Comparative Evaluation of Architectural and Code-Level Approaches for Finding Security Vulnerabilities

Radu Vanciu

Ebrahim Khalaj

Marwan Abi-Antoun

Department of Computer Science

Wayne State University

Motivating Example

- A hash code needs to be encrypted
- The developer follows the steps of an encryption algorithm
- The developer forgets one of the crucial steps of the algorithm
- The encryption is now broken

```
class CryptoStep {
   void missingStep() {
     MsgDigest md = new MsgDigest("") ;
     IO io = new IO();
     String hashInput = new String("ABCDEFG123456");
     // Injected vulnerability: comment out next call
     // md.update(hashInput);
     io.writeLine(io.toHex(md.digest()));
   }
}
```

- Which approaches report the security vulnerability with good precision and recall?

Contributions

- Comparative evaluation using test cases with injected vulnerabilities of:
  - **Scoria** as an **architecture-level** approach
  - **FlowDroid** as a **code-level** approach
- We compare approaches in terms of:
  - True positives (TP), Higher is better
  - False Positives (FP), Lower is better
  - False Negatives (FN), Lower is better
- Building a benchmark
  - Benchmarking is a common way to do the comparison, e.g., for compiler optimization
  - Enables reproducibility
- Introduce Architectural Flaw Index (AF-Index) to classify security vulnerability along the continuum

Research Questions

To find security vulnerabilities, approaches make tradeoffs

- Scoria tradeoffs: (Architecture-Level)
  - Sound and possibly less precise
  - Analyst-assisted approach
  - Special purpose constraints
  - Separate extraction and constraints
  - Extracts high-level representation for consumption by Security Information Workers (SIW)
- FlowDroid tradeoffs: (Code-Level)
  - Unsound and possibly more precise
  - More automated approach;
  - General purpose constraints
  - Combined extraction and constraints
  - Extract low-level representation used by the tools

- Which of the above tradeoffs leads to higher recall and precision?

Finding security vulnerabilities that are closer to **architectural flaws** is harder

Architectural flaw	**Coding bug**
e.g., missing authentication	e.g., hard-coded password

- We are more interested in test cases that are closer to architectural flaws

ScoriaBench

- We built a benchmark with hand-selected test cases from different equivalence classes:
  - 43 hand-selected test cases
  - Android and Java applications
  - 13 different equivalence classes
- Selected test cases from:
  - DroidBench(DB)
  - NIST SAMATE Reference Dataset (SRD)
  - Examples from CERT Secure Coding Standard for Java
  - Designed by us (US)

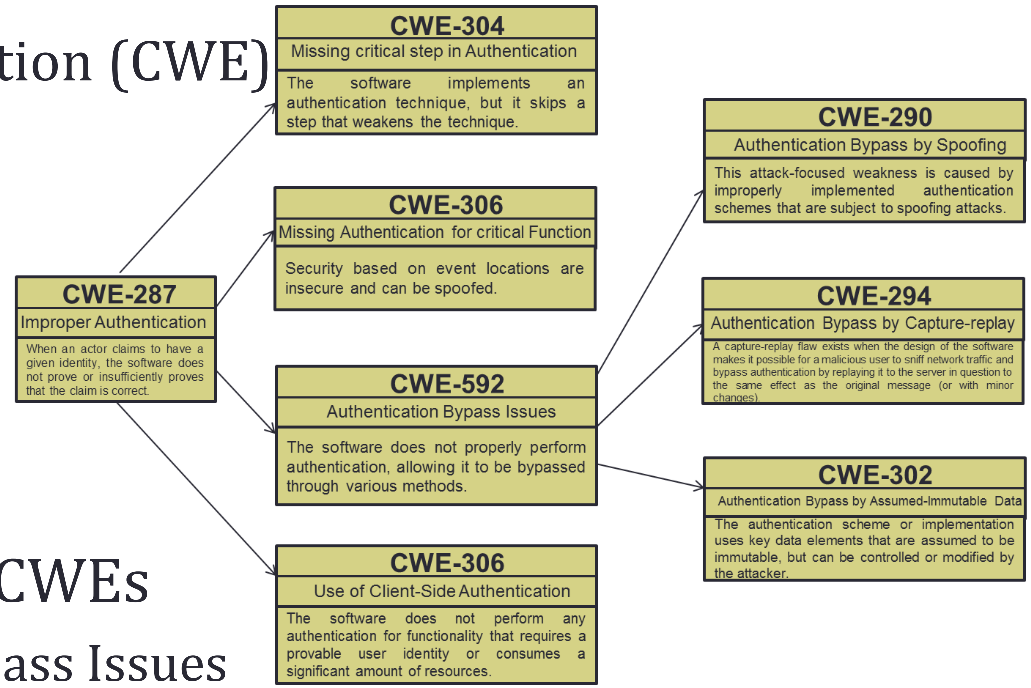
ScoriaBench Equivalence Classes

equivalence classes	#tests cases	DB	SRD	CERT	US
Arrays and Lists	1	✓			
Callbacks	5	✓			
Field and Object Sensitivity	7	✓			✓
Inter-App Communication	3	✓			
Lifecycle	5	✓			
General Java	3	✓			
Android-Specific	6	✓			✓
Implicit Flows	4	✓			
Missing Encryption	**4**		✓	✓	
Bypass Authentication	**2**				✓
Command Injection	**1**			✓	
Exploitable Service	**1**				✓
Least Privilege Violation	**1**				✓
Total	**43**	**8**	**1**	**2**	**5**

*Testcases in bold equivalence classes contain more architectural flaws. We added them on top of DroidBench equivalence classes.

