THE UNIVERSITY OF ILLINOIS AT CHICAGO

UIC

POLITECNICO DI MILANO

Static Detection and Automatic Exploitation of Intent Message Vulnerabilities in Android Applications

Daniele Gallingani, **Rigel Gjomemo**,
V.N. Venkatakrishnan, Stefano Zanero

Android Message Passing Mechanism

Android apps are composed of different components

Intents carry messages among components and applications

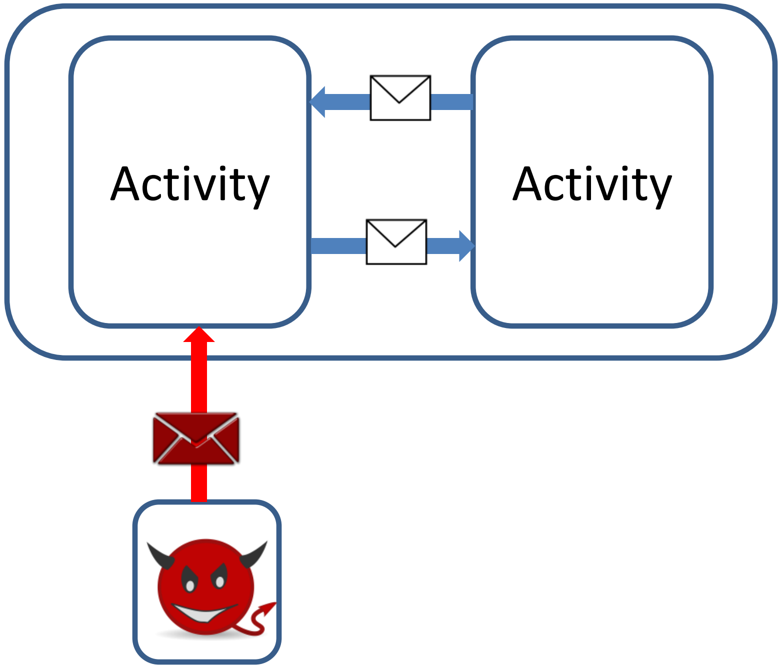
Components declare the types of intents they are willing to receive

Intents can be sent explicitly or implicitly

Motivation

Problem: Android Components have no message origin verification capabilities

An attacker can spoof legitimate intents and send malicious input



Questions

- Could we check if applications validate input?

- If so, can we automatically generate exploit opportunities?

Contributions

- Static analysis method to automatically detect data flows leading to sensitive operations
  - Formulation of the problem as an IFDS problem
- Method for automatically generating exploits that trigger malicious behavior
- Results
  - Automatically generated exploits for 26 applications and showed they are vulnerable to user interface spoofing attacks

Outline

- Problem Statement
- Approach
- Implementation
- Results

Problem Statement

```
String host = intent.getStringExtra("hostname");
String file = intent.getStringExtra("filename");
String url="http://www.example.com";
if (host.contains("example.com"))
     url = "http://" + host + "/";
if (file.contains(".."))
     file = file.replace("..", "");
String httpPar = toBase64(file);
 . . .
DefaultHttpClient httpC = new DefaultHttpClient();
HttpGet get = new HttpGet(url+httpPar);
. . .
httpC.execute(get);
```

Problem Statement

```
String host = intent.getStringExtra("hostname");
String file = intent.getStringExtra("filename");
String url="http://www.example.com";
if (host.contains("example.com"))
    url = "http://" + host + "/";
if (file.contains(".."))
    file = file.replace("..", "");
String httpPar = toBase64(file);
 . . .
DefaultHttpClient httpC = new DefaultHttpClient();
HttpGet get = new HttpGet(url+httpPar);
. . .
httpC.execute(get);
```

Source

Problem Statement

```
String host = intent.getStringExtra("hostname");
String file = intent.getStringExtra("filename");
String url="http://www.example.com";
if (host.contains("example.com"))
      url = "http://" + host + "/";
if (file.contains(".."))
      file = file.replace("..", "");
String httpPar = toBase64(file);
 . . .
DefaultHttpClient httpC = new DefaultHttpClient();
HttpGet get = new HttpGet(url+httpPar);
. . .
httpC.execute(get);
```

