Provably Secure Self-Protecting Systems (PROSSES)

Narges Khakpour

Department of Computer Science,
Linnaeus University

Introduction

- Evolving and complex systems
- Reactive security mechanisms (e.g. IDSs, Firewalls etc) are not sufficient
- Requires efficient proactive security mechanisms
- Sound security mechanisms
- Unknown multistage attacks

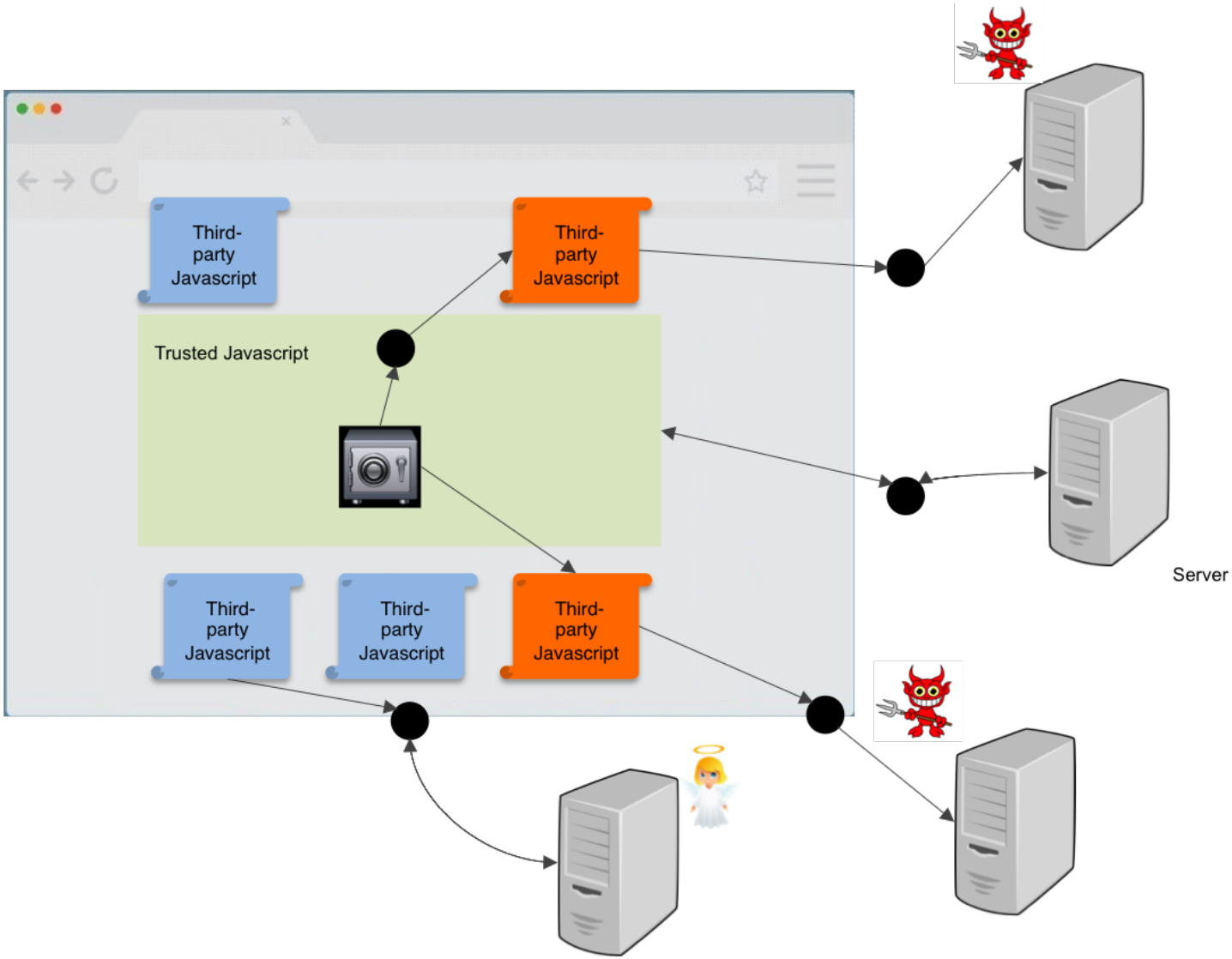
The First Phase

- Developing a sound protecting layer for web-applications
- Use formal methods
- Controls information flow

