PENNSTATE

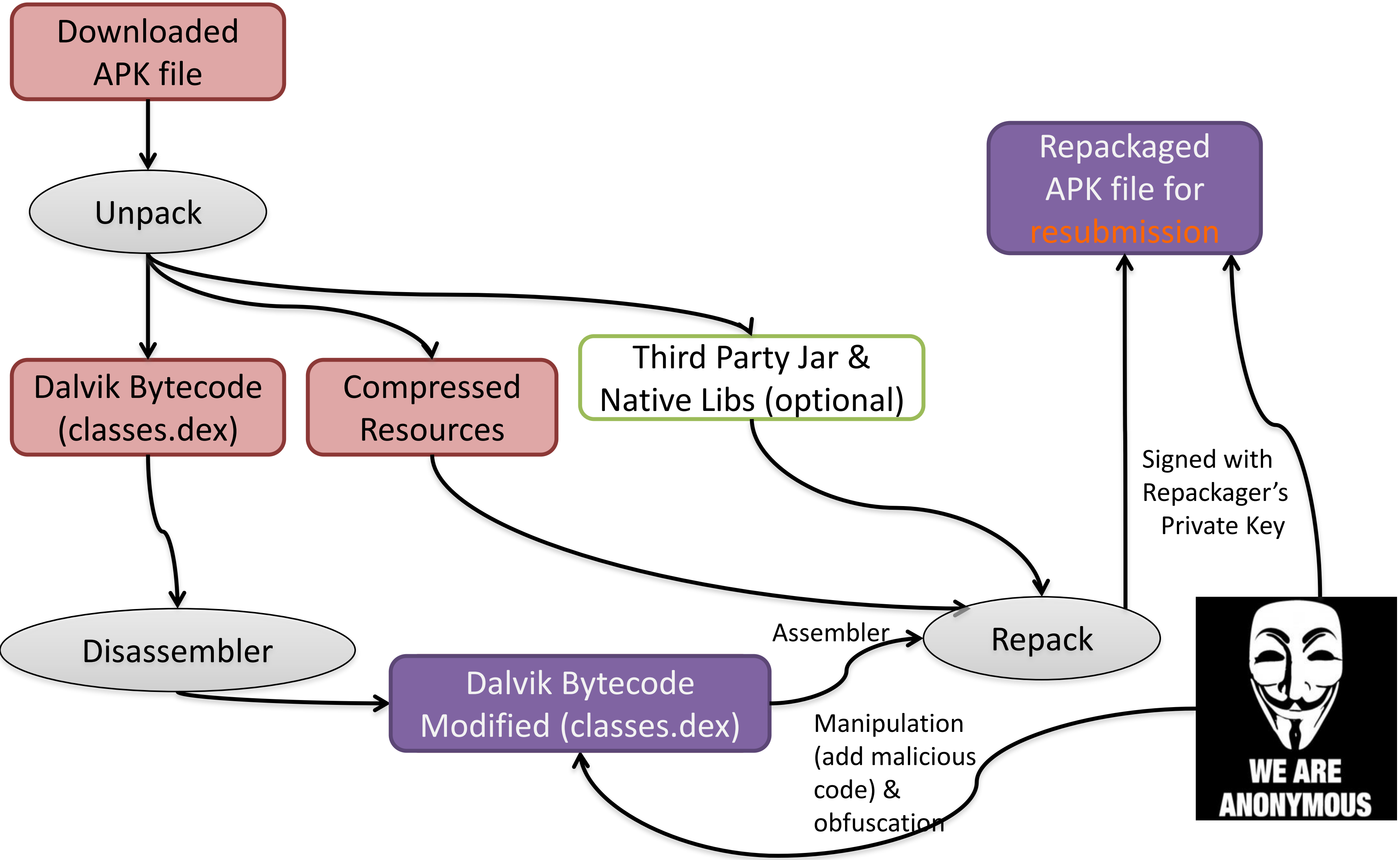
1855

A Framework for Evaluating Mobile App Repackaging Detection Algorithms

Heqing Huang, PhD Candidate.

Sencun Zhu, Peng Liu (Presenter) & Dinghao Wu, PhDs

Repackaging Process


Downloaded APK file
Unpack
Dalvik Bytecode (classes.dex)
Compressed Resources
Third Party Jar & Native Libs (optional)
Repackaged APK file for resubmission
Signed with Repackager's Private Key
Disassembler
Assembler
Repack
Dalvik Bytecode Modified (classes.dex)
Manipulation (add malicious code) & obfuscation
WE ARE ANONYMOUS

