# 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` procedure + caller.

For efficient testing and 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.

# Domain Partitioning

**Domain Partitioning:** Given function $f$, partition domain($f$) into equivalence classes $EC_0, EC_1, \ldots$ such that $x, y \in EC_i$ implies $f(x) = f(y)$.

**Symbolic representations** (formulas) are used, because the partition may be infinite, and each equivalence class may be infinite.

**Reactive system:** model it as a function
$f(curState, input) = \langle nextState, output \rangle$.

Prior work on domain partitioning focuses on arithmetic.
This work focuses on **objects and cryptography**.

# 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`: $\{EC_{\mathsf{err}}, EC_0, EC_1, \ldots\}$

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

# 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,
- static variables,
- 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 $p$.

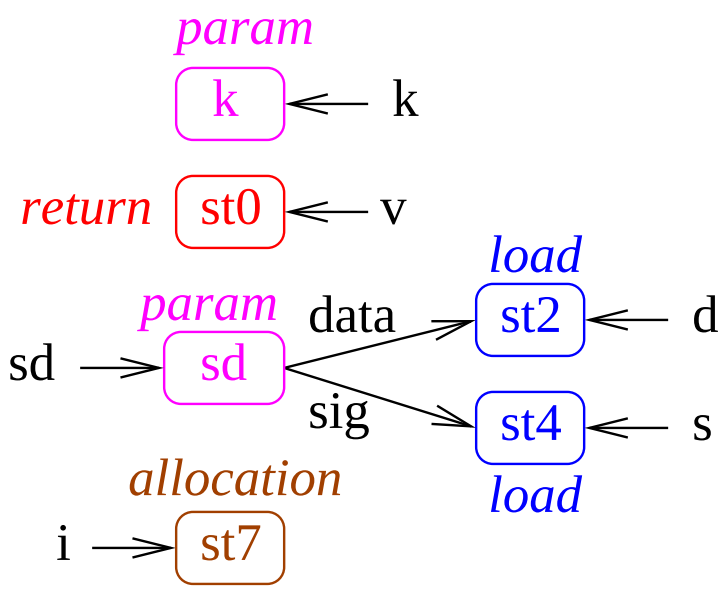
The **load node** $n_{st}$ for a load statement $st : l_1 = l_2.f$ represents objects that $l_2.f$ might point to.

