Hybrid Static-Runtime Information Flow and Declassification Enforcement

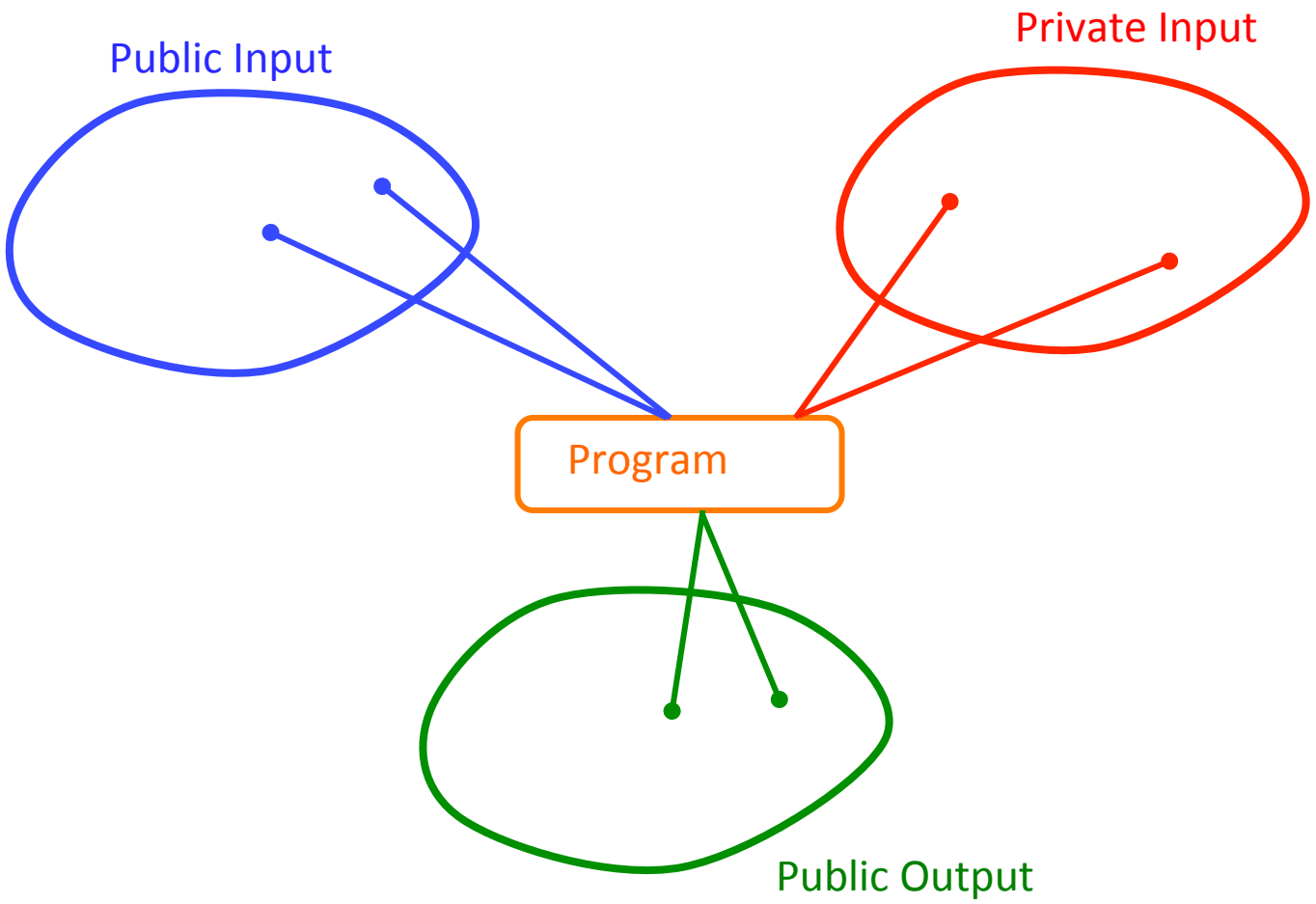
Bruno P. S. Rocha
Mauro Conti
Sandro Etalle
Bruno Crispo

Submitted

Introduction

- Language-based information flow aims to analyze programs with respect to flow of information between channels of different security levels
- **Non-interference** is a formal property for specifying valid flows *(Goguen & Meseguer 1982)*

