig) \end{aligned}$$

In $\rho(n_{st3})$, 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 $\rho(n) = \perp$ 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* $\llbracket m \rrbracket$ 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* $\llbracket st \rrbracket$ 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 $\llbracket m \rrbracket$ is supplied. Consider an invocation statement st of the form $l = l_0.m(l_1, \dots, l_k)$. 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 $\llbracket m \rrbracket_{\alpha(\bullet st)}(l, l_0, l_1, \dots, l_k, n_{st}, n_{st}^{exc}, \rho)$. Recall that α is the result of PTE analysis; the PTE graph $\alpha(\bullet st)$ is an argument to $\llbracket m \rrbracket$. 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 $\llbracket m \rrbracket$, respectively, is a dummy value.

The call to $\llbracket m \rrbracket$ returns a pair of functions $\langle f_{env}, f_{guard} \rangle$, both with domain $thrownExc(m)$. $f_{env}(e)$ is the environment obtained by updating ρ to reflect the effect of execution of m when m terminates in the manner described by e . $f_{guard}(e)$ is (an over-approximation of) the pre-condition for m to terminate in the manner described by e . $f_{guard}(e)$ is in $D_{boolean}$. If m returns a reference or primitive value, the return value is reflected in $f_{env}(\text{normal})$ by an updated binding for n_{st} or l , respectively. For $e \neq \text{normal}$, $f_{env}(e)$ contains an updated binding for n_{st}^{exc} . 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 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. $getVal_g(\rho, v)$ 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:

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

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

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

$setVal_g(\rho, l, d)$ 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 $singular_g(l)$. In this case, $setVal_g(\rho, l, d)$ 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

```

[[Signature.verify]([B]Z)]g(l, l0, l1, n, nexc, ρ) =
  let this = getValg(ρ, l0)
  and sig = getVal(ρ, l1) in
  if this = Signature(alg, mode, key, data)
  for some alg, mode, key, data then
    let exns = {SignatureException, NullPointerException.}
    and result = verify(alg, key, data, sig)
    and this' = {Signature(alg, mode, key, [])} in
    let fenv = {⟨normal, setValg(ρ, l0, this')[l ↦ result]⟩}
      ∪ ∪cl ∈ exns {⟨cl, ρ[nexc ↦ cl()]⟩}
    and fguard =
      {⟨normal, getMode(this) = verifying ∧ sig ≠ null⟩,
       ⟨SignatureException, getMode(this) ≠ verifying⟩,
       ⟨NullPointerException, getMode(this) = verifying
        ∧ sig = null⟩} in
    ⟨fenv, fguard⟩
  else ⟨(λ exc. ⊤), (λ exc. true)⟩

```

Figure 3: Abstraction for Signature.verify.

location will actually be updated by the assignment. This is called *weak update*. If $\text{type}(l) \in \text{PrimTy}$, strong update is always used.

```

setValg(ρ, l, d) = if type(l) ∈ Class then
  if singularg(l) then ρ[∀n ∈ Ig(l) : n ↦ d]
  else ρ[∀n ∈ Ig(l) : n ↦ ρ(n) ⊔D d]
  else ρ[l ↦ d]

```

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 [8].

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[]`. `getMode` is a selector for $D_{\text{Signature}}$, defined by: `getMode(Signature(alg, mode, key, data)) = mode`. If `getMode(this) ≠ 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. If `getMode(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 `verify(alg, key, data, sig)`, which is described in Section 7.1.

7.2.2 Transfer Functions

For a statement st , the transfer function $\llbracket st \rrbracket$ characterizes the effect of execution of st on the environment (a more

```

[[l = c]](ρ) = if type(l) ∈ Class then ρ else ρ[l ↦ {c}]
[[l = v]](ρ) = if type(l) ∈ Class then ρ else ρ[l ↦ ρ(v)]
[[stload]](ρ) =
  let g = α(•stload) in
  if type(l1) ∈ Class then
    let SE = {n ∈ Ig(l2) | eg(n) ≠ ∅} in
    ρ[nstload ↦ ⊔D {load(ρ(n), f) | n ∈ SE}]
  else ρ[l1 ↦ ⊔D {load(ρ(n), f) | n ∈ Ig(l2)}]
[[ststore]](ρ) =
  let τ = type(l2)
  and g = α(•ststore)
  and d = getValg(ρ, l1) in
  if singularg(l2) ∧ singularFld(τ, f) then
    ρ[(∀n ∈ Ig(l2) : n ↦ storeτ(ρ(n), f, d))]
  else ρ[(∀n ∈ Ig(l2) : n ↦ storeτ(ρ(n), f,
    d ⊔D loadτ(ρ(n), f)))]
[[l1 = l2[l3]]](ρ) = [[l1 = l2.elements]](ρ)
[[l2[l3] = l1]](ρ) = [[l2.elements = l1]](ρ)
[[stid]](ρ) = ρ
[[stcall]](ρ) =
  let ⟨fenv, -⟩ = [[m]]α(•stcall)(l, l0, l1, ..., lk, nstcall, nstcallexc)
  in fenv
[[l := l1 op l2]](ρ) = ρ[l ↦ ⊤]