Selection Process

- Common Weakness Enumeration (CWE)
  - Basis for a vulnerability
  - Parent-child relation



- We found such interesting CWEs
  - CWE-592: Authentication Bypass Issues
  - CWE-302: Authentication Bypass by Assumed-Immutable Data
  - CWE-290: Authentication Bypass by Spoofing
  - CWE-325: Missing Required Cryptographic Step
- We searched SRD to find test cases for those CWEs

Selection Process (cont'd)

- Some CWEs had an interesting test case
  - CWE-325 : Missing Required Cryptographic Step
  - We added the test case to ScoriaBench
- Some CWEs had no corresponding test case:
  - CWE-592: Authentication Bypass Issues
  - CWE-290: Authentication Bypass by Spoofing
  - CWE-302: Authentication Bypass by Assumed-Immutable Data
- There is a **gap** that needs to be filled

Architecture-Level Approach: Scoria

Add and typecheck annotations

- Annotations express design intent

Extract high-level representation

- Sound over-approximation of runtime structure

Write constraints to find vulnerabilities

- Enriched representation with security properties and queries

Refine annotations

Scoria: Add Annotations

Code:

```
class CryptoStep {
   void missingStep() {
     MsgDigest md = new MsgDigest("") ;
     IO io = new IO();
     String hashInput = new String("ABCDEFG123456");
     io.writeLine(io.toHex(md.digest()));
   }
}
```

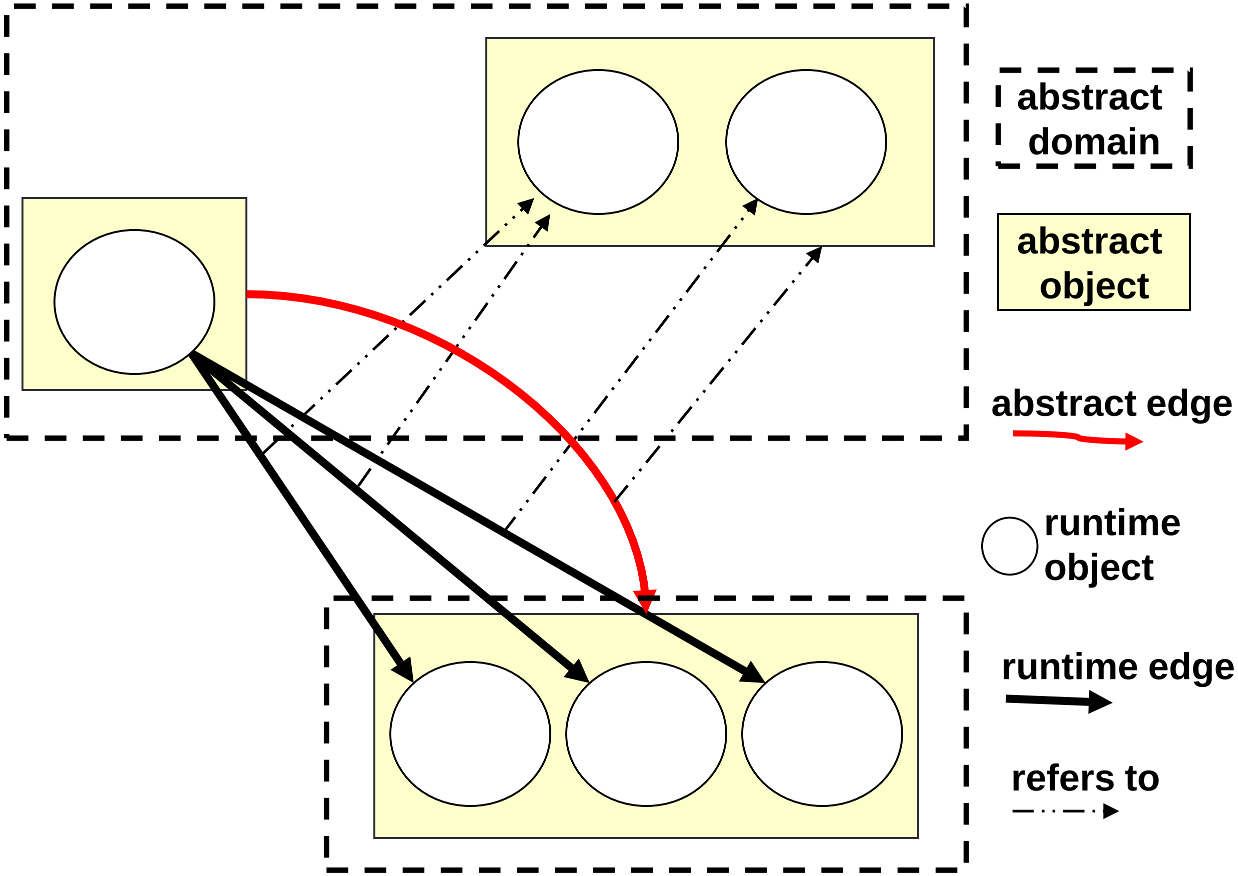
- Express design intent
  - Impose arbitrary hierarchy on objects [in the object graph]
- Consistent with the code:
  - Implement a type system
  - Not tags, they have meaning
- Support legacy code

Annotated code:

```
@Domains("OWNED")
class CryptoStep {
   void missingStep() {
     @Domain(“OWNED") MsgDigest md = new MsgDigest("") ;
     @Domain("OWNED") IO io = new IO();
     @Domain("unique") String hashInput = new String("ABCDEFG123456");
     io.writeLine(io.toHex(md.digest()));
   }
}
```

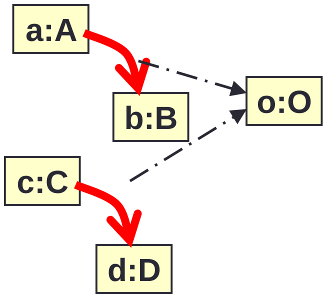
Scoria: Extraction Static Analysis

- Soundness
- Aliasing
- Precision
- Summarization
- High-level View
- Traceability