Source

Sink

Problem Statement

```
String host = intent.getStringExtra("hostname");
String file = intent.getStringExtra("filename");
```

Source

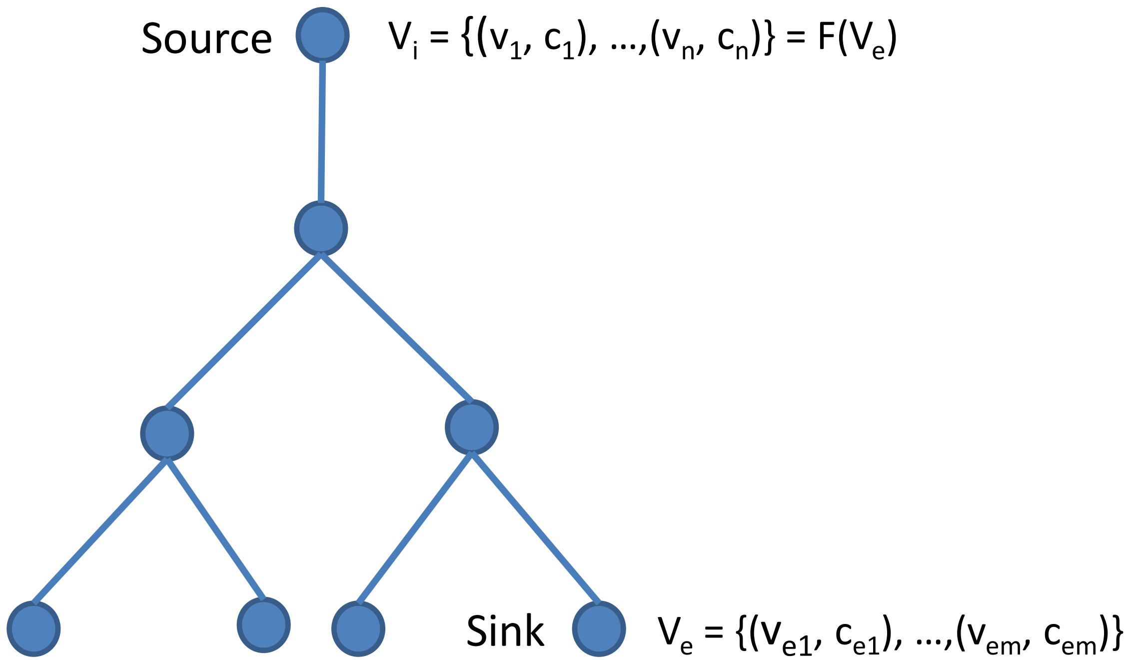
- Finding paths from sources to sinks is not sufficient
- **Question: Are those paths feasible for an attack?**

```
HttpGet get = new HttpGet(url+httpPar);
. . .
httpC.execute(get);
```

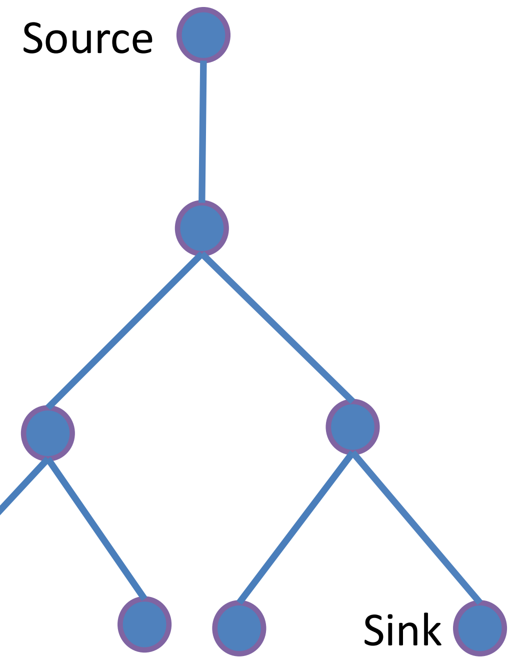
Sink

Approach

- Input state: V_I
- Exploit state(s): V_e
  Value patterns related to sinks
- Find relationship F between V_I and $V_{e,}$ such that V_i=F(V_e)

