AndrEnsemble: Leveraging API Ensembles to Characterize Android Malware Families

Omid Mirzaei, Guilermo Suarez-Tangil, José M. de Fuentes,
Juan Tapiador, **Gianluca Stringhini**

14th ACM Asia Conference on Computer and Communications Security
ASIACCS 2019

Introduction

- Android is the most popular OS as of today.
- Many Android apps are freely available via formal & third-party online app stores.
- Apps on third-party stores are weakly vetted.
- Therefore, Android malware find their way easily into app stores.
- Also, they are commonly hardened with advanced anti-analysis techniques.

Outline

- Background and Contributions
- System Overview
- Results
- Takeaway
- Conclusion

Outline

- **Background and Contributions**
- System Overview
- Results
- Takeaway
- Conclusion

Background and Contributions

Malware Labeling

- Main idea:
  - Assigning a family name to newly discovered malware specimens.
- How?
  - Based on some static features and other Indicators of Compromise (IoC)
- AV vendors use different criteria to name samples and families
- Consequences? **Inconsistencies**
  - Labels are not consistent with the actual behavior of apps.
  - Each AV engine produces a different risk score and security report for a malicious app.
  - Not all samples associated to a family are always related.
  - It is common to find two samples in different families with related behavior.
- Solutions?
  - Considering sub-families (variants)
  - Extracting a unique behavioral core

Background and Contributions

Main Contributions

- A characterization method for Android malware families based on common ensembles of sensitive API calls.
- Study of common and rare ensembles in three types of malware:
  - Ransomware
  - SMS Trojans
  - Banking Trojans
- Summary of anomalies observed in current family labeling of Android malware.

Outline

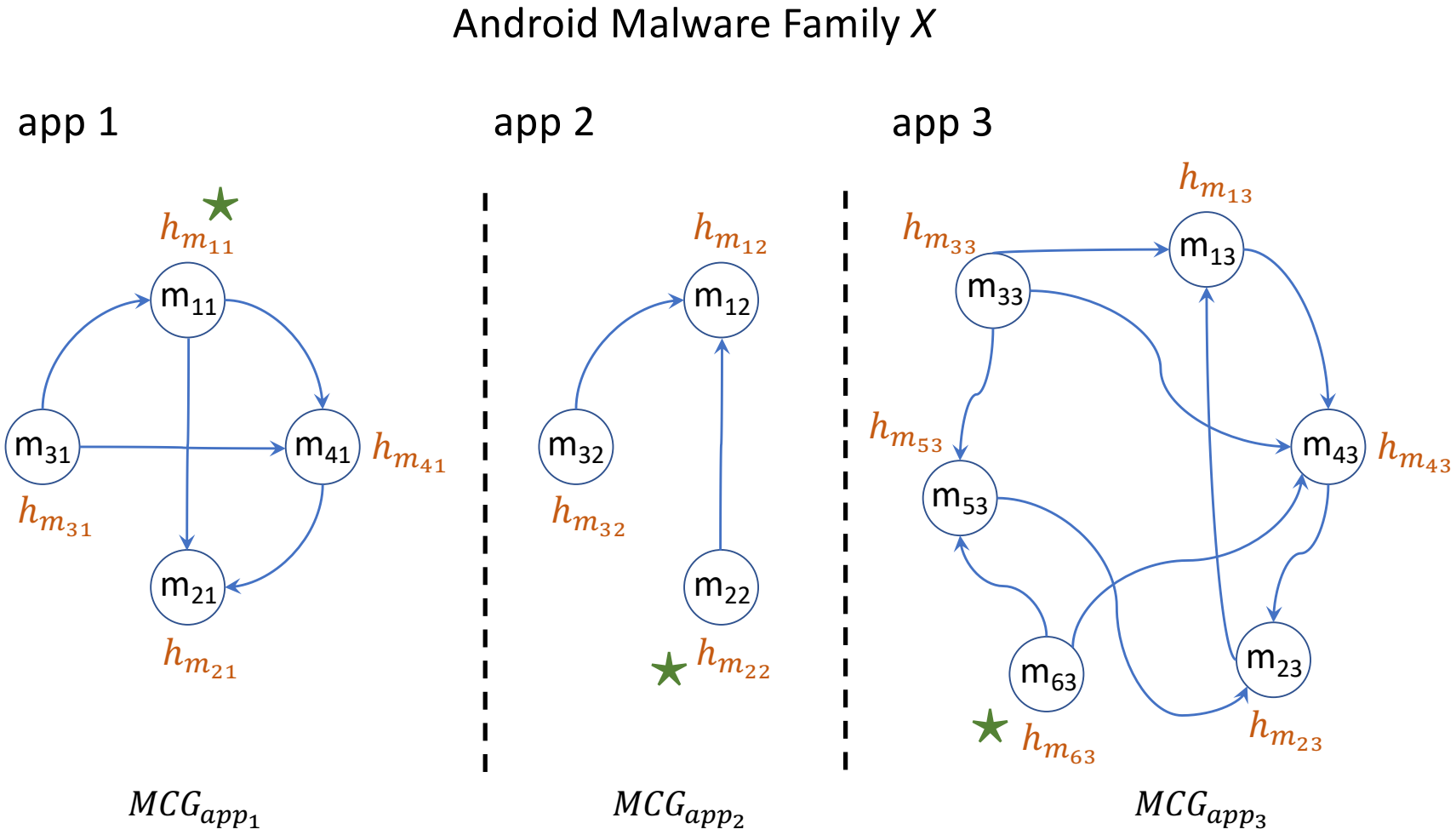
- Background and Contributions
- **System Overview**
- Results
- Takeaway
- Conclusion