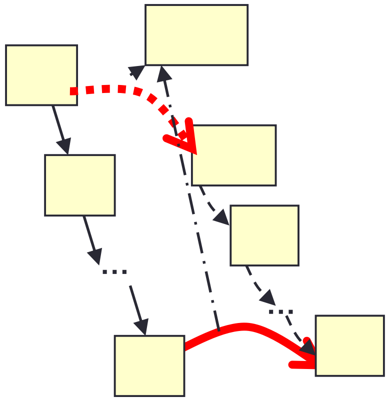


Scoria: Constraints

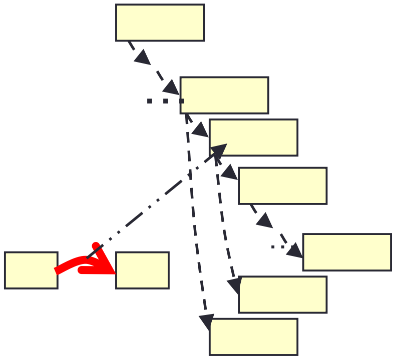
Object Provenance



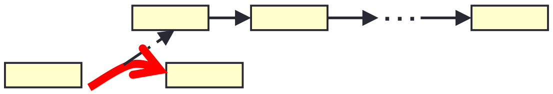
Indirect Communication



Object Hierarchy

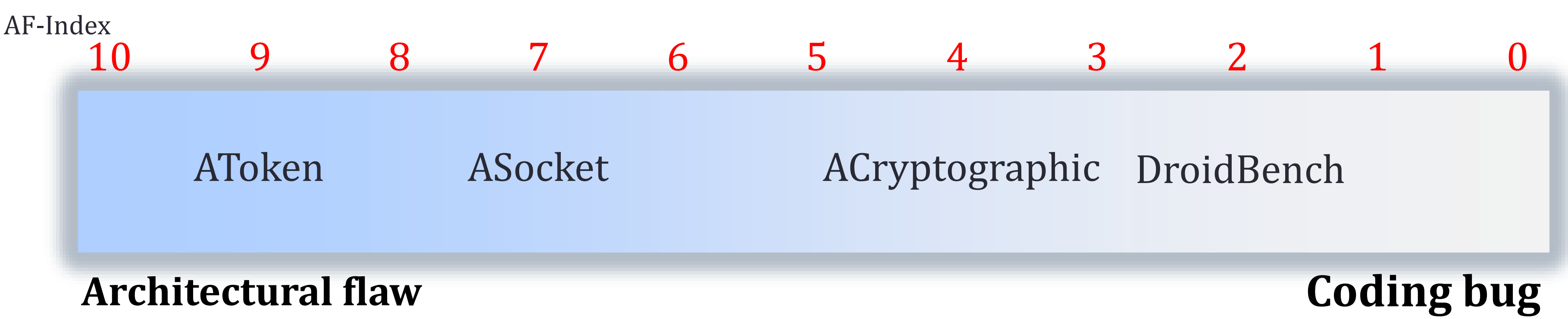


Object Reachability



Architectural Flaw Index (AF-index)

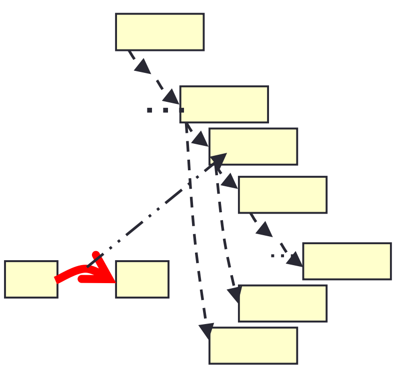
- Measures if a test case is an architectural flaw or coding bug
- In ScoriaBench we have AF-index from **0** to **10**:
  - 0 is the completely code-level vulnerability
  - 10 is the completely architecture vulnerability
- **Higher** AF-index means a test case is placed closer to the **left** of the continuum



How to compute AF-index?

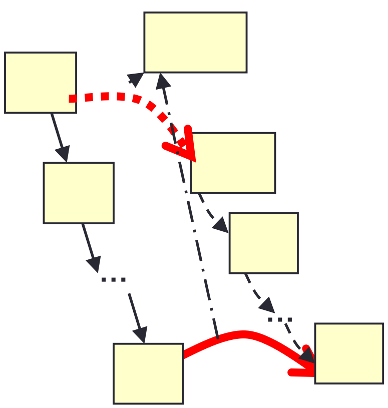
- Assign a weight to each Scoria constraint
- Compute the summation of weighted constraints that are used to find the vulnerability using Scoria