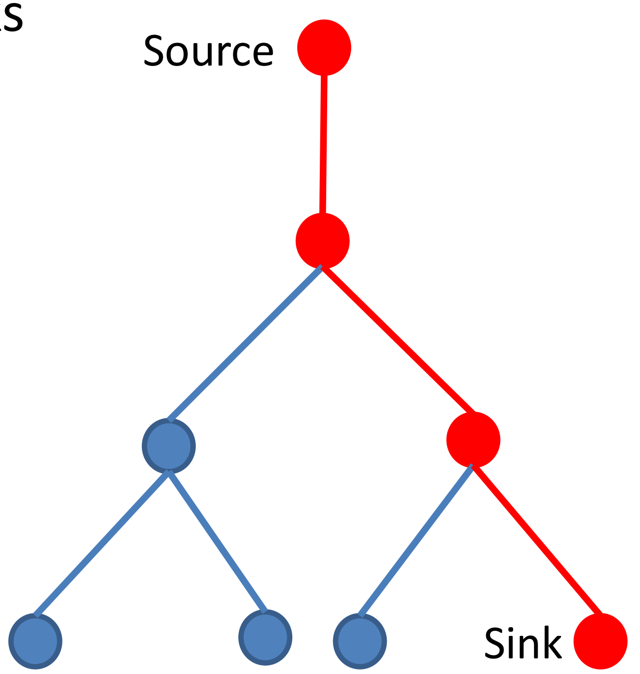


Approach Overview



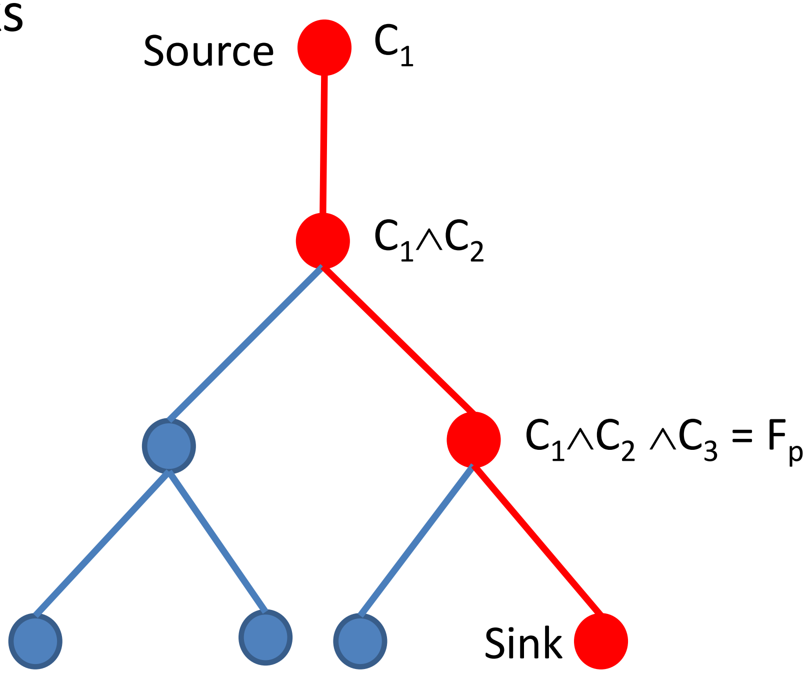
Approach Overview

- Path Computation
  - Find all paths from sources to sinks



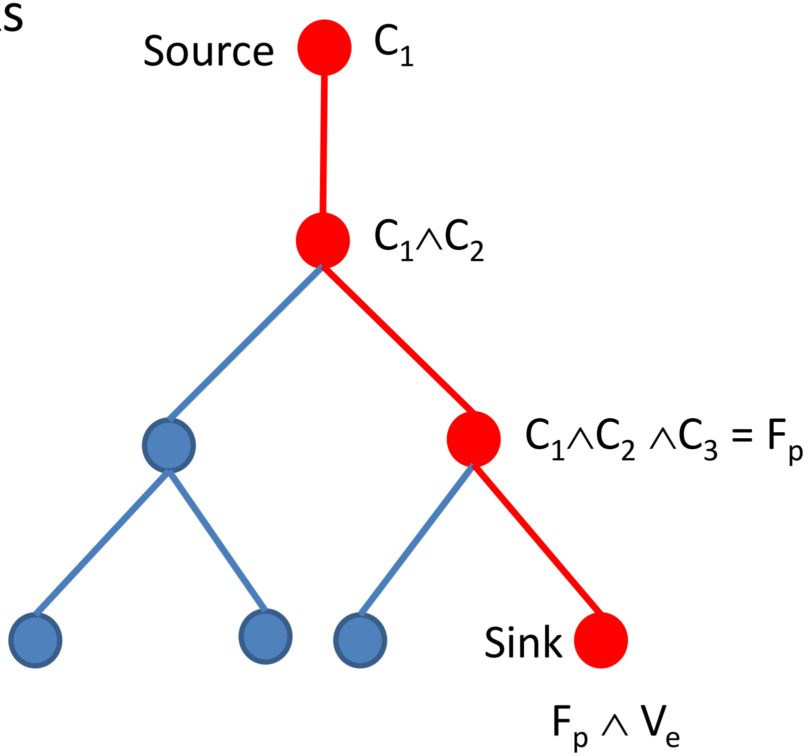
Approach Overview

- Path Computation
  - Find all paths from sources to sinks