System Overview
Method Hashing

- Main idea:
  - Creating a fuzzy hash value for each method based on some features.
- List of features:
  - Control flow graph signature created by Cesare's grammar
  - Method name and its class name
  - Method's intents
  - Sensitive API calls
  - Native and incognito methods
- Fuzzy hashing vs. regular hashing:
  - In fuzzy hashing, the feature to be hashed is segmented into pieces.
  - A rolling hash is used to join all the hashes of these segments together and create a fuzzy hash.
- Why fuzzy hashing? Suitable for **comparative purposes**.

System Overview

AHG Construction: A Simple Example (1/10)

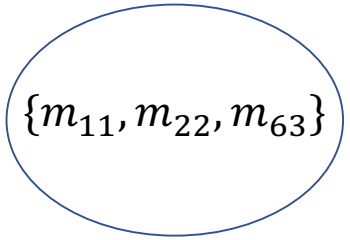


System Overview

AHG Construction: A Simple Example (2/10)

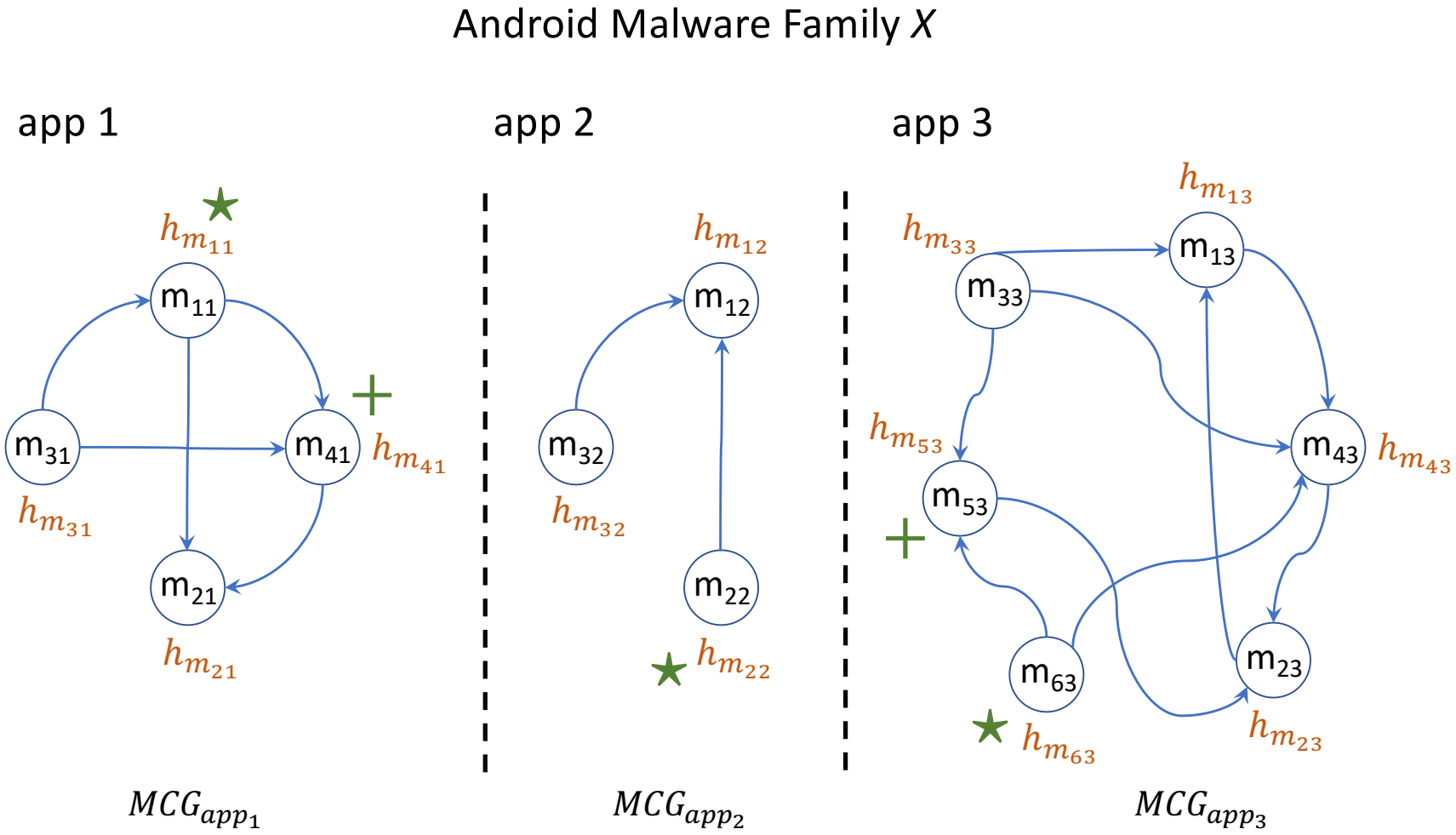
Updating the Aggregated Hash Graph

$$N_1 = h_{m_{11}} = h_{m_{22}} = h_{m_{63}}$$



System Overview

AHG Construction: A Simple Example (3/10)



System Overview

AHG Construction: A Simple Example (4/10)