Non-interference



Change in the private input should not change the output

Declassification

- Non-interference is **too stringent** for most practical applications
- Classic examples:
  - Average salary
  - Password verification
  - Encryption
- In many occasions, it is necessary to downgrade the security level of specific data i.e., to **declassify** that data

Current approaches

- **Static analysis**
  - + Detects **implicit flows**
  - + Easy to code **declassification** in the program.
  - - Policies that need runtime information cannot be enforced, e.g. runtime security labels, constraints on control flow execution
  - - If security labels vary between systems, analysis must be done on target machine
- **Runtime enforcers**
  - Some runtime **overhead** is incurred
  - Cannot handle implicit flows
    - needs to check code that was not executed
  - Support to **declassification is too expensive**
    - needs to keep track of every operation performed on data

Hybrid approaches

- Solution seems obvious: a hybrid mechanism that combines the strenghts of both static and runtime approaches
- Some solutions do exist: direct combination of static and runtime mechanisms originally designed to work independently

Static analysis
- Ensures type-safety
- Detects implicit flows
- Supports declassification in the code

Annotated programming language: needs input from programmer
System-dependent analysis

Runtime enforcement
- Ensures that code deemed insecure by previous step is not executed
- Checks every instruction

Overhead still not negligible
What if type-safety and declassification also needs runtime information?

Current mechanisms handle policies that need **either** static or runtime information, but not policies that need **both**

Our contribution

- A hybrid mechanism designed to:
  1. Support policies that need **both** static and runtime information
  2. Has minimum runtime overhead (works in smartphones)
- And that, additionally:
  - Does not require specially annotated code
    - declassification policies decoupled from the program's code
    - with possibly runtime constraints
  - Handles information-flow at the level of program variables
  - Performs system-independent static analysis (can be performed by a different, external system)
  - Supports security labels only known at runtime

Three Phases

Static analysis – *(an extension of graph-based PCR analysis, published at IEEE S&P 2010)*

- Detects information flows and points where declassification occurs
- Unannotated language: declassification policies specified separately
- Symbolic analysis: generates a ***flow report*** which is independent from security labels

Pre-load check

- Performed just before program is executed
- Maps the flow report on the security labels of the I/O channels
- For labels known only at runtime (e.g. filename), and runtime constraints associated to declassifications, a ***runtime checklist*** is generated, for the next stage

Runtime enforcement

- For each element of the *runtime checklist*, a call to the runtime enforcer is injected in the application's bytecode, at the program point where the check is necessary
- As a result, only statements that need runtime information to be validated are checked and, due to the code injection, the enforcer performs only constant time computation (i.e. it never loops), making it extremely lightweight

Two examples

```
secureConn := secConnect("some.host");
myLoc := getLocation();
myTz := timezone(myLoc);
otherLoc := recv(secureConn);
otherTz := timezone(otherLoc);
if myTz = otherTz then
    send("ACK", secureConn);
    near := isNear(myLoc, otherLoc);
    if near then
          print("Host is nearby!");
          print("Location:", otherLoc);
```

Declassification policy allows:

- `timezone` of any location
- `isNear` of any two locations

OBS: the secure connection has dynamic runtime security labels, which are set after each transmission

```
sum := 0;
num := 0;
db := openDBConnection();
while !exitSignal do
    rec := fetch(db);
    prop := getProperty(rec);
    sum := sum + prop;
    num := num + 1;
avg := sum / num;
output(avg);
```

Declassification policy allows:

- An average of several `getProperty` values, from several records of the DB
- Average must include at least 25 distinct records to be allowed

Static Analysis (1)

```
secureConn := secConnect("some.host");
myLoc := α2;
myTz := timezone(myLoc);
otherLoc := β4;
otherTz := timezone(otherLoc);
if myTz = otherTz then
    δ6 := "ACK"
    near := isNear(myLoc, otherLoc);
    if near then
        γ9 := "Host is nearby!"
        γ10 := "Location:", otherLoc
```

```
sum := 0;
num := 0;
db := openDBConnection();
while !exitSignal do
    rec := α2;
    prop := getProperty(rec);
    sum := sum + prop;
    num := num + 1;
avg := sum / num;
γ10 := avg;
```

1. Identifies input/output channels
2. Detects points where declassification may happen
3. Generates the flow report

The flow report

- α_2 is declassified at `myTZ`, then implicitly flows to δ_6, γ_9 and γ_{10}
- β_4 is declassified at `otherTZ`, then it implicitly flows to δ_6, γ_9 and γ_{10}
- β_4 flows explicitly to γ_{10}
- α_2 and β_4 are declassified at `near`, then flow to $\gamma_9$and γ_{10}

