Uranine: Real-time Privacy Leakage Monitoring without System Modification for Android

Vaibhav Rastogi[1], Zhengyang Qu[2], Jedidiah McClurg[3], Yinzhi Cao[4], and Yan Chen[2]

[1] University of Wisconsin and Pennsylvania State University
[2] Northwestern University
[3] University of Colorado Boulder
[4] Lehigh University

The Privacy Problem

- Third-party smartphone apps becoming increasingly important
- Apps regularly leak private information without informing users
- Private information leakage is a concern for both consumers and enterprises

Goal make information about privacy leaks transparent and accessible to the user

Outline


Requirements and Approach
Challenges
Design
Implementation and Evaluation
Conclusion

Outline


Requirements and Approach
Challenges
Design
Implementation and Evaluation
Conclusion

Requirements

- Real-time detection: enable situationally-aware decision making
- No platform modification: enable deployment on all devices
- Easily configurable: enable privacy leakage monitoring for just the apps user wants, no overhead for the rest of the system
- Portable: across different architectures and language runtimes
- Others: accuracy, performance

Requirements

	TaintDroid	Phosphor
Real time	Yes	Yes
System Modification	Yes	Yes
Configurability	Little	Little
Portability	No	Yes
Runtime performance	Good	Good
Accuracy	Good	Good

Enck, William, et al. "TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones." *ACM Transactions on Computer Systems (TOCS)* 32.2 (2014): 5.

Bell, Jonathan Schaffer, and Gail E. Kaiser. "Phosphor: Illuminating Dynamic Data Flow in the JVM." OOPSLA (2014).

Uranine

- Inline taint tracking. Add information flow tracking code to the application
- Do not touch platform code
  - No modification to the *runtime*
  - No modification to the *framework libraries*
- Approximate information flow through platform code

Requirements

	TaintDroid	Phosphor
Real time	Yes	Yes
System Modification	Yes	Yes
Configurability	Little	Little
Portability	No	Yes
Runtime performance	Good	Good
Accuracy	Good	Good

Enck, William, et al. "TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones." *ACM Transactions on Computer Systems (TOCS)* 32.2 (2014): 5.

Bell, Jonathan Schaffer, and Gail E. Kaiser. "Phosphor: Illuminating Dynamic Data Flow in the JVM." (2014).

