# Composing Security Policies with Polymer

Meera Sridhar

# An Introduction

- we want a mechanism by which
  - we can create security policies at a high level
  - we have a centralized security policy implementation, instead of a combination of mechanisms scattered

# An Introduction

- Polymer does these and more
  - allows policies to be first class objects → makes room for much richer policy modeling
  - allows policies to keep track of security state
  - allows bookkeeping, catching exceptions

# Key Contributions of Polymer

- new programming methodology
  - enables policy composition
  - policy separated into two types:
    - effectless method that gives suggestions and is okay to execute at any time
    - effectful methods that update security state only under certain conditions
- library of first-class, higher-order policies
  - use them to build a security policy for untrusted email clients
- formal semantics for the language & proof that it is type safe

# Polymer System Overview: Constructing a Secure Executable

- Polymer has two main tools:
  - a policy compiler that compiles program monitors defined in the Polymer language into Java language and then Java bytecode
  - a bytecode rewriter that processes ordinary Java bytecode, inserting calls to the monitor in all the necessary places

# Polymer System Overview: Constructing a Secure Executable

1. write the *action declaration file* – all methods that might have an impact on system security
2. instrument the system libraries specified in the action declaration file
3. write and compile security policy → bytecode
4. start JVM with modified libraries
5. load target application (during loading, specialized class loader rewrites target code)
6. execute secured application

# The Polymer Language: Core Concepts

- Actions:
  - objects containing information about methods that the monitors need
    - static information: method signature
    - dynamic information: calling object, actual parameters
  - can be matched against *action patterns*

# The Polymer Language: Core Concepts

- suggestions – when an untrusted application attempts to execute a security-relevant action, the monitor suggests a way to handle this action – conveyed through a `Sug` object

| SUGGESTION TYPE | SUGGESTION FOR TRIGGER ACTION | POINTS TO NOTE | TARGET |
|---|---|---|---|
| `IrrSug` | irrelevant to SP – can execute | | continue executing |
| `OKSug` | relevant to SP, but can execute | | continue executing |
| `InsSug` | deferred | final decision can't be made until auxiliary code is executed & effects evaluated | |
| `ReplSug` | return value of action replaced with value supplied by SP | may use `InsSug` to compute the alternate value | |
| `ExnSug` | should not execute | Polymer can throw a SecurityException that the target can catch before continuing execution | continue executing |
| `HaltSug` | should not execute | | should be halted |

# The Polymer Language: Core Concepts

- encode a run-time monitor in Polymer by extending the base Policy class
- new policy:
  - provide an implementation of the **`query`** method
  - optionally override **`accept`** and **`result`** methods

| METHOD | |
|---|---|
| **`query`** | -analyzes trigger action<br>-returns suggestion |
| **`accept`** | -indicates to policy that suggestion is going to be followed<br>-policy can do bookkeeping |
| **`result`** | -access to return value from **`InsSug`** or **`OKSug`**<br>-result has three arguments:<br>-original suggestion<br>-return value or action(null if return type was void, or Exception value if the action completed abnormally)<br>-flag indicating whether action completed abnormally |

# The Polymer Language: Simple Policies

```
public abstract class Policy {
  public abstract Sug query(Action a);
  public void accept(Sug s) { };
  public void result(Sug s, Object result,
                     boolean wasExnThn) { };
}
```

**Figure 2.** The parent class of all policies

```
public class DisSysCalls extends Policy {
   public Sug query(Action a) {
     aswitch(a) {
       case <* java.lang.Runtime.exec(..)>:
         return new HaltSug(this, a);
     }
     return new IrrSug(this);
   }

   public void accept(Sug s) {
     System.out.println("Illegal method called: " +
       s.getTrigger());
   }
}
```

**Figure 4.** Policy that disallows `Runtime.exec` methods

```
public class Trivial extends Policy {
  public Sug query(Action a)
    { return new IrrSug(this); }
}
```

