# Domain Partitioning for Open Reactive Systems

Scott D. Stoller

Computer Science Department

State University of New York at Stony Brook

http://www.cs.sunysb.edu/~stoller/

# The Problem

Consider open reactive system with typed method-call interface.

Program for environment is often unavailable or unsuitable for model-checking (state-space exploration) or thorough testing.
**Goal:** Generate a suitable program that models the environment.

Many inputs are **equivalent**, that is, lead to same output (system state and return value).
**Examples:** secure distributed voting system + insecure network, getLen.

For efficient explicit-state model-checking:

- Use static analysis to partition inputs into **equivalence classes**.
- Generate model of environment that uses **one representative** of each equivalence class.

# Running Example: getLen

```
class SD { byte[] data; byte[] sig; }   // Signed Data

Integer getLen(SD sd, PublicKey k) {
  if (sd.sig is a valid signature of sd.data
      with respect to k)
   return new Integer(sd.data.length);
  else return null;
 }
```

**Analysis result for** `getLen`: $\{EC_{\text{err}}, EC_0, EC_1, \ldots\}$

$$
\begin{aligned}
EC_{\text{err}} &= \{\langle \mathtt{sd}, \mathtt{k}\rangle \mid \mathtt{sd} = \text{null} \ \vee\ \mathtt{k} = null \\
&\qquad\qquad\quad \vee\ \mathtt{sd} \text{ is not correctly signed WRT } \mathtt{k}\} \\
EC_i &= \{\langle \mathtt{sd}, \mathtt{k}\rangle \mid \mathtt{sd} \neq \text{null} \ \wedge\ \mathtt{k} \neq \text{null} \\
&\qquad\qquad\quad \wedge\ \mathtt{sd} \text{ is correctly signed WRT } \mathtt{k} \\
&\qquad\qquad\quad \wedge\ \text{sd.data.length} = i\}
\end{aligned}
$$

# Analysis Method: Three Steps

1. Use **points-to escape (PTE) analysis** [Whaley & Rinard 1999] to analyze flow of references (storage locations).

2. Use **data-flow analysis** to analyze flow of values.
The abstract domains and transfer functions typically embody symbolic evaluation.

3. Construct **equivalence classes** based on what information about inputs is revealed by the return value and updates to global storage

**Exceptions and static fields (global storage):**
handled in the paper; usually ignored in this talk.

# Step 1: Points-to Escape (PTE) Analysis

**Program representation:** like Java bytecode, with variables instead of operand stack.

**Analysis result:** a PTE graph $\langle Nodes, Edges, esc \rangle$ at each program point.

**node:** represents set of objects

**edge**: represents possible references

$esc(n)$: set of ways by which objects represented by node $n$ may escape from method $m$:
- return value,
- global storage,
- parameters of $m$,
- arguments of methods called by $m$

# Step 1 (PTE Analysis): Some Kinds of Nodes

There is one kind of node for each way a program can obtain references.

The **allocation node** $n_{st}$ for a `new` statement $st$ represents objects allocated at $st$.

The **parameter node** $n_p$ for a reference parameter $p$ represents the object bound to $p$.

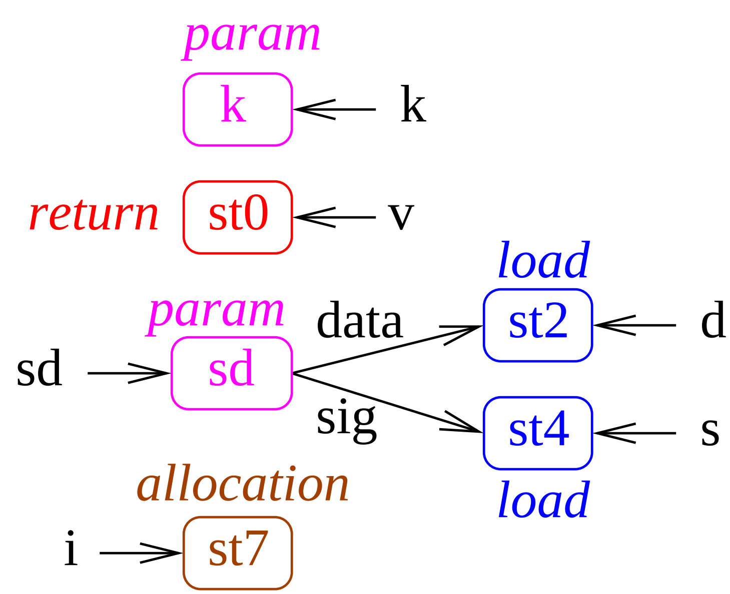
The **load node** $n_{st}$ for a load statement $st : l_1 = l_2.f$ represents objects that $l_2.f$ might point to.

The **return node** $n_{st}$ for a method invocation statement $st$ represents objects returned by invocations at $st$.