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;
  }
```




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

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

# Step 2 (Data-Flow Analysis): Domains

There is an **abstract domain** for each class and primitive type.
Each **abstract value** represents a set of concrete values.

**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 $cl =$ `byte[]`) and functional methods (*e.g.*, k.getAlgorithm() for $cl =$ `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.

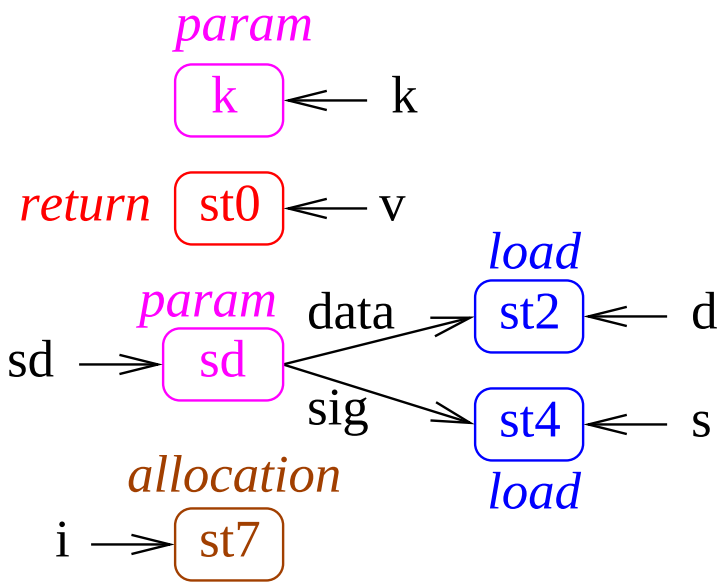
Each statement $st$ determines a **transfer function** $[\![st]\!]$.
valuation at $st\bullet = [\![st]\!]$(PTE graph at $\bullet st$, valuation at $\bullet st$)

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 (be revealed) 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

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

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

# Case Study: Distributed Voting System

Described in paper about Phalanx [Malkhi and Reiter, 1998].
Any voter can vote at any polling station (PS).

1. When a voter $V$ tries to vote, the PS contacts other PSs to ensure $V$ has not voted elsewhere.

2. When balloting is finished, the PSs co-operate to agree on the final tallies.

Voting system must be **safe** and **live** even if up to $k$ PSs are
- Byzantine faulty (**fault-tolerance**)
- compromised (**intrusion-tolerance**)

How many PSs need to be contacted?
A quorum.

# Quorum System

A $b$-**dissemination quorum system** is a set $QS$ of quorums (sets of servers) such that $(\forall Q_1, Q_2 \in QS : |Q_1 \cap Q_2| \geq b + 1)$.

**Suppose:**

- at most $b$ servers are faulty.
- every operation is performed by a quorum.
- quorum $Q_1$ performs $op_1$.
- quorum $Q_2$ performs $op_2$.

**Then** $Q_1 \cap Q_2$ contains at least one uncompromised server $S$. $S$ can inform all servers in $Q_2$ that $op_1$ was done.

# Distributed Voting System: Initialization

Central authority gives:

- a **private key**, a **secret** $s_i$, and all **public keys** to $\mathrm{PS}_i$.
- an unguessable **voter ID** (VID) to each voter, secret from PSs.
- an **access tag** $\langle h(\mathrm{VID}), \{h(f(s_i, \mathrm{VID}))\}_{1 \leq i \leq n} \rangle$ for each voter to $\mathrm{PS}_i$.

**Notation:**

$h$ is a one-way collision-resistant function.

$f$ is a pseudo-random function.

$n$ is the number of PSs

# Distributed Voting System: Initiator's Algorithm

$\text{PS}_i$: On receiving a VID from a voter:

**1.** Compute $v = h(\text{VID})$ and $sv = f(s_i, \text{VID})$.

**2.** Search for an access tag $\langle v, S \rangle$ for some $S$.
If not found, reject.

**3.** Send $[\text{voteReq}(v, sv)]_i$ to (at least) a quorum of PSs.

**4.** Wait for replies from a quorum of PSs.

**5.** If all replies are signed copies of the above request,
allow the vote. Otherwise, reject.

**Notation:**

$[msg]_i$ is *msg* signed by $\text{PS}_i$.

# Distributed Voting System: Responder's Alg.

**Overview:** $\mathsf{PS}_j$ accepts and stores the first legitimate voteReq $vreq$ for each voter. If $\mathsf{PS}_j$ receives another voteReq's for the same voter, it replies with $vreq$.

$\mathsf{PS}_j$: On receiving $vreq = [\mathsf{voteReq}(v, sv)]_i$:

1. Search for an access tag $\langle v, S\rangle$ for some $S$.
   If not found, ignore the request.
2. Check that $sv \in S$. If not, ignore the request.
3. Check whether *voted* contains a voteReq $vreq'$ for voter $v$.
4. If so, reply with $[vreq']_j$.
   If not, insert $vreq$ in *voted* and reply with $[vreq]_j$.

# Code for Environment (Adversary)

**Domain Partitioning:** 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 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 is written manually, could be generated semi-automatically.

# Checking the Distributed Voting System

First apply three transformations [Stoller and Liu, 2001]:

**centralization**: merge processes into one process

**RMI removal**: replace RMI with local invocation and copy

**pseudo-crypto**: replace `java.security` with `javacheck.security`, which "simulates" crypto

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

# Result of Model Checking

Found a violation of the **safety property**: if any polling station believes voter VID voted at $\mathsf{PS}_i$, then VID voted at $\mathsf{PS}_i$.

This is due to the accidental omission of part of an integrity check for incoming voteReqs.

An access tag should contain a sequence $\langle\!\langle h(f(s_i, \mathsf{VID}))\rangle\!\rangle_{1\leq i\leq n}$ not a set $\{h(f(s_i, \mathsf{VID}))\}_{1\leq i\leq n}$.

**Responder's Algorithm** (corrected):

$\mathsf{PS}_j$: On receiving $vreq = [\mathsf{voteReq}(v, sv)]_i$:

1. Search for an access tag $\langle v, S\rangle$ for some $S$.
   If not found, ignore the request.
2. Check that $sv = S[i]$. If not, ignore the request.

...

## 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.