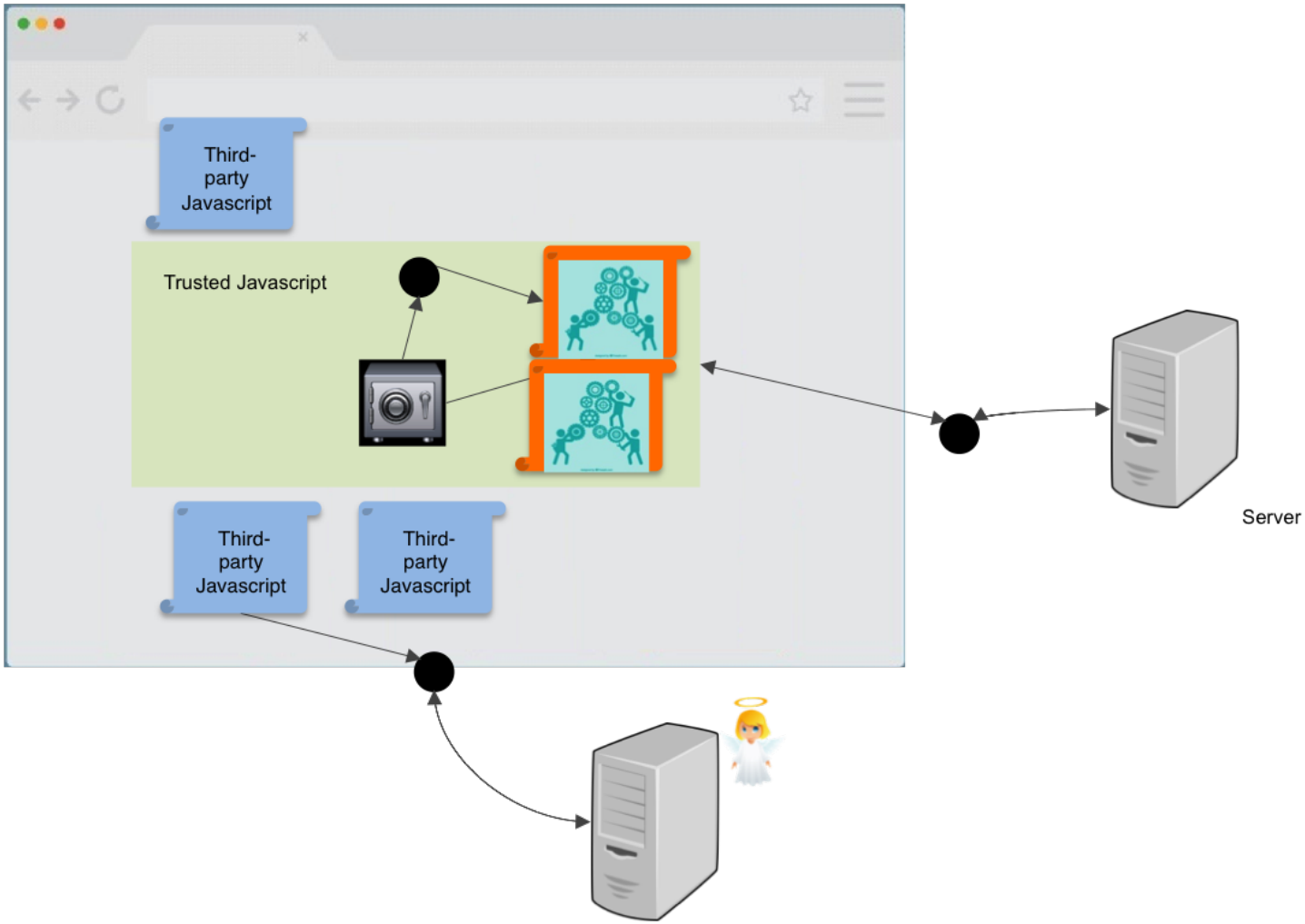
Information Flow Security

- Information Flow Security deals with
  - Confidentiality: sensitive information should not be disclosed
  - Integrity: data should not be altered illegally
- Today's software trends
  - mobile code, executable content
  - open source and platform-independence
  - large-scale and complex systems

