Defeating Android security solutions by exploiting fuzzy hashing (updated)

Arash Vahidi
RISE

Øresund Security Day, 2019

The early morning opening slide...

There are two types of crypto talks...

1. " We present a third preimage attack on reduced round Shenanigans-256, improving attack complexity from 2^{∞} to a much more reasonable $2^{\infty * 0.91}$ ".
2. " We poked at this thing until it fell apart ".

Motivation

- Automatic analysis of foreign files is sometimes the only line of defence in computer systems.
- A number of Android security tools depend on reliable automatic analysis of apps (APKs).
- But how can a computer program learn to recognize classes of unwanted/vulnerable/malicious software? And how easy can it be fooled?

We consider two types of threats

1. **Concealment**: a component is not detected
2. **Forgery**: a component is misidentified

Example

- Facebook SDK libraries included in many apps call home without user consent[1].
- Android privacy & security tools such as REAPER, PINPOINT, SweetDroid, and ART attempt to isolate the offending library and cut its access to your data and the network.

[1] https://privacyinternational.org/report/2647/how-apps-android-share-data-facebook-report

Current approaches

- *"Reliable Third-Party Library Detection in Android and its Security Applications"*, Backes et al. use a Merkle tree on simplified bytecode.
- *"Orlis: Obfuscation-Resilient Library Detection for Android"*, Wang et al. use a two stage detection methods using two different fuzzy hash algorithms.
- *"LibRadar: Fast and Accurate Detection of Third-party Libraries in Android Apps"*, Ma et al. use a Merkle tree on class API calls.
- *"LibD: Scalable and Precise Third-party Library Detection in Android Markets"*, Li et al. use a Merkle tree on CFG hash chain.

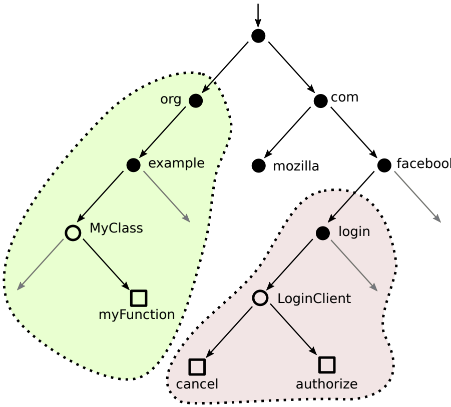
Regarding targeted algorithms...

(this page was added after the presentation)

As mentioned during the presentation, we noted that the published description does not always match the provided implementation.

Since the goal of this paper is demonstrating fuzzy hashing issues, we will consider two generic approaches (A and B) and will make no further claims about breaking any particular algorithms.

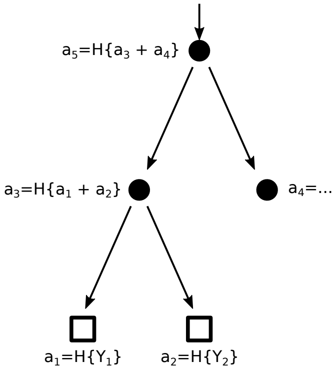
Anatomy of an Android application



```
.method authorize()
const/4 v2, #22
mul-int v1, v2, v3
move v4, v3
...
```

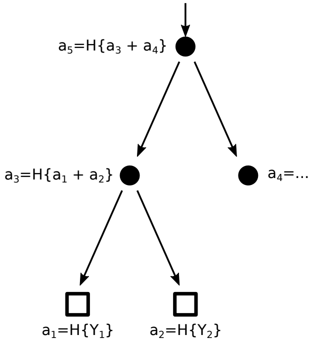
Hash trees

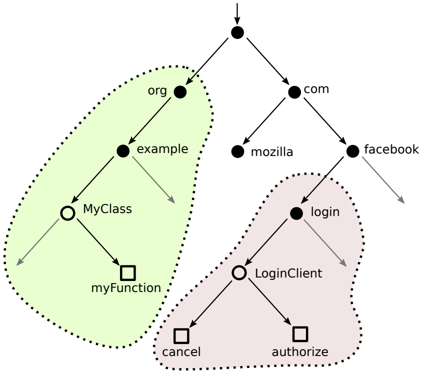
In a **hash tree** each node has a message that is a combination of that nodes data and labels of its children. A nodes label is the digest of this message:



(Merkle trees are a variation of hash trees)

Do you see where this is going?





A naive identification strategy

A naive approach would be to store label of all library nodes in a database. Unfortunately, this approach is very fragile due to the following issues:

1. Package, class and method names may have been obfuscated (or faked) [2]
2. Use of different toolchains and optimization options
3. Minor changes to the code

Hence a method is needed that allows *similar* components to be identified as "equal".

[2]For example, com.facebook.login.LoginClient.authorize() could be stored as a.b.c.A.b()

Fuzzy hashing

A fuzzy hash H_ϕ is a context-aware hash that satisfies the following property: Given two unequal inputs ($f_1 \neq f_2$), the probability of $H_\phi(f_1) = H_\phi(f_2)$ should be higher the more similar the two are:

$$\phi(f_1, f_2) \geq 1 - \epsilon \,, \; 0 < \epsilon \ll 1$$

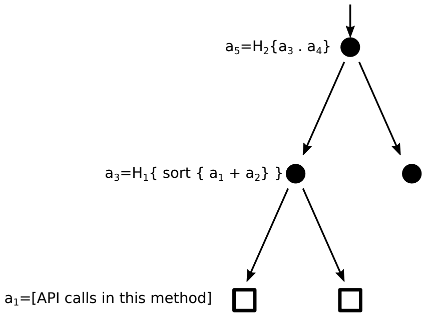
A common example is

$$H_\phi(x) = H(C(x))$$

where H is a normal hash function and C is a context aware lossy compression.