**Figure 3.** Policy that allows all actions

# EMAIL logging/Spam

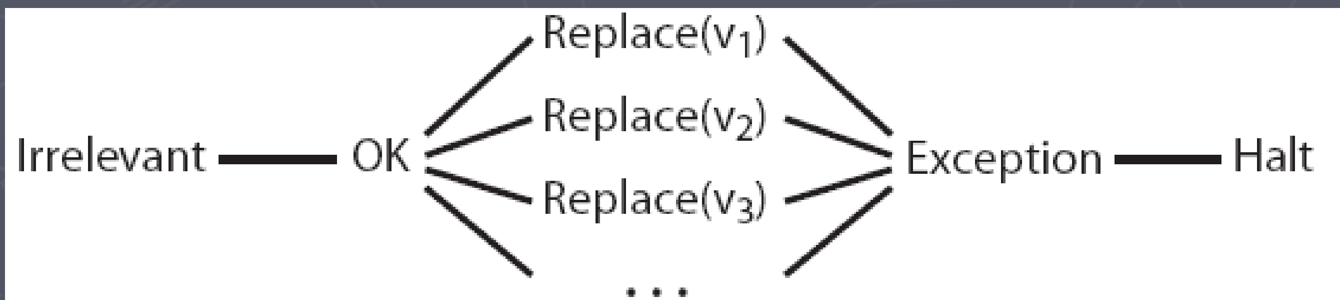
```
public class IncomingMail extends Policy {
  ...
  public Sug query(Action a) {
    aswitch(a) {
      case <abs * examples.mail.GetMail()>:
        return new OKSug(this, a);
      case <* MimeMessage.getSubject()>:
      case <* IMAPMessage.getSubject()>:
        String subj = spamifySubject(a.getCaller());
        return new ReplSug(this, a, subj);
      case <done>:
        if(!isClosed(logFile))
          return new InsSug(this, a, new Action(
            logFile, "java.io.PrintStream.close()"));
    }
    return new IrrSug(this, a);
  }
  public void result(Sug sugg, Object res,
                     boolean wasExnThn) {
    if(!sugg.isOK() || wasExnThn) return;
    log(GetMail.convertResult(sugg.getTrigger(), result));
  }
}
```

# The Polymer Language: Policy Combinators

- every policy is a first-class object
- subpolicies, superpolicies
- two kinds:
  - Conjunctive
  - Precedence
  - Selectors
  - Policy Modifiers

# Conjuctive Combinator

- composes two policies, can be extended to n
- either subpolicy InsSug → combinator suggests insertions starting from left conjunct to right; recursively examine inserted actions if security-relevant
- neither subpolicy InsSug → combinator returns least upper bound of two suggestions



# Conjunctive Combinator: Example

```
public class Conjunction extends Policy {
  private Policy p1, p2;
  public Conjunction(Policy p1, Policy p2) {
    this.p1 = p1; this.p2 = p2;
  }
  public Sug query(Action a) {
    Sug s1=p1.query(a), s2=p2.query(a);
    if(s1.isInsertion()) return SugUtils.getNewSug(
      s1, this, a, new Sug[]{s1});
    if(s2.isInsertion()) return SugUtils.getNewSug(
      s2, this, a, new Sug[]{s2});
    if(s1.isHalt() && s2.isHalt())
      return SugUtils.getNewSug(s1, this, a,
        new Sug[]{s1,s2});
    if(s1.isHalt()) return SugUtils.getNewSug(
      s1, this, a, new Sug[]{s1});
    ...
  }
  public void accept(Sug sug) {
    //notify subpolicies whose suggestions were accepted
    Sug[] sa = sug.getSuggestions();
    for(int i = 0; i < sa.length; i++) {
      sa[i].getSuggestingPolicy().accept(sa[i]);
    }
  }
  ...
}
```

# Precedence Combinators

- Precedence Combinators
  - gives precedence to one subpolicy over another
  - examples
    - TryWith: first policy returns IrrSug, OKSug, InSug → return same suggestion, else use second policy
    - Dominates: always follows the suggestion of the first conjunct if that considers the trigger action security relevant

# Selectors

- choose to enforce only one of their policies
- example:
  - IsClientSigned:
    - target cryptographically signed → weaker policy, else stronger policy

# Policy Modifiers

- enforce single policy & perform some other actions at the same time
- examples:
  - **`Audit`** – wrapper – does whatever original policy does plus logging
  - **`Filter`** – blocks policy from seeing certain actions