# Step 1 (PTE Analysis): Example

```
class SD { byte[] data; byte[] sig; }

 Integer getLen(SD sd,
                PublicKey k) {
0  Sig v = Sig.getInstance();
1  v.initVerify(k);
2  byte[] d = sd.data;
3  v.update(d);
4  byte[] s = sd.sig;
5  boolean b = v.verify(s);
6  if (b) {
7   i = new Integer(d.length);
8  •return i; }
9  else return null;
 }
```




$esc(n_{\text{st7}})$: return value

$esc(n_{\text{st4}})$: param sd, call st5

# Step 2 (Data-Flow Analysis): Domains

There is an abstract domain for each class and primitive type.

**Default domain** for class $cl$ is the union of:

- expressions representing values of type $cl$ retrieved from read-only inputs by field accesses (*e.g.*, sd.data for $cl = \texttt{byte}[]$) and functional methods (*e.g.*, k.getAlgorithm() for $cl = \texttt{String}$).
- the cross-product of the domains for the fields of $cl$.

**Custom domains** may be supplied for selected classes and types. They typically embody symbolic evaluation.

**Example:** Custom abstractions related to `Signature`.

$sign(key, data)$ represents return val of **`sign`**,

$verify(key, data, sig)$ represents return val of **`verify`**, *etc*.

# Step 2 (Data-Flow Analysis): Algorithm

**Valuation**: a function from (1) nodes in the PTE graph and (2) variables with primitive types to abstract values.

**Analysis result:** a valuation $\rho$ at each program point.

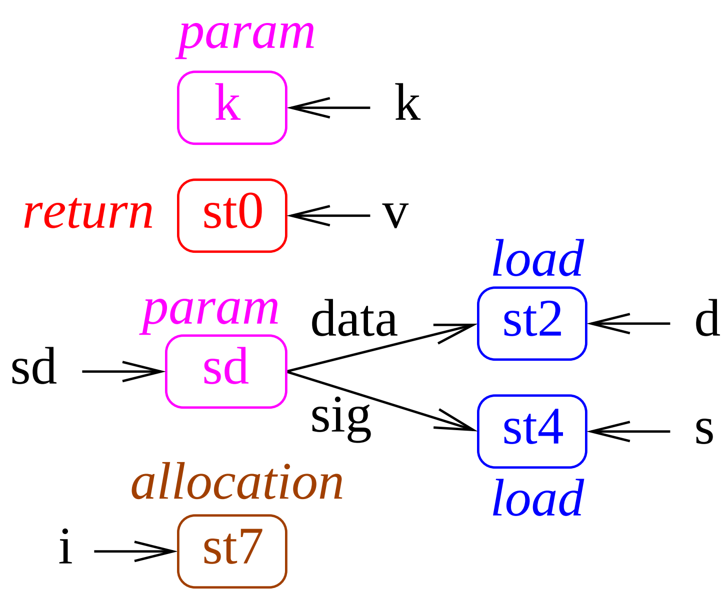
Each statement $st$ determines a **transfer function** $[\![st]\!]$.
valuation at $st\bullet$ = $[\![st]\!]$(PTE graph at $\bullet st$, valuation at $\bullet st$)

User may supply **custom method abstractions** $[\![m]\!]$.
$[\![m]\!]$ is used by transfer functions for statements that invoke $m$.
$[\![m]\!]$ distinguishes behavior for different outcomes (exceptions).
Other methods are inlined.

Analysis is expressed as a set of constraints on valuations.
Constraint for $st$ uses $[\![st]\!]$ to relate valuations at $\bullet st$ and $st\bullet$.
Contraints are solved by a worklist algorithm.

# Step 2 (Data-Flow Analysis): Example

```
 Integer getLen(SD sd,
                PublicKey k) {
0  Sig v = Sig.getInstance();
1  v.initVerify(k);
2  byte[] d = sd.data;
3  v.update(d);
4  byte[] s = sd.sig;
5  boolean b = v.verify(s);
6  if (b) {
7   i = new Integer(d.length);
8  •return i; }
9  else return null;
 }
```




$\rho(n_{\mathsf{st0}}) = \mathit{Signature}(\mathsf{verifying}, [\,], \ldots)$
$\rho(n_{\mathsf{st7}}) = \mathit{Integer}(\mathsf{sd.data.length})$
$\rho(b) = \mathit{verify}(\mathsf{k}, \mathsf{sd.data}, \mathsf{sd.sig}, \ldots)$

# Step 3: Construct Input Partition

Information about inputs may escape by being
- **part of** the return value (*e.g.*, sd.data.length), or
- **inferrable from** return value (*e.g.*, validity of sd.sig)

$StmtEsc$: statements that can cause values to escape:
return, throw, method invoc., store into escaping object.

$esc(st)$: abstract value that escapes at statement $st$
$type(st)$: type of value that escapes at statement $st$

$escStruct(st)$: concrete structures that could escape at $st$, *i.e.*, set of values of type $type(st)$, quotiented by structural equality (graph isomorphism) for selected objects (*e.g.*, new objects).