Motivation

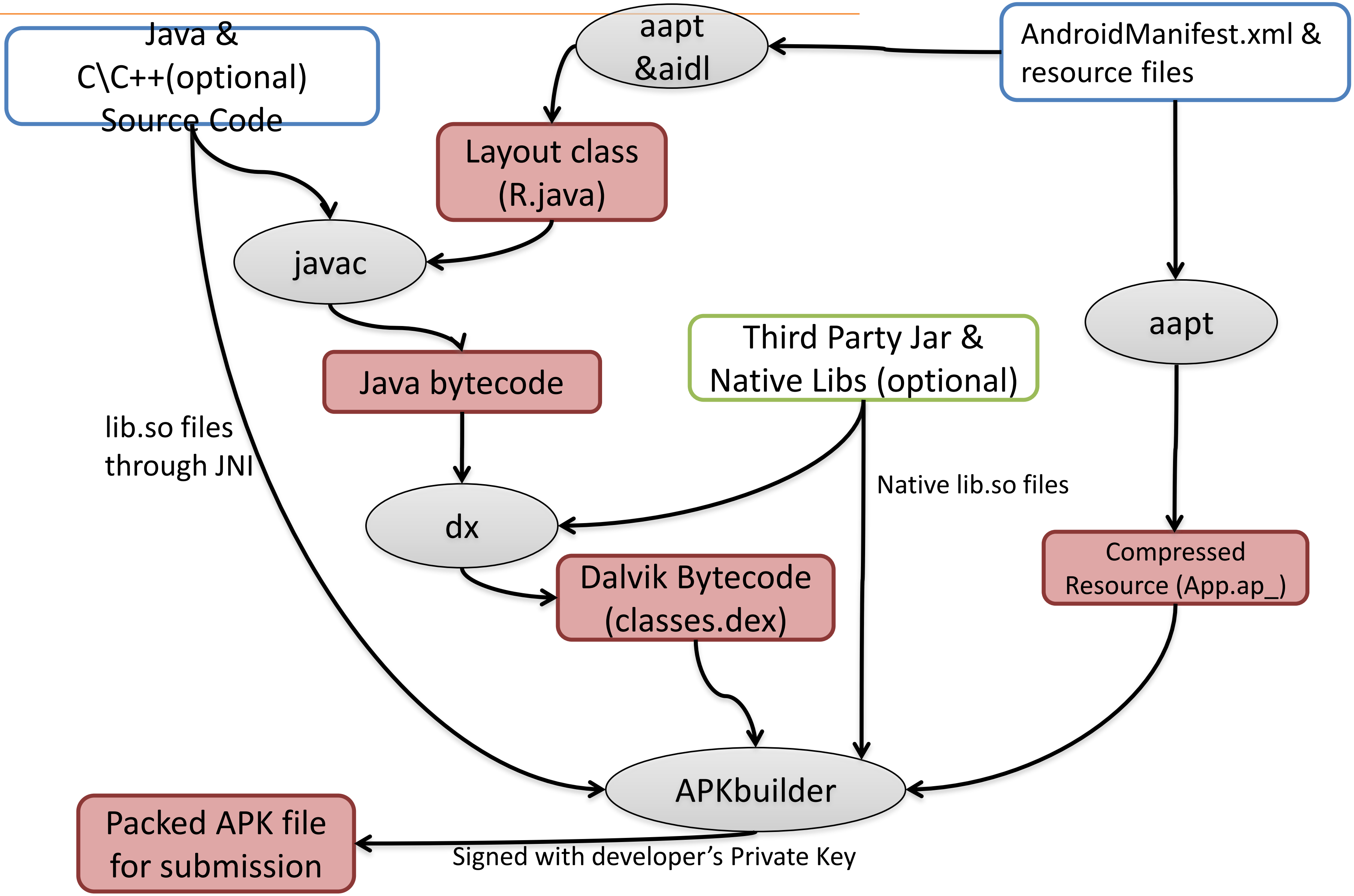
- **Android apps repackaging (plagiarism) problem**
  - 5-13% of apps in third party app markets are plagiarism of applications from the official Android market
    - 1 in 10 apps are repackaged apps!
  - And 86.0% of malware were repackaged (1083/1260)

- **Repackaging Detection Algorithms (RDAs) do exist**
  - With very ad hoc evaluation on their false negatives
  - Potential advanced code obfuscations could appear in the market at any time

Evaluation Framework

- **RDAs need false negatives evaluation**
  - What code obfuscation methods can produce more false negatives?
- **Help tune the RDAs against various code obfuscations**
  - How to choose a specific ***k*** for the k-gram based feature used in the Feature Hashing mapping?

Original Android App Developing Process


Java & C\C++(optional) Source Code
aapt &aidl
AndroidManifest.xml & resource files
Layout class (R.java)
javac
aapt
Java bytecode
Third Party Jar & Native Libs (optional)
lib.so files through JNI
Native lib.so files
dx
Compressed Resource (App.ap_)
Dalvik Bytecode (classes.dex)
APKbuilder
Packed APK file for submission
Signed with developer's Private Key

Why repackaging so attractive?

- **Dalvik Bytecode easy to be reverse engineered**
  - RE tools for automation: Basmali/Smali, Apktool and Dare, etc.
  - Dalvik Virtual Machine: register-based bytecode easy to read

Why repackaging so attractive?