# Empirical Evaluation: Implementation & Performance

- Apache BCEL API for bytecode rewriting

| | Time |
|---|---|
| Per method instrumentation | 3.7 ms |
| Loading non-library classes (without instrumenting methods) | 12 ms |
| Transferring control to and from a policy while executing target | 0.62 |

- Complexity at runtime governed by computations peformed by policy

# Empirical Evaluation: Case Study on Securing Email Clients

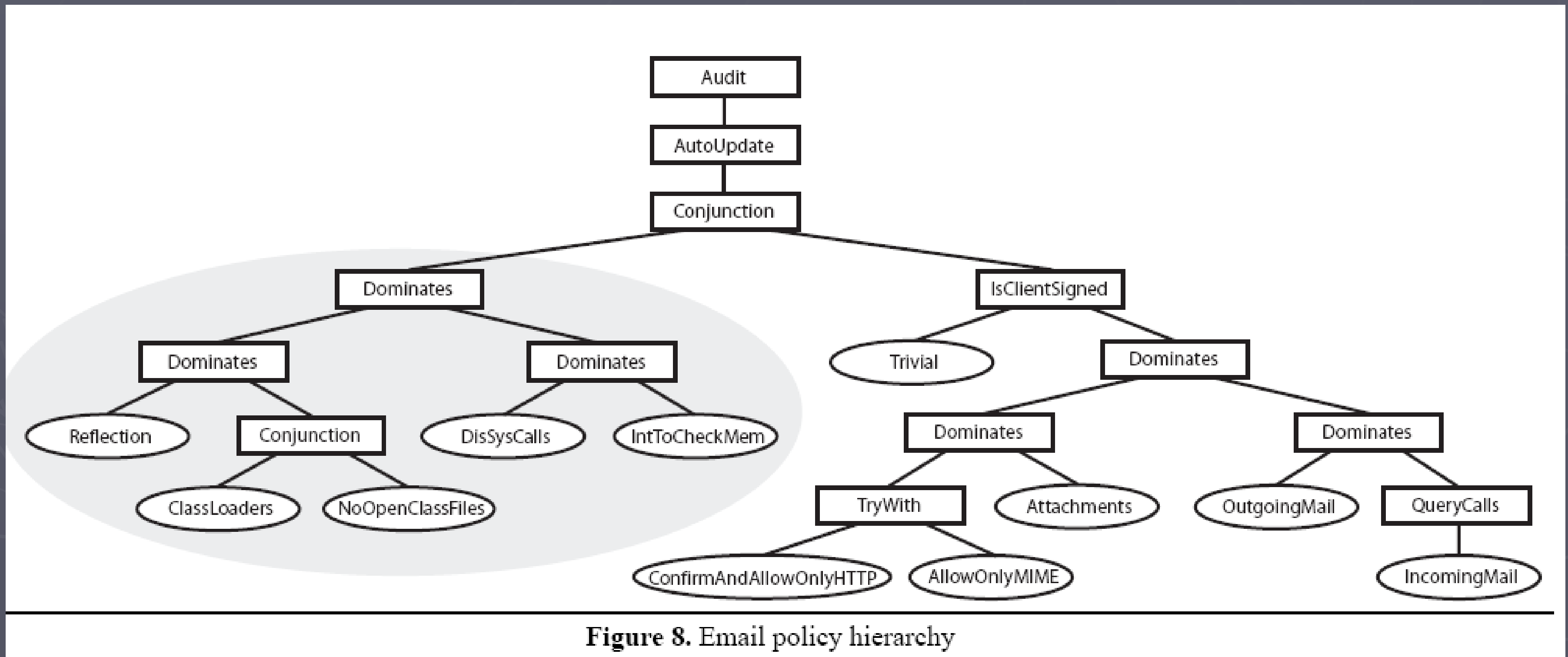



**Figure 8.** Email policy hierarchy

# Formal Semantics: Formal Syntax

types :

$\tau \quad ::= \text{Bool} \mid (\vec{\tau}) \mid \tau\ \text{Ref} \mid \tau_1 \rightarrow \tau_2 \mid \text{Poly} \mid \text{Sug} \mid \text{Act} \mid \text{Res}$

programs :

