**Current Applications**

**Security Problems**
- Break-ins are costly & hard to prevent
- Security depends on weakest link
  - E.g., library bug exposes whole app
- Programs exhibit *aberrant behavior*
  - E.g., unexpected default behaviors

**Passe Development Cycle**
- Develop plain Django app
- Training on E2E tests defines security policies
- Secure execution using these security policies

**Passe Training**
- **Constraints** learned through analysis of queries and their results
- **Taint tracking** to detect data-flow and control-flow dependencies between queries and user requests
- **View boundaries** inferred along natural Django view boundaries

**Passe Secure Execution**

**Isolation of Components**
- **Views** separated into processes
- **Processes** isolated with AppArmor

**Restricted Data Queries**
- **Database Proxy** mitigates database access
- **Query Types** are whitelisted
- **Integrity Constraints** on query arguments

**Passe Architecture**

**Training: Data-Flow Dependencies**

- Flows captured with equality or set membership:
  - Avoids path explosion and over-tainting in analysis

**Training: Dynamic Analysis with Taint Tracking**

- Taints associate results with later query args.
- If an arg is always the result of a data-flow, Passe infers a constraint.
- Simple control-flow dependencies also create constraints.

**Enforcing Data-Flows with Token Constraints**

- Passing of cryptographic token allows stateless database proxies to check the “source” of an argument

**Preliminary Results: Inferred Policies**

- Analyzed and ran off-the-shelf Django applications
- Measured by inspecting constraints and app source code
- “Red” policies required fixing *fetch-before-check* (e.g., app fetches page from db before checking if user can read page)