- α_5 is declassified at `avg`, then flows to γ_{10}

Loop must run at least 25 times

Pre-load checklist

- α_2 is declassified at `myTZ`, then implicitly flows to δ_6, γ_9 and γ_{10}
- β_4 is declassified at `otherTZ`, then it implicitly flows to δ_6, γ_9 and γ_{10}
- β_4 flows explicitly to γ_{10}
- α_2 and β_4 are declassified at `near`, then flow to $\gamma_9$and γ_{10}

- α_5 is declassified at `avg`, then flows to γ_{10}

Loop must run at least 25 times

- α_2 = high
- β_4 = data
- δ_6 = data
- γ_9 = low
- γ_{10} = low

- α_5 = high
- γ_{10} = low

1. Translates input/output channels to their respective labels
   "*data*"'s security label is only known at runtime
2. Validates declassification constraints
3. Verifies flows that can be checked at this point (alert in case of unsafe flows)
4. Filters list of flows, leaving only those that need runtime information
5. Generates *runtime checklist*

Runtime checklist

- α_2 is declassified at `myTZ`, then implicitly flows to δ_6, γ_9 and γ_{10}
- β_4 is declassified at `otherTZ`, then it implicitly flows to δ_6, γ_9 and γ_{10}
- β_4 flows explicitly to γ_{10}
- α_2 and β_4 are declassified at `near`, then flow to $\gamma_9$and γ_{10}

- α_2 = high
- β_4 = data
- δ_6 = data
- γ_9 = low
- γ_{10} = low

- α_5 is declassified at `avg`, then flows to γ_{10}

Loop must run at least 25 times

- α_5 = high
- γ_{10} = low

Runtime enforcer

```
secureConn := secConnect("some.host");
myLoc := getLocation();
myTz := timezone(myLoc);
otherLoc := recv(secureConn);
otherTz := timezone(otherLoc);
if myTz = otherTz then
    send("ACK", secureConn);
    near := isNear(myLoc, otherLoc);
    if near then
          print("Host is nearby!");
          Enforcer.checkInput(4,low);
          print("Location:", otherLoc);
```

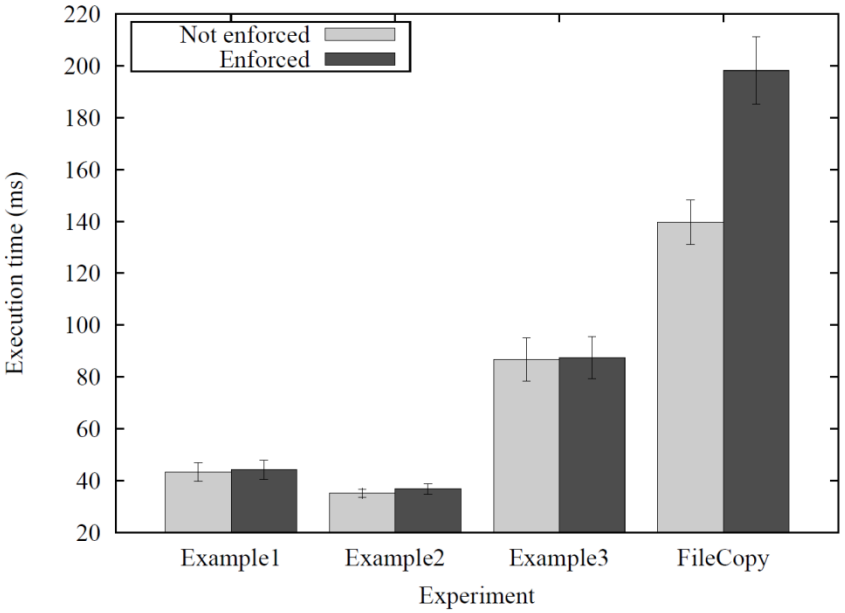
```
sum := 0;
num := 0;
db := openDBConnection();
while !exitSignal do
    Enforcer.countIter(4);
    rec := fetch(db);
    prop := getProperty(rec);
    sum := sum + prop;
    num := num + 1;
avg := sum / num;
Enforcer.eval(Enforcer.iter(4) >= 25);
output(avg);
```