Weight = 2



Object hierarchy

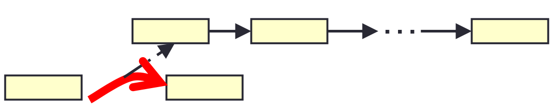
Weight = 3



Indirect communication

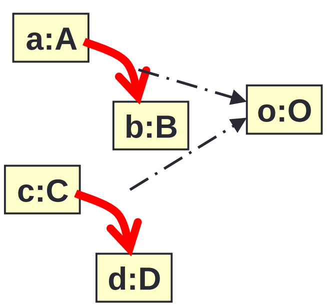
Weight = 1
Property

Weight = 2



Object reachability

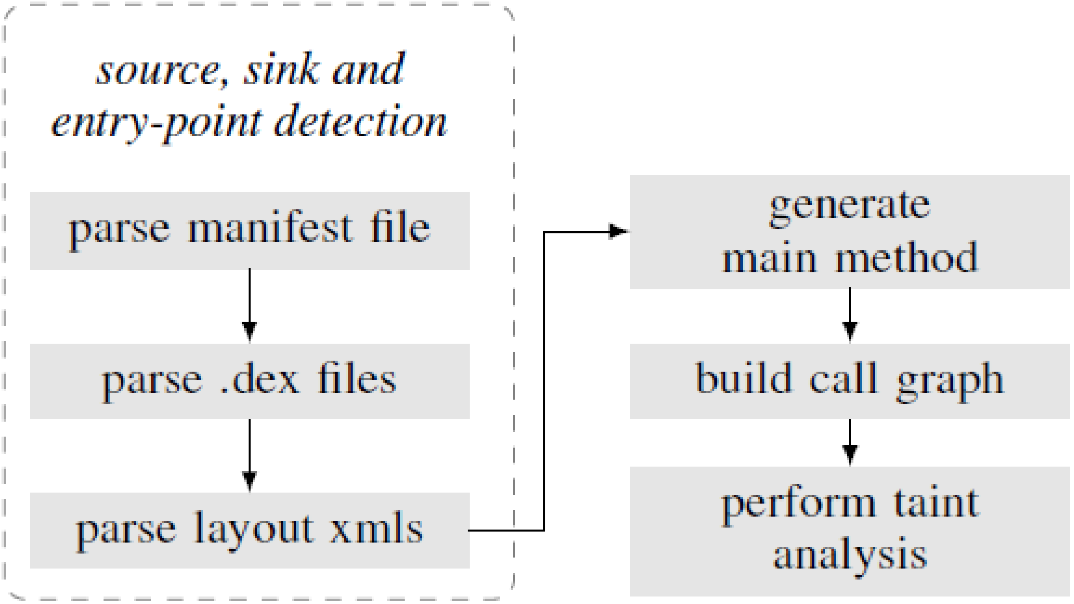
Weight = 3



Object Provenance

Code-Level Approach: FlowDroid

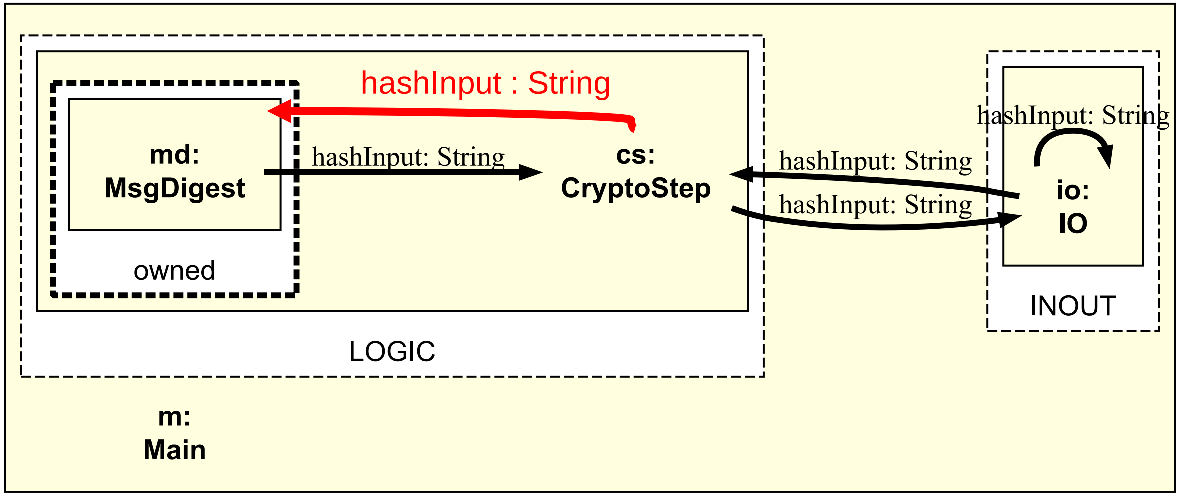
- FlowDroid [Arzt et al., PLDI, 2014]
  - Reasons about information flow at the level of variables
  - Combines extractions and constraints
  - Find a path form a source to a sink
  - Mainly developed for Android apps



ACryptographic

The SIW can notice the missing edge.

```
class CryptoStep {
   void missingStep() {
     MsgDigest md = new MsgDigest("") ;
     IO io = new IO();
     String hashInput = new String("ABCDEFG123456");
     md.update(hashInput);
     io.writeLine(io.toHex(md.digest()));
   }
}
class Main {
   public static void main(String[] args) {
     Main m = new Main();
     m.run();
   }
  void run(){
    IO io = new IO();
    CryptoStep cs = new CryptoStep();
    cs.missingStep();
   }
}
```

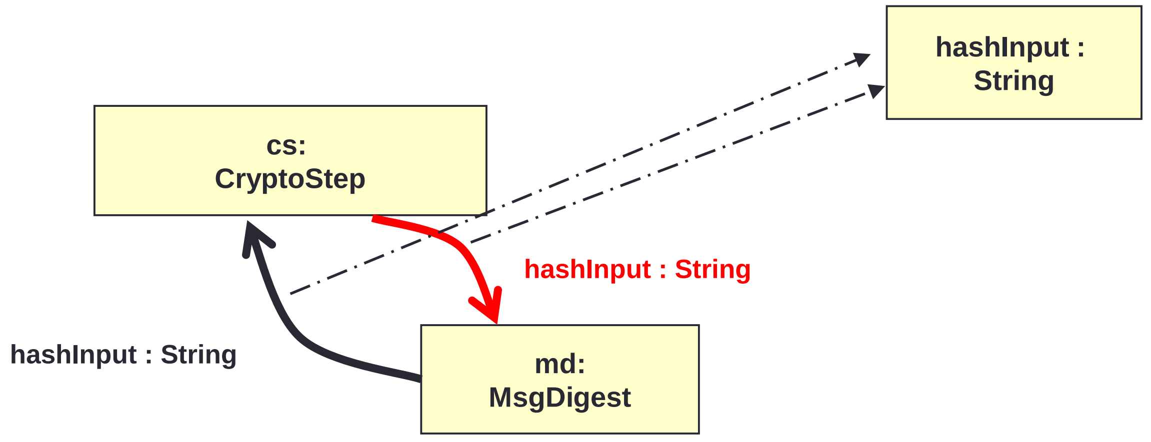


Scoria's constraint for ACryptographic

- Found it using object provenance

$$(\mathbf{MsgDigest}, \mathbf{cs.OWNED}) \xrightarrow{\mathbf{hashInput:String}} (\mathbf{CryptoStep}, \mathbf{main.LOGIC})$$

$$(\mathbf{CryptoStep}, \mathbf{main.LOGIC}) \xrightarrow{\mathbf{hashInput:String}} (\mathbf{MsgDigest}, \mathbf{cs.OWNED})$$



