# Tackling Android's Native Library Malware with Robust, Efficient and Accurate Similarity Measures

Anatoli Kalysch (**Speaker**), Oskar Milisterfer, Mykolai Protsenko, Tilo Müller

August 29, 2018

Friedrich-Alexander-Universität Erlangen-Nürnberg
Department of Computer Science
IT Security Infrastructures Lab
Software Security Research Group

ARES Conference
*International Conference on Availability, Reliability and Security*

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

# Outline

# Obfuscation on Android

| Year | Obfuscation Technique |
|---|---|
| 2011 | Symmetric Encryption (partly custom) String Encoding |
| 2012 | Proguard, Steganography, Dalvik Level Encryption |
| 2013 | Protector (Dexguard), Non-dalvik Encryption |
| 2014 | Packers, Protectors, and Native Code |
| 2015 | Packers, Protectors, and Native Code comb. w/ Obfuscators |

Table: Malware obfuscation chronology (excerpt) [4].

# Solutions for your obfuscation needs

| Packer/Protector | Obfuscation Techniques | Native Library |
|---|---|---|
| Dexguard | obfuscation, hooking, anti-dynamic | ✓ |
| Aliprotect | native and dex obf. | ✓ |
| Tencent | native and dex obf., MLU | ✓ |
| Qihoo | native and dex obf., MLU | ✓ |
| Bangcle | native and dex obf., hooking, MLU | ✓ |
| Ijiami | native and dex obf., MLU | ✓ |

Table: Protection Measures in Packers and Protectors (excerpt) [3].

# What makes native code so popular?

- Written in C/C++ and compiled, meaning no smali byte code is available.
- Huge performance boost if executed on the Dalvik Virtual Machine (DVM), minimal performance boost on the Android RunTime (ART).
- Direct usage of system resources (permission model still applies) and ability to manipulate own process components.
- Breaks most Android reverse engineering tools, and less meta data is available compared to smali byte code making reverse engineering harder

# Tackling malicious native libraries on Android

- Need for a solution to the threat posed by malicious native libraries. And ideally this solution is
  - automated,
  - accurate,
  - efficient, and
  - robust (regarding code obfuscation).
- Currently we see wide employment of code-similarity measures to detect known malicious code, e.g., hash and signature-based solutions.

# Outline

# Process Overview

- Create a 3D vector for every function in the native library based on the control-flow graph (CFG)
- From the 3D vectors we create a centroid from the sum of its edge weights
- The centroids only differ if the underlying functions differ as well
- This encoding introduces an abstraction layer that disregards certain obfuscation techniques

# Creation of a 3D vector

Each basic block (BB) in the CFGs is given a coordinate in the three dimensions

- sequence,
  - defining the order in which basic blocks (BB) of the CFG are executed
- selection, and
  - represents the number of outgoing edges for each BB
- repetition.
  - reflecting the loop depth of the current basic block

After all the BBs were assigned coordinates in the 3D system a unifying vector can be created.

# Creating Centroids

A Centroid of a 3D-CFG vector is defined as

$\vec{c} =< c_x, c_y, c_z, \pi >$, with

$$c_x = \frac{\sum_{e(p,q)\in 3D-CFG}(\pi_p x_p + \pi_q x_q)}{\pi},$$

and $c_y$ and $c_z$ accordingly [1].

The $\pi$ coordinate is encoded as $\pi = \sum_{e(p,q)\in 3D-CFG}(\pi_p + \pi_q)$ where $e(p,q)$ refers to an edge in the 3D-CFG, which connects the two nodes $p$ and $q$.

# Comparison

- Due to monotonicity properties of centroids [2] the same methods will be mapped to the same centroid
- Centroids are sortable [2], enabling a faster comparison
- Comparison of two centroids is performed through the computation of the Centroid Difference Degree (CDD)

## Definition (Centroid Difference Degree)

Given two centroids, $\vec{c}$ and $\vec{d}$, the CDD is computed as

$$CDD(\vec{c}, \vec{d}) = \max\left(\frac{|c_x - d_x|}{c_x + d_x}, \frac{|c_y - d_y|}{c_y + d_y}, \frac{|c_z - d_z|}{c_z + d_z}, \frac{|\pi_c - \pi_d|}{\pi_c + \pi_d}\right).$$

# Putting Centroids to Work on ARM Libraries

- Before application to ARM the right combination of variables and weights needs to be found
- Heuristics for a sane CDD need to be found / defined
- Equally, a Library Similarity Degree (LSD) needs to be defined

# Outline

# The Dataset and Malware Families

- 3rd party APK Stores
  - 18 different app stores
  - 508,745 apps
  - 2,346,005,582 methods
- 29 Malware-Families including
  - Bios.A
  - DroidDream
  - Godless
  - KungFu
  - OldBoot
  - Rootnik
  - TatooHack
  - VikingHorde
  - Ycchar

# Accuracy

- Detection of library versions
  - comparisons of 1,500 unrelated library pairs
  - testing different pairs of CDD / LSD yielded false positive rates (FPR) as low as 1% for libraries with more than 100 functions
  - singnificantly small libraries with less than 100 functions performed worse with FPR around 10%
- Database clustering
  - 146,264 native libraries from 40 size-based clusters were categorized into 4,201 clusters
  - A name-based library comparison and in some cases a method level CFG comparison concluded FPRs of less than 2%