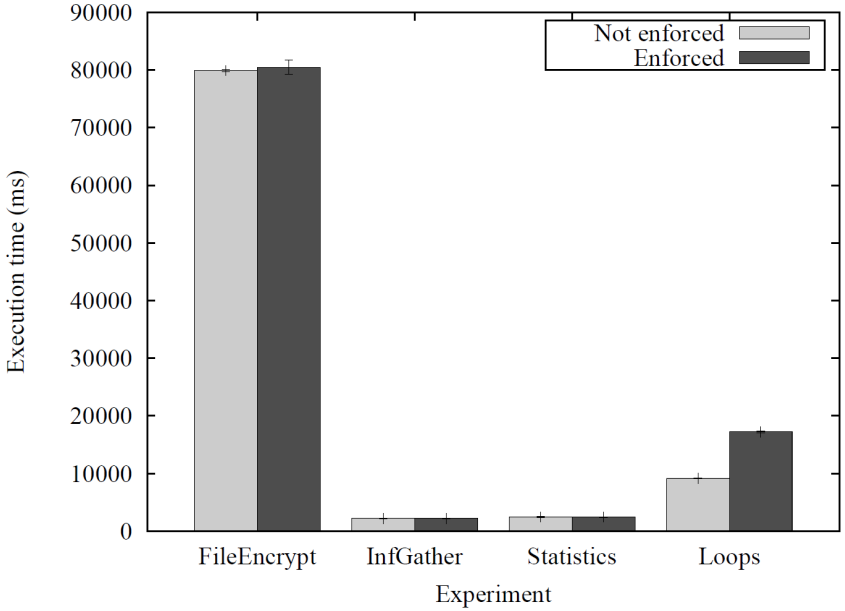
1. Injects code of checks from the checklist, prior to execution
   (this is actually done in the bytecode, source code shown here for clarity)
2. Program is then executed normally

Runtime enforcer overhead

- Implemented an Android version of the runtime enforcer
- Set of benchmarking programs used:
  - The 3 examples of the paper (2 in this presentation).
  - *FileCopy* performs a copy between files, but each 1KB block has a dynamic runtime label. The label of the source has to be applied at the target, for each transmission (stress benchmark).
  - *FileEncrypt* is the same as the above, but each block is encrypted before written (i.e. computation added between checks).
  - *InfGather* accesses inputs from 10 different sources, all with labels only known at runtime, then performs a single output combining them all.
  - *Statistics* is the same as above, but labels are static, violating non-interference. However, some statistical calculation is allowed by a declassification policy, with runtime constraints.
  - *Loops* consists of several small loops, all of which need to have their number of iterations counted (stress benchmark).

Execution time





Stress tests cause significant overhead, but represent extreme and unlikely scenarios. For the more realistic tests, overhead is barely measurable.

Memory usage

Values are the overhead between enforced and non-enforced programs

Program	Allocation Count	Allocated Size
Example1	1.2%	0.3%
Example2	1.9%	0.2%
Example3	< 0.1%	< 0.1%
FileCopy	2.2%	1.3%
FileEncrypt	< 0.1%	< 0.1%
InfGather	8.4 %	< 0.1%
Statistics	25.3 %	0.1%
Loops	341.3%	0.1%

Loop counting introduces many more memory allocations, but since tracked value is small (an integer per loop), the allocated size (the value that really matters here) remains very low. Highest overhead of 1.3%, on a stress test.

Thank you!

Questions?

Recent hybrid approaches

- Yu et al. Leakprober: a framework for profiling sensitive data leakage paths. *CODASPY'11.*
- Russo and Sabelfeld. Dynamic vs. Static Flow-Sensitive Security Analysis. *CSF'10.*
- Askarov and Sabelfeld. Tight enforcement of information-release policies for dynamic languages. *CSF'09.*
- Chong and Myers. End-to-End Enforcement of Erasure and Declassification. *CSF'08.*
- Le Guernic. Automaton-based confidentiality monitoring of concurrent programs. *CSF'07.*
- Shroff et al. Dynamic dependency monitoring to secure information flow. *CSF'07.*
- Nentwich et al. Cross-site scripting prevention with dynamic data tainting and static analysis. *NDSS'07.*
- Le Guernic et al. Automata-based confidentiality monitoring. *ASIAN'06.*
- Qin et al. A hybrid security framework of mobile code. *COMPSAC'04.*
- Schneider et al. A language-based approach to security. 2001. (first proposed)