Updating the Aggregated Hash Graph

$N_1 = h_{m_{11}} = h_{m_{22}} = h_{m_{63}}$

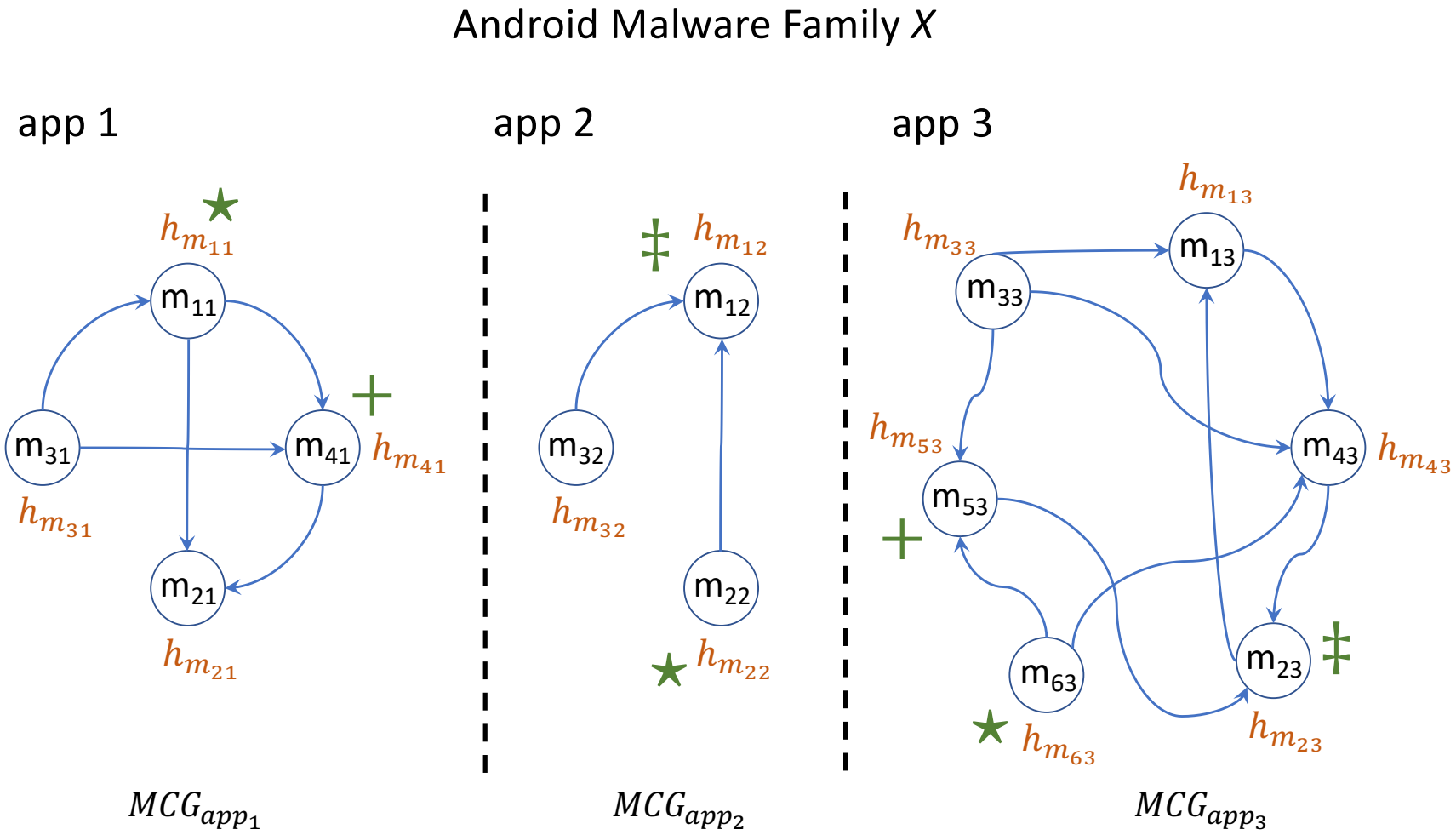
$\{m_{11}, m_{22}, m_{63}\}$

$N_2 = h_{m_{41}} = h_{m_{53}}$

$\{m_{41}, m_{53}\}$

System Overview

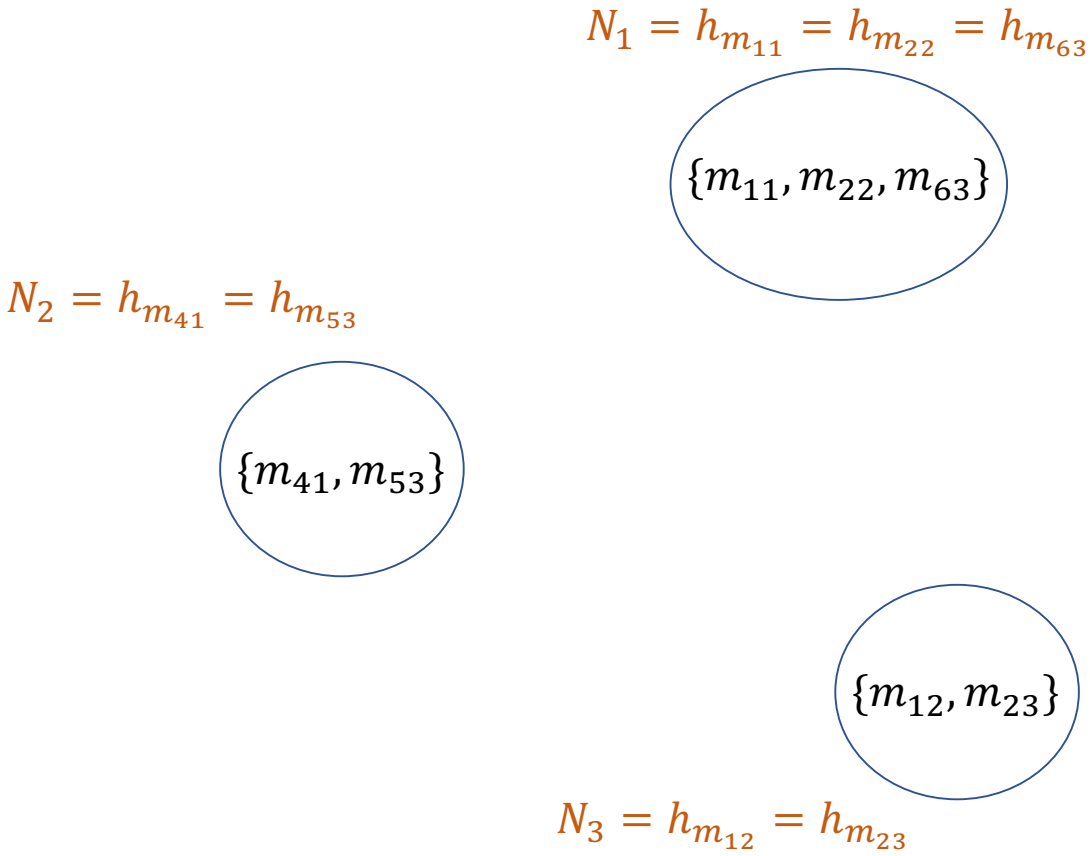
AHG Construction: A Simple Example (5/10)



