
BRITISH COLUMBIA

THE UNIVERSITY
OF BRITISH COLUMBIA

Detecting Mobile Malware via Program Analysis and Machine Learning: Challenges and Ways Forward

Julia Rubin
The University of British Columbia, Vancouver, Canada

November 2022

Julia Rubin

Associate Professor, University of British Columbia, Vancouver, Canada
Canada Research Chair in Trustworthy Software
Lead of UBC Research Excellence Cluster on Trustworthy ML

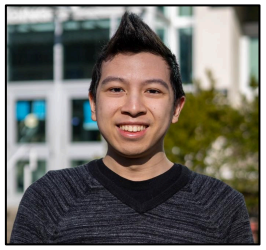
Earlier:

- Research Staff Member and Research Group Manager in IBM Research
- Postdoctoral researcher at EECS, MIT, USA
- PhD in Computer Science – University of Toronto, Canada

Based on Joint Work

Khaled Ahmed

Sahar Badihi



Michael Cao

Duling Lai

Shabrooneel Pal

Jaskeerat Sarin

Michael Tegegn

Lina Qiu

Yingying Wang

Peiyu (Gabby) Xiong

Junbin Zhang

[1] Michael Cao, Khaled Ahmed, Julia Rubin. *Spoiled Apples Ruin the Bunch: Anatomy of Google Play Malware*. ICSE 2022.

[2] Khaled Ahmed, Mieszko Lis, and Julia Rubin. *MANDOLINE: Dynamic Slicing of Android Applications with Trace-Based Alias Analysis*. ICST 2021, **Distinguished Paper Award**.

[3] Junbin Zhang, Yingying Wang, Lina Qiu, and Julia Rubin. *Analyzing Android Taint Analysis Tools: FlowDroid, Amandroid, and DroidSafe*. IEEE Transactions on Software Engineering, 2021.

[4] Michael Cao, Sahar Badihi, Khaled Ahmed, Peiyu Xiong, and Julia Rubin. *On Benign Features in Malware Detection*. ASE 2020, NIER Track.

[5] Duling Lai and Julia Rubin. *Goal-Driven Exploration for Android Applications*. ASE 2019.

[6] Lina Qiu, Yingying Wang, and Julia Rubin. *Analyzing the Analyzers: FlowDroid/IccTA, AmanDroid, and DroidSafe*. ISSTA 2018, **ACM SIGSOFT Distinguished Paper Award**.

Two papers under review

Why Mobile?

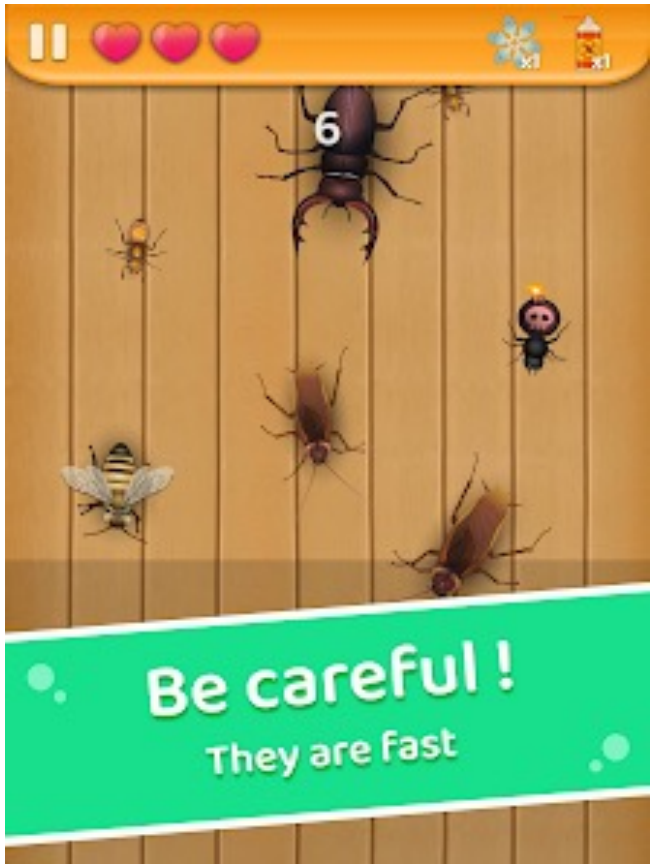
15 Billion Mobile Devices

>

https://www-statista-com.eu1.proxy.openathens.net/statistics/245501/multiple-mobile-device-ownership-worldwide/
https://www.statista.com/statistics/273495/global-shipments-of-personal-computers-since-2006/

Be Careful! Mobile Malware

- Steals user credentials and private data
- Mines cryptocurrency on user devices
- Demands ransom
- …



Example: the BugSmasher App

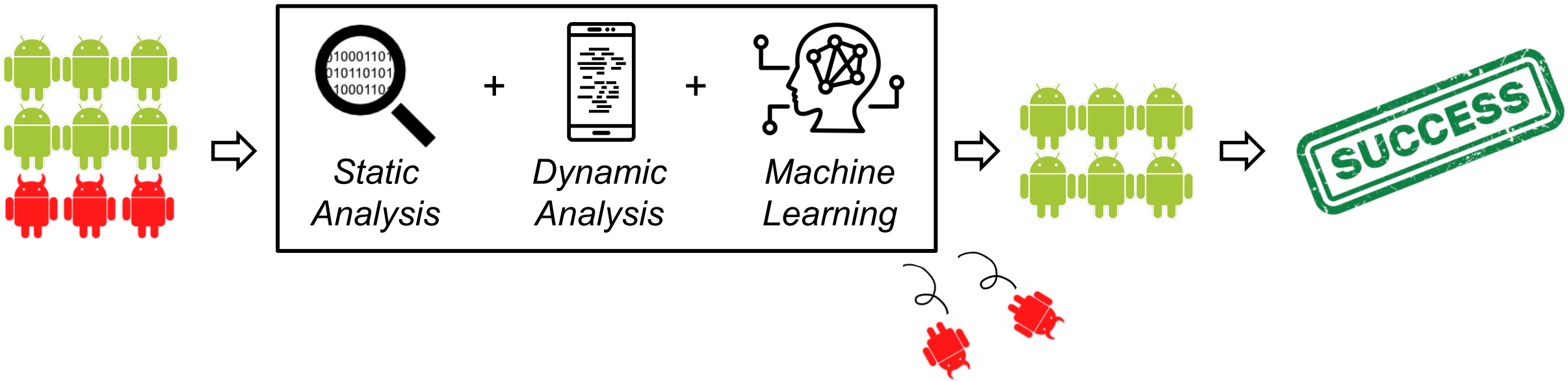
1. *Activates when device screen is turned off*
2. *Checks whether the phone is plugged*
3. *Mines cryptocurrency without user consent!*

Detecting Malware

- Official and alternative stores
- On-device protection
- Antivirus companies
- Companies whose apps are impersonated by malicious apps (e.g., Meta/WhatsApp)
- …

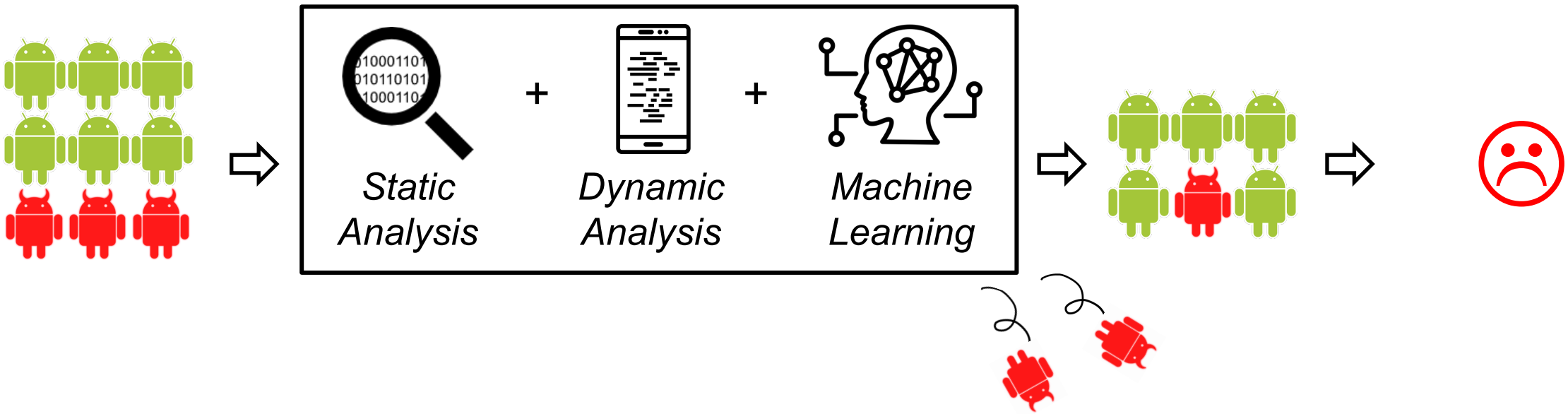
Detecting Malware

- Official and alternative stores
- On-device protection
- Antivirus companies
- Companies whose apps are impersonated by malicious apps (e.g., Meta/WhatsApp)
- …



Detecting Malware

- Official and alternative stores
- On-device protection
- Antivirus companies
- Companies whose apps are impersonated by malicious apps (e.g., Meta/WhatsApp)
- …



Detecting Malware

- Official and alternative stores
- On-device protection
- Ar
- C
-

FORTUNE

Android Ma

36.5 Mi

Android Google Play app with 100 million downloads starts to deliver malware

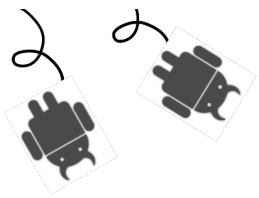
been available since 2010, has recently started installing malware.

as Many

Home > News > Security > Android malware apps with 2 million installs found on Google Play

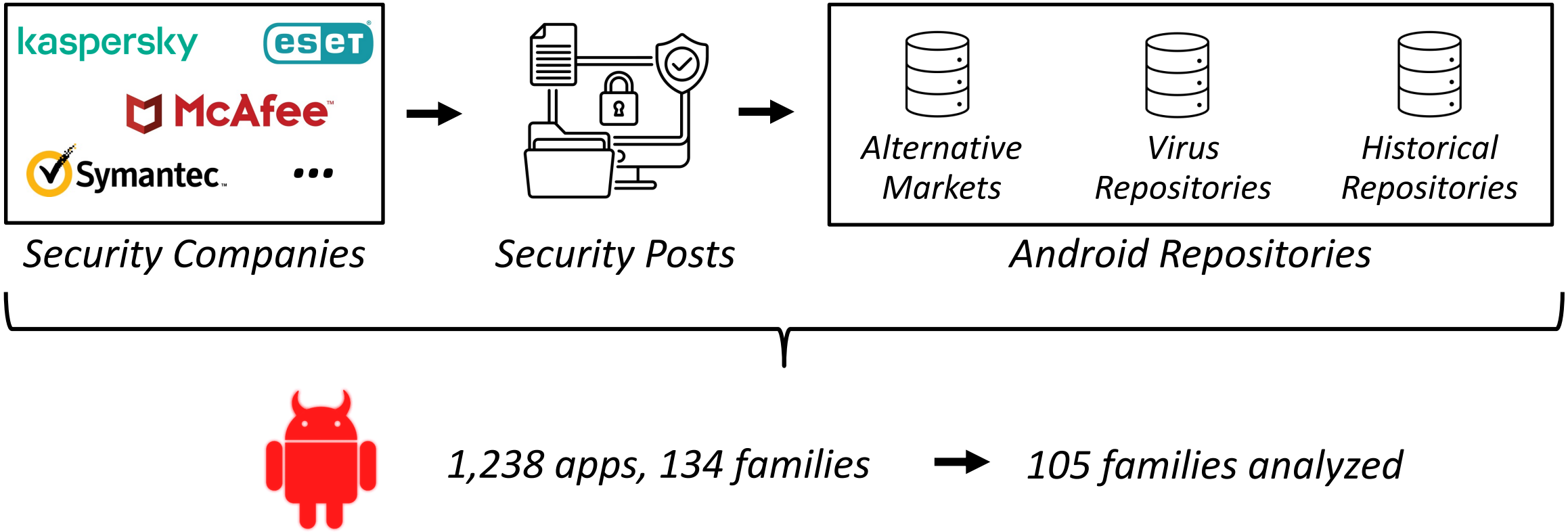
Can

Android malware apps with 2 million installs found on Google Play



Why is this happening?

A Collection of Recent Malware



[1] Michael Cao, Khaled Ahmed, and Julia Rubin. *Rotten Apples Spoil the Bunch: an Anatomy of Google Play Malware*, ICSE 2022.