----------The original bytecode pattern from Skype classes.dex ----------------

```
1. invoke-static {v1}, Ljava/lang/Integer;->valueOf(I)Ljava/lang/Integer;
2. move-result-object v1
3. const-string v2, "TYPE"
4. invoke-interface {v0, v1, v2}, Ljava/util/Map;->
      put(Ljava/lang/Object;Ljava/lang/Object;)Ljava/lang/Object;
```

Why repackaging so attractive?

- Reverse engineering is easy
- Repackaging is easy
  - Easy to insert malicious code
  - Easy to do obfuscations
- Marketing is easy
  - Self-signed Certificates without authorization
  - Little vetting on submitted apps from Google Side
  - Decentralized Markets of Android Apps

Current Repackaging Detection Algorithms (RDA)

- **RDAs**
  - Fuzzy Hashing based RDA (CODASPA’ 12)
  - Program Dependence Graph based RDA (ESORICS’ 12)
  - Feature Hashing based RDA (DIMVA’ 12)
  - AndroGuard (Blackhat’ 11)
- **False Negatives of RDAs?**
  - Specific code manipulation to blur the core features used by these detectors
  - Potential advanced obfuscations
- **False Positives**
  - Requires manual check; not a goal of our evaluation framework

Current Repackaging Detection Algorithms (RDA)

- **Fuzzy Hashing**
  - ▫ A hash is computed for each segment of opcode
  - ▫ Identify lazy repackaging efficiently

- **False Negatives**
  - – Adding noisy code chunks
  - – Use different ad libraries

Potential Obfuscation

----------The original bytecode pattern from Skype classes.dex ----------------

```
1. invoke-static {v1}, Ljava/lang/Integer;->valueOf(I)Ljava/lang/Integer;
2. move-result-object v1
3. const-string v2, "TYPE"
4. invoke-interface {v0, v1, v2}, Ljava/util/Map;->
       put(Ljava/lang/Object;Ljava/lang/Object;)Ljava/lang/Object;
```

Potential Obfuscation

----------The original bytecode pattern from Skype classes.dex ----------------

```
1. invoke-static {v1}, Ljava/lang/Integer;->valueOf(I)Ljava/lang/Integer;
2. move-result-object v1
3. const-string v2, "TYPE"
4. invoke-interface {v0, v1, v2}, Ljava/util/Map;->
        put(Ljava/lang/Object;Ljava/lang/Object;)Ljava/lang/Object;
```

--------The semantics-preserving bytecode pattern after manipulation--------

```
1. invoke-static {v1}, Ljava/lang/Integer;->valueOf(I)Ljava/lang/Integer;
2. move-result-object v1
3. move-object v3, v1          <<----- use extra virtual register "v3"
4. const-string v2, "TYPE"
5. move v4, v2                 <<----- use extra virtual register "v4"
6. invoke-interface {v0, v3, v4}, Ljava/util/Map;->
       put(Ljava/lang/Object;Ljava/lang/Object;)Ljava/lang/Object;
7. move v2, v4             <<----- update the register "v2" according to register “v4”
8. move-object v1, v3      <<----- update the register "v1" according to register “v3”
```

Current Repackaging Detection Algorithms (RDA)

- **PDG: Program Dependence Graph**
  - Identify repackaged apps with similar data dependency graph of a set of methods

- **False Negatives**
  - Resilient against dummy code insertion
  - Advanced control and data dependency obfuscators

Potential Obfuscation

----------The original bytecode pattern from Skype classes.dex ----------------

```
1. invoke-static {v1}, Ljava/lang/Integer;->valueOf(I)Ljava/lang/Integer;
2. move-result-object v1
3. const-string v2, "TYPE"
4. invoke-interface {v0, v1, v2}, Ljava/util/Map;->
       put(Ljava/lang/Object;Ljava/lang/Object;)Ljava/lang/Object;
```