```

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 $l = l_0.m(l_1, \dots, l_k)$.

formal requirement appears in [8]). $Stmt_{\text{invoc}}$ is the set of invocation statements. If $st \notin Stmt_{\text{invoc}}$, then $\llbracket st \rrbracket(\rho)$ is the environment resulting from execution of st in environment ρ . If $st \in Stmt_{\text{invoc}}$, then for $x \in \text{thrownExc}(st)$, $\llbracket st \rrbracket(\rho)(e)$ 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 $\llbracket st \rrbracket(\rho)(e)$ 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 .

The transfer functions in Figure 4 are reasonably straightforward, except perhaps for load and store statements. Consider a load statement st_{load} . If $\text{type}(l_1) \in \text{PrimTy}$, the transfer function for st_{load} updates the environment in a straightforward way. If $\text{type}(l_1) \in \text{Class}$, the transfer function for st_{load} updates $\rho(n_{st_{\text{load}}})$ 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 $\rho(n_{st_{\text{load}}})$ 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 `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 $\llbracket m \rrbracket$ is ignored in $\llbracket st_{\text{call}} \rrbracket$. Currently, branch conditions and guards are used only in Step 3. They could also be used to increase the accuracy of data-flow analysis.

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 $\text{thrownExc}(st) \rightarrow \text{Env}$. 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 ρ_0 is: $\rho_0(x) = \{x\}$ if $x \in \text{Param}_{\text{prim}}$ or x is a parameter node, and $\rho_0(x) = \perp$ otherwise.

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 $\sqcup_{\text{Env}} \emptyset = \perp$. \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. With the assumptions in Section 6, it is easy to characterize N_{esc} using the escape function e at $\text{exit}_{m\bullet}$: N_{esc} contains nodes that escape through non-read-only parameters or **Runnable** objects [8]. 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.

$$\begin{aligned} \beta(\text{enter}_{m\bullet}) &= \rho_0 \\ \beta(\bullet st) &= (\sqcup_{\text{Env}} \{\beta(st'\bullet) \mid st' \in \text{pred}(st) \setminus \text{Stmt}_{\text{thrower}}\}) \\ &\quad \sqcup_{\text{Env}} (\sqcup_{\text{Env}} \{\beta(st'\bullet)(e) \mid st' \in \text{pred}(st) \cap \text{Stmt}_{\text{thrower}} \\ &\quad \quad \quad \wedge e \in \text{excns}(st', st)\}) \\ \beta(st\bullet) &\geq_{\text{Env}} \llbracket st \rrbracket(\beta(\bullet st)) \quad \text{for } st \neq \text{enter}_m \\ \beta(st\bullet)(n) &= \top \quad \text{for } st \in \text{Stmt} \text{ and } n \in N_{\text{esc}} \end{aligned}$$

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 $st \in \text{Stmt}_{\text{esc}}$, an abstract value $\text{esc}(st)$ representing information that possibly escapes at st ; and (3) for each $st \in \text{Stmt}$ and $st' \in \text{succ}(st)$, a predicate $\text{guard}(st, st')$ 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 $I_{\alpha(\bullet st)}(l_2) \cap N_{\text{esc}} \neq \emptyset$, where N_{esc} is defined in the last paragraph of Section 7. For $st \in \text{Stmt}_{\text{esc}}$, $\text{var}(st)$ denotes the variable denoted by l_1 in Section 3. For a statement st in Stmt_{esc} , $\text{esc}(st) = \text{getVal}_{\alpha(\bullet st)}(\beta(\bullet st), \text{var}(st))$. For statements other than branches and method invocations, $\text{guard}(st, st')$ is the constant predicate “true”. If st is an invocation statement and st' is a successor of st , then $\text{guard}(st, st')$ is determined by the conditions in f_{guard} (returned by the method abstraction) and $\text{excns}(st, st')$ in a straightforward way. If st is a branch statement, then the predicate is constructed in a straightforward way from st and $\beta(\bullet st)$.