Characterization Schema

Initial entry point of malicious path

Event conditions	#	%
System	80	76.2
Boot status	44	41.9
Device status	30	28.6
Network status	26	24.8
→ Developer-defined	26	24.8
Package changes	22	21.0
→ Service bind	11	10.5
SMS delivery	11	10.5
Battery status	9	8.6
Call status	8	7.6
USB status	1	1.0
User	95	90.5
Application launch	92	87.6
→ Button click	46	43.8
→ Sensitive input	28	26.7
→ Permissions	25	23.8
→ App install	15	14.3
→ Clipboard text	1	1.0
Scheduling	58	55.2
Scheduling	58	55.2

Guarding conditions along malicious path

Check conditions	#	%
External server	79	75.2
Internet	78	74.3
SMS	2	1.9
Device	82	78.1
Software specs	62	59.0
Network	43	41.0
→ Hardware specs	28	26.7
→ Sensors	4	3.8
Environment	41	39.0
Time	38	36.2
Location	6	5.7
Application	42	40.0
Permissions	31	29.5
Data format	10	9.5
→ Probability	8	7.6
Version	1	1.0

Main malicious behavior

Payloads	#	%
Information stealing	69	65.7
Ad abuse	54	51.4
Premium charges	10	9.5
→ Cryptomining	5	4.8
Root exploit	4	3.8
→ Clipboard hijacking	3	2.9
→ Port forwarding	3	2.9
Ransom	1	1.0
Unknown	16	15.2

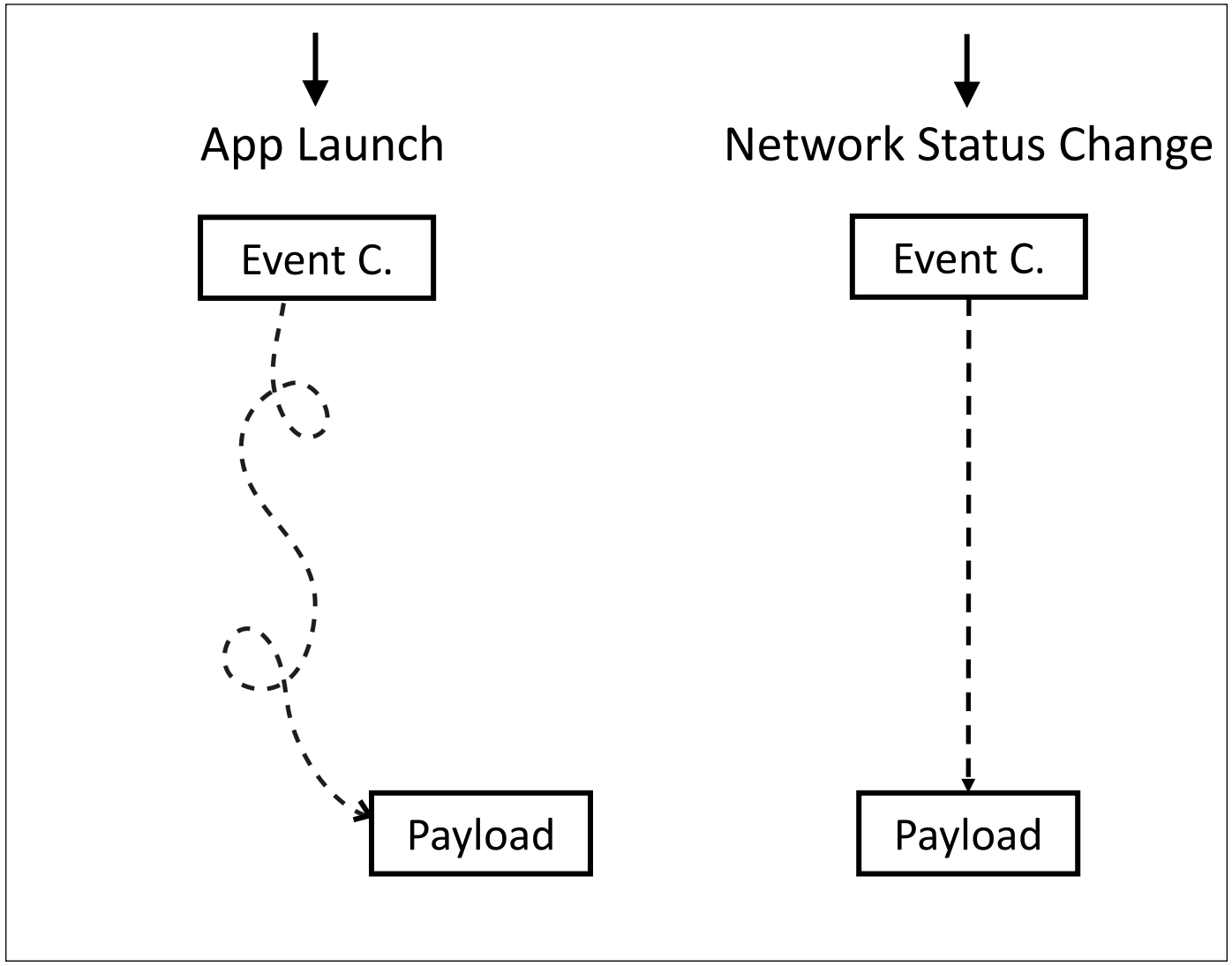
Hide maliciousness from user

Hiding techniques	#	%
Rich functionality	65	61.9
Icon manipulation	34	32.4
Device admin	15	14.3
Information blocking	12	11.4
→ Self-uninstallation	6	5.7
→ Automated gesture input	5	4.8
Screen locking	3	2.9

Implementation location and language

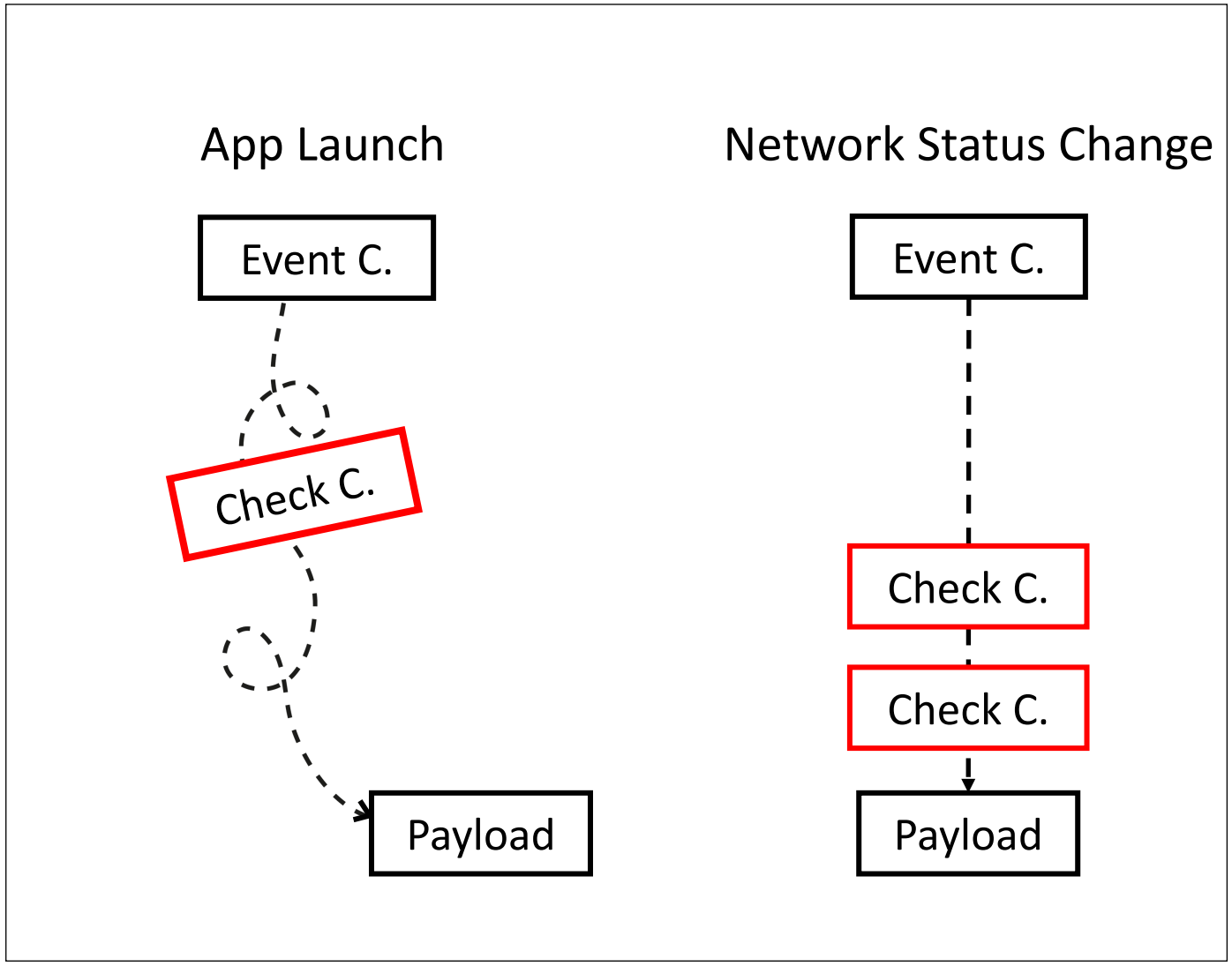
Code properties	#	%
Location		
Direct	97	92.4
→ Downloaded (Remote)	60	57.1
→ Hidden (Resources)	25	23.8
Language		
Bytecode	102	97.1
→ Web	29	27.6
Native	11	10.5

Event and Check Conditions



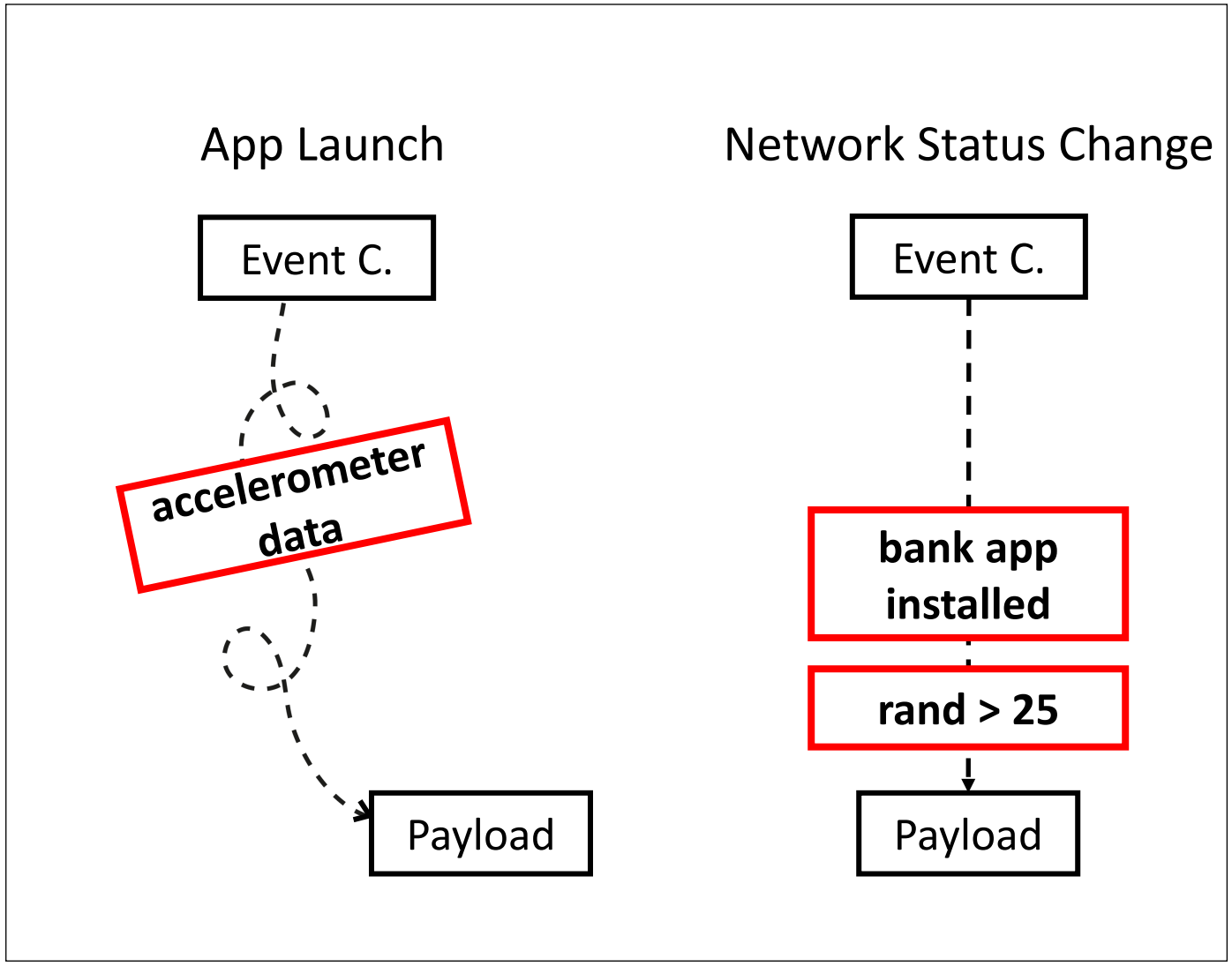
- 1/3 of the payloads are not triggered on app launch

Event and Check Conditions



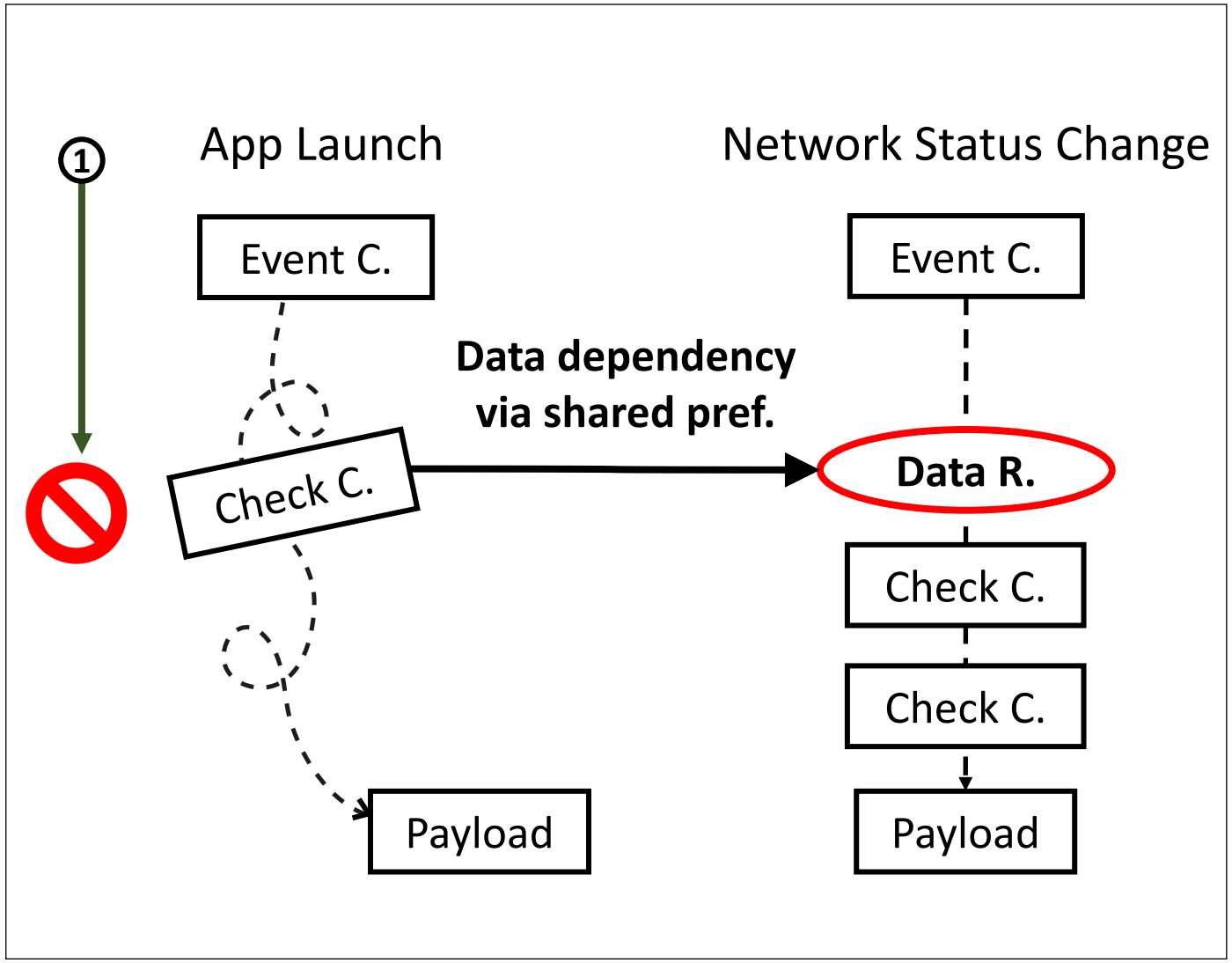
- 1/3 of the payloads are not triggered on app launch
- A variety of checks that guard the malicious payload