System Overview

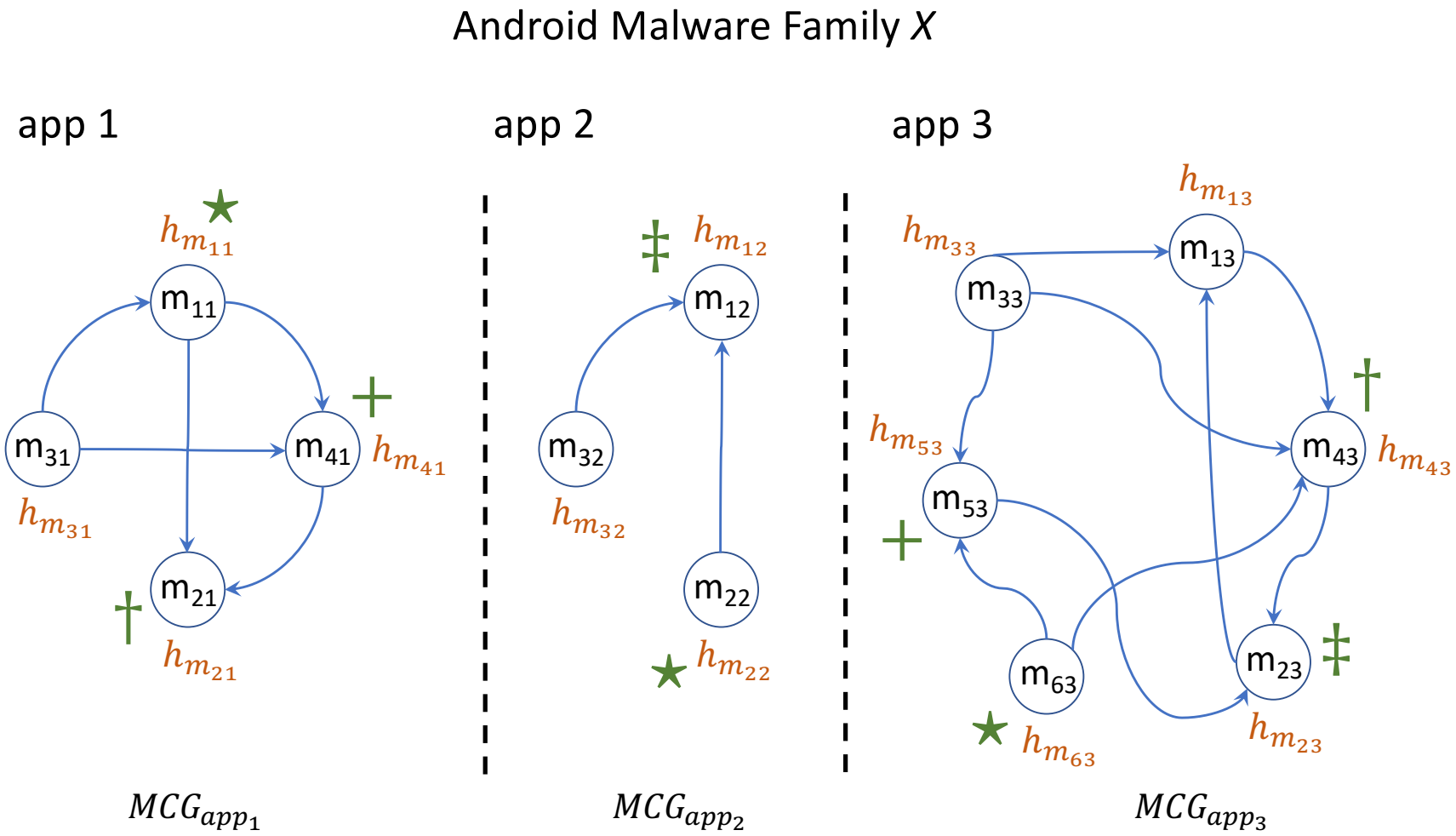
AHG Construction: A Simple Example (6/10)

Updating the Aggregated Hash Graph