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 enter_m to exit_m . For $\sigma \in \text{Paths}$, let $\text{guard}(\sigma)$ be the conjunction of the guards of the edges in σ . 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 $\sigma \in \text{Paths}$, let st_1, \dots, st_n be the sequence of statements st in Stmt_{esc} that appear on σ and satisfy $\text{esc}(st) \notin \{\perp, \top\}$. The information that escapes along σ is represented abstractly (symbolically) by $\text{esc}(\sigma) = \langle \text{esc}(st_1), \dots, \text{esc}(st_n) \rangle$. The set of distinct concrete data structures that might escape along σ is

$$\begin{aligned} \text{escStruct}(\sigma) &= \text{struct}(\text{type}(\text{var}(st_1))) \times \dots \\ &\quad \times \text{struct}(\text{type}(\text{var}(st_n))) \end{aligned}$$

where $\text{struct}(\tau)$ is Con_τ quotiented by structural equality; in other words, $\text{struct}(\tau)$ 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 $\text{esc}(\sigma) = \text{esc}(\sigma')$. For $\hat{\sigma} \in \text{PPaths}$, let $\text{guard}(\hat{\sigma}) = \bigvee_{\sigma \in \hat{\sigma}} \text{guard}(\sigma)$ and $\text{escStruct}(\hat{\sigma}) = \bigcup_{\sigma \in \hat{\sigma}} \text{escStruct}(\sigma)$.

If there exists $st \in \text{Stmt}_{\text{esc}}$ such that $\text{esc}(st) = \top$, 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 $\text{guard}(\hat{\sigma})$ can lead to output s along $\hat{\sigma}$.

$$\begin{aligned} \text{partn}(m) &= \bigcup_{\substack{\hat{\sigma} \in \text{PPaths} \\ s \in \text{escStruct}(\hat{\sigma})}} \{\{param \mid \text{esc}(\hat{\sigma}) \in s \wedge \text{guard}(\hat{\sigma})\}\} \end{aligned}$$

where $param$ is a tuple containing the parameters of m . Occurrences of parameters in $\text{esc}(\hat{\sigma})$ and $\text{guard}(\hat{\sigma})$ are captured (bound) by $param$ 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 $struct(cl)$ with unions over the values of those fields. If $x \in esc(\hat{\sigma})$ is a constant (e.g., null), we simplify $\bigcup_{s \in escStruct(\hat{\sigma})} \{\{param \mid \dots\}\}$ to $\{\{param \mid guard(\hat{\sigma})\}\}$.

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

$$escStruct(\sigma) = \{\{o \in Con_{Integer} \mid o.value = val\} \mid val \in int\},$$

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

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

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

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

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 (1) below. Equation (1) 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.

$$\begin{aligned} partn(C.getLength) = & \\ & \{\{(this, sba) \mid \neg normalGetLength\}\} \\ & \cup \bigcup_{i \in int} \{\{(this, sba) \mid sba.data.length = i \\ & \quad \wedge normalGetLength\}\} \end{aligned} \quad (1)$$

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 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} .

Analysis results. The analysis result for `contend` appears in [8]. The partition contains: an equivalence class containing all inputs that cause `contend` to throw `VotingExc`; an equivalence class containing all inputs that cause `contend` to throw `VotingExc`; a family of equivalence classes, each corresponding to an input that causes `contend` to return its argument `so`. For any fixed values of read-only global storage, this family is finite, although the form in which it is initially expressed (namely, sets defined by characteristic predicates) 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. For example, following the general pattern described in Section 7.1, the abstraction for `Hashtable.containsKey` introduces predicates of the form $containsKey(h, k)$, which holds if hashtable h contains an entry with key k . Such a predicate corresponds naturally to a union over the set of keys in h . Analysis of an expression like `x.equals(v.elementAt(i))` (cf. `y20K` in Figure 5) introduces a characteristic predicate of the form $equals(x, element(v))$, where $element(v)$ represents the set of possible elements of vector v ; this predicate corresponds naturally to a union over the elements of v . Introduction of unions is also a step towards generation of code for the environment; the next step is to translate unions into iterations (loops).

Input partition analysis of all the remote methods produces about 25 equivalence classes or families of equivalence classes. Some of these are for arguments of remote methods. Others are for return values of remote methods; this is necessary because the return values are also intercepted and fabricated by the adversary. Partition analysis for return values can be reduced to partition analysis for arguments [8].

```

// 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.

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 $n - 1$; the scheduler explores all of these values. This feature is used to non-deterministically select which remote method to invoke next and which equivalence class to use for the arguments or return value of an RMI.

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 and using 28 MB RAM.

Acknowledgments. I thank the reviewers for their helpful comments.

10. 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. Hammer, 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.