Event and Check Conditions



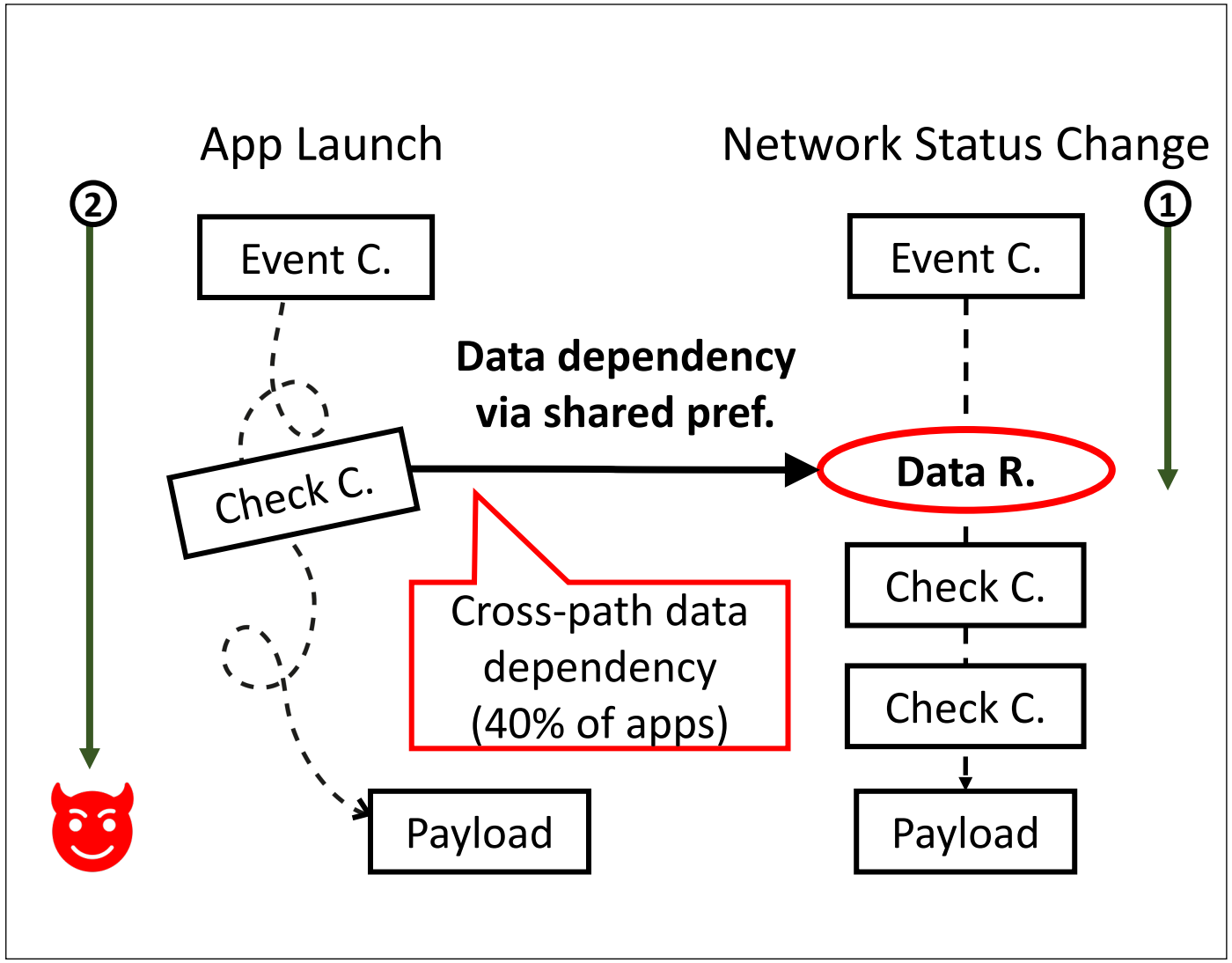
- 1/3 of the payloads are not triggered on app launch
- A variety of checks that guard the malicious payload, e.g., execute if
  - accelerometer data implies real device
  - a certain app is opened
  - at random

Cross-path Dependencies



- Order of events matters
  - Payload is executed only if events are triggered in a certain order

Cross-path Dependencies

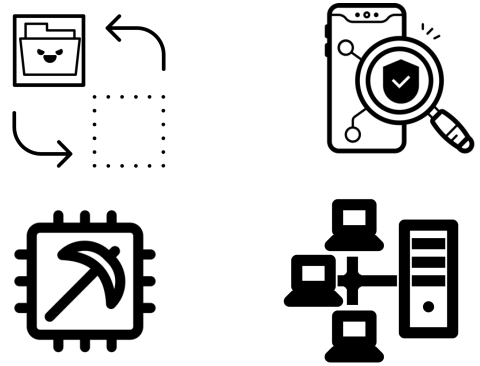


- Order of events matters
  - Payload is executed only if events are triggered in a certain order

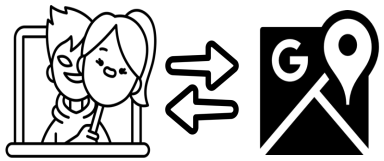
Payloads

- Most frequent payloads: info stealing (69 apps) and ad abuse (54 apps)

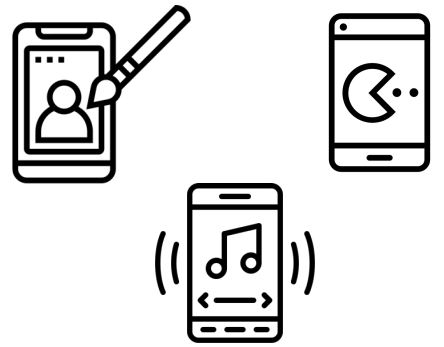
- Newer behaviors
  - Collaboratively mine cryptocurrency (5 apps)
  - Clipboard hijacking (3 apps)
  - …



Hiding Techniques



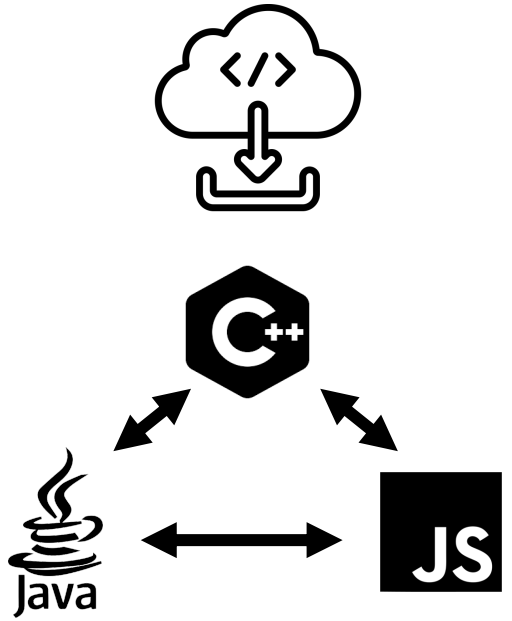
- Hide the app from user
  - E.g., hide the icon or change the icon to pretend being Google Maps
- Avoid uninstall
  - E.g., by auto-clicking back button when uninstall is attempted
- Distract user with useful benign behaviors
  - E.g., photo editing, music players, mobile games, …



Avoid detection by users and thus prolong lifetime

Code Properties

- Malicious code resides in a variety of locations
  - Dex files (97 apps)
  - Downloaded remotely (60 apps)
  - Hidden in resource files (25 apps)
- Implemented in a variety of languages
  - Java bytecode, Native, Web/Javascript



Tool Collection

Goal: Understand gaps in existing defenses

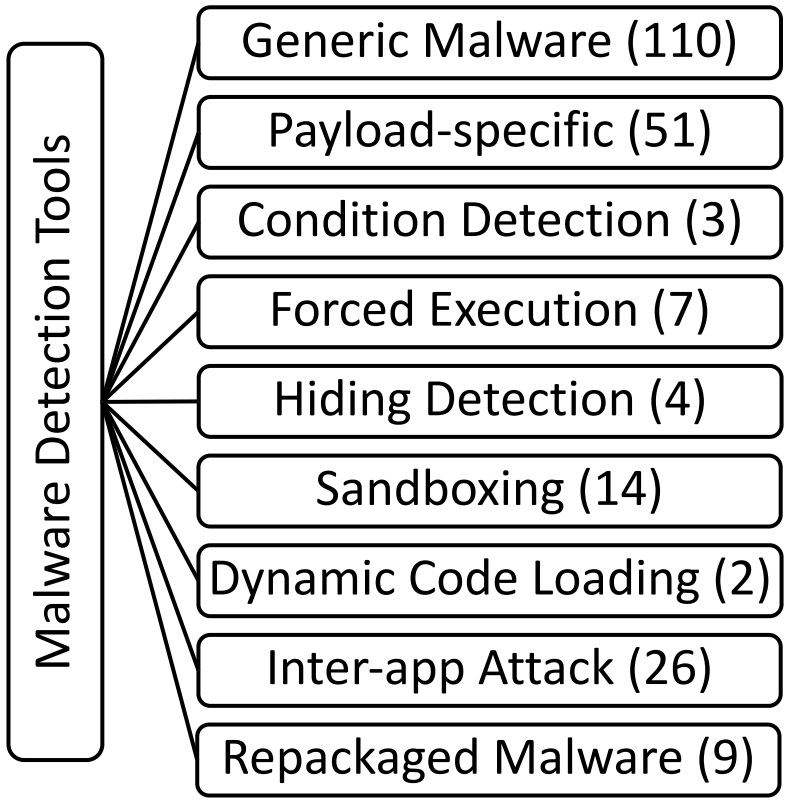
- Top conferences and journals
  - Between 2010 – 2021
- Seven Android security surveys
  - Between 2015 – 2020

237 papers in total

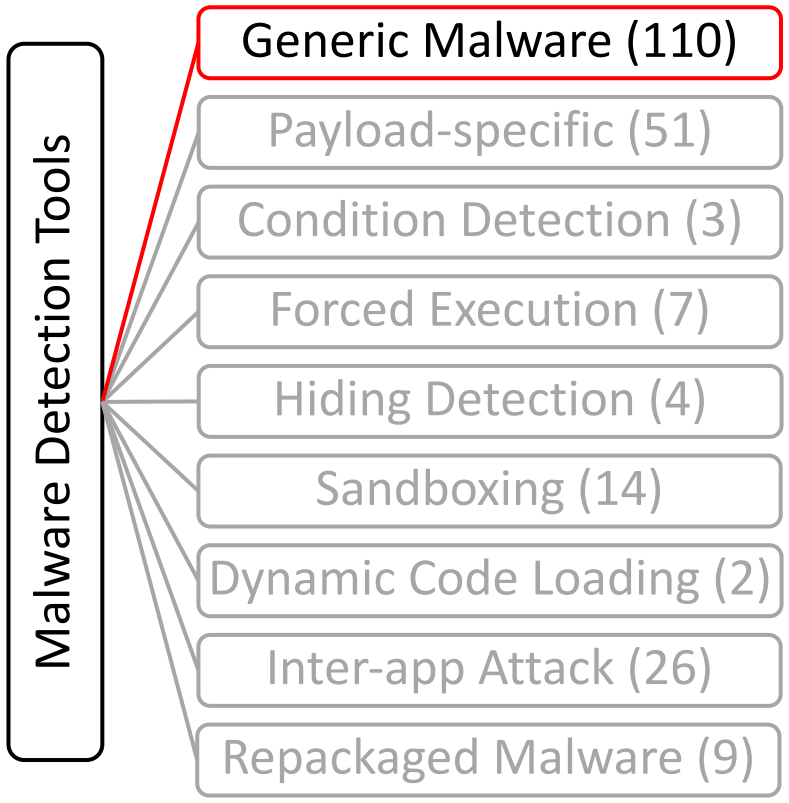
Category	Conference/Journal Name
Software Engineering	Automated Software Engineering (ASE)
	Foundations on Software Engineering (FSE)
	International Conference on Software Engineering (ICSE)
	International Symposium on Software Testing and Analysis (ISSTA)
	Mining Software Repositories (MSR)
	Empirical Software Engineering (ESE)
	IEEE Software
	IEEE Transactions on Software Engineering (TSE)
	Information and Software Technology (IST)
	Journals of Systems and Software (JSS)
	Transactions on Software Engineering and Methodology (TOSEM)
Security	Annual Computer Security Applications Conference (ACSAC)
	Conference on Computer and Communications Security (CCS)
	IEEE Symposium on Security and Privacy (S&P)
	The Network and Distributed System Security Symposium (NDSS)
	USENIX Security Symposium (USENIX Security)
	Computers & Security (CS)
	IEEE Security & Privacy
	IEEE Transactions no Dependable and Secure Computing (TDSC)
	IEEE Transactions on Information Forensics and Security (TIFS)
	International Journal of Information Security (JIS)
	International Journal of Information Security Science (JISS)

[1] Michael Cao, Khaled Ahmed, and Julia Rubin. *Rotten Apples Spoil the Bunch: an Anatomy of Google Play Malware*, ICSE 2022.