$P \quad ::= (\vec{F}, M, e_{\text{pol}}, e_{\text{app}})$

monitored functions :

$F \quad ::= \text{fun} f(x{:}\tau_1){:}\tau_2\{e\}$

memories :

$M \ ::= \cdot \mid M, l : v$

values :

$v \quad ::= \text{true} \mid \text{false} \mid (\vec{v}) \mid l \mid \lambda x{:}\tau.e \mid \text{pol}(v_{\text{query}}, v_{\text{acc}}, v_{\text{res}}) \mid$
$\qquad \text{irrs} \mid \text{oks} \mid \text{inss}(v) \mid \text{repls}(v) \mid \text{exns} \mid \text{halts} \mid \text{act}(f, v) \mid$
$\qquad \text{result}(v{:}\tau)$

expressions :

$e \quad ::= v \mid x \mid (\vec{e}) \mid e_1; e_2 \mid \text{ref}\ e \mid !e \mid e_1{:=}e_2 \mid e_1\ e_2 \mid$
$\qquad \text{pol}(e_{\text{query}}, e_{\text{acc}}, e_{\text{res}}) \mid \text{inss}(e) \mid \text{repls}(e) \mid \text{act}(f, e) \mid$
$\qquad \text{invk}\ e \mid \text{result}(e{:}\tau) \mid \text{case}\ e_1\ \text{of}\ (p \Rightarrow e_2 \mid _ \Rightarrow e_3) \mid$
$\qquad \text{try}\ e_1\ \text{with}\ e_2 \mid \text{raise exn} \mid \text{abort}$

patterns :

$p \quad ::= x \mid \text{true} \mid \text{false} \mid (\vec{p}) \mid \text{pol}(x_1, x_2, x_3) \mid \text{irrs} \mid \text{oks} \mid$
$\qquad \text{inss}(p) \mid \text{repls}(p) \mid \text{exns} \mid \text{halts} \mid \text{act}(f, p) \mid \text{result}(p{:}\tau)$

# Formal Semantics: Static Semantics

$\boxed{S; C \vdash e : \tau}$

$$\frac{S; C \vdash e_{\text{query}} : \text{Act} \rightarrow \text{Sug} \quad S; C \vdash e_{\text{acc}} : (\text{Act}, \text{Sug}) \rightarrow () \quad S; C \vdash e_{\text{res}} : \text{Res} \rightarrow ()}{S; C \vdash \text{pol}(e_{\text{query}}, e_{\text{acc}}, e_{\text{res}}) : \text{Poly}}$$

$$\frac{}{S; C \vdash \text{irrs} : \text{Sug}} \qquad \frac{}{S; C \vdash \text{oks} : \text{Sug}}$$

$$\frac{S; C \vdash e : \text{Act}}{S; C \vdash \text{inss}(e) : \text{Sug}} \qquad \frac{S; C \vdash e : \text{Res}}{S; C \vdash \text{repls}(e) : \text{Sug}}$$

$$\frac{}{S; C \vdash \text{exns} : \text{Sug}} \qquad \frac{}{S; C \vdash \text{halts} : \text{Sug}}$$

$$\frac{C(f) = \tau_1 \rightarrow \tau_2 \quad S; C \vdash e : \tau_1}{S; C \vdash \text{act}(f, e) : \text{Act}} \qquad \frac{S; C \vdash e : \text{Act}}{S; C \vdash \text{invk } e : \text{Res}}$$

$$\frac{S; C \vdash e : \tau}{S; C \vdash \text{result}(e{:}\tau) : \text{Res}}$$

$$\frac{S; C \vdash e_1 : \tau' \quad C \vdash p : (\tau'; C') \quad S; C, C' \vdash e_2 : \tau \quad S; C \vdash e_3 : \tau}{S; C \vdash \text{case } e_1 \text{ of } (p \Rightarrow e_2 \mid _ \Rightarrow e_3) : \tau}$$