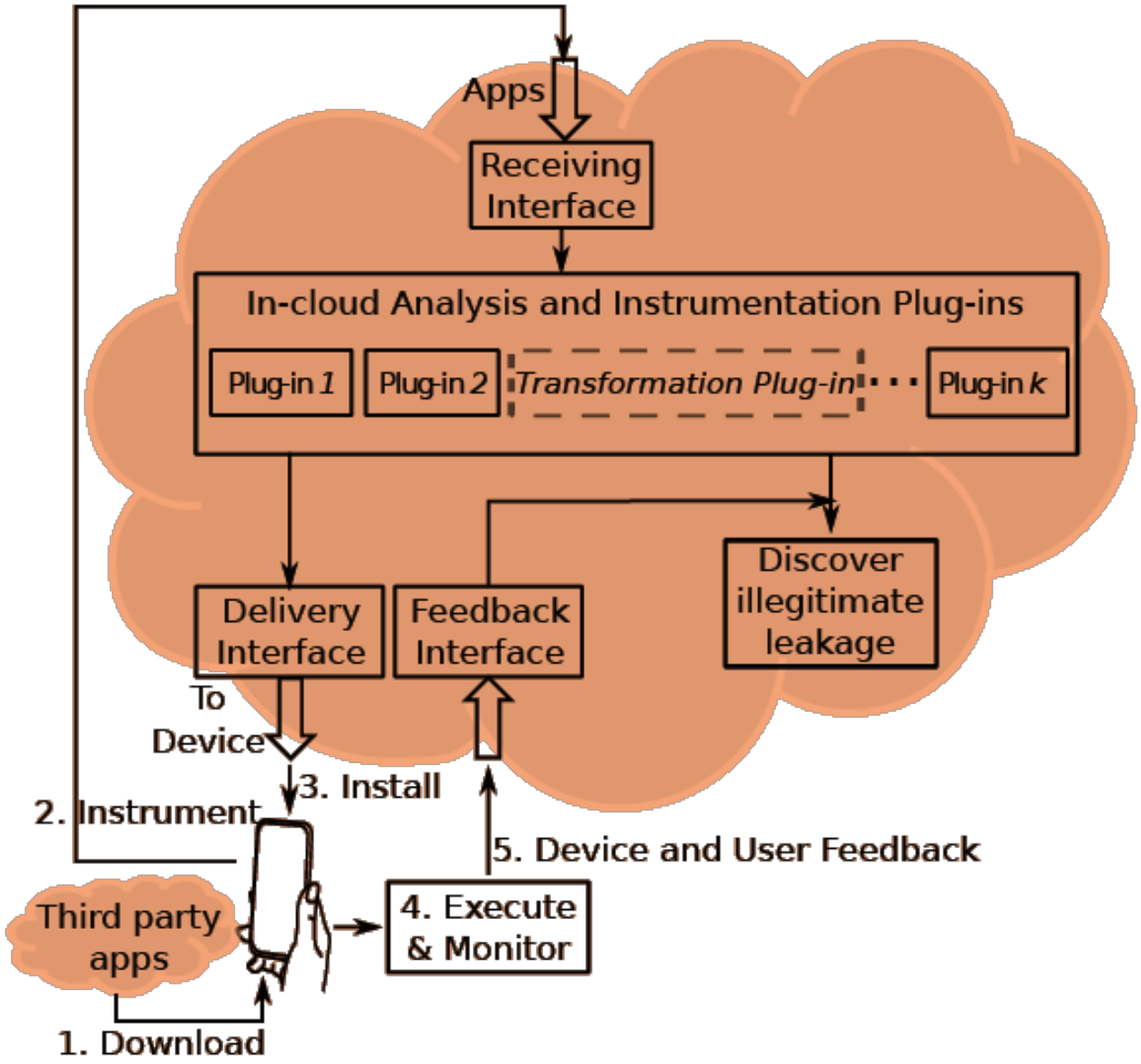
Requirements

	TaintDroid	Phosphor	Uranine
Real time	Yes	Yes	Yes
System Modification	Yes	Yes	No
Configurability	Little	Little	High
Portability	No	Yes	Yes
Runtime performance	Good	Good	Good
Accuracy	Good	Good	Good

Enck, William, et al. "TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones." *ACM Transactions on Computer Systems (TOCS)* 32.2 (2014): 5.

Bell, Jonathan Schaffer, and Gail E. Kaiser. "Phosphor: Illuminating Dynamic Data Flow in the JVM." (2014).