Potential Obfuscation

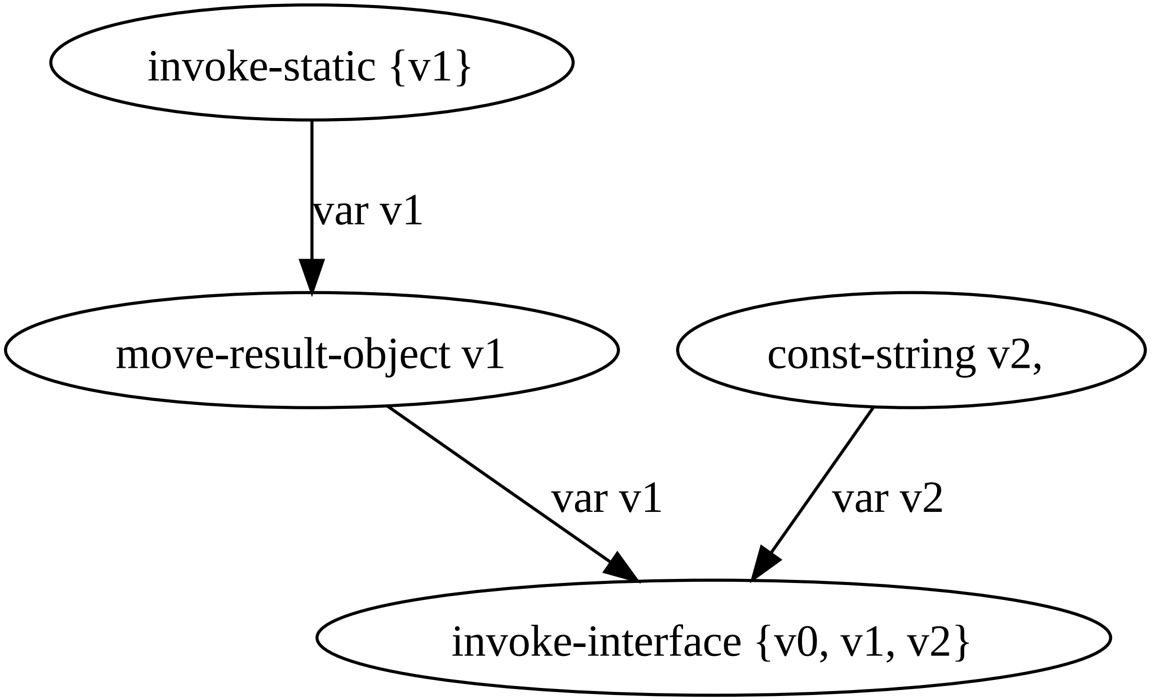
----------The original bytecode pattern from Skype classes.dex -----------------

```
1. invoke-static {v1}, Ljava/lang/Integer;->valueOf(I)Ljava/lang/Integer;
2. move-result-object v1
3. const-string v2, "TYPE"
4. invoke-interface {v0, v1, v2}, Ljava/util/Map;->
       put(Ljava/lang/Object;Ljava/lang/Object;)Ljava/lang/Object;
```

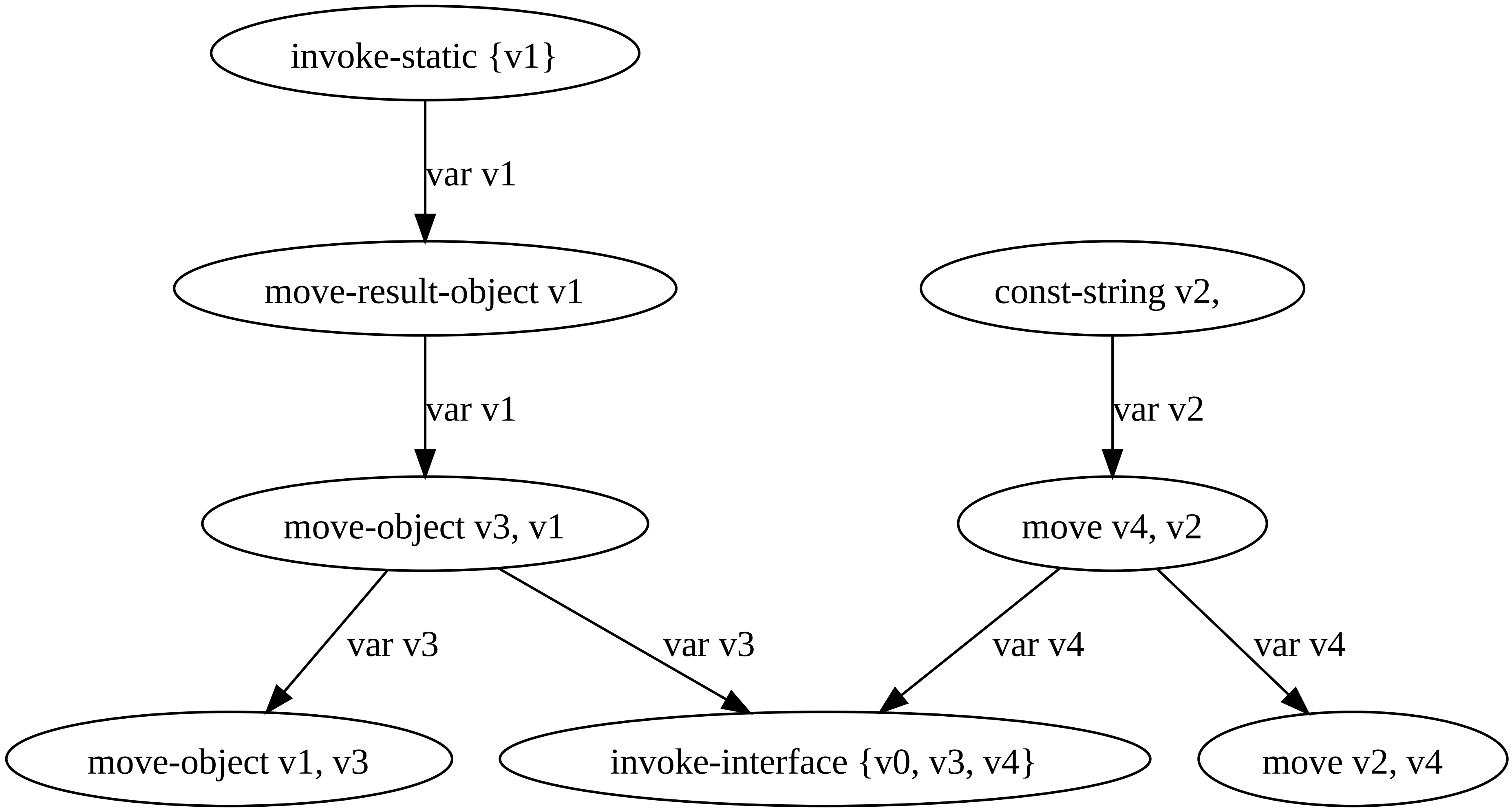
--------The semantics-preserving bytecode pattern after manipulation--------

```
1. invoke-static {v1}, Ljava/lang/Integer;->valueOf(I)Ljava/lang/Integer;
2. move-result-object v1
3. move-object v3, v1            <<----- use extra virtual register "v3"
4. const-string v2, "TYPE"
5. move v4, v2                   <<----- use extra virtual register "v4"
6. invoke-interface {v0, v3, v4}, Ljava/util/Map;->
      put(Ljava/lang/Object;Ljava/lang/Object;)Ljava/lang/Object;
7. move v2, v4          <<----- update the register "v2" according to register “v4”
8. move-object v1, v3   <<----- update the register "v1" according to register “v3”
```