Information Flow Security

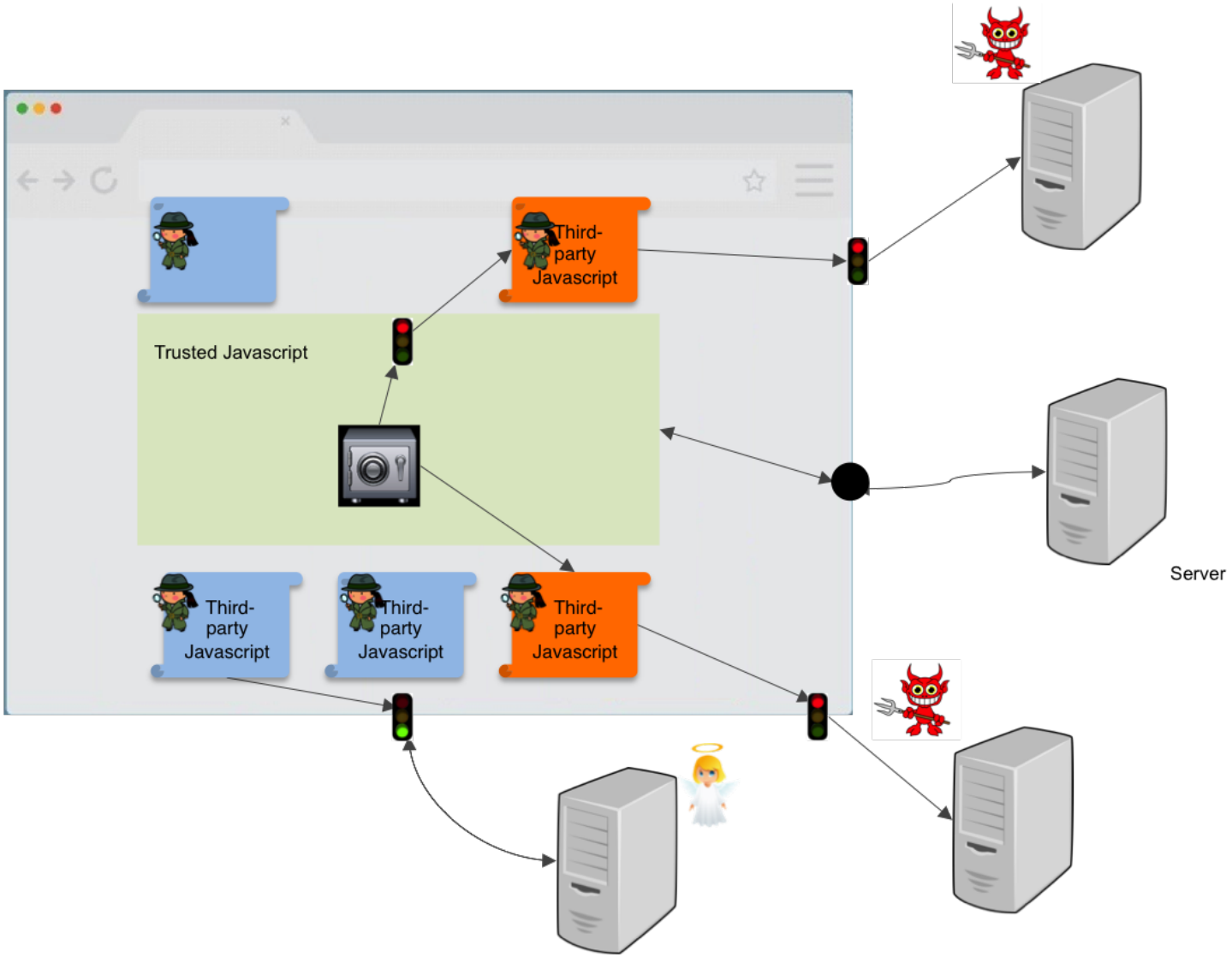


Information Flow Security



Information Flow Security

Information Flow Security



Information Flow Security

- Explicit flow vs implicit flow

Prog1 :

```
L = salary;
print (L);
```

Prog 2 :

```
if(salary>50000)
    L=L*2;
print(L);
```

Information Flow Control Mechanisms

- Static Analysis that rejects the program, if its execution causes a harm
- Rewrite the code before executing to prevent from any potential harm
- Monitor to detect the harm before it happens
- Auditing to repair the program after occurrence of a possible harm