System Overview

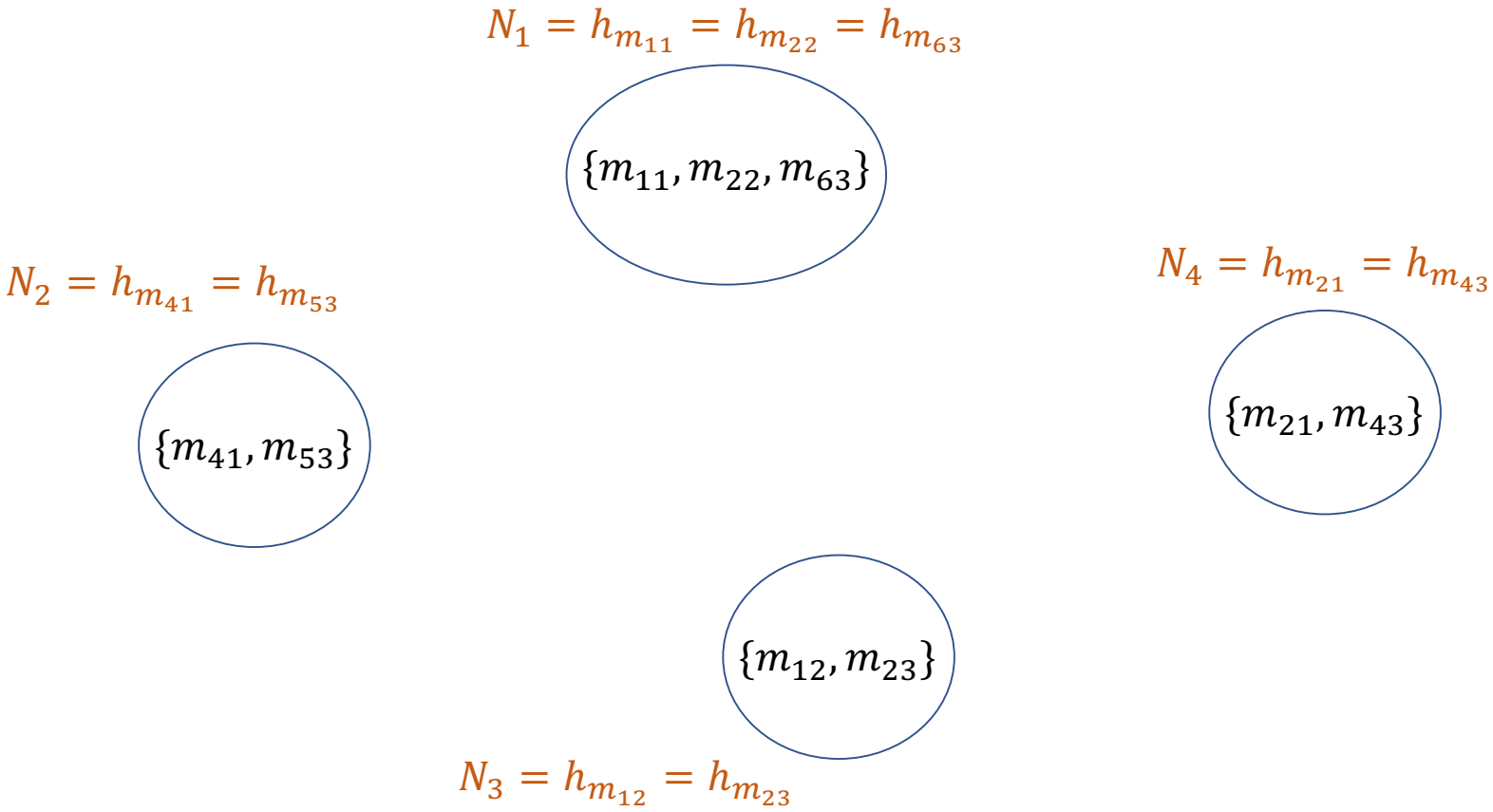
AHG Construction: A Simple Example (7/10)