Before Obfuscation



After Obfuscation



Current Repackaging Detection Algorithms (RDA)

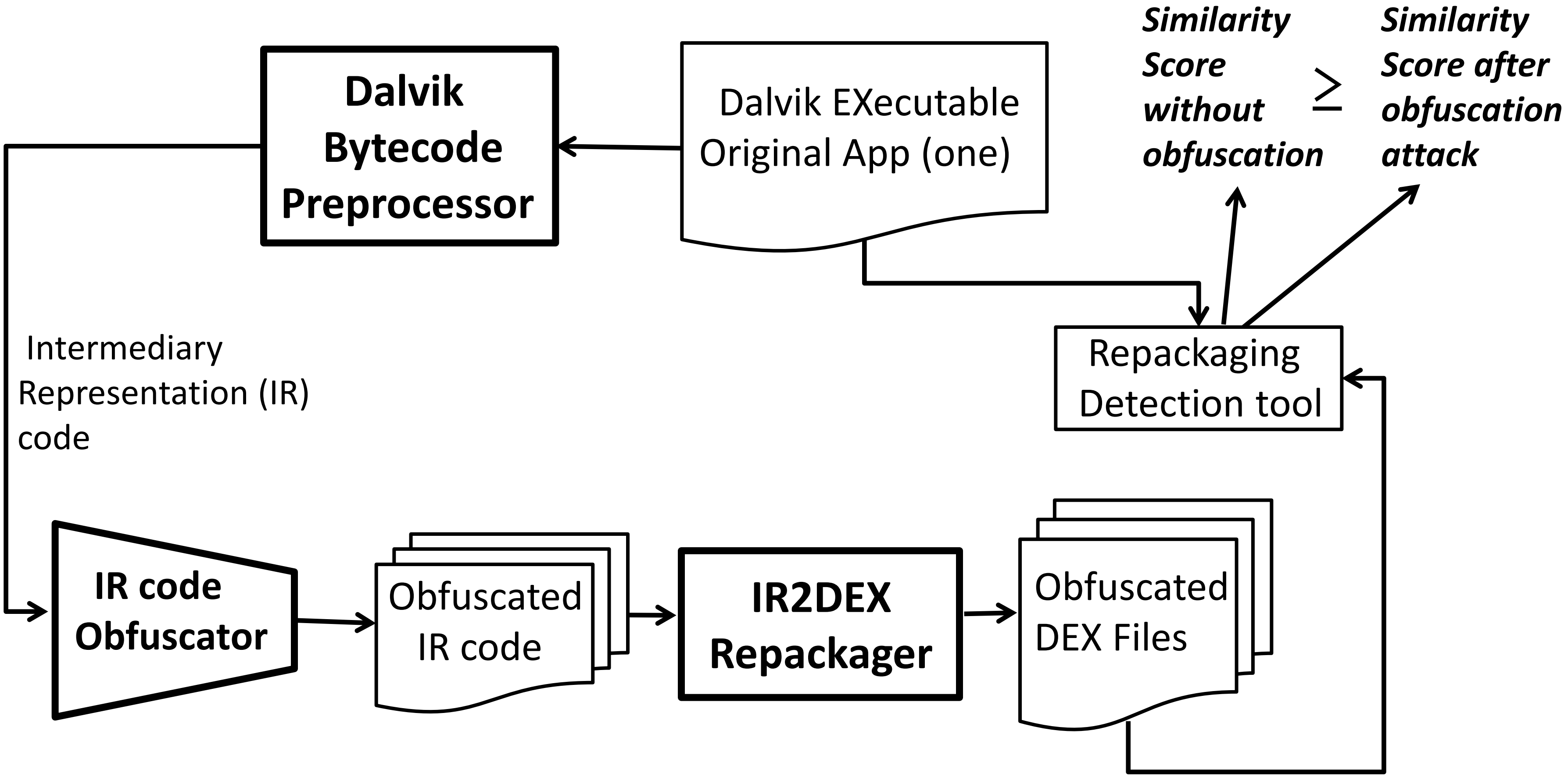
- **Feature Hashing**
  - Identify repackaged apps with similar features
  - Feature is defined as k-grams of various opcode sequence patterns within each program’s basic block