Tool Categorization



Tool Categorization



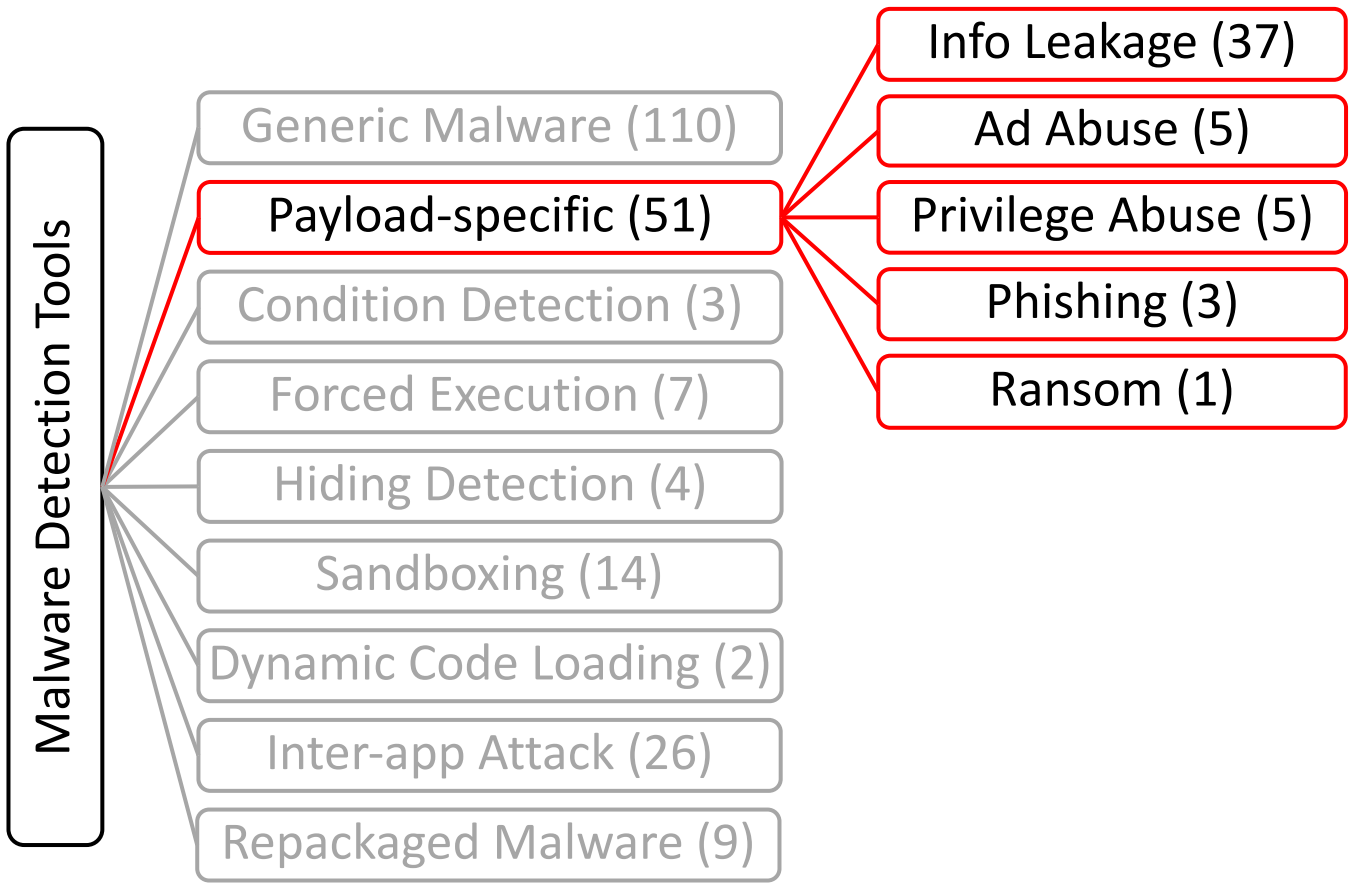
Extract generic features separating benign and malware samples

E.g., DREBIN [*] features

- Android components
- Permissions
- API calls
- Network addresses

[*] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. "Drebin: Effective and explainable detection of android malware in your pocket." In NDSS, vol. 14, pp. 23-26. 2014.

Tool Categorization



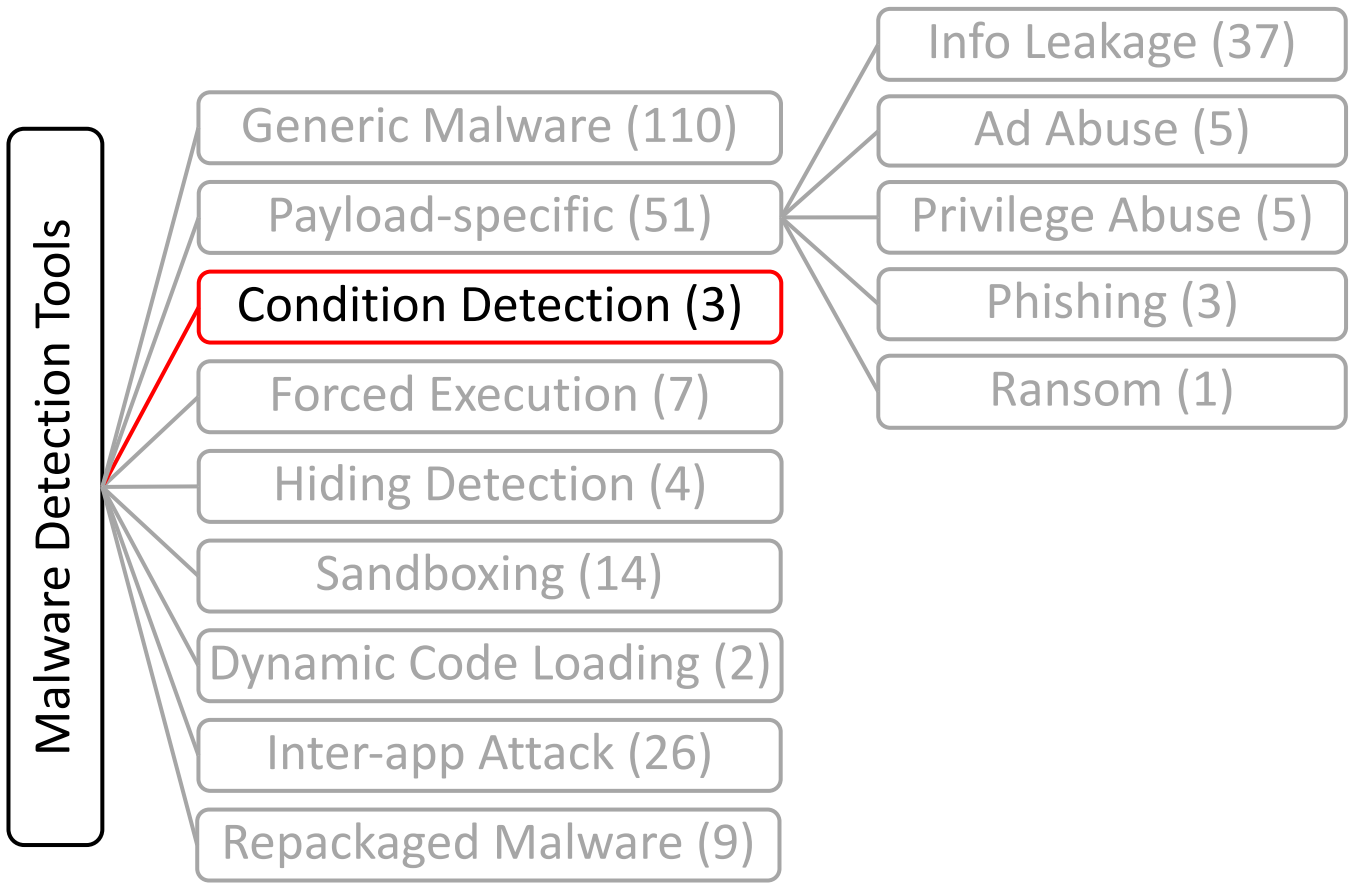
Extract patterns specific to a particular payload

E.g., FlowDroid [*]

- Apply taint analysis to identify info leakage payloads

[*] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps." ACM SIGPLAN Notices 49, no. 6 (2014): 259-269.

Tool Categorization



Detect **narrow** conditions leading to malicious behaviors

E.g., TriggerScope [*]

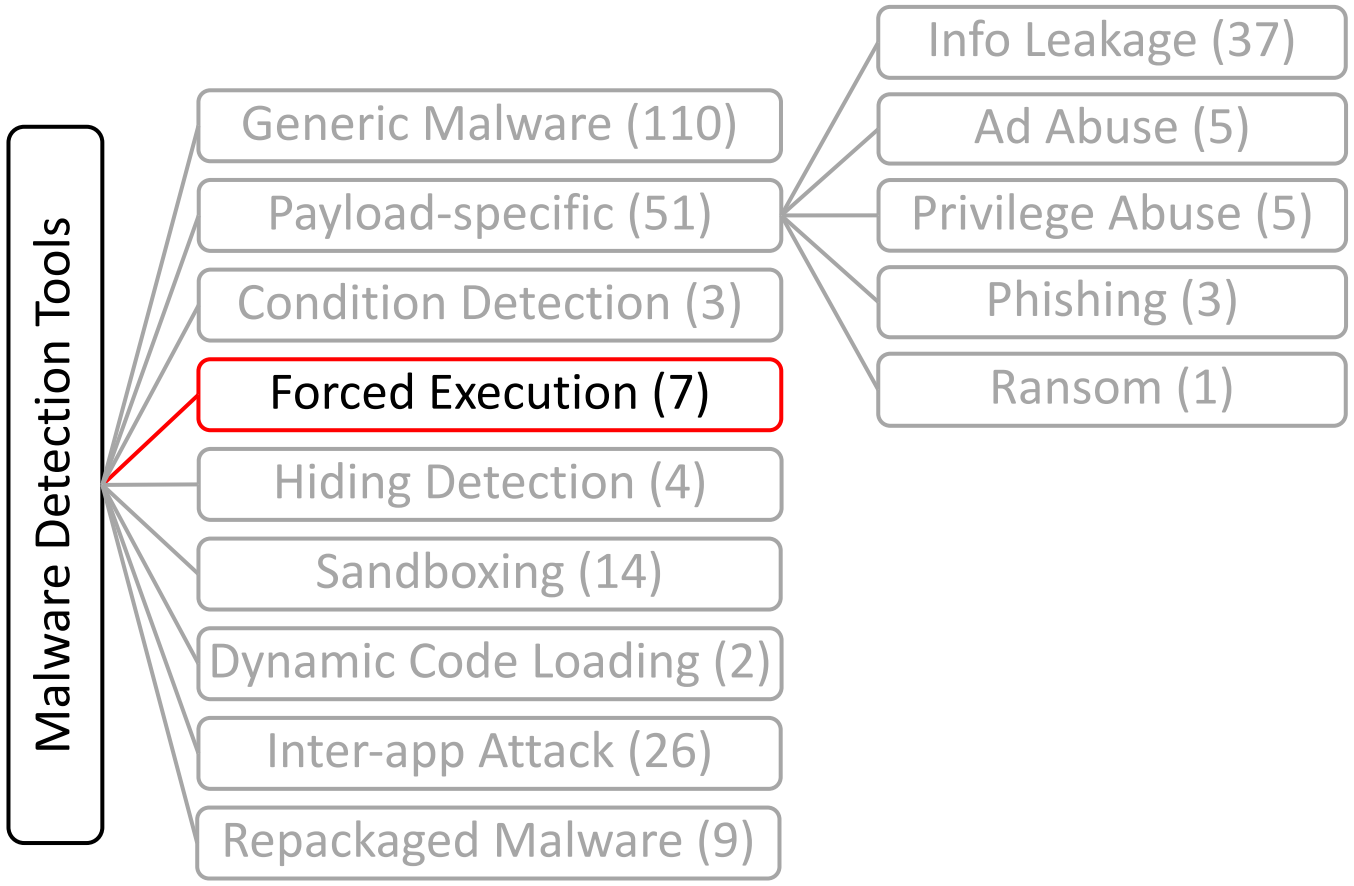
Time

Constant value

```
if (new Date().getYear() == 2023) {
    mineCryptocurrency();
}
else {
    … benign functionality
}
```

[*] Yanick Fratantonio, Antonio Bianchi, William Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. "Triggerscope: Towards detecting logic bombs in android applications." In 2016 IEEE Symposium on Security and Privacy (SP), pp. 377-396. IEEE, 2016.

Tool Categorization



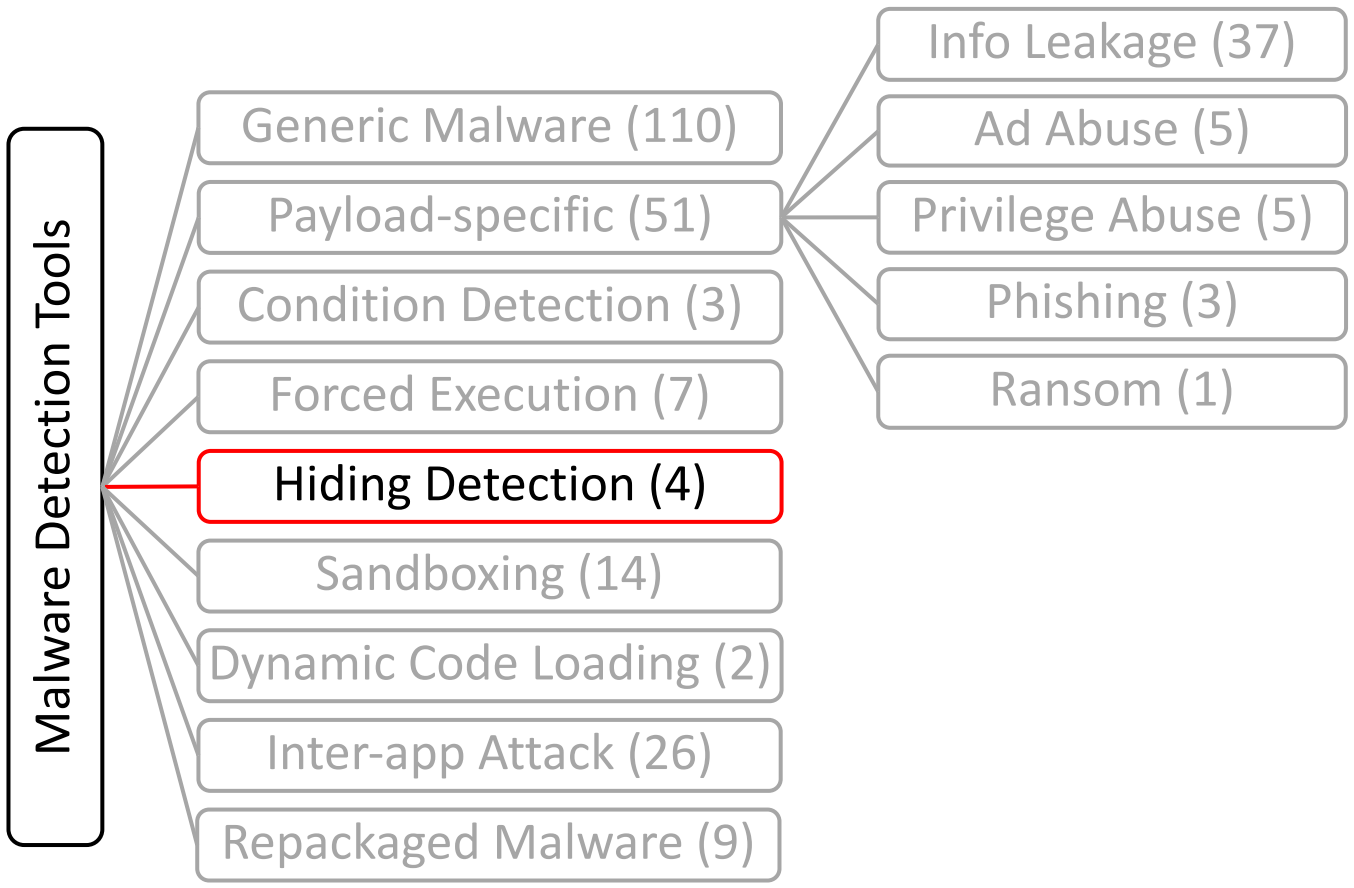
Dynamically force app execution towards the malicious behavior