- Symbolic Execution
  - Generate a symbolic formula F_p

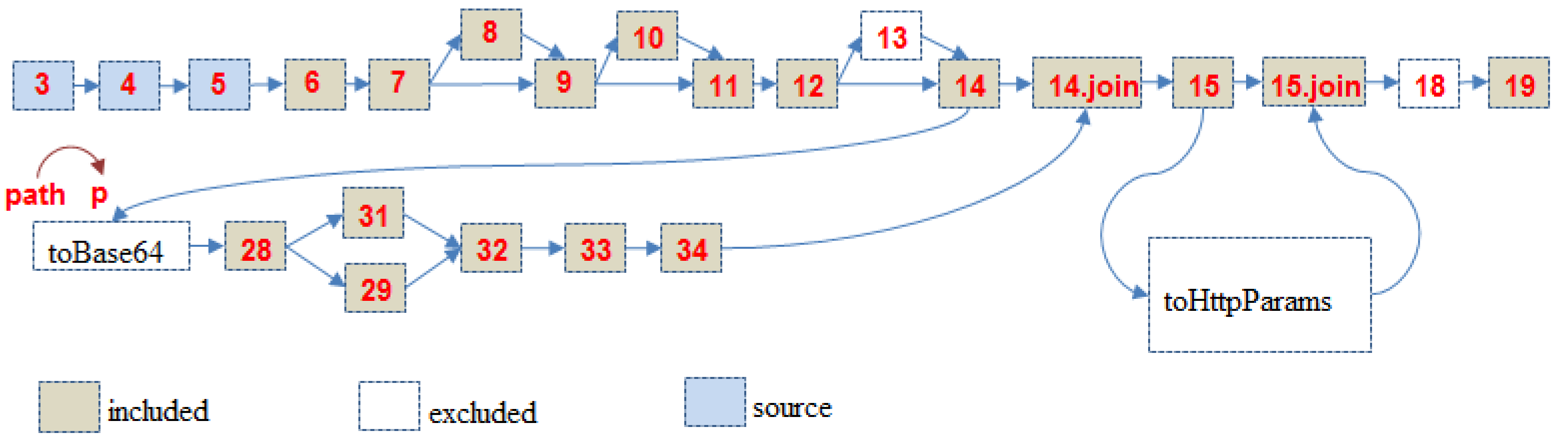


Approach Overview

- Path Computation
  - Find all paths from sources to sinks
- Symbolic Execution
  - Generate a symbolic formula F_p
- Exploit generation
  - Solve $F_p \wedge V_e \rightarrow V_l$