Deployment Model



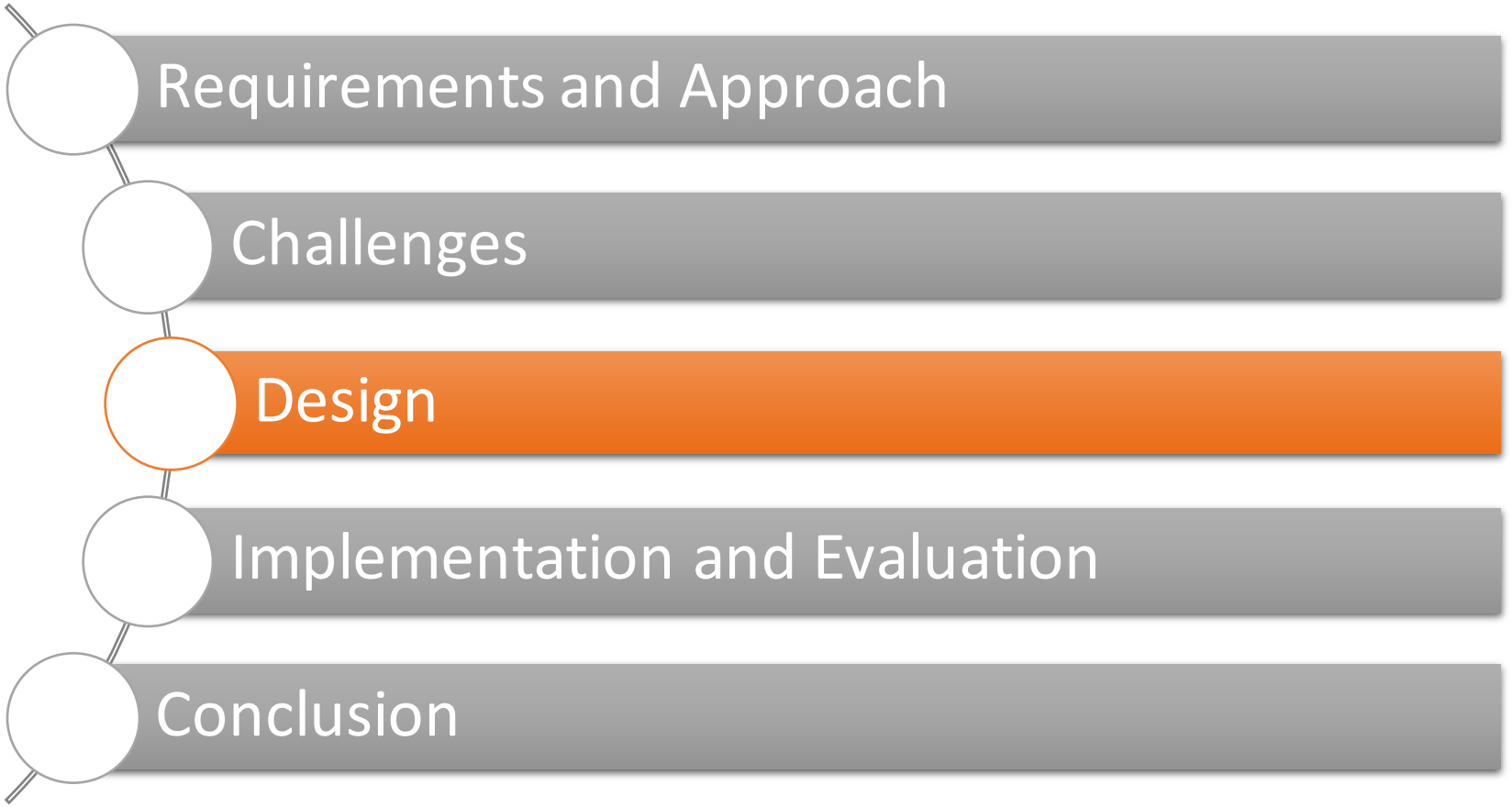
Outline


Requirements and Approach
Challenges
Design
Implementation and Evaluation
Conclusion