Approach A

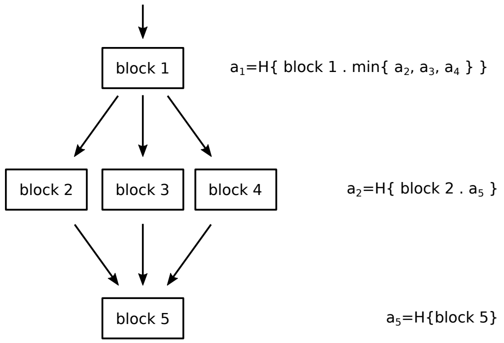
The first approach only consider calls to framework API:


a_5=H_2{a_3 . a_4}
a_3=H_1{ sort { a_1 + a_2} }
a_1=[API calls in this method]

The rationale behind this idea is that the calls to the Android APIs should represent a good summary of what the class does. Note that sorting is required since order in the bytecode may change due to obfuscation.

Approach B

With approach B, the leaf label is computed from the control flow graph (CFG) of the corresponding methods. A block contains simplified version of the bytecode where (almost) all instruction parameters have been removed:


block 1
block 2
block 3
block 4
block 5
a_1=H{ block 1 . min{ a_2, a_3, a_4 } }
a_2=H{ block 2 . a_5 }
a_5=H{block 5}

The idea behind this design is to discard some details in each method but still retain the core structure.

Concealment

For approach A, in each package that has no sub-packages ones adds a new class or performs an API call.

1. $a'_1 = [API_0, API_1, ..., \underline{, API_{666}}]$
2. $a'_5 = H\{a_3.a_4\underline{.a_{666}}\}$

For approach B, one can modify or re-arrange the code to affect at least one block that contributes to the final output. For example:

1. $min\{a'_2, a_3, a_4\} \neq min\{a_2, a_3, a_4\}$
2. $block1' \neq block1$

Note that we must ensure our modifications are not removed by the obfuscator / optimizer.

Concealment - example

Add the following code to the very beginning of a random function to defeat both approaches:

```
Date d = new Date();
if(d.getMonth() == 42) {                     // false
    Animator a = new Animator ();            // API call
    System.out.println(a.isRunning()); // use it
}
```

Forgery - approach A

The key to forgery in approach A is to remember that only classes with API calls contribute to the final label. Hence we will use the following recipe:

1. Select a victim library that uses a superset of the required API
2. Empty all classes, move API calls to dedicated methods
3. At this point the library should retain it's original label
4. Populate the empty classes with own code
5. Instead of making direct API calls, use the dedicated functions as proxies

(this makes some assumptions about class inheritance that may not always hold)

Forgery - approach B

Approach B ignored two types of data: bytecode parameters (e.g. A and B in "mov A, B") and CFG blocks that have a sibling with a smaller label. The forgery attack uses this to create two disjoint paths, one executed and one measured:

1. Select a victim library with a large number of methods that start with an if-statement
2. In each victim function, find the first ignored block
3. Change the branch condition to always execute this block
4. Replace the victim block with own code (can be of any size, as long as its label is larger)

This requires more computation than forgery for approach A.

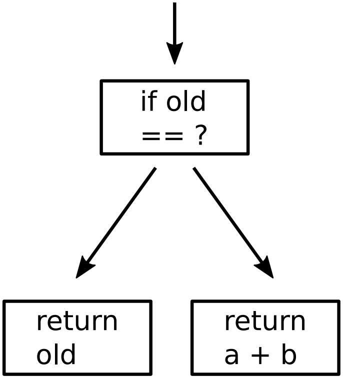
Forgery - approach B - example

```
void method1(old, a, b, ...) {
 if(old != 0)
  return old;
 else
  return a + b;
}
```

a_1=H{ "if old == ?" . min{ a_1, a_2} }

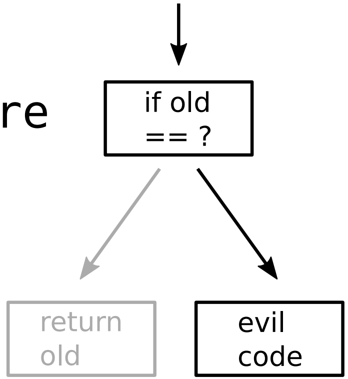
a_2=H{ block "return old" }

a_3=H{ block "return a + b" }



Forgery - approach B - example

```
void method1(old, a, b, ...) {
 if(old == 5)
  return old;
 else {
  // evil code here
 }
}
```



a_1=H{ "if old == ?" . a_1}

a_2=H{ block "return old" }

Countermeasures

The main problem in these examples was that the behavior of the compression function $C(x)$ could easily be anticipated and circumnavigated. To avoid such trivial attacks we recommend that:

1. More narrow properties of x are included, and it possible a fuzzy parameter or threshold is applied
2. $C(x)$ includes multiple overlapping properties of x
3. $C(x)$ does not rely on properties that are easily translated to code

While their quality and attack resilience is yet to be tested, this might be a good fit for certain algorithms that use machine learning and extract features from a large pool of feature candidates.

THANK YOU