System Overview

AHG Construction: A Simple Example (8/10)

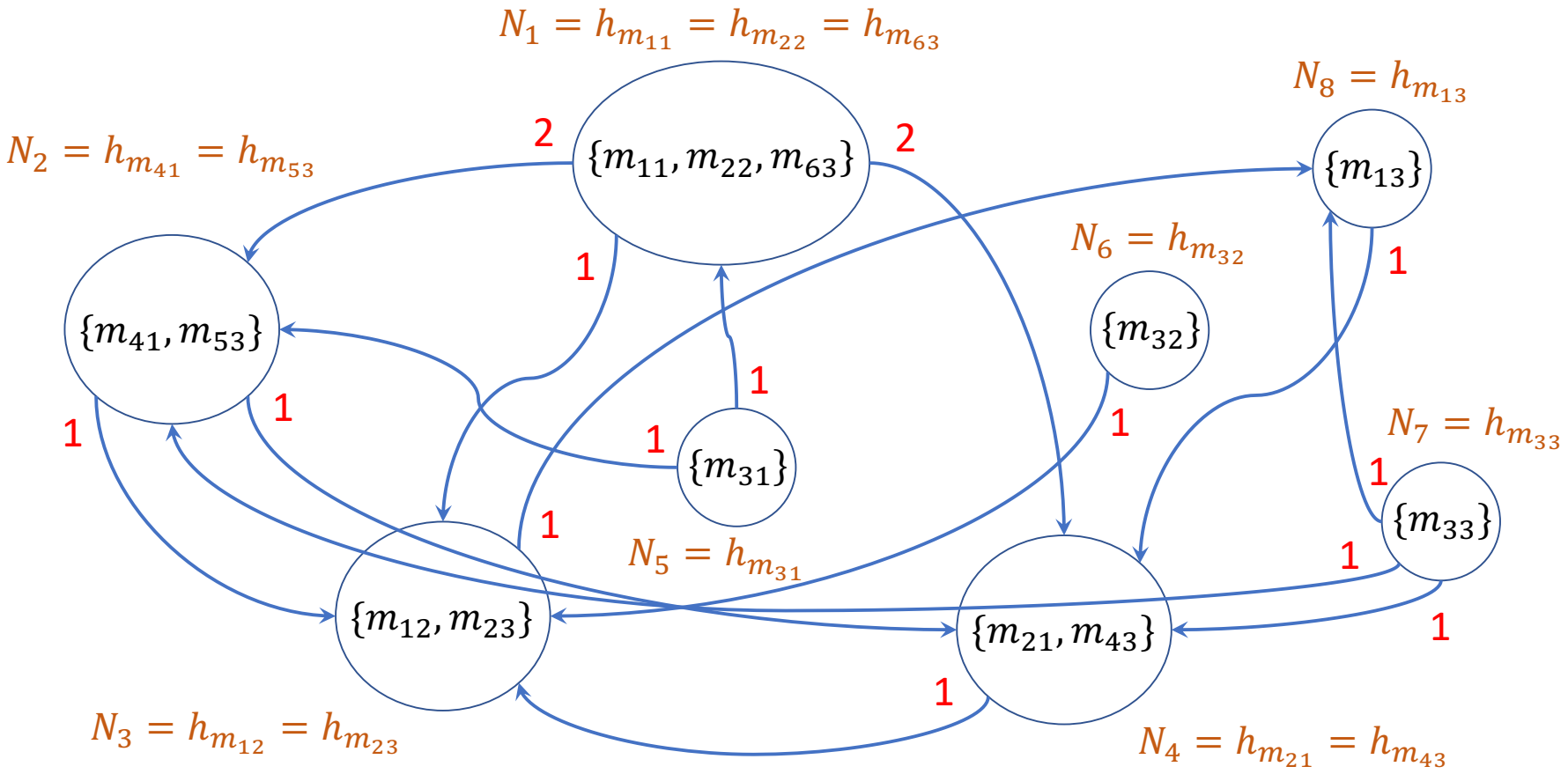
Updating the Aggregated Hash Graph



System Overview

AHG Construction: A Simple Example (9/10)