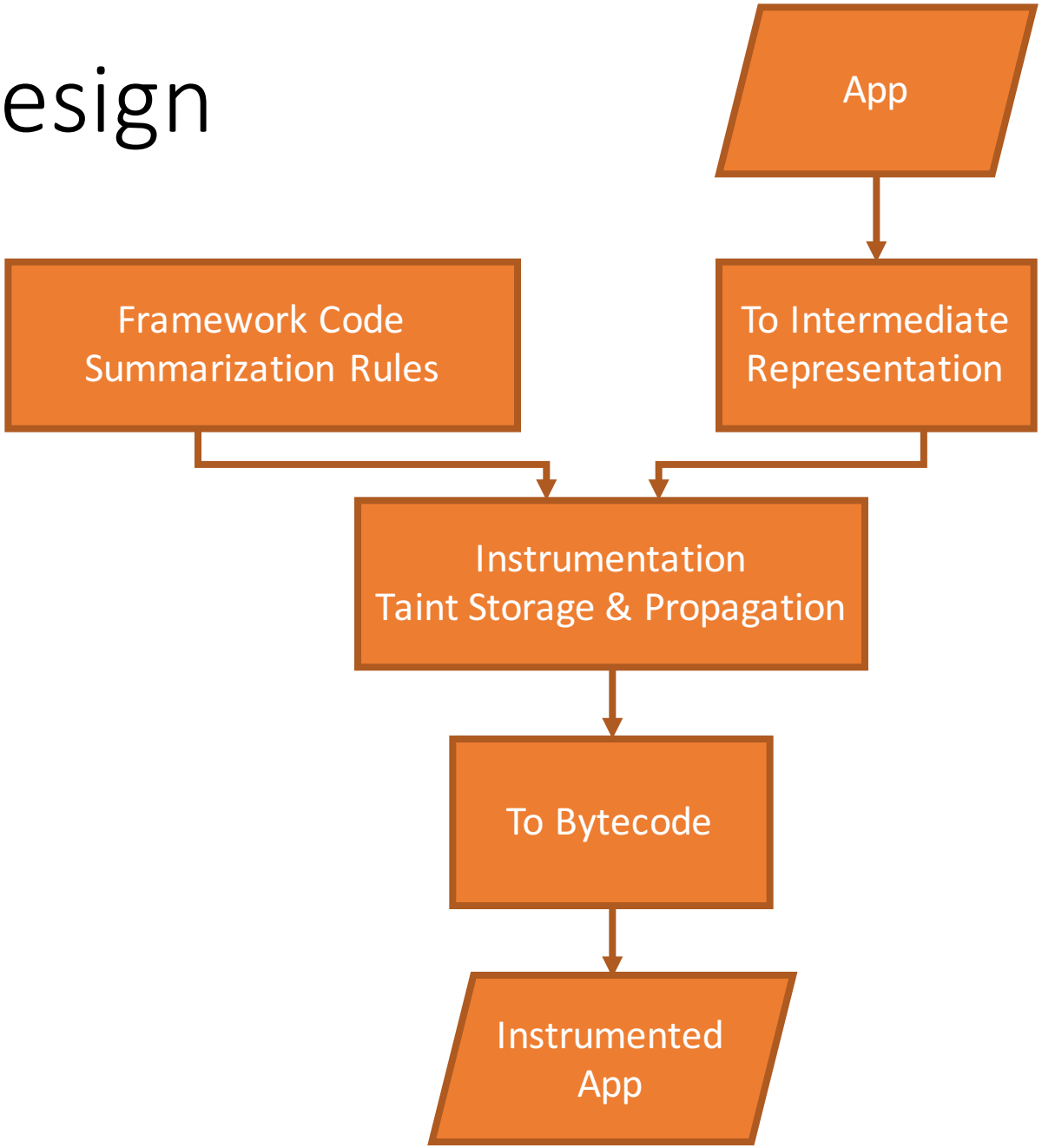
Challenges

- Tracking taint across calls to framework libraries
- Accounting for the effects of callbacks
- Tainting objects while following Java reference semantics

Outline


Requirements and Approach
Challenges
Design
Implementation and Evaluation
Conclusion

Design


App
Framework Code
Summarization Rules
To Intermediate
Representation
Instrumentation
Taint Storage & Propagation
To Bytecode
Instrumented
App

Taint Storage and Propagation

- Shadow taint location for each location

```
class A {
    String field;
}
```



```
class A {
    String field;
    int field_t;
}
```

- Similar for method parameters
  - Add additional parameters for carrying taints
  - Return taint returned via parameter
- Taint propagation for various operations

```
p = q + r;
```



```
p = q + r;
p_t = q_t | r_t;
```

Taint Storage and Propagation

- Introduce taint at sources

```
String id = tm.getDeviceId();
```

```
String id = tm.getDeviceId();
int id_t = 1;
```

- Check for taint reaching sinks

```
socket.write(deviceLocation);
```

```
if (deviceLocation_t != 0)
    sendAlert();
socket.write(deviceLocation);
```

Tracking Taint across library calls

- Pre-defined rules for summarization
- Catch-all policy: Combine taint of all parameters and set to the return taint and the taint of object on which method is called (*receiver*)
- Above summarization not sufficient: additionally propagate taint to all objects that refer to the object being tainted

Callbacks

```
class A {
    private String id;
    public A(TelephonyManager m) {
        id = m.getDeviceId();
    }
    public toString() {
        return id;
    }
}
```

- `toString()` may be called by framework code and the returned string used elsewhere
- Solution: treat like framework code and propagate return taint to receiver

Java Reference Semantics

- Problem: tainting objects, not just object references
- If an object gets tainted, all references should show the taint
- Storing object taints should not affect garbage collection
- Solution: Use a weak hashtable to map objects to taints

Outline


Requirements and Approach
Challenges
Design
Implementation and Evaluation
Conclusion

Implementation

- Employ `dexlib` to convert bytecode to IR
- A class hierarchy analysis to identify callbacks and guide the instrumentation
- A fine-grained instrumentation framework on top of IR
  - Generates bytecode sequences that pass the Dalvik verifier
- 6000 lines of Scala code