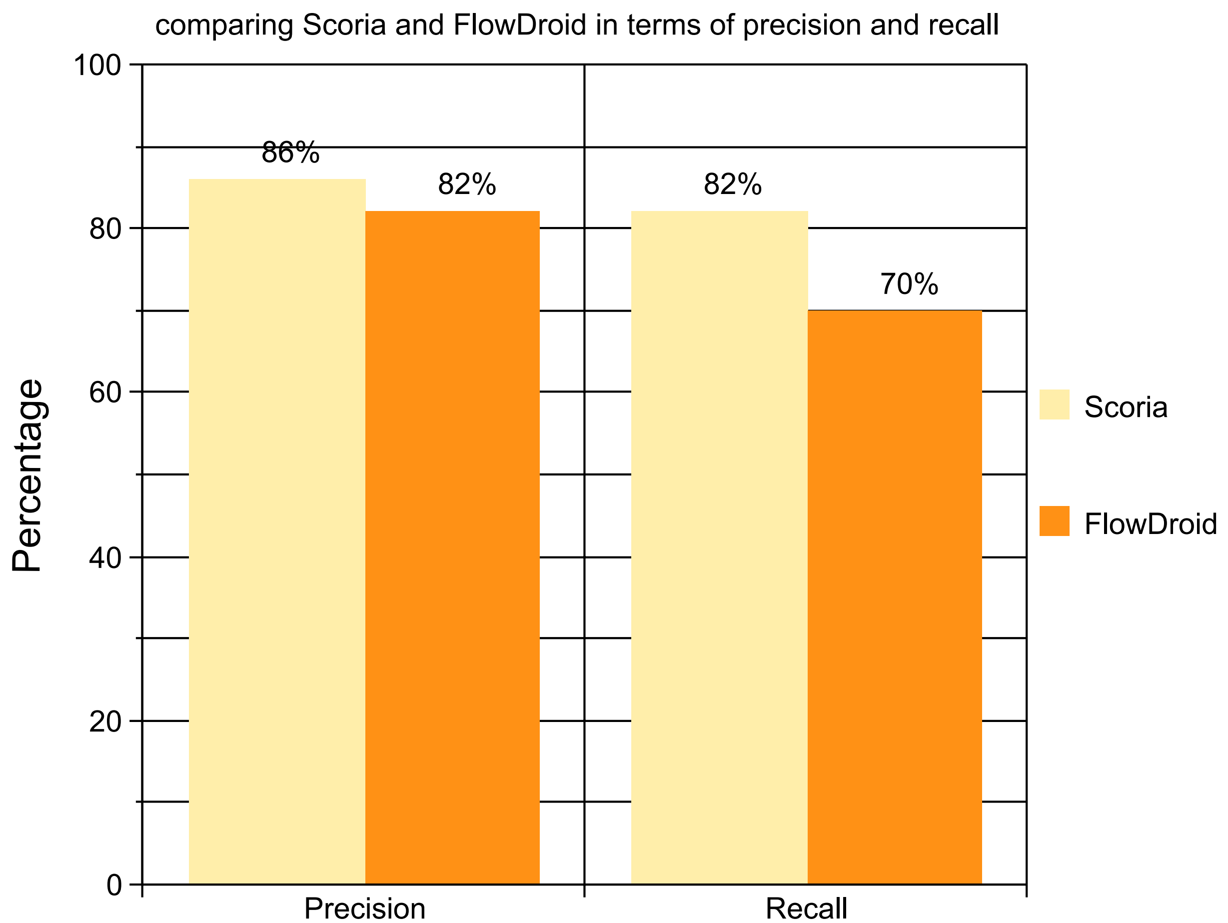
FlowDroid's constraint for ACryptographic

- The vulnerability in ACryptographic cannot be found by FlowDroid
- FlowDroid always looks for a confidential information flows to an untrusted sink
  - The confidential data that is supposed to flow to an untrusted sink is missing in ACryptographic