- **False Negatives**
  - Modify the normal opcode sequence patterns by code injection
  - Reduce k to defend against code injection but may raise false positives

Evaluation Framework Requirements

- **Provide a standard evaluation for RDAs**
  - RDAs based on static program analysis
  - Dalvik Bytecode as the original inputs
- **Evaluation should be efficient and effective**
- **Contain a good set of obfuscation algorithms to analyze the effective of the RDAs**
- **Can provide standard evaluation schemes**
  - Broadness and depth analysis metrics

Our Evaluation Framework



Dalvik Bytecode Preprocessor
Dalvik EXecutable Original App (one)
Similarity Score without obfuscation
≥
Similarity Score after obfuscation attack
Intermediary Representation (IR) code
Repackaging Detection tool
IR code Obfuscator
Obfuscated IR code
IR2DEX Repackager
Obfuscated DEX Files

Our Evaluation Framework

- **Dalvik Bytecode Preprocessor**
  - ▫ Convert Dalvik Bytecode into Java Bytecode
  - ▫ Optimize corresponding Java Bytecode by Soot
  - ▫ Preprocess and verify the Java Bytecode by Byte Code Engineering Library
- **IR code Obfuscator**
  - ▫ Leverage obfuscators from SandMark
  - ▫ Obfuscate programs for broadness and depth analysis
- **IR2DEX Repackager**
  - ▫ Use DX tool from Android Platform to compile the obfuscated Java Bytecode down to Dalvik EXcutable

Our Evaluation Framework

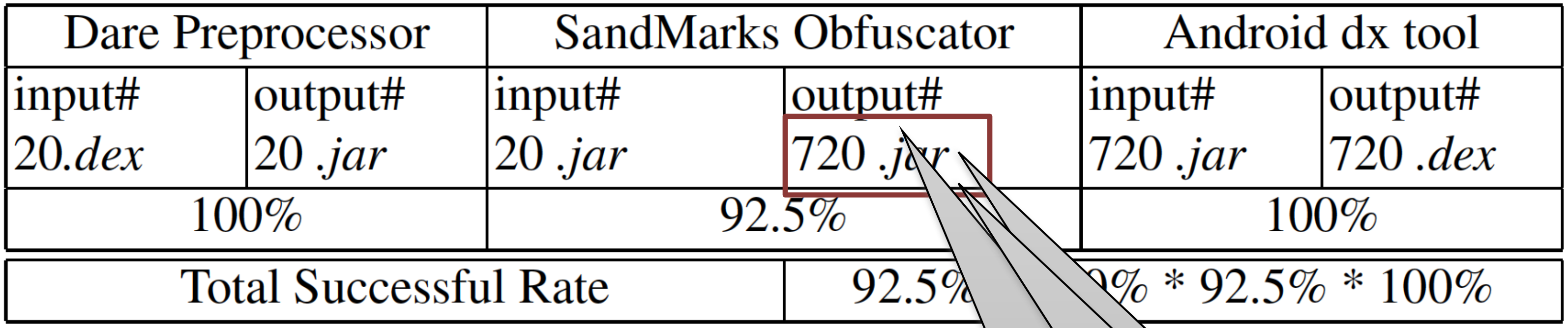
- **Broadness Analysis**
  - ▫ Perform obfuscations in a controlled manner (one obfuscator per evaluation)
  - ▫ To identify the strength and pinpoint the weakness
- **Depth Analysis**
  - ▫ Perform advanced obfuscations by serializing several obfuscators for each evaluation
  - ▫ To further analysis the obfuscation resilience of the detection algorithm

Framework Success Rates

- **Single obfuscator**
  - Perform obfuscations in a controlled manner (one obfuscator per evaluation)
  - 36/39 single obfuscators from SandMarks

Framework Success Rates

By “success”, we mean whether an evaluation workflow crashes.