Accuracy Evaluation

- Use TaintDroid as ground truth
- Small-scale manual as well as large-scale automated tests
- Large-scale automated runs with Android Monkey on 1490 apps
- Privacy leakage results consistent with TaintDroid
- 4 cases were identified to be Uranine false positives

Performance Evaluation

- Performance expected to be good: framework code, which does the real heavy-lifting, runs without overhead
- Measuring performance is difficult
  - No macrobenchmarks for Android
  - Microbenchmarking will not show true performance on real workloads
- Created 6 macrobenchmarks from real apps from Google Play
- Overhead less than 50% for 5 benchmarks, and around 10% in four benchmarks
- Compares favorably with TaintDroid (30%) and Phosphor (50%)

Scope for Optimizations

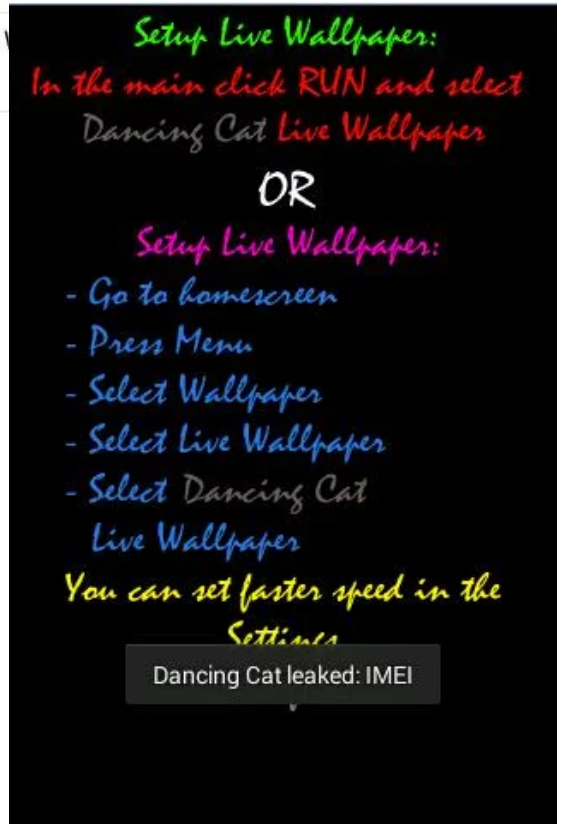
- Static analysis may be used to identify code paths that will not leak information
- Thus only a few paths need to be instrumented
- Such optimizations not possible for TaintDroid or Phosphor

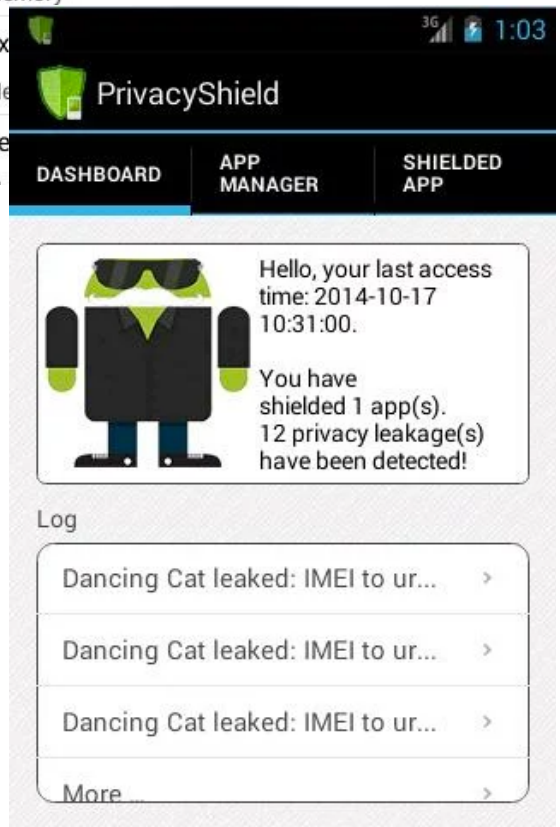
Outline


Requirements and Approach
Challenges
Design
Implementation and Evaluation
Conclusion

Conclusion

- Privacy is a major issue in the present digital revolution
- Private information leakage should be transparent
- Uranine tracks private information leakage in Android apps without platform modification
- A step towards bringing information leakage transparency to the masses





https://play.google.com/store/apps/details?id=com.webshield.privacyshield

Thank you!