Retrofitting Programs for Information-Flow Security

Dave King, Somesh Jha, Sanjit A. Seshia, Trent Jaeger

Information-flow security is a desirable security property: information-flow secure systems will not leak secret information to public channels when run. *Security-typed languages* allow programmers to write programs that will be certified as being information-flow secure. However, these languages require programmers to start from scratch and provide a large number of manual annotations. This is not a scalable way to build real systems. We propose a different approach: retrofit programs for security by finding and resolving their information-flow errors using a semi-automated procedure.

Challenges Faced By Retrofitting

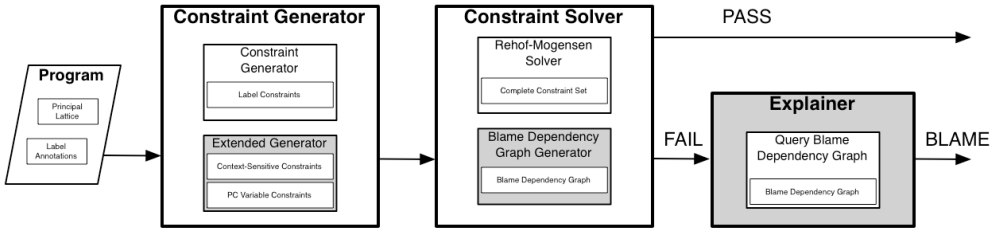
Over the last few years, a number of information-secure systems have been written using security-typed languages: JPMail (e-mail client), SIF (web application server), FlowWall (firewall).

Each of these applications was developed from scratch; however, there are a lot of existing applications that we would like to certify as information-flow secure.

There are two main challenges with this approach:

- Code is not likely to be information-flow secure "out of the box"
- Information-flow errors can be quite complex, involving conditionals, loops, exceptions, and span across methods.

A common way to verify information-flow security in code is to check the solvability of a constraint system. To better track errors in program code, we expand the constraint solver with a *blame dependency graph*, which provides errors with a *complete* and *minimal* explanation. If an error is detected, we use the *explainer* to determine what caused that failure.

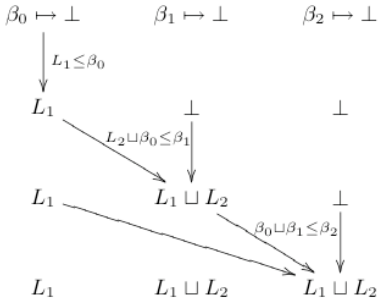


Explaining Errors

The blame dependency graph keeps track of the changes performed by the solver, allowing us to search back through the history to see which constraints caused an error.

In order to better explain errors from program code, we modify the constraint generator to create intermediary constraints that allow for better explanations.

Visualization of the Dependency Graph



Resolving Errors

Each of the errors highlighted by our system is a legitimate information-security security leak, and has a number of possible fixes based on system policy. Based on heuristics, we search for *resolution sites* that are compatible with system policy.

Groupings of similar errors can be used to solve more than one error at once; the dependency graph allows us to determine if two errors have similar causes, which may suggest a single fix. We can also introduce *trusted declassifiers* -- declassification statements that can be controlled at the policy level -- in order to make sure the application is not performing any unknown declassification.

Integrity Violation and the Program Slice Witnessing the Violation:

Error Report: implicit flow from the APDU to number of transactions since the last PIN verification

```
// Purse.java, line 1276
failed: {pc1461; appVerifyPin_reciever} <= {Untainted}
error trace:
// Purse.java, line 1273
1: {pc1461} == {verifyPin:value_returned; pc1460}
// Purse.java, line 1273
2: {verifyPin:value_returned} ==
   {verifyPin_rv; apdu@appVerifyPin:pc1460}
// Purse.java, line 838
3: {apdu@call:appVerifyPin} ==
   {apdu@call:appVerifyPin2; pc1463}
// PurseApplet.java, line 467
4: {apdu@call:appVerifyPin2} == {Tainted; pc2333}
unsatisfiable: {Tainted} <= {Untainted}
```

Error Slice:

```
1  public void process{Tainted}(APDU{Tainted} apdu) {
2     // PurseApplet.java, line 467
3     appVerifyPin(apdu);
4  }
5
6  void appVerifyPin(APDU apdu) {
7     // Purse.java, line 838
8     purse.appVerifyPin(apdu);
9  }
10
11 void appVerifyPin(APDU apdu) {
12    // Purse.java, line 1273
13    if(verifyPin(apdu, ISO7816.OFFSET_CDATA)) {
14       // Purse.java, line 1276
15       transactionWOPIN = 0;
16    }
17 }
```

Papers:

[1] Vinod Ganapathy, Dave King, Trent Jaeger, Somesh Jha. *Mining Security Sensitive Operations in Legacy Code using Concept Analysis.* In the proceedings in ICSE '06.
[2] Dave King, Somesh Jha, Sanjit A. Seshia, Trent Jaeger. *Effective Blame for Information-Flow Violations.* In submission.

Supported by DTO (Defense Technology Objectives)