Path Computation



- Supergraph contains CFGs of all the functions
- Taint Propagation
  - Identifies statements that can be influenced by attacker
  - Reduces size of the problem

Implementation (Background)

- Path Computation: IFDS framework (Soot&Heros)
  - Transforms dataflow problems into graph reachability problems
  - Framework user defines a **fact**
  - Framework user defines update rules for a fact
- Exploit Generation: Kaluza
  - Efficient string solver
  - Native support for many string operations

Implementation

- **Path Computation**
  - A **fact** contains path and taint information for every node
  - Different rules update the fact information during graph traversal
- **Exploit Generation**
  - Translate $F_p \wedge V_e$ into a Kaluza formula
  - Additional string operations modeled using the Kaluza language

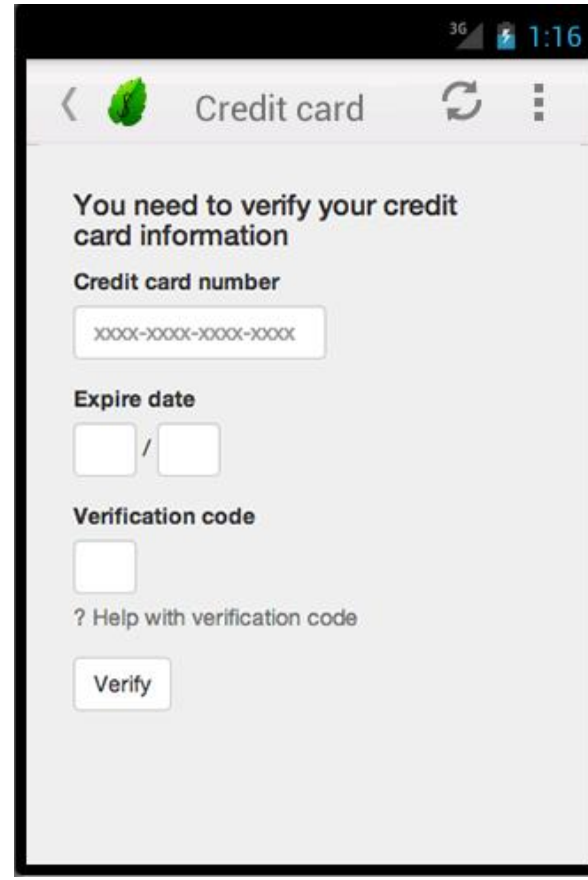
    E.g.,: *a.contains("test")* $\rightarrow$ *a \in CapturedBrack(/.*test.*/);*

Results Overview

- 64 applications of different sizes
  - 26 exploits generated and manually verified
- Sink statements: GUI operations
- V_e chosen to change apps GUIs (phishing)
- Different GUI targets
  - Entire screen change
  - Alerts screen change
  - User input fields
  - Other Components

Results

	App	Attack
Entire Screen	Mint	Display an arbitrary web page inside an Activity
	GoSMS	Prompt to the user notification about a new message with arbitrary sender and SMS content
User Input	GoSMS	Prompt notification about a new message received with arbitrary sender and receiver
	Yelp	Modify venue review draft screen and enter review on behalf of the user
Alert Screen	Poste Pay	Modify and show the application prompt alerts with arbitrary messages
Other Components	Craigslist	Change the Action Bar title, compromising the interface integrity



Results

	Min	Max	Avg
Per-application execution time	2.4 min	33.2 min	12.3 min
Per-application components	3	31	24.5
Per-application vulnerable paths	2	19	4.2
Per-path statements	5	81	17.2
Per-path if-statements	0	3	0.98

- Very few validation checks present
  - Mostly null pointers
- 31% of the String library functions approximated with Kaluza

Limitations

- Untainted variables contribute to application state. May introduce false positives
- Solver approximations. May introduce false positives

Conclusions

- Conclusions
  - We present an automatic method to discuver vulnerable paths inside Android application components
  - Our method is modelled as an IFDS problem
  - We provide proofs for the vulnerabilities under the form of actual exploits, generated automatically.

Questions?