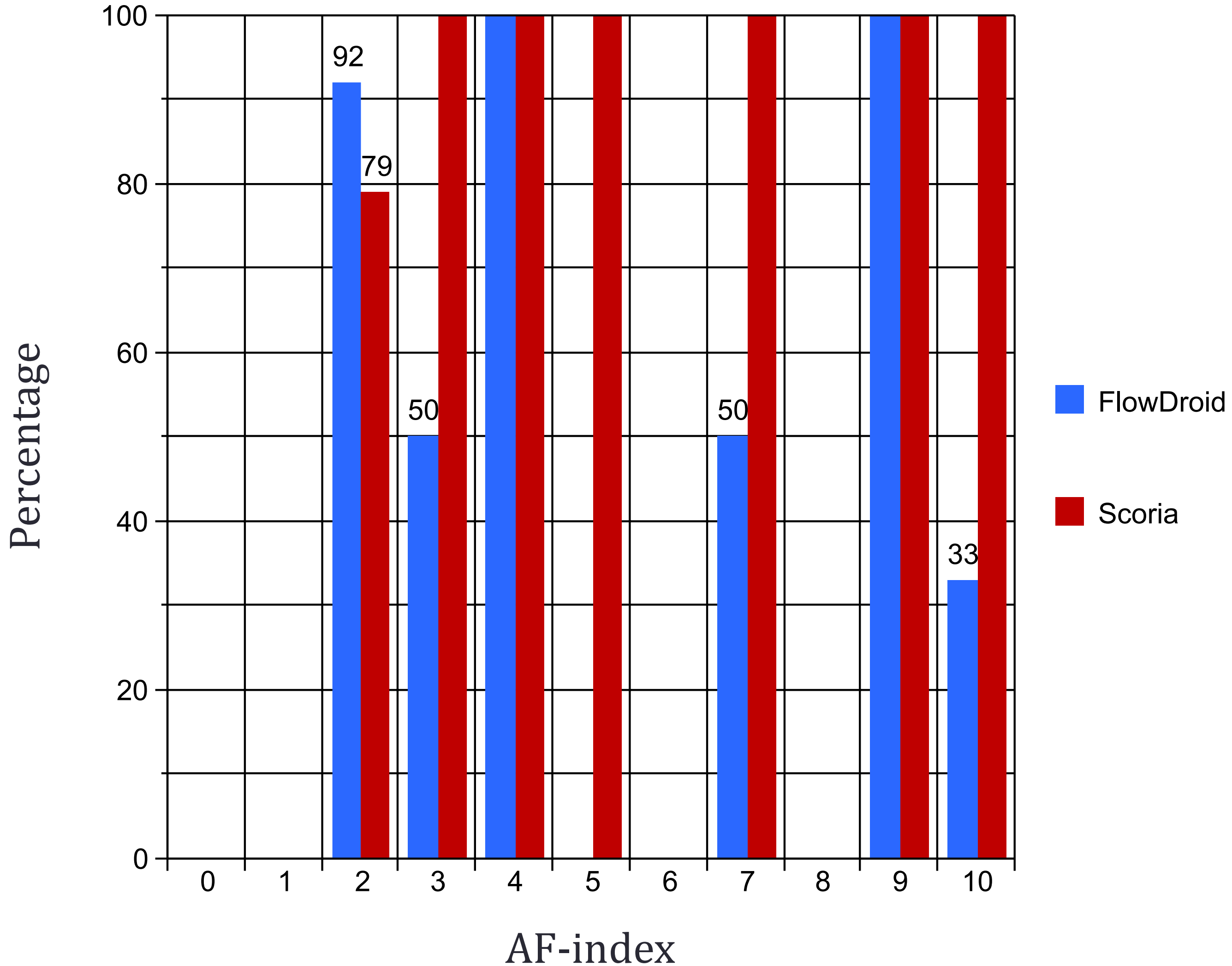
- So we cannot map the test case information flow into FlowDroid's constraint form
  - *(C1,md1, Property1)* $\longrightarrow^{*}$ *(C2,md2, Property2)*
  - Cannot write such a constraint

Legend
C: class
md: method declaration
Property: security property, e.g., Source, Sink, Sanitizer