# Efficiency - Computation

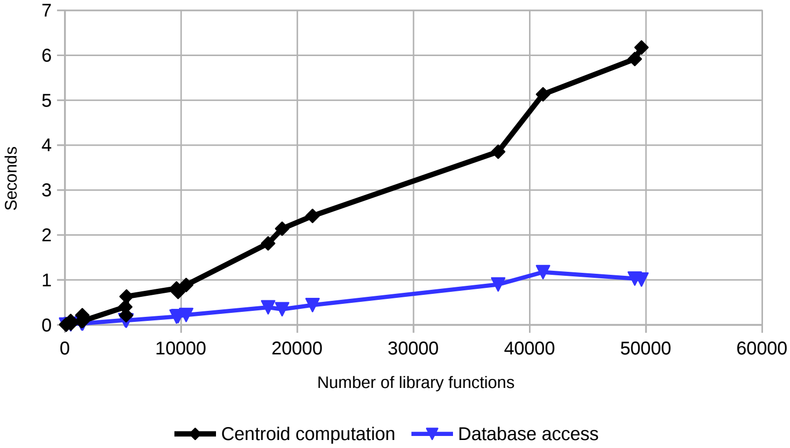


Figure: Computation of a centroid with and without database access.

# Efficiency - Comparison

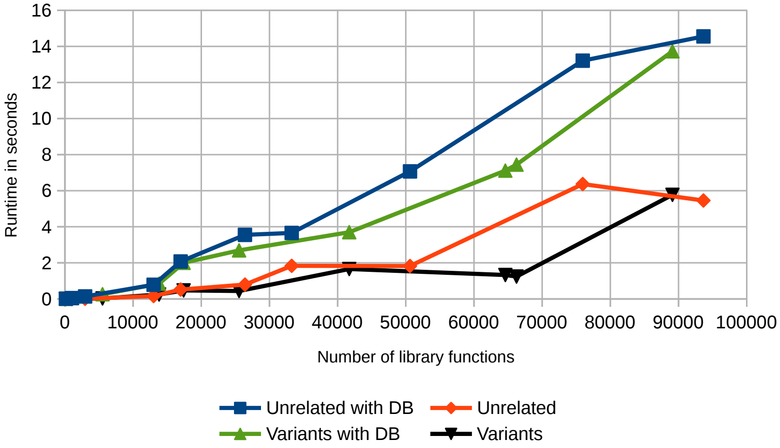


Figure: Comparison between related and unrelated libraries.

# Robustness to Obfuscation

| Obfuscation Technique | Category | Detection |
|---|---|---|
| modified APK meta data | string-based | ✓ |
| native library relocation | file hiding | ✓ |
| native library renaming | string-based | ✓ |
| variable name obfuscation | string-based | ✓ |
| binary stripping | string-based | ✓ |
| native library payload placement | code insertion | ✓ |
| junk function insertion | code insertion | ✓ |
| literal/arithmetic encoding | code insertion | ✗ |
| BB segment reordering | control flow obfuscation | ✓ |
| opaque predicates | control flow obfuscation | ✗ |
| function in/outlining | control flow obfuscation | ✗ |
| control flow flattening | control flow obfuscation | ✗ |

# Findings

| Market Name | Malicious NLs | Detected NLs |
|---|---|---|
| playmob | 26 | 1089 |
| mumayi | 36 | 151 |
| baidu | 44 | 368 |
| apkmirror | 80 | 3393 |
| nduo | 128 | 396 |
| up2down | 219 | 4195 |
| apkworld | 307 | 1880 |

Table: Selection of detected malicious native libraries among ARM 32-bit native libraries.

# Comparison to VirusTotal

- APKs from detected malicious clusters were uploaded to VirusTotal
  - Roughly half were detected as malicious
- Next we extracted the native library and uploaded it to VirusTotal as well
  - Note that we analyzed malware that actively uses native code for exploitation
  - less than 4% were considered malicious

# Outline

## Android Malware is Going Native
### Android Obfuscation in Context
### Why Native Libraries?

## Introducing Dimensional Encoding
### Centroids
### Comparison Procedure

## Bringing it All Together
### Accurate, Efficient, and Robust?
### Hunting Malware

## Conclusion

# Conclusion

- Improved version of the centroid similarity measure
  - Defined heuristics to use with ARM libraries
  - Increased efficiency and accuracy
  - Robustness against certain obfuscation techniques
- Large-scale study of native library malware in Android third party apps
  - 18 third party app stores checked for infection
  - 508,745 apps analyzed
  - Infection rates of up to 17.05% detected
  - Detection rates outperform VirusTotal

# Thank you.

Questions?

# References

[1] Kai Chen, Peng Liu, and Yingjun Zhang.
Achieving accuracy and scalability simultaneously in detecting application clones on android markets.
In *Proceedings of the 36th International Conference on Software Engineering*, pages 175–186. ACM, 2014.

[2] Kai Chen, Xueqiang Wang, Yi Chen, Peng Wang, Yeonjoon Lee, XiaoFeng Wang, Bin Ma, Aohui Wang, Yingjun Zhang, and Wei Zou.
Following devil's footprints: Cross-platform analysis of potentially harmful libraries on android and ios.
In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 357–376. IEEE, 2016.

[3] Yue Duan, Mu Zhang, Abhishek Vasisht Bhaskar, Heng Yin, Xiaorui Pan, Tongxin Li, Xueqiang Wang, and X Wang.
Things you may not know about android (un) packers: a systematic study based on whole-system emulation.
In *25th Annual Network and Distributed System Security Symposium, NDSS*, pages 18–21, 2018.

[4] Parvez Faruki, Hossein Fereidooni, Vijay Laxmi, Mauro Conti, and Manoj Singh Gaur.
Android code protection via obfuscation techniques: Past, present and future directions.
*CoRR*, abs/1611.10231, 2016.

[5] Kimberly Tam, Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, and Lorenzo Cavallaro.
The evolution of android malware and android analysis techniques.
*ACM Computing Surveys (CSUR)*, 49(4):76, 2017.