Dare Preprocessor		SandMarks Obfuscator		Android dx tool	
input#	output#	input#	output#	input#	output#
20.*dex*	20 .*jar*	20 .*jar*	720 .*jar*	720 .*jar*	720 .*dex*
100%		92.5%		100%	

Total Successful Rate	92.5% 0% * 92.5% * 100%

Array rel
req
c
c
bytecod

aget-wide from Dalvik VM can be potentially mapped to iaload and faload from JVM

Framework Success Rates

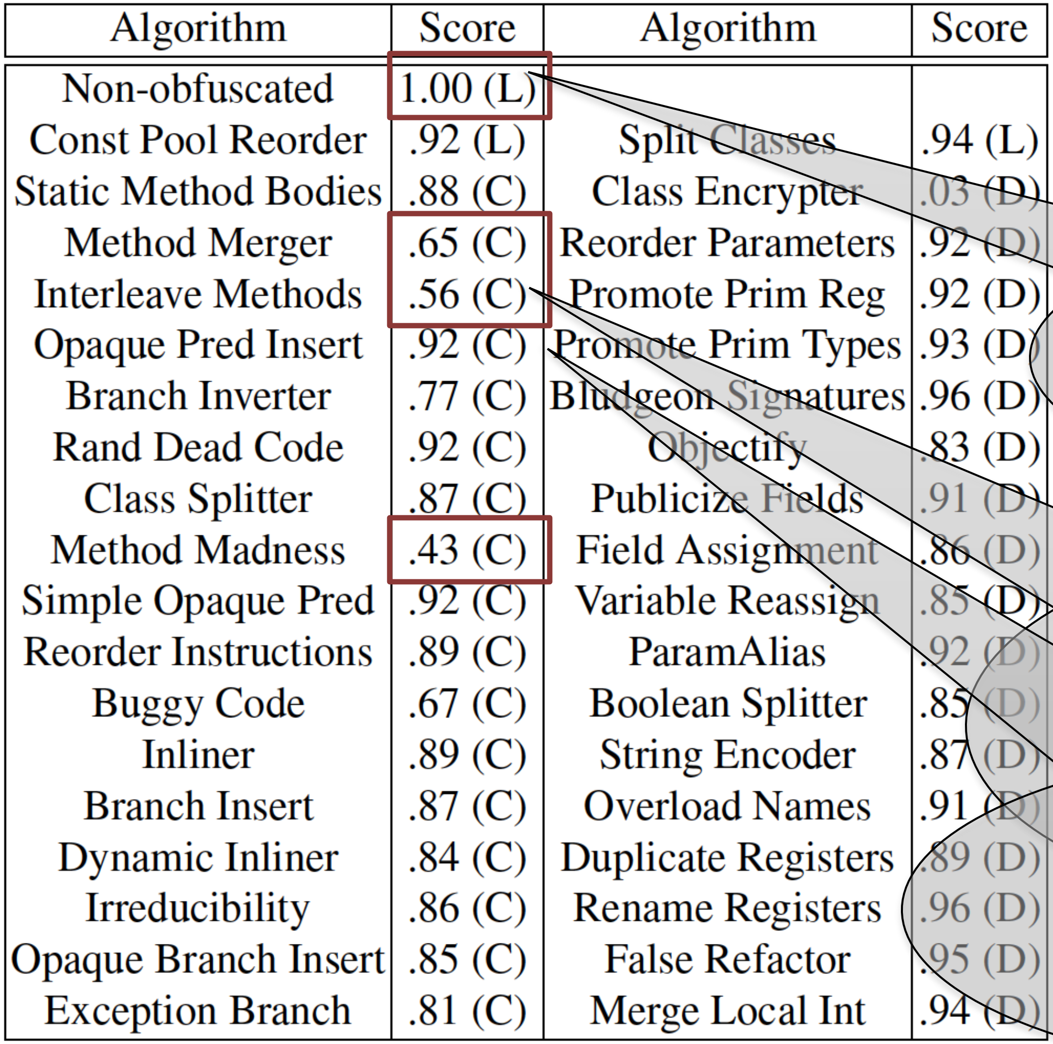
- **Multiple obfuscators**
  - Conflicts might appear among obfuscators
  - Tested various combination of obfuscators from the most effective single obfuscators

Case Study on AndroGuard

AndroGuard (The only open sourced RDA)

- Use regular expression to describe apps’ control flow structure into string
- Use Normalized Compression Distances to compare the string pairs of corresponding method pairs
- Similarity score is derived from method relevant metrics to “new method”, “diff method” and “match method”