E.g., IntelliDroid [4]

- Given a target API, find reachable paths and constraints to execute API

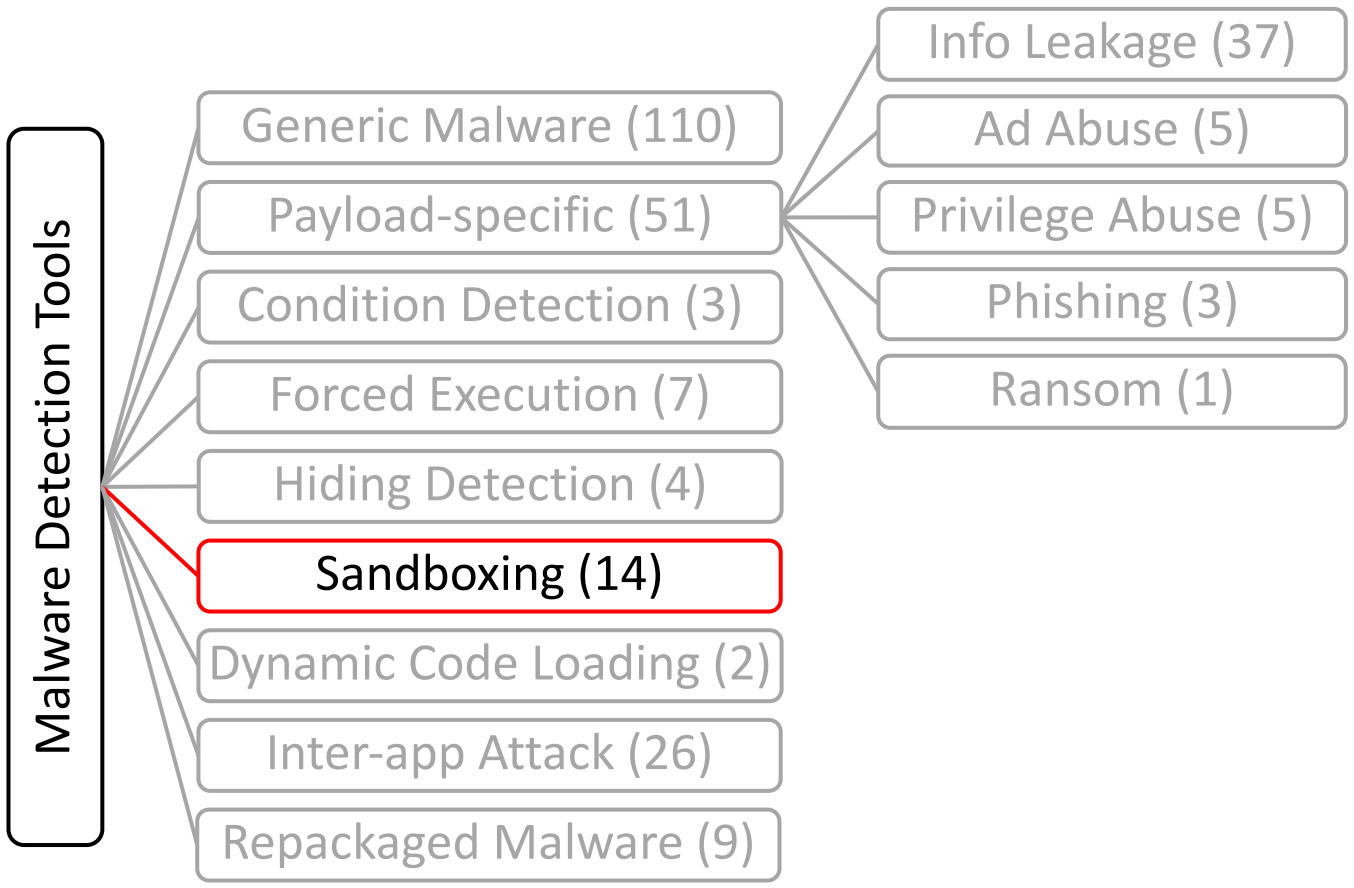
[4] Michelle Y. Wong and David Lie. "Intellidroid: a targeted input generator for the dynamic analysis of android malware." In NDSS, vol. 16, pp. 21-24. 2016.

Tool Categorization



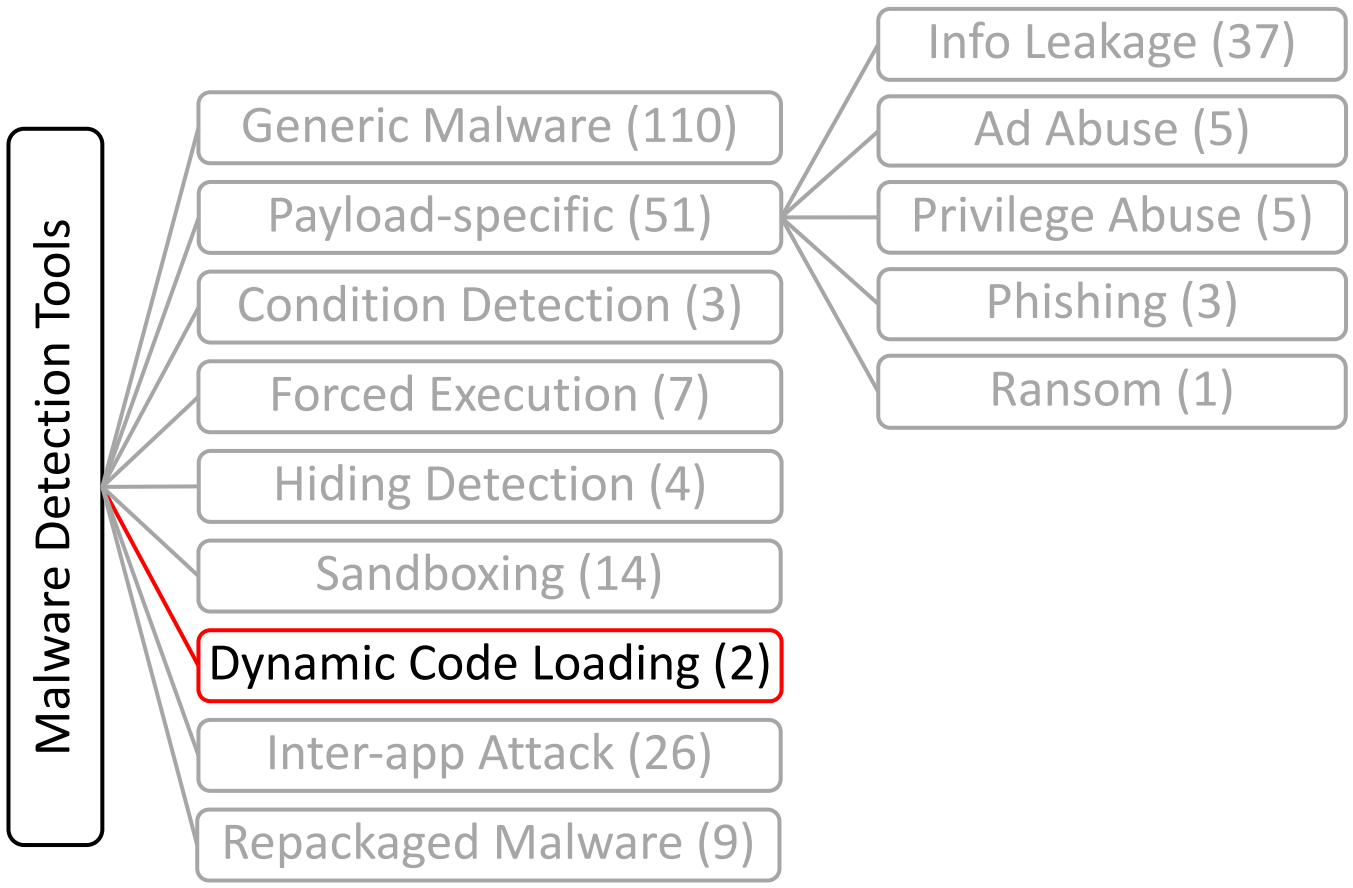
Identify apps that hide malicious actions from the user

Tool Categorization



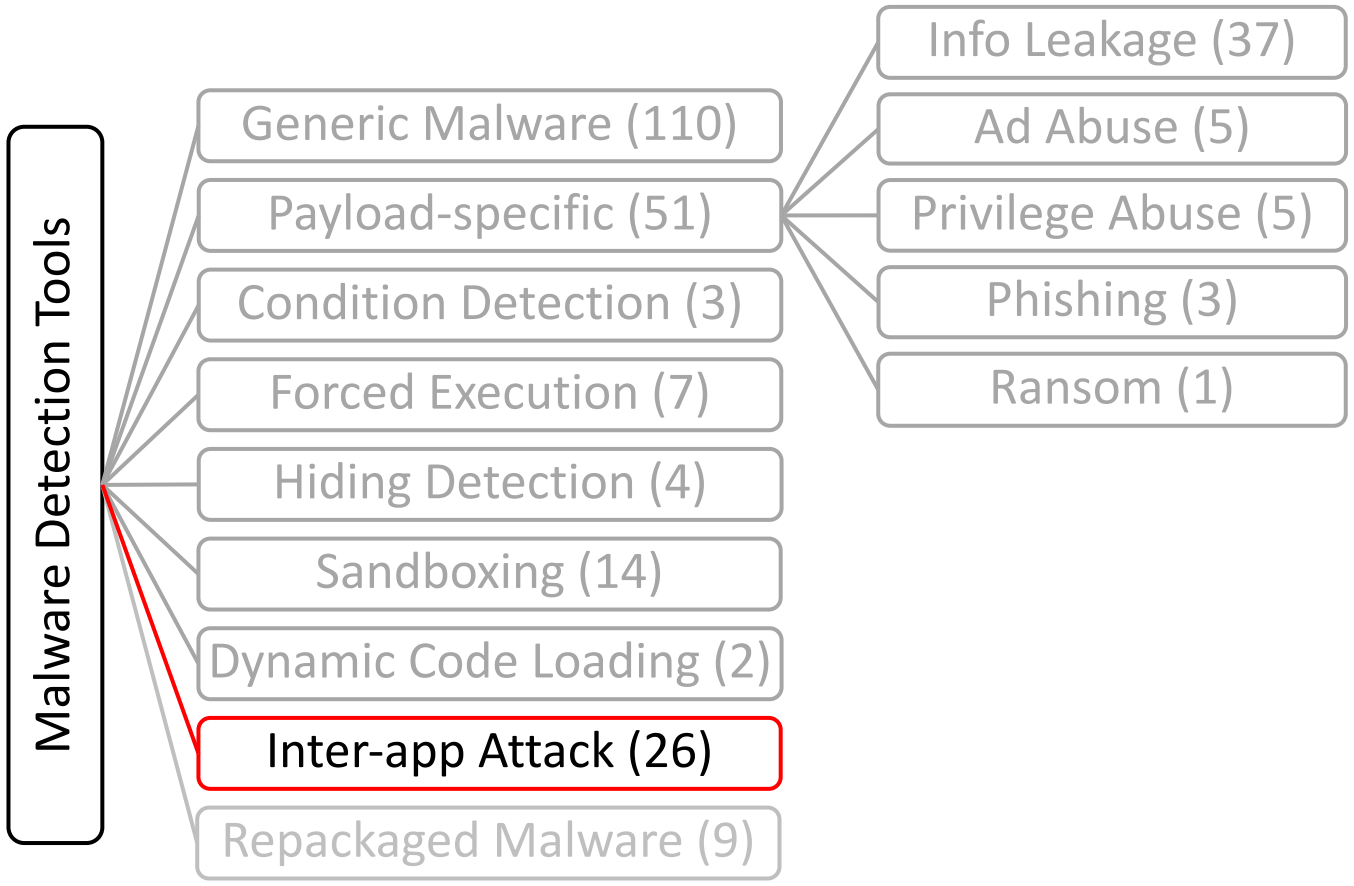
Analysis platforms to help researchers in analyzing malicious apps