Information Flow Control Mechanisms

```
x=h;
if(a>0) {
     while(b>0)
       {
           x=0;
           b=b-1;
     }
}
else
      x=1;
l=x;
f(l);
print(l);
```

- $a \leq 0 \implies$ secure
- $a > 0 \wedge b > 0 \implies$ secure
- $a > 0 \wedge b \leq 0 \implies$ insecure

Information Flow Control Mechanisms

- Static methods are strict:
  they accept a program only if ALL its possible executions guarantee the confidentiality, otherwise is rejected
- Dynamic methods are more permissive but have runtime overhead
- Hybrid methods combine the static and dynamic methods
- Early vs late detection
- E.g. f can be storing the data in database or sending it over the network

The General Method

Synthesize a symbolic predictive monitor that

- observes a (Java) program/system at certain checkpoints,
- predicts future information flow violations, and
- applies suitable countermeasures to prevent information leakage or unauthorized data tampering.

Step 1: Identify Checkpoints, Observation Points

- The monitor can only observe the program in the checkpoints
- An external observer (or attacker) can observe the system in the observation points

Step 1: Identify Checkpoint, Observation Points

```
 9       /* @EntryPoint */
10       /* @CheckPoint */
11       public void run(/*@SecurityInit(securityLevel="H", policyType="X")*/int sl) {
12           int estimate = 0;
13           estimate = estimatLocation(sl);
14           /* @ObservationPoint (default="System.out.println(-10)")*/
15           System.out.println(/* @SecurityPolicy(securityLevel="L", policyType="X") */estimate);
16       }
17
18       /*@CheckPoint*/
19       int estimatLocation(int strangerLoc) {
20           int x = strangerLoc * location;
21           if (x > 0) {
22               int d = getDistance(location, strangerLoc);
23               boolean exist = true;
24               if (d < MaxDistance) {
25                   x = location;
26                   exist = true;
27               } else {
28                   boolean b = false;
29                   int i = 0;
30                   while (!b && i < friendsNum) {
31                       int friendLoc = getFriendLocAt(i);
32                       d = getDistance(friendLoc, strangerLoc);
33                       if (d < MaxDistance) {
```

Step 2: Security Policies

$$a := b \times c$$

- Include the security typing information into the program
- Assign a security type to each variable which shows its security level
- Manipulate them through the execution

$$L(a) = L(b) \vee L(c)$$

- Security policies are defined over the security types in the observation points

Step 3: Synthesize the Monitor

Generated guard in estimatLocation: $ifriendsNum \geq 1$

```
9       /* @EntryPoint */
10      /* @CheckPoint */
11      public void run(/*@SecurityInit(securityLevel="H", policyType="X")*/int sl) {
12          int estimate = 0;
13          estimate = estimatLocation(sl);
14          /* @ObservationPoint (default="System.out.println(-10)")*/
15          System.out.println(/* @SecurityPolicy(securityLevel="L", policyType="X") */estimate);
16      }
17
18      /*@CheckPoint*/
19      int estimatLocation(int strangerLoc) {
20          int x = strangerLoc * location;
21          if (x > 0) {
22              int d = getDistance(location, strangerLoc);
23              boolean exist = true;
24              if (d < MaxDistance) {
25                  x = location;
26                  exist = true;
27              } else {
28                  boolean b = false;
29                  int i = 0;
30                  while (!b && i < friendsNum) {
31                      int friendLoc = getFriendLocAt(i);
32                      d = getDistance(friendLoc, strangerLoc);
33                      if (d < MaxDistance) {
```

Step 4: Design the Countermeasures

- What to do if a violation is predicted in a checkpoint?
- Apply sound countermeasures including declassification
- Declassification means deliberately disclosing information
- Necessary to design practical systems, e.g. calculating the average salary or log-in process

Ongoing Work and Conclusions

- Safety checking
- Improving the permissiveness
- Evaluating the effectiveness and scalability