Broadness analysis on AndroGuard



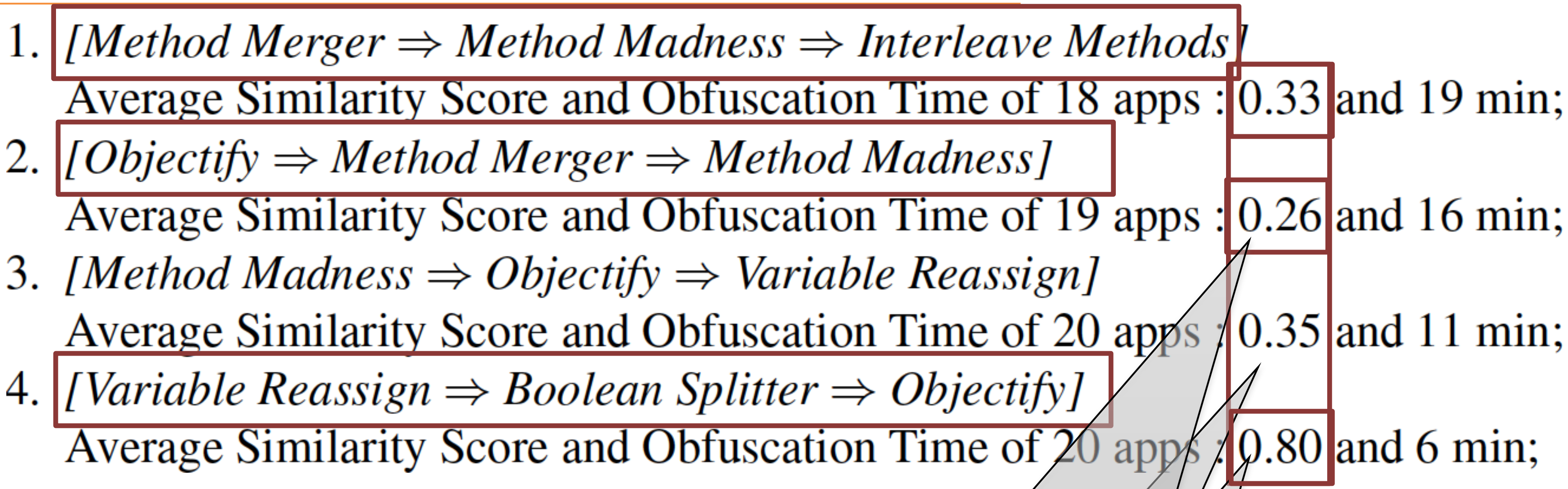
Algorithm	Score	Algorithm	Score
Non-obfuscated	1.00 (L)		
Const Pool Reorder	.92 (L)	Split Classes	.94 (L)
Static Method Bodies	.88 (C)	Class Encrypter	.03 (D)
Method Merger	.65 (C)	Reorder Parameters	.92 (D)
Interleave Methods	.56 (C)	Promote Prim Reg	.92 (D)
Opaque Pred Insert	.92 (C)	Promote Prim Types	.93 (D)
Branch Inverter	.77 (C)	Bludgeon Signatures	.96 (D)
Rand Dead Code	.92 (C)	Objectify	.83 (D)
Class Splitter	.87 (C)	Publicize Fields	.91 (D)
Method Madness	.43 (C)	Field Assignment	.86 (D)
Simple Opaque Pred	.92 (C)	Variable Reassign	.85 (D)
Reorder Instructions	.89 (C)	ParamAlias	.92 (D)
Buggy Code	.67 (C)	Boolean Splitter	.85 (D)
Inliner	.89 (C)	String Encoder	.87 (D)
Branch Insert	.87 (C)	Overload Names	.91 (D)
Dynamic Inliner	.84 (C)	Duplicate Registers	.89 (D)
Irreducibility	.86 (C)	Rename Registers	.96 (D)
Opaque Branch Insert	.85 (C)	False Refactor	.95 (D)
Exception Branch	.81 (C)	Merge Local Int	.94 (D)

It is a Layerout obfuscation Perform the conversion of JVM bytecode <-> Dalvik VM bytecode by our framework with no obfuscation

Control flow structure obfuscation performed on method granularity are more efficient to destruct the method relevant metric used by AndroGuard

Will it be better if AndroGuard calculate similarity metric on basic block granularity? Or add Data dependency similarity comparison metrics?

Depth analysis on AndroGuard



1. *[Method Merger ⇒ Method Madness ⇒ Interleave Methods]*
   Average Similarity Score and Obfuscation Time of 18 apps : 0.33 and 19 min;
2. *[Objectify ⇒ Method Merger ⇒ Method Madness]*
   Average Similarity Score and Obfuscation Time of 19 apps : 0.26 and 16 min;
3. *[Method Madness ⇒ Objectify ⇒ Variable Reassign]*
   Average Similarity Score and Obfuscation Time of 20 apps : 0.35 and 11 min;
4. *[Variable Reassign ⇒ Boolean Splitter ⇒ Objectify]*
   Average Similarity Score and Obfuscation Time of 20 apps : 0.80 and 6 min;

Combining one data flow obfuscation with the other top-2 control flow obfuscations could further reduce the similarity score

The top-3 control flow structure obfuscations is much more power than the top-3 data flow obfuscations

All the serialized obfuscations further bring the similarity score to a low level

Limitations and Future work

- **Support only static analysis based RDAs**
  - Try to enhance the framework for dynamic analysis based RDAs
- **Not all the obfuscators can be completed successfully**
  - Leverage other obfuscation tools
  - Try to fix the type inference and other bugs from the current Dalvik bytecode preprocessor

Conclusion

- Security research requires benchmarks
- A framework to check the potential FNs of RDAs
- Propose Broadness and Depth evaluations to pinpoint the weakness of the RDAs
- Help tune the RDAs' design and configuration

Thank You!