Tool Categorization



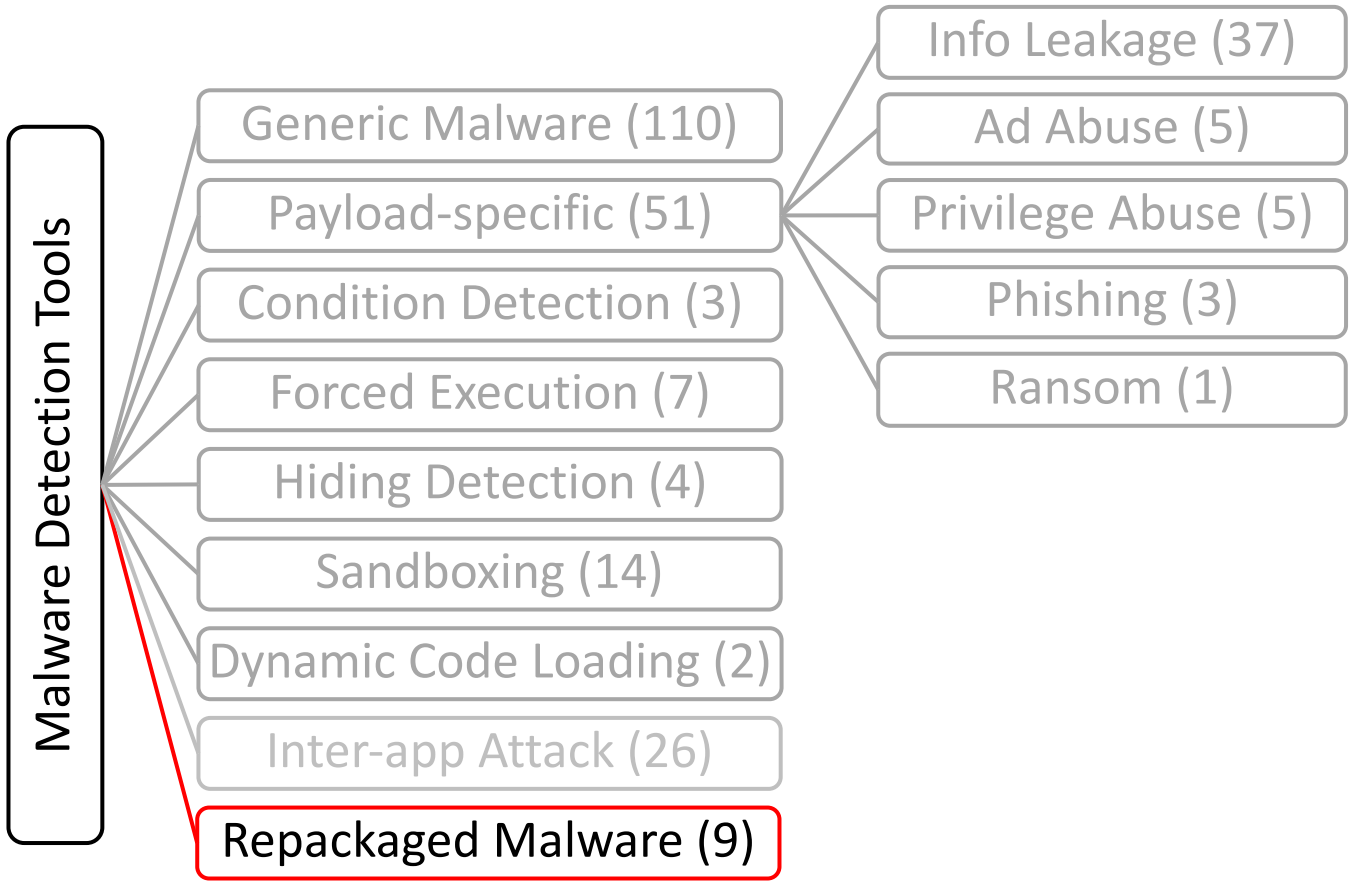
Locate unintended dynamic code loading

Tool Categorization



Identify apps that collude to perform joint attacks

Tool Categorization



Identify repackaged malware

Why Does Malware Evade Detection?

Static Analysis

Cannot reason about dynamically-loaded and obfuscated code
Hard to scale for real apps

Why Does Malware Evade Detection?

Static Analysis

Cannot reason about dynamically-loaded and obfuscated code
Hard to scale for real apps

Benchmark	Tool	Precision	Recall	F-measure
DroidBench (158 apps)	FlowDroid	88	72	79
	Amandroid	66	56	61
	DroidSafe	88	89	88
IccBench (24 apps)	FlowDroid	100	65	79
	Amandroid	85	100	92

FlowDroid

Apps	# Exp. Flows	# Detected Flows
App #13	15	17 (15 TP, 2 FP)
App #21	12	4 (4 TP, 0 FP)
12 apps	36	0
11 apps	19	0 [EX/OM]
Total	**82**	**21**

Exceptions,
out of memory (256 GB)

[3] Junbin Zhang, Yingying Wang, Lina Qiu, and Julia Rubin. *Analyzing Android Taint Analysis Tools: FlowDroid, Amandroid, and DroidSafe*. IEEE Transactions on Software Engineering, 2021.

[6] Lina Qiu, Yingying Wang, and Julia Rubin. *Analyzing the Analyzers: FlowDroid/IccTA, AmanDroid, and DroidSafe*. ISSTA 2018, **ACM SIGSOFT Distinguished Paper Award**.

Why Does Malware Evade Detection?

Static Analysis

Cannot reason about dynamically-loaded and obfuscated code
Hard to scale for real apps

Benchmark	Tool	Precision	Recall	F-measure
DroidBench (158 apps)	FlowDroid	88	72	79
	Amandroid	66	56	61
	DroidSafe	88	89	88
IccBench (24 apps)	FlowDroid	100	65	79
	Amandroid	85	100	92

FlowDroid

Apps	# Exp. Flows	# Detected Flows
App #13	15	17 (15 TP, 2 FP)
App #21	12	4 (4 TP, 0 FP)
12 apps	36	0
11 apps	19	0 [EX/OM]
Total	**82**	**21**

reflection , framework modeling, etc

[3] Junbin Zhang, Yingying Wang, Lina Qiu, and Julia Rubin. *Analyzing Android Taint Analysis Tools: FlowDroid, Amandroid, and DroidSafe*. IEEE Transactions on Software Engineering, 2021.

[6] Lina Qiu, Yingying Wang, and Julia Rubin. *Analyzing the Analyzers: FlowDroid/IccTA, AmanDroid, and DroidSafe*. ISSTA 2018, **ACM SIGSOFT Distinguished Paper Award**.

Why Does Malware Evade Detection?

Static Analysis	*Cannot reason about dynamically-loaded and obfuscated code* *Hard to scale for real apps*
Dynamic Analysis	*Cannot reach hard-to-trigger code*

```
1 List apps = getRunningApps();
2 if (apps.contains("com.bankapp")) {
3    openOverlayWindow();
4    String password = getInputText();
5    sendToInternet(password);
6 }
```

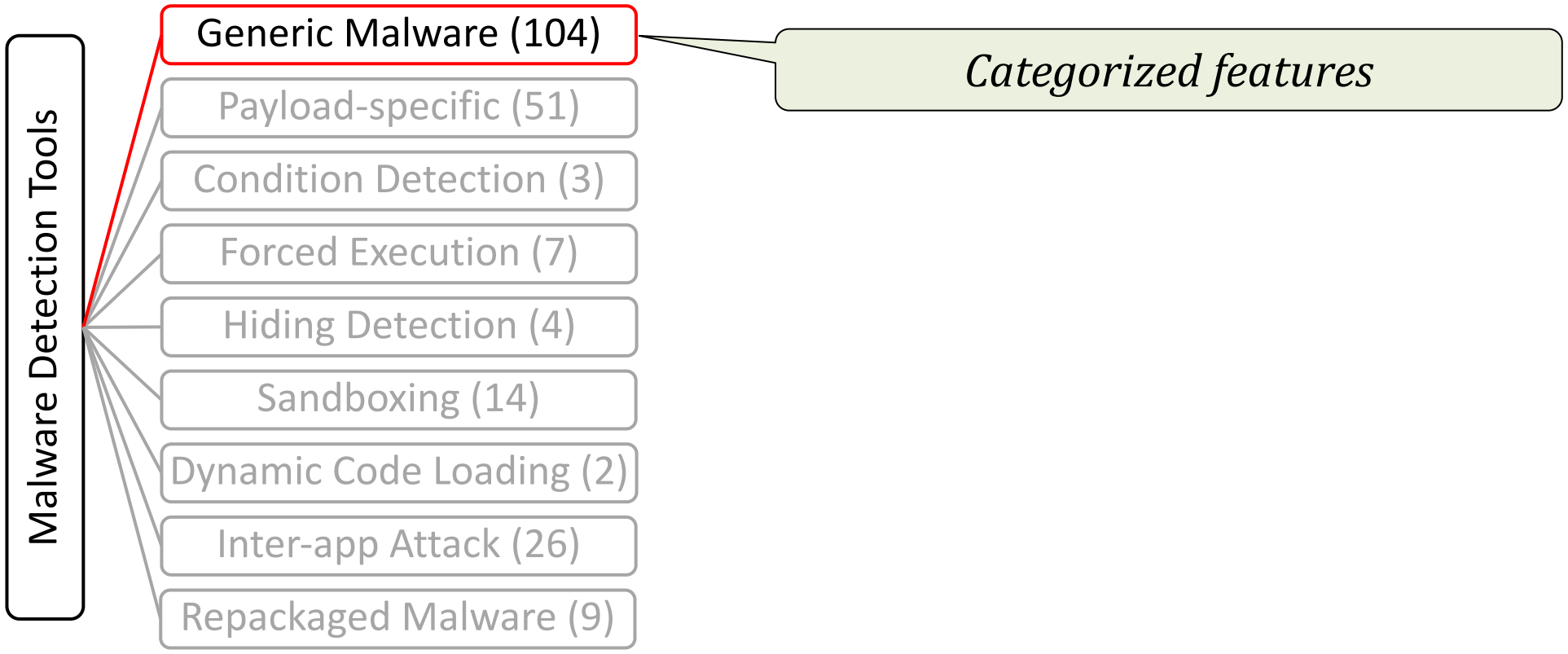
[5] Duling Lai and Julia Rubin. *Goal-Driven Exploration for Android Applications*. ASE 2019.

Why Does Malware Evade Detection?

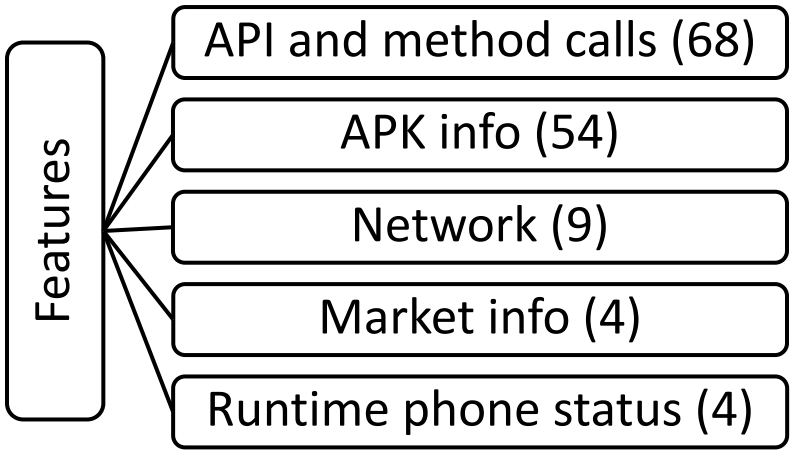
Static Analysis	*Cannot reason about dynamically-loaded and obfuscated code* *Hard to scale for real apps*
Dynamic Analysis	*Cannot reach hard-to-trigger code*
ML-based Detection	*Not reliable and not explainable*

[4] Michael Cao, Sahar Badihi, Khaled Ahmed, Peiyu Xiong, and Julia Rubin. On Benign Features in Malware Detection. ASE 2020, NIER Track.