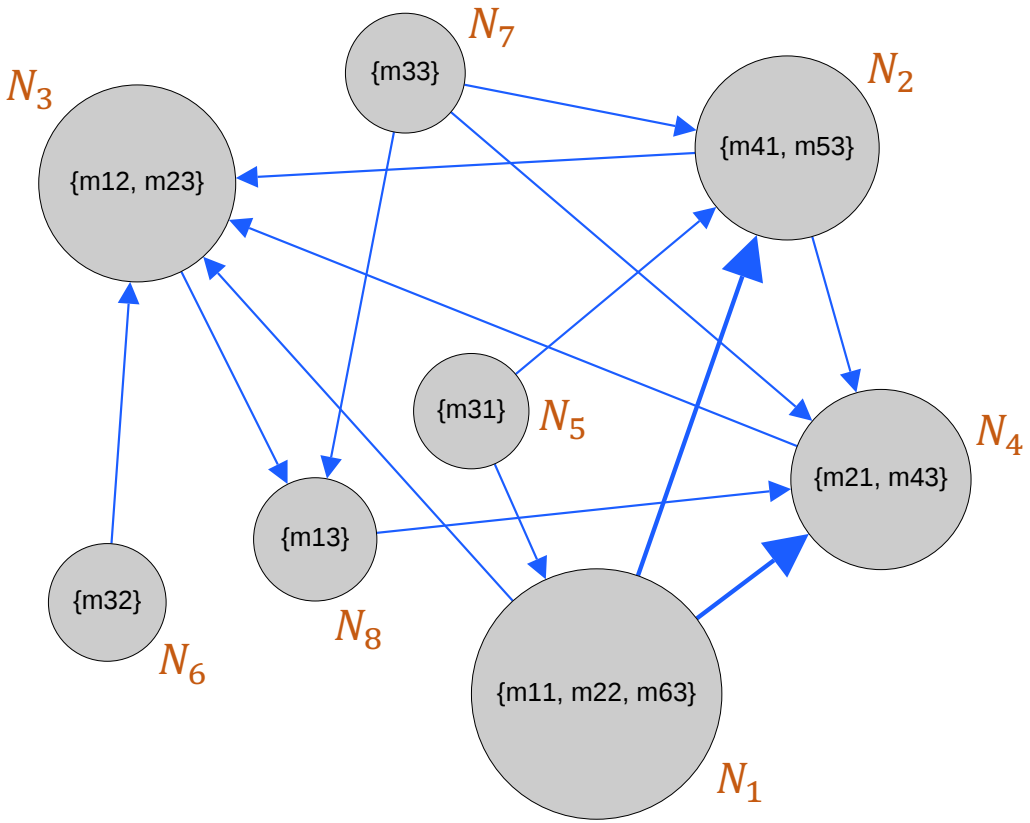
Updating the Aggregated Hash Graph



System Overview

AHG Construction: A Simple Example (10/10)

Updating the Aggregated Hash Graph



System Overview
AHG Construction

- Main idea:
  - Creating an Aggregated Hash Graph, a specific form of a call graph, for each malware family.
- What is an AHG?
  - A bi-directional weighted graph.
- How to create an AHG for each family?
  - Building a Hash Graph (HG) for each application in the family.
  - Merging all HGs to end up with an AHG.
- Thus, in AHG:
  - Each node (representative of one or more methods) is a fuzzy hash value obtained from the previous step.
  - Each edge shows whether or not there are connections between pairs of hashes.
  - The weight of each edge shows how many apps do share that particular edge in the family.

System Overview
API Ensemble Extraction

- Main idea:
  - Identifying ensembles of API methods exercised by the majority of app in each family.
- Why APIs?
  - API calls are appropriate representatives of an app's behavior.
- How to extract API ensembles?
  - Identifying all source methods
  - Extracting all paths originating from source methods with respect to maximum length
  - Recording sensitive API calls appearing in each path

System Overview
Feature Vectors Creation

- Assigning a binary feature vector to each app based on the extracted ensembles
- Vector's length is equal to the total number of extracted ensembles from the whole dataset
- How to measure similarities and differences?
  - Cosine distance
  - It is 0: when vectors are very similar
  - It is 1: when vectors are completely different

Outline

- Background and Contributions
- System Overview
- **Results**
- Takeaway
- Conclusion

Results

Dataset and Settings

- AndroZoo contains around 8M apps from more than 3,000 families.
- Around 97% of apps are collected from GooglePlay, Anzhi and AppChina.
- Around 1%, 33% and 17% of apps in the above three markets are malware.