**Example:** $esc(\texttt{return i}) = Integer(\text{sd.data.length})$
$escStruct(\texttt{return i}) = \bigcup_{i \in \text{int}} \{\ [\texttt{Integer}(i)]\ \}$

# Step 3: Construct Input Partition

$Path$: edge-simple paths $p$ from $\texttt{enter}_m$ to $\texttt{exit}_m$
$guard(p)$: conjunction of guards on edges in $p$
$esc(p)$: abstract val that escapes along $p$, *i.e.*, $\bigotimes_{st \in p \cap StmtEsc} esc(st)$
$escStruct(p)$: structures that could escape along $p$, *i.e.*,
$\bigotimes_{st \in p \cap StmtEsc} escStruct(st)$

$PATH = Path$ quotiented by: $p \equiv p'$ iff $esc(p) = esc(p')$
Extend $guard$ and $escStruct$ to $PATH$:
$guard(P) = \bigvee_{p \in P} guard(p)$, $\quad escStruct(P) = \bigcup_{p \in P} escStruct(p)$

$param$: tuple of parameters of $m$

$$partn(m) = \bigcup_{\substack{P \in PATH \\ s \in escStruct(P)}} \{\{param \mid esc(P) \in s \ \wedge \ guard(P)\}\}$$

# Step 3: Construct Input Partition: Example

$$
\begin{aligned}
&\mathit{partn}(\texttt{getLen}) = \\
&\quad \{\{\langle \mathsf{sd}, \mathsf{k} \rangle \mid \neg \mathit{normalGetLen}\}\} \\
&\cup \textstyle\bigcup_{i \in \mathsf{int}} \{\{\langle \mathsf{sd}, \mathsf{k} \rangle \mid \mathsf{sd.data.length} = i \ \wedge \ \mathit{normalGetLen}\}\}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{normalGetLen} = \\
&\qquad\qquad \mathit{availableSigAlg}(\text{"SHA1withDSA"}) \\
&\qquad \wedge\ \mathsf{sd} \neq \mathsf{null} \ \wedge\ \mathsf{k} \neq \mathsf{null} \\
&\qquad \wedge\ \mathit{compatible}(\mathsf{k.getAlgorithm()}, \text{"SHA1withDSA"}) \\
&\qquad \wedge\ \neg \mathit{verify}(\text{"SHA1withDSA"}, \mathsf{k}, \mathsf{sd.data}, \mathsf{sd.sig})
\end{aligned}
$$

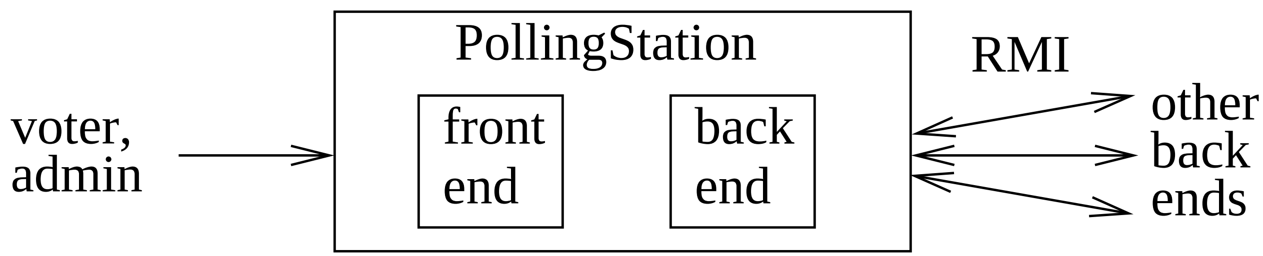
# Case Study: Distributed Voting System

Described in paper about Phalanx [Malkhi and Reiter, 1998].

Voting system is fault-tolerant and intrusion-tolerant.

Any voter can vote at any polling station.

Design is based on Byzantine quorums.




Partitions (for all methods) represented by approx 25 expressions.

Number of equiv classes with 6 quorums, 2 voters, 2 candidates, 5 polling stations: approx 425

## Code for Environment (Adversary)

Code for adversary is similar to [Roscoe and Goldsmith, 1996], but deals with equivalence classes (and RMI).

*known* := $\{E \in \text{Partn} \mid E \cap \text{InitialKnowledge} \neq \emptyset\}$
while (true) {
    non-deterministically choose an equiv. class $E$ in *known*;
    send a message in $E$ to system
    intercept response *res*;
    *known* = closure(*known* $\cup$ equivalenceClass(*res*))
}

Code for adversary is written manually, but could be generated semi-automatically from partition, by transforming predicates to unions to loops.

# Checking the Distributed Voting System

**Model checker:** state-less search with sleep sets, as in Verisoft [Godefroid 1996]. It controls non-deterministic choices by adversary and scheduler.

Found a violation of the **safety property**: if any polling station believes voter $V$ voted at polling station $S$, then $V$ voted at $S$.

This is due to the accidental omission in [Malkhi and Reiter, 1998] of part of an integrity check for requests from other polling stations.

## Related Work

Partition Analysis [Richardson and Clarke, 1985]

Auto. Closing Open Reactive Systems [Colby *et al.*, 1998]

## Summary

The analysis extracts a declarative description of the information about inputs that escapes from a method invocation.

The analysis result provides a basis for manual or semi-automatic generation of code that models the environment of an open reactive system.