$\boxed{\vdash (\vec{F}, M, e_{\text{pol}}, e_{\text{app}}) : \tau}$

$$\frac{\vdash \vec{F} : C \quad C \vdash M : S \quad S; C \vdash e_{\text{pol}} : \text{Poly} \quad S; C \vdash e_{\text{app}} : \tau}{\vdash (\vec{F}, M, e_{\text{pol}}, e_{\text{app}}) : \tau}$$

# Formal Semantics: Dynamic Semantics

$$\boxed{(\vec{F}, M, e_{\mathrm{pol}}, e_{\mathrm{app}}) \mapsto (\vec{F}, M', e'_{\mathrm{pol}}, e'_{\mathrm{app}})}$$

$$\frac{(\vec{F}, M, \mathrm{Triv}, e) \to_\beta (M', e')}{(\vec{F}, M, E[e], e_{\mathrm{app}}) \mapsto (\vec{F}, M', E[e'], e_{\mathrm{app}})}$$

where $\mathrm{Triv} = \mathrm{pol}(\lambda x{:}\mathrm{Act.irrs}, \lambda x{:}(\mathrm{Act}, \mathrm{Sug}).(), \lambda x{:}\mathrm{Res}.())$

$$\frac{(\vec{F}, M, v_{\mathrm{pol}}, e) \to_\beta (M', e')}{(\vec{F}, M, v_{\mathrm{pol}}, E[e]) \mapsto (\vec{F}, M', v_{\mathrm{pol}}, E[e'])}$$

$$\boxed{(\vec{F}, M, v_{\mathrm{pol}}, e_{\mathrm{app}}) \to_\beta (M', e'_{\mathrm{app}})}$$

$$\frac{}{(\vec{F}, M, v_{\mathrm{pol}}, (\lambda x{:}\tau.e)v) \to_\beta (M, e[v/x])}$$

$$\frac{F_i \in \vec{F} \quad F_i - \mathrm{fun} f(x{:}\tau_1){:}\tau_2\{e\}}{(\vec{F}, M, v_{\mathrm{pol}}, \mathrm{invk\ act}(f, v)) \to_\beta (M, \mathrm{Wrap}(v_{\mathrm{pol}}, F_i, v))}$$

where $\mathrm{Wrap}(\mathrm{pol}(v_{\mathrm{query}}, v_{\mathrm{acc}}, v_{\mathrm{res}}), \mathrm{fun} f(x{:}\tau_1){:}\tau_2\{e\}, v) =$

$$\begin{array}{l} \mathrm{let}\ s = v_{\mathrm{query}}(\mathrm{act}(f, v))\ \mathrm{in} \\ \mathrm{case}\ s\ \mathrm{of} \\ \quad \begin{array}{ll} \ \ \mathrm{irrs} & \Rightarrow \mathrm{let}\ x = v\ \mathrm{in\ result}(e{:}\tau_2) \\ \mid \mathrm{oks} & \Rightarrow v_{\mathrm{acc}}(\mathrm{act}(f, v), s); \\ & \quad \mathrm{let}\ x = v\ \mathrm{in\ let}\ r = \mathrm{result}(e{:}\tau_2)\ \mathrm{in}\ v_{\mathrm{res}}\ r;\ r \\ \mid \mathrm{repls}(r) & \Rightarrow v_{\mathrm{acc}}(\mathrm{act}(f, v), s);\ r \\ \mid \mathrm{exns} & \Rightarrow v_{\mathrm{acc}}(\mathrm{act}(f, v), s);\ \mathrm{raise\ exn} \\ \mid \mathrm{inss}(a) & \Rightarrow v_{\mathrm{acc}}(\mathrm{act}(f, v), s);\ v_{\mathrm{res}}(\mathrm{invk}\ a);\ \mathrm{invk\ act}(f, v) \\ \mid _ & \Rightarrow \mathrm{abort} \end{array} \end{array}$$

# Food for Thought

- How do we verify that the rewritten program is safe? – Our current TCB includes the rewrite.

# References


- Lujo Bauer, Jay Ligatti, David Walker. Composing Security Policies with Polymer, *ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, Chicago, Illinois, June 2005.
- Lujo Bauer, Jarred Ligatti, David Walker. Types and Effects for Non-Interfering Program Monitors. Department of Computer Science, Princeton University.