Malware Type	#Apps	#Families	Avg. Size
Ransomware	824	7	4.98
SMS Trojan	1,967	98	9.88
Banking Trojan	259	12	10.20
Total	3,050	117	8.35

- Edges are common in more than 70% of apps
- We do not mine paths with lengths higher than 2.

Results

Most Common & Rarest Ensembles

- Out of 25 ensembles extracted from Ransomware families:
  - 11 ensembles are present in more than 70% of apps in different families
  - Few ensembles are present in less than 2% of apps in different families
- Out of 168 unique ensembles extracted from SMS Trojan families:
  - 3 ensembles are present in more than 50% of apps in all families
  - 91 ensembles (54%) are present in less than 2% of apps in various families
- Out of 50 ensembles extracted from Banking Trojan families:
  - 2 ensembles are shared by 50% of apps in different families
  - 9 API ensembles are common in less than 5% of apps in various families

Results

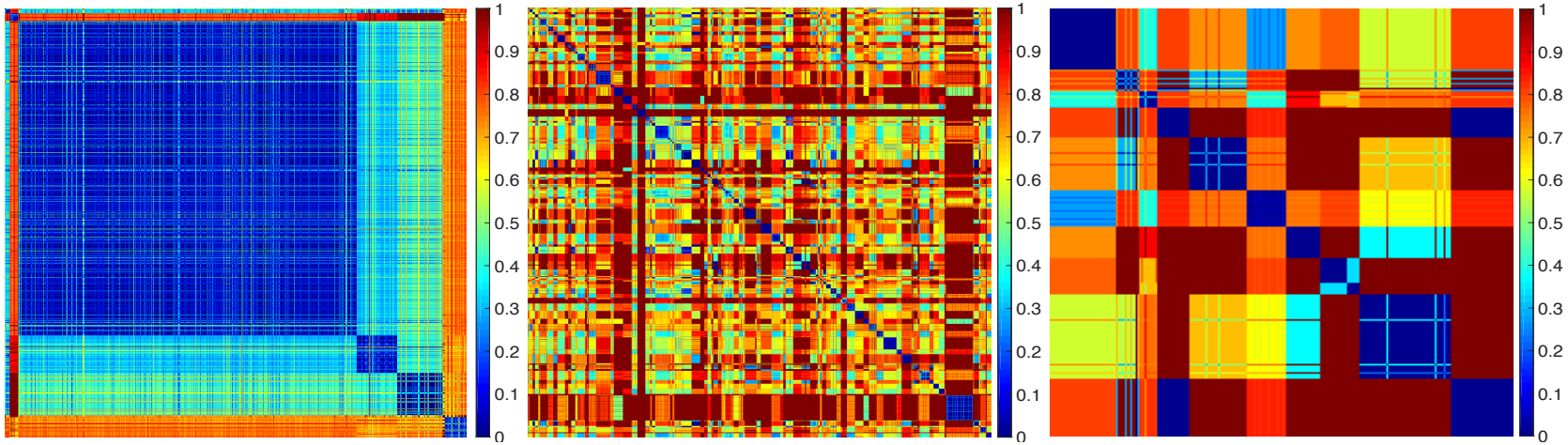
Case Study: Fareac

- 37 different specimens
- All samples (100%) share 3 API methods:
  - isWifiEnabled( ) AND loadLibrary( ) AND getClassLoader( )
- Other API ensembles which are common in more than 70% of apps:
  - <setFlags( ), getApplicationInfo( )>
  - query( )
  - getNetworkOperator( )
  - addFlags( )
  - crypto
  - <openConnection( ), connect( )>
  - exists( ) AND delete( )
  - getInputStream( )
  - <killProcess( ), myPid( )>

Results

Intra-family Characterization

- Apps from two families with **exactly the same** signature:
  - *4654EC...48F2.apk from slocker* AND *8905B3...99DC.apk from gepew*
- Apps from two families with **slightly different** signatures:
  - *C3829A...03DB.apk from svpeng* AND *877D3B...2AE4.apk from slocker*
  - Share all ensembles except two:
    - The first app overlays its window on top of others (addFlags( ))
    - It also has a keyword database to identify encryption-related words in UI widgets (query( ))



Takeaway

- Background and Contributions
- System Overview
- Results
- **Takeaway**
- Conclusion

Takeaway

- Malicious operations do not necessarily contain several sensitive API methods
  - A considerable number of common ensembles contain only one sensitive API method

≈72% in ransomware, ≈ 21% in SMS Trojans, and ≈ 52% in banking Trojans

- Opposite to ransomware and banking Trojans, ensembles of two API methods were the most common in SMS Trojans.

- We found several samples with identical ensembles though belonging to different families.

Outline

- Background and Contributions
- System Overview
- Results
- Takeaway
- **Conclusion**

Conclusion

- Characterizing Android malware families based on ensembles of API methods
- Building an Aggregated Hash Graph (AHG) per family
- A greedy graph-mining algorithm based on the maximum length of paths