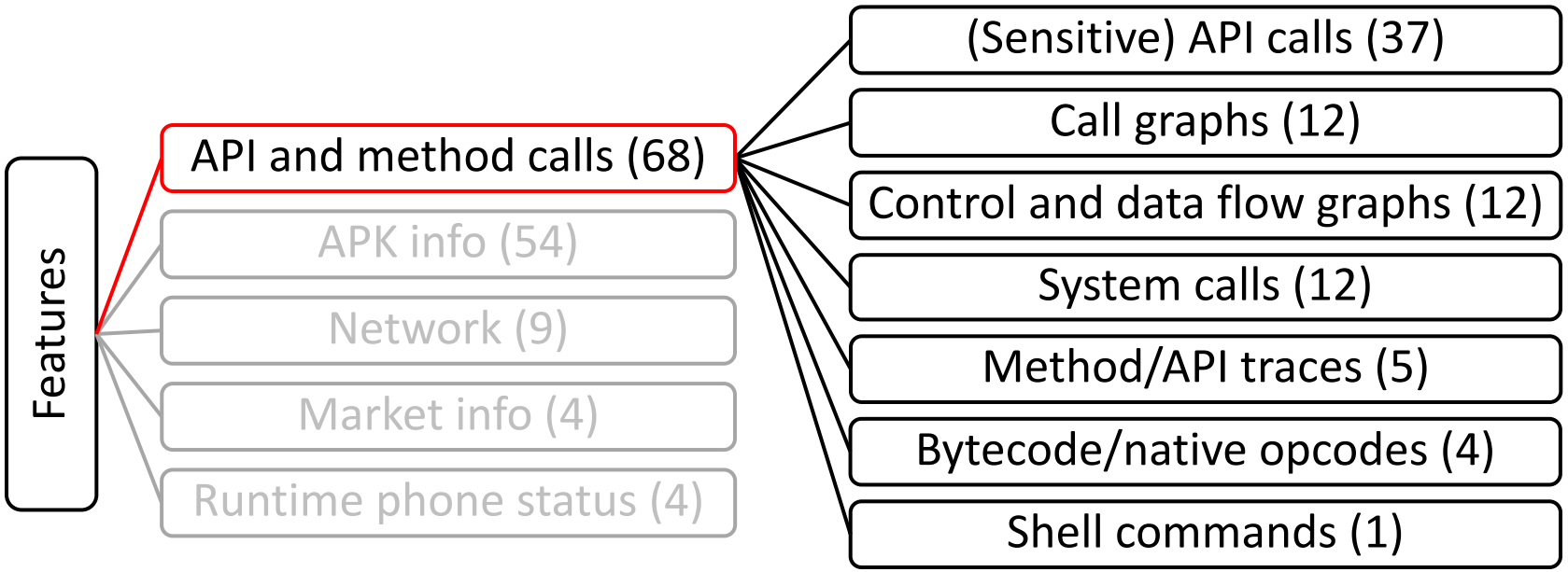
Tool Categorization



Features Commonly Used for Malware Detection

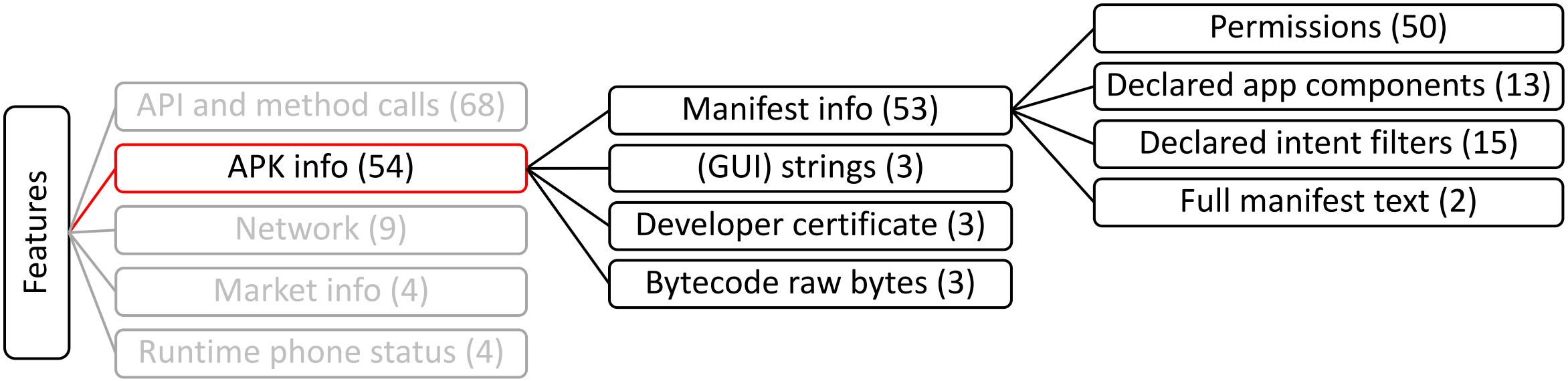


Features Commonly Used for Malware Detection

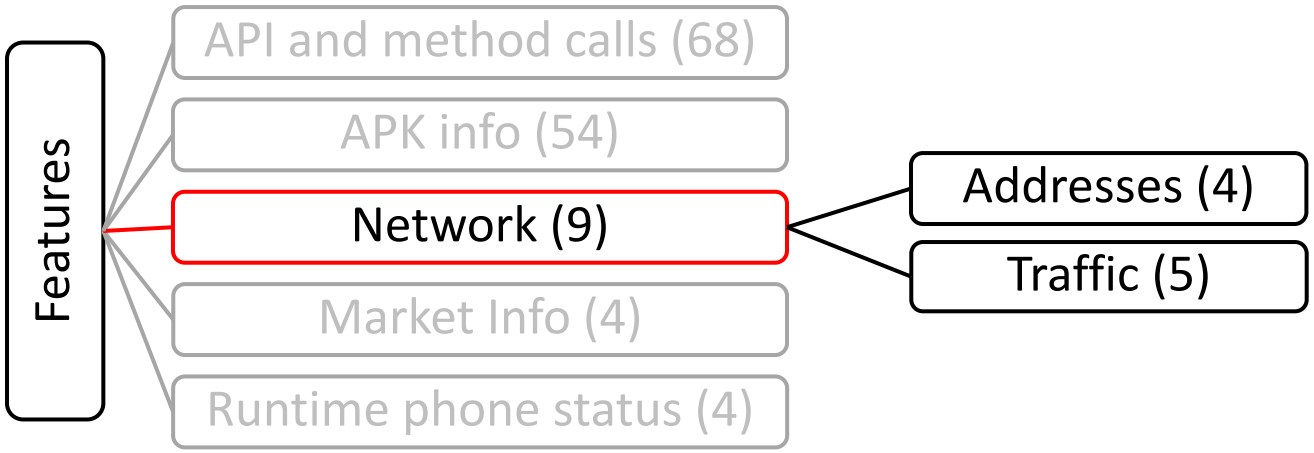


Statically or dynamically extracted from bytecode, native code, hidden APKs

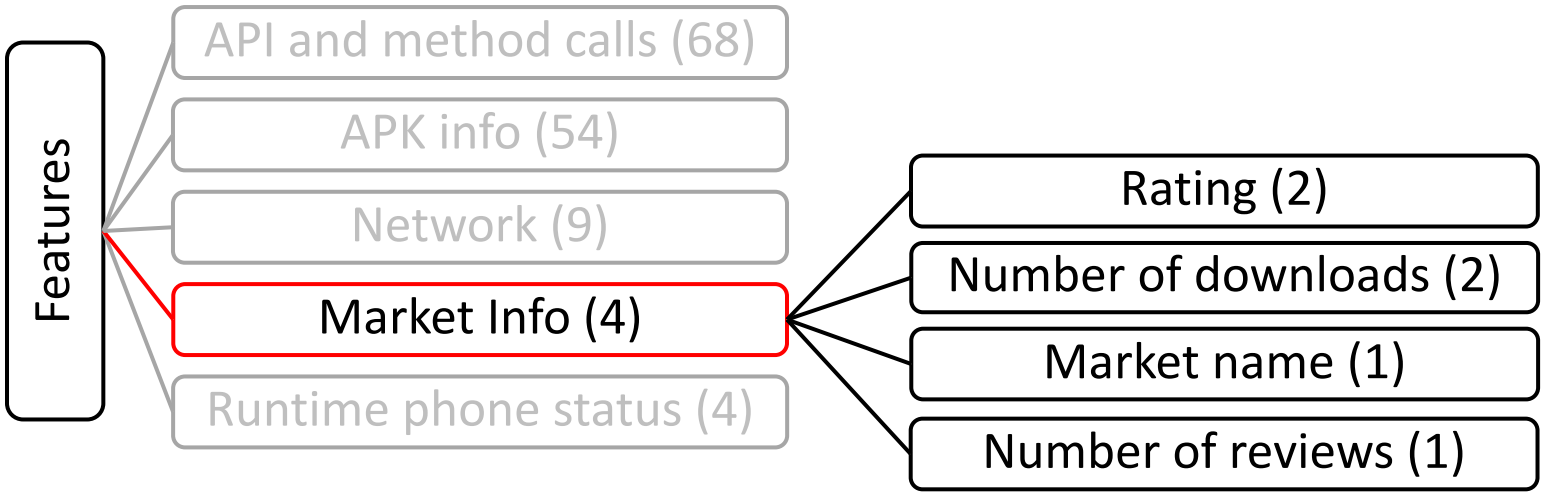
Features Commonly Used for Malware Detection



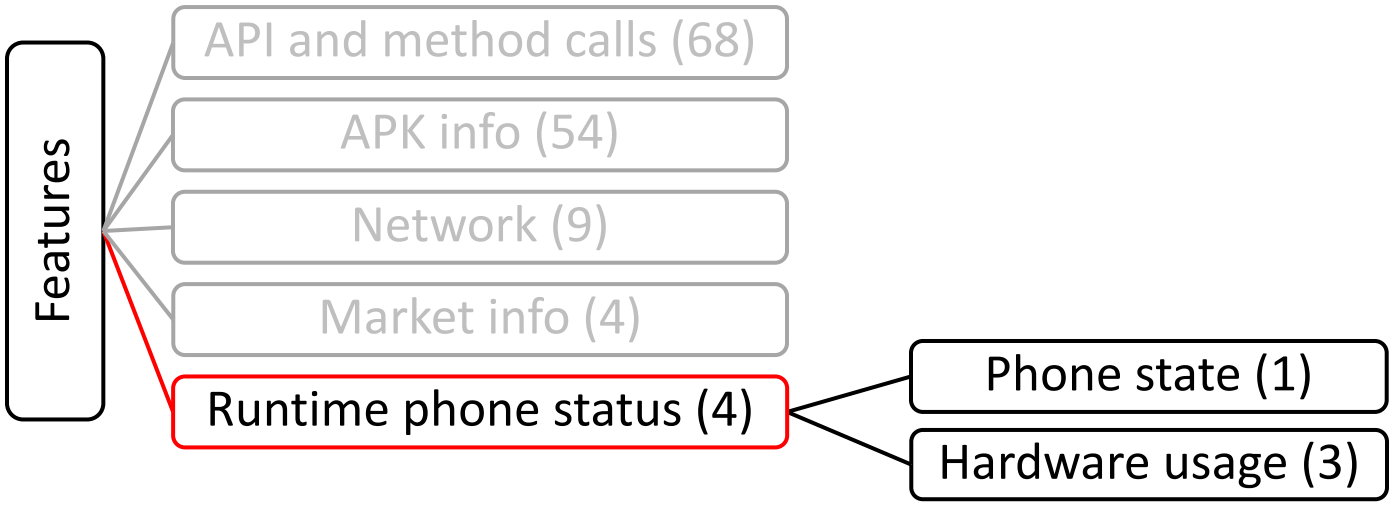
Features Commonly Used for Malware Detection



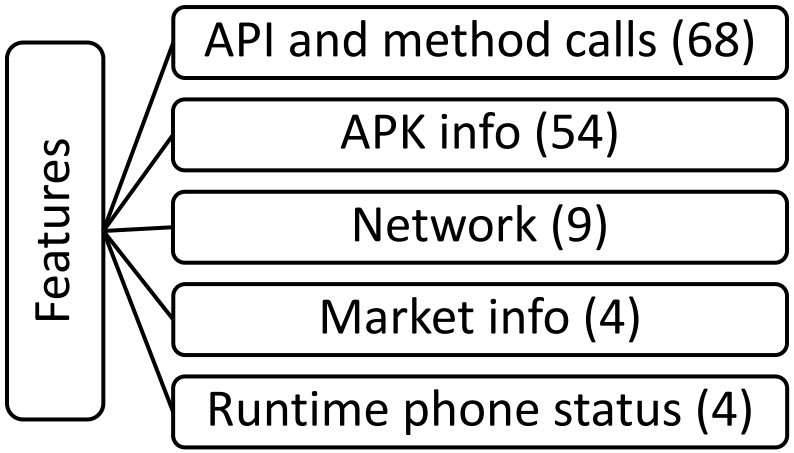
Features Commonly Used for Malware Detection



Features Commonly Used for Malware Detection

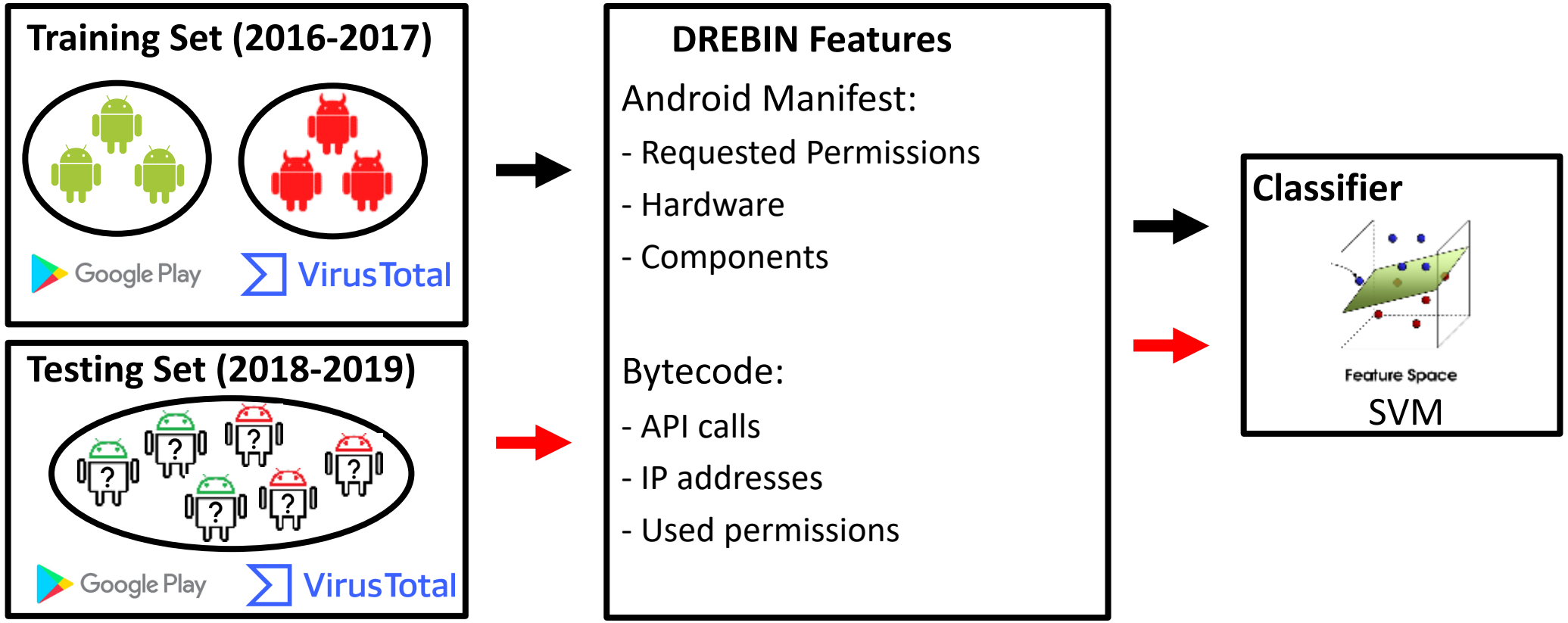


Observations



- Most commonly used:
  Permissions (50) and API calls (37)
- Commonly extracted via static analysis

Case Study



[4] Michael Cao, Sahar Badihi, Khaled Ahmed, Peiyu Xiong, and Julia Rubin. On Benign Features in Malware Detection. ASE 2020, NIER Track.

Case Study: Lessons Learned

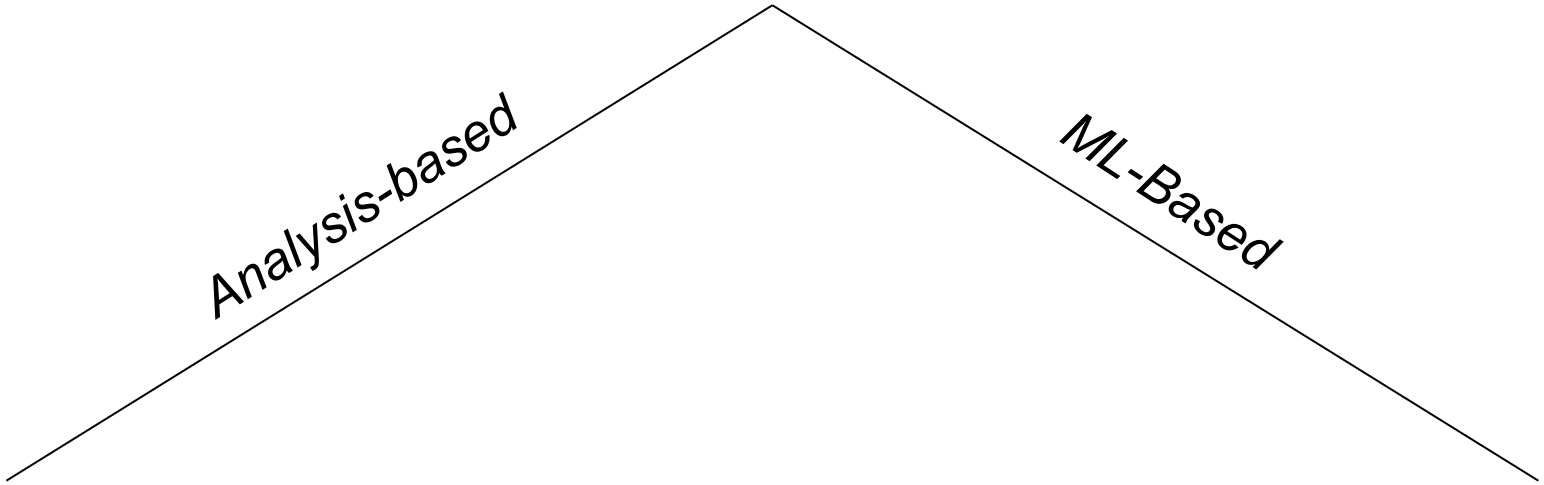
- Reproduced high performance (F1-measure = 96%) but…
- Learned “unreliable” correlations,
  e.g., that benign apps use analytics and crash logging libraries

[4] Michael Cao, Sahar Badihi, Khaled Ahmed, Peiyu Xiong, and Julia Rubin. On Benign Features in Malware Detection. ASE 2020, NIER Track.

