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/
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_{err}, EC_0, EC_1, \ldots\}

\begin{align*}
    EC_{err} &= \{ (sd, k) \mid sd = \text{null} \lor k = \text{null} \\
               &\quad \lor \text{sd is not correctly signed WRT } k \}
    \\
    EC_i &= \{ (sd, k) \mid sd \neq \text{null} \land k \neq \text{null} \\
            &\quad \land \text{sd is correctly signed WRT } k \\
            &\quad \land \text{sd.data.length} = i \}
\end{align*}
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 \( \langle \text{Nodes}, \text{Edges}, \text{esc} \rangle \)
at each program point.

node: represents set of objects

edge: represents possible references

\( \text{esc}(n) \): 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) {
    Sig v = Sig.getInstance();
    v.initVerify(k);
    byte[] d = sd.data;
    v.update(d);
    byte[] s = sd.sig;
    boolean b = v.verify(s);
    if (b) {
        i = new Integer(d.length);
        return i;
    } else return null;
}
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 $cl = \text{byte}[]$) and functional methods (e.g., k.getAlgorithm() for $cl = \text{String}$).
- 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.

$\text{sign}(\text{key}, \text{data})$ represents return val of $\text{sign}$,
$\text{verify}(\text{key}, \text{data}, \text{sig})$ represents return val of $\text{verify}$, 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 \( \llbracket st \rrbracket \). valuation at \( st\bullet = \llbracket st \rrbracket \)(PTE graph at \( \bullet st \), valuation at \( \bullet st \))

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

Analysis is expressed as a set of constraints on valuations. Constraint for \( st \) uses \( \llbracket st \rrbracket \) to relate valuations at \( \bullet st \) and \( st\bullet \). Constraints are solved by a worklist algorithm.
Step 2 (Data-Flow Analysis): Example

```java
Integer getLen(SD sd, PublicKey k) {
    Sig v = Sig.getInstance();
    v.initVerify(k);
    byte[] d = sd.data;
    v.update(d);
    byte[] s = sd.sig;
    boolean b = v.verify(s);
    if (b) {
        i = new Integer(d.length);
        return i;
    } else return null;
}
```

\[
\begin{align*}
\rho(n_{st0}) &= Signature(\text{verifying}, [], \ldots) \\
\rho(n_{st7}) &= Integer(sd.data.length) \\
\rho(b) &= verify(k, sd.data, sd.sig, \ldots)
\end{align*}
\]
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:**
\[
\begin{align*}
esc(\text{return } i) &= \text{Integer}(\text{sd.data.length}) \\
escStruct(\text{return } i) &= \bigcup_{i \in \text{Int}} \{ [\text{Integer}(i)] \}
\end{align*}
\]
Step 3: Construct Input Partition

Path: edge-simple paths $p$ from $\text{enter}_m$ to $\text{exit}_m$

$\text{guard}(p)$: conjunction of guards on edges in $p$

$\text{esc}(p)$: abstract val that escapes along $p$, i.e., $\bigotimes_{st \in p \cap \text{StmtEsc}} \text{esc}(st)$

$\text{escStruct}(p)$: structures that could escape along $p$, i.e., $\bigotimes_{st \in p \cap \text{StmtEsc}} \text{escStruct}(st)$

$\text{PATH} = \text{Path}$ quotiented by: $p \equiv p'$ iff $\text{esc}(p) = \text{esc}(p')$

Extend $\text{guard}$ and $\text{escStruct}$ to $\text{PATH}$:

$\text{guard}(P) = \bigvee_{p \in P} \text{guard}(p)$, $\text{escStruct}(P) = \bigcup_{p \in P} \text{escStruct}(p)$

$\text{param}$: tuple of parameters of $m$

$\text{partn}(m) = \bigcup_{P \in \text{PATH}} \{ \{ \text{param} \mid \text{esc}(P) \in s \land \text{guard}(P) \} \}$

$\text{param} \in \text{escStruct}(P)$ $s \in \text{escStruct}(P)$
Step 3: Construct Input Partition: Example

\[\text{partn}(\text{getLen}) =
\{ \langle \text{sd}, k \rangle \mid \neg \text{normalGetLen} \}\]
\[\cup \bigcup_{i \in \text{int}} \{ \langle \text{sd}, k \rangle \mid \text{sd.data.length} = i \land \text{normalGetLen} \}\]\n
\[\text{normalGetLen} =
\text{availableSigAlg}("\text{SHA1withDSA}")
\land \text{sd} \neq \text{null} \land k \neq \text{null}
\land \text{compatible}(k.\text{getAlgorithm}(), "\text{SHA1withDSA}")
\land \neg \text{verify}("\text{SHA1withDSA}", k, \text{sd.data}, \text{sd.sig})\]
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).

\[
\text{known} := \{ E \in \text{Partn} \mid E \cap \text{InitialKnowledge} \neq \emptyset \}
\]

while (true) {
  non-deterministically choose an equiv. class \( E \) in \text{known};
  send a message in \( E \) to system
  intercept response \text{res};
  \text{known} = \text{closure}(\text{known} \cup \text{equivalenceClass(\text{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.