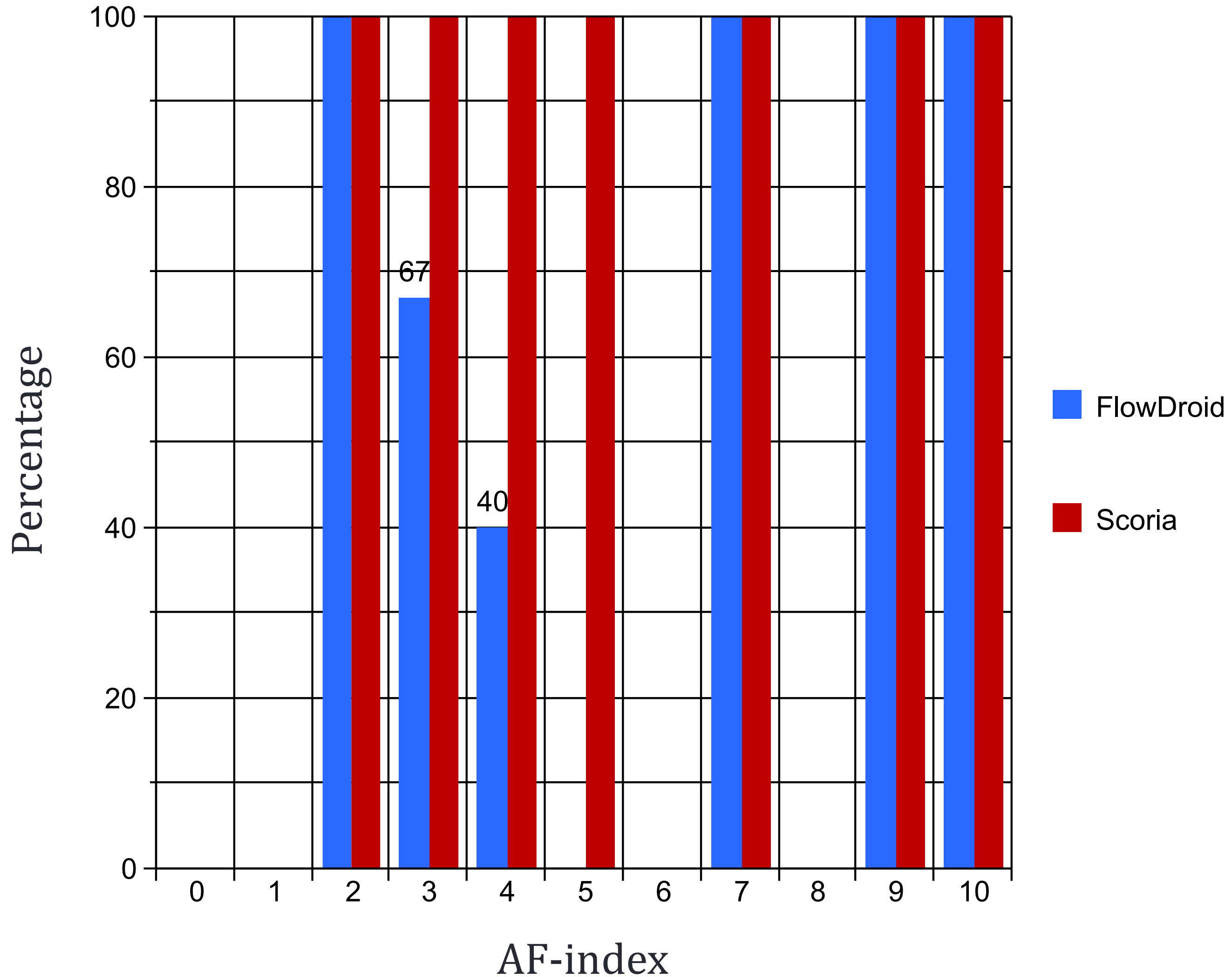
Overall Precision/Recall on ScoriaBench



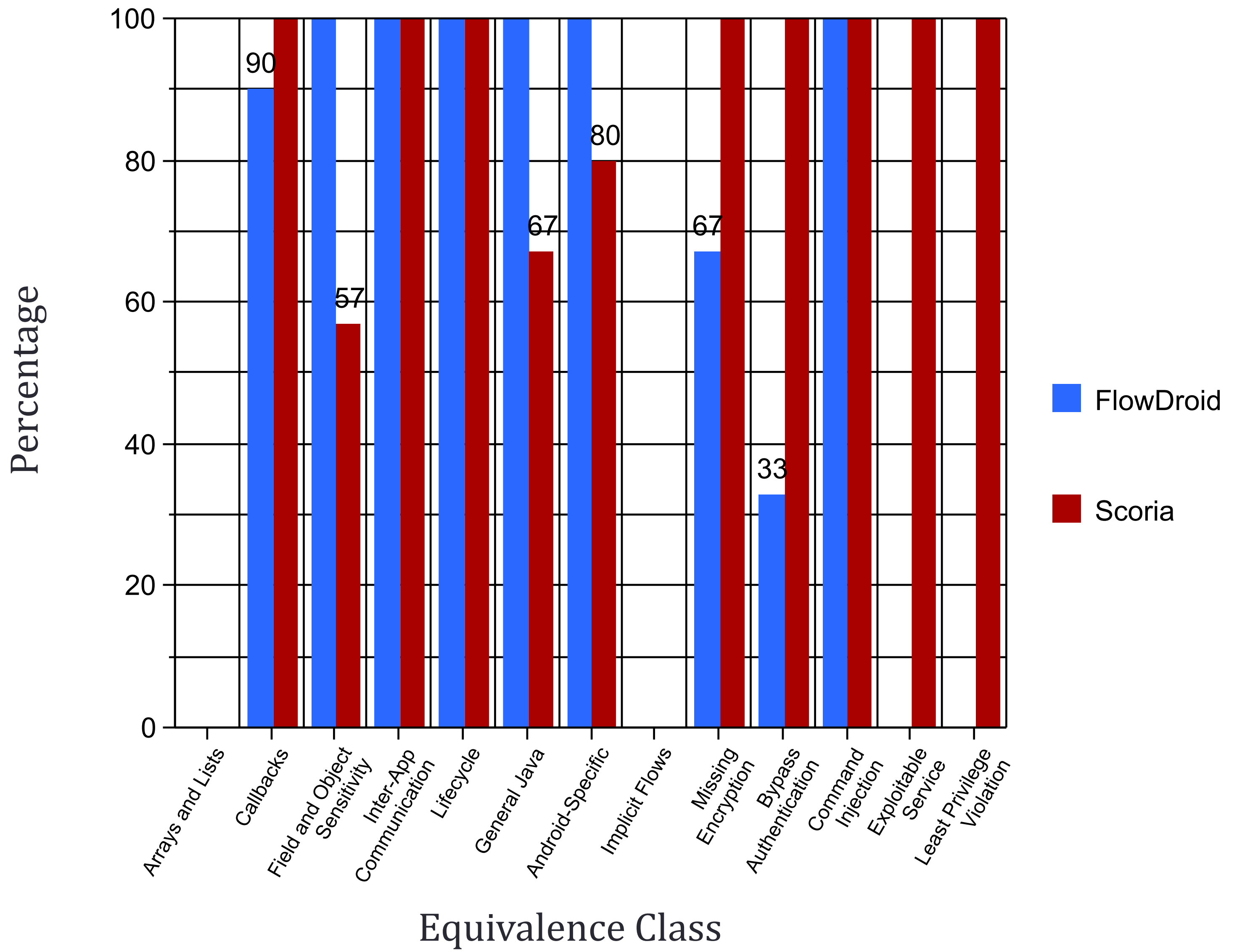
Precision per AF-index



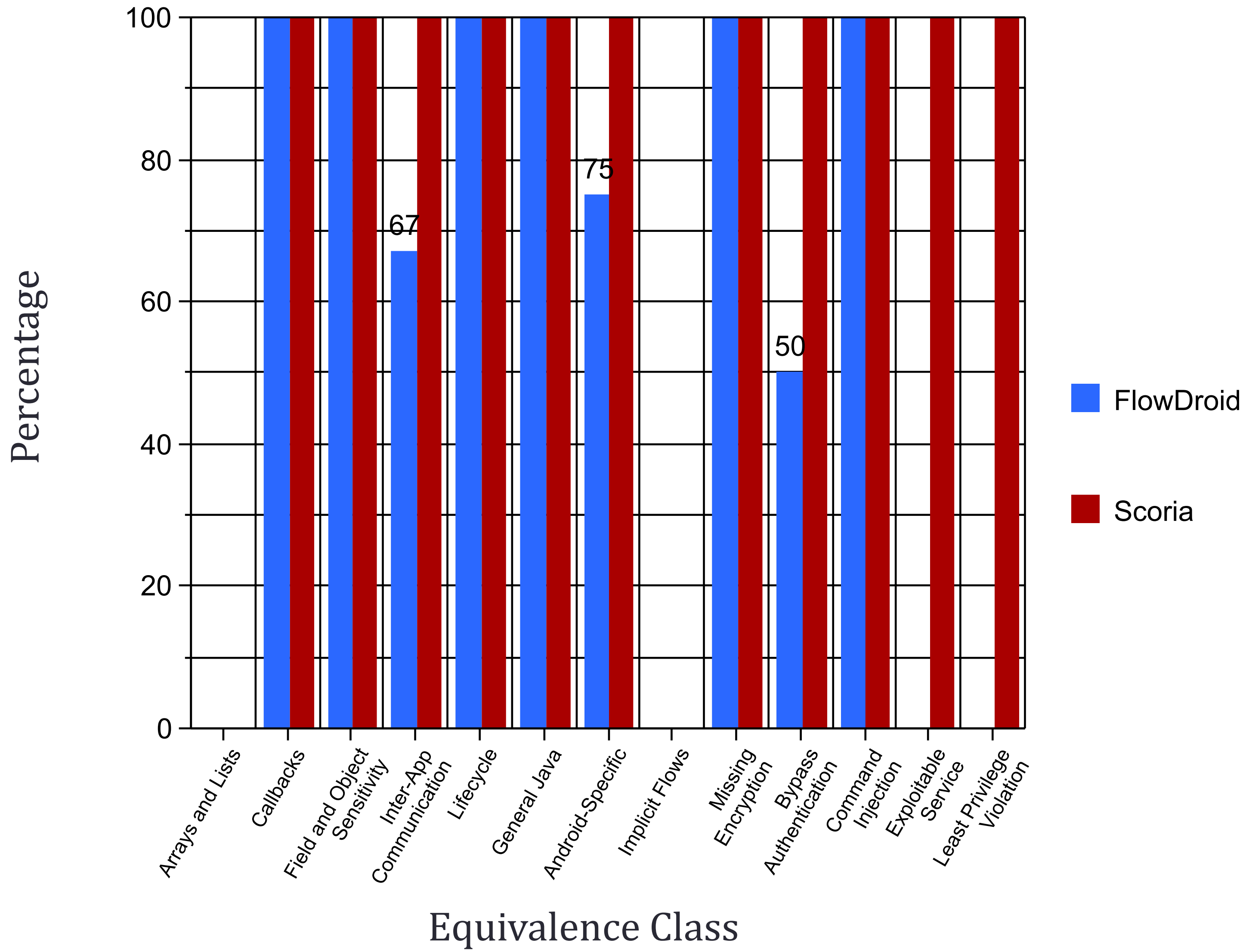
Recall per AF-index



Precision per Equivalence Class



Recall per Equivalence Class



Lessons Learned

- An architecture-level representation helps a SIW understand the system more than reading the code
- Many tools focus on coding bugs, not enough tools focus on architectural flaws
- Code-level approaches are less effective at finding architectural flaws than architecture-level ones

Detailed Results

	AF_index	provenance	hierarchy	reachability	traceability	Indirect Com.	Object. Props.	Edge. Props.	Typechecker	Extraction	FlowDroid			Scoria			Origin
											TP	FP	FN	TP	FP	FN	
Missing Encryption																	
ACipher	3	✓					0	0	0	1	0	0	0	0	0	0	SRD
ACipher2	3	✓					0	0	0	0	0	1	0	0	0	0	SRD
ASocket	7		✓			✓	2	0	2	5	1	1	0	1	0	0	CERT
ACryptographic	4	✓			✓		0	0	2	2	0	0	1	1	0	0	SRD
Bypass Authentication																	
AToken1	9		✓	✓		✓	2	0	0	6	3	0	0	3	0	0	US
AToken2	9		✓	✓		✓	2	1	0	6	1	2	0	1	0	0	US
Android Specific																	
AActivity	4			✓			2	0	0	3	0	0	1	1	0	0	US
Command Injection																	
ARuntime	2				✓		1	0	0	6	1	0	0	1	0	0	CERT
Exploitable Service																	
AChat	3	✓					0	0	0	2	0	1	1	1	0	0	US
Least Privilege Violation																	
SecretViewer2	4	✓					1	0	3	1	0	0	1	1	0	0	US

Limitations

- We did not measure **effort** and **learnability**
  - In future work, we will measure effort
  - If an approach is good but requires a lot of effort to apply, people are not going to use it

Related Work

- Security benchmarks
  - SecuriBench Micro [Livshits, 2006]
    - Focuses on injection
    - Aliasing, collection and dataflow communication
  - MalGenome [Zhou et al., SSP, 2012]
    - A collection of 1200 malware Android applications
- Applications with injected vulnerabilities
  - Web apps
    - SecuriBench [Livshits et al., USS, 2005]
      - A collection of web applications with different sizes
  - Android apps
    - InsecureBank [Paladion, 2013]
      - Information disclosure to an external memory card
- Case studies on real-world applications
  - Evaluating some other approaches [Enck et al., USENIX, 2011]
  - Dynamic analyses [Enck et al., OSDI, 2010][Falcone et al., RV, 2013]

Future Work

- Add more test cases to ScoriaBench
  - More architectural flaws
- Evaluate more approaches
  - Architectural level [Almorsy et al., ICSE, 2013]
  - Type system such as Tainting Checker [Dietl et al., ICSE, 2011]
  - Sound static analysis such as JOANA [Graf et al., ATPS, 2013]

Conclusion

- Comparative evaluation of two approaches
  - Scoria, architectural level
  - FlowDroid, code level
- Introduce AF-index
  - A measure to classify test cases on the vulnerability continuum
- ScoriaBench
  - Consists of hand-selected test cases from different sources
  - Our extensions focus on architectural flaws

Call for action

- We encourage you to evaluate your approach on the benchmark
  - If you are interested let us know
- We encourage you to contribute test cases to the benchmark
  - We will run Scoria on your test cases