Why Does Malware Evade Detection?

Static Analysis	*Cannot reason about dynamically-loaded and obfuscated code* *Hard to scale for real apps*
Dynamic Analysis	*Cannot reach hard-to-trigger code*
ML-based Detection	*Not reliable and not explainable*

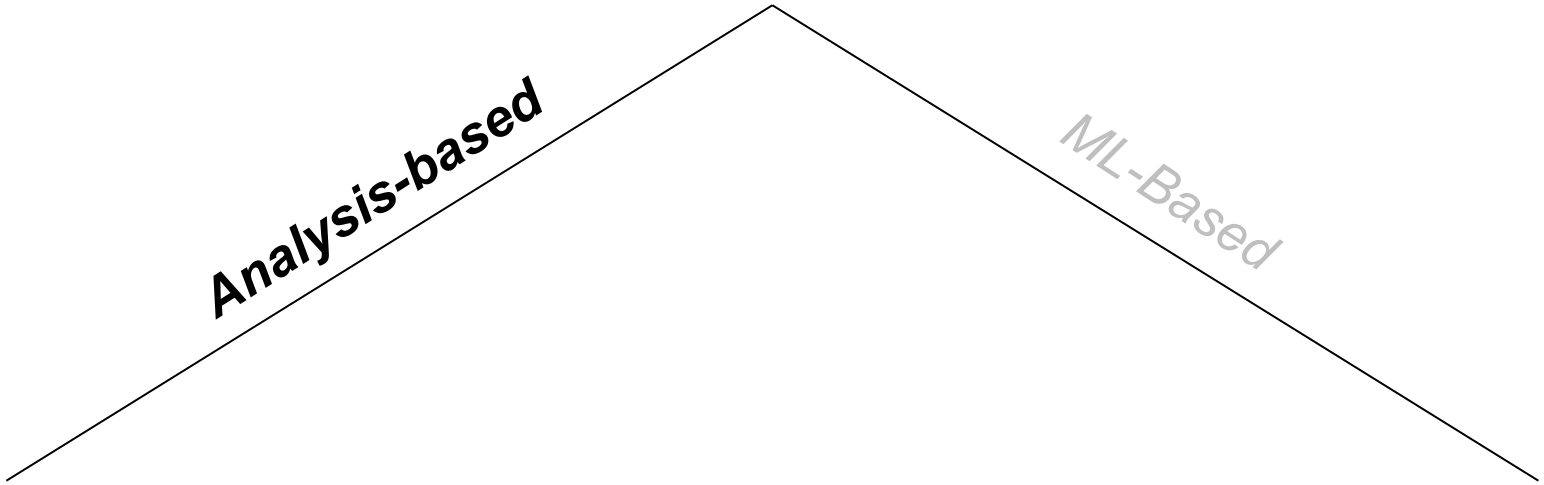
Our Current Work



On-device malware detection

Feature and Data Representation

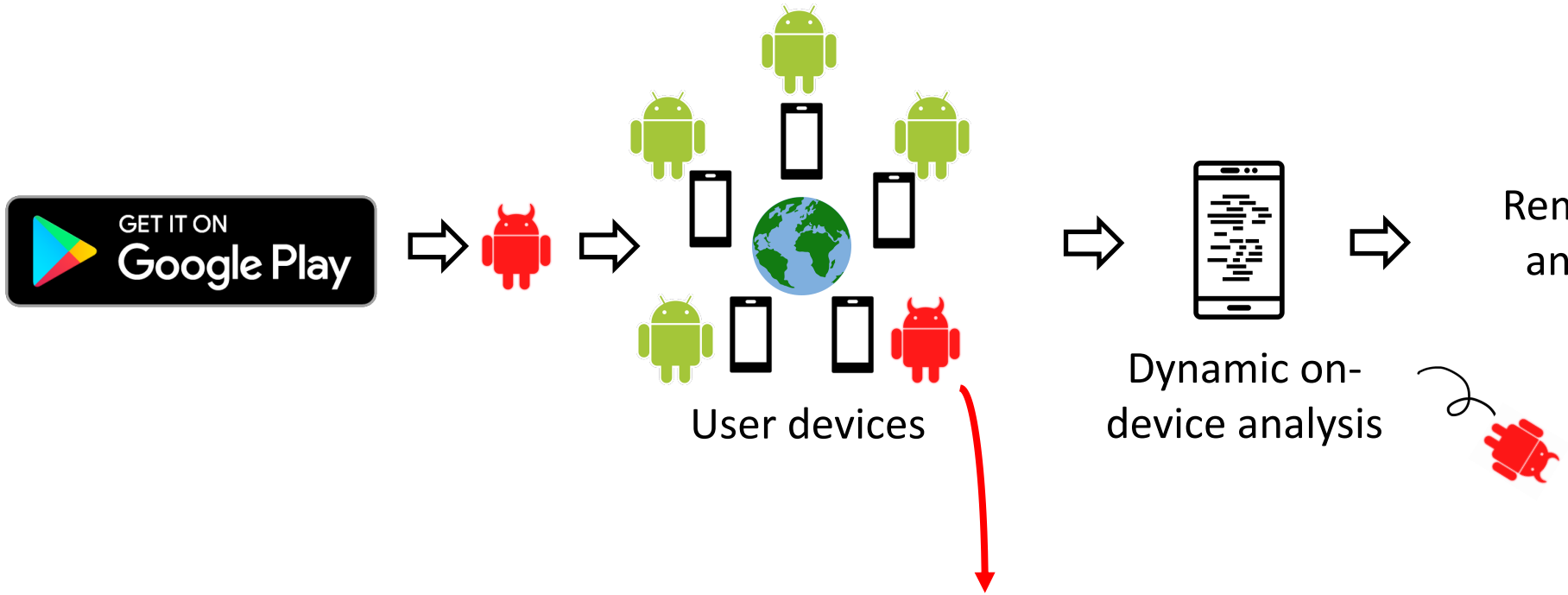
Our Current Work



On-device malware detection

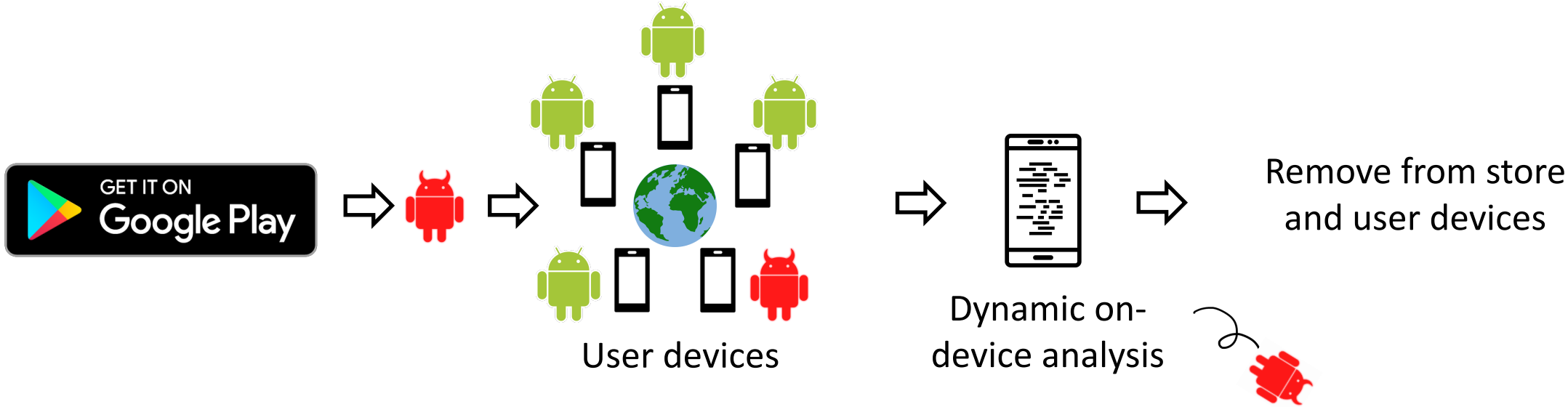
Feature and Data Representation

On-Device Malware Detection




```
1 List apps = getRunningApps();
2 if (apps.contains("com.bankapp")) {
3    openOverlayWindow();
4    String password = getInputText();
5    sendToInternet(password);
6 }
```

On-Device Malware Detection



Requirements:

1. Efficient: no visible slowdown or battery life degradation
2. Explainable: help the analyst to verify the malicious behavior

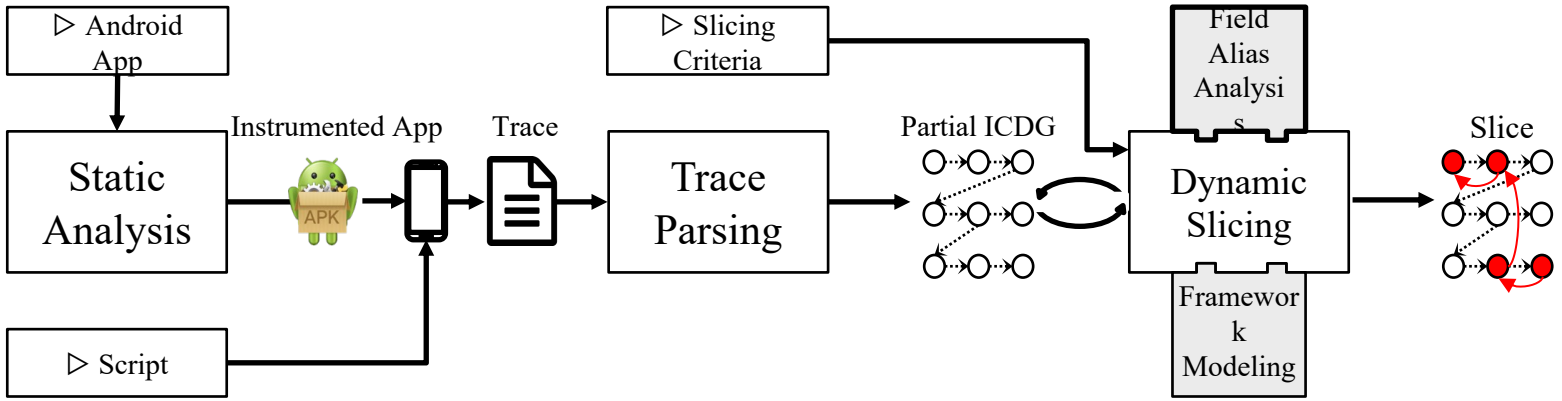
On-Device Malware Detection: Main Idea

Track a range of malicious patterns via
lightweight control- and data-flow analysis techniques

Dynamic Analysis Infrastructure

1. Dynamic slicing [2]
   - Light instrumentation, only to collect traces
   - Efficient analysis of the trace to recover data flows by alias analysis

MANDOLINE



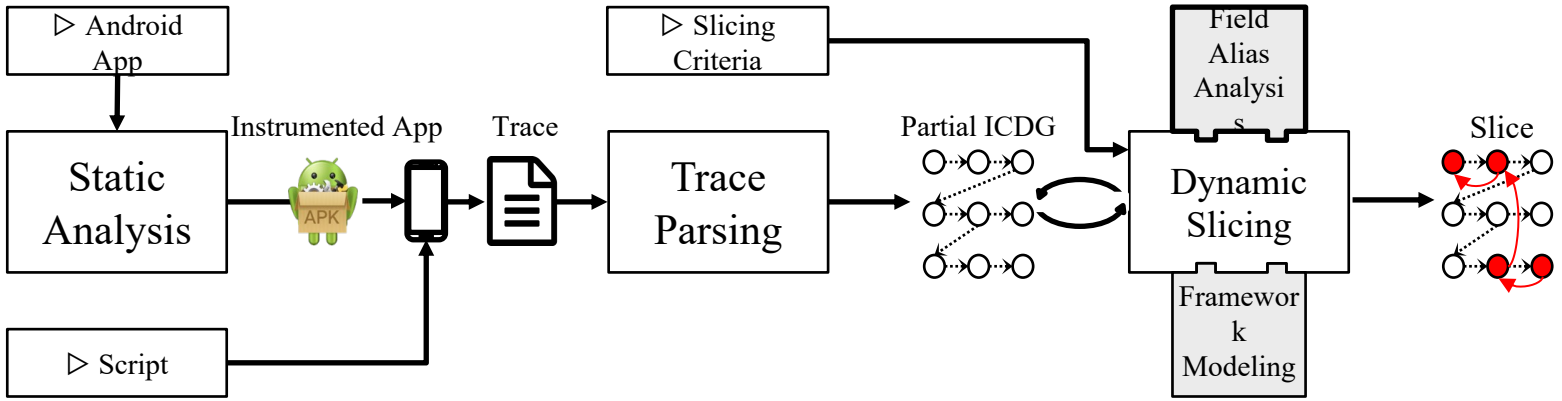
2. Path-Aware Taint Analysis (current work)

[2] Khaled Ahmed, Mieszko Lis, and Julia Rubin. *MANDOLINE: Dynamic Slicing of Android Applications with Trace-Based Alias Analysis*, ICST 2021, **Distinguished Paper Award**

Dynamic Analysis Infrastructure

1. Dynamic slicing [2]
   - Light instrumentation, only to collect traces
   - Efficient analysis of the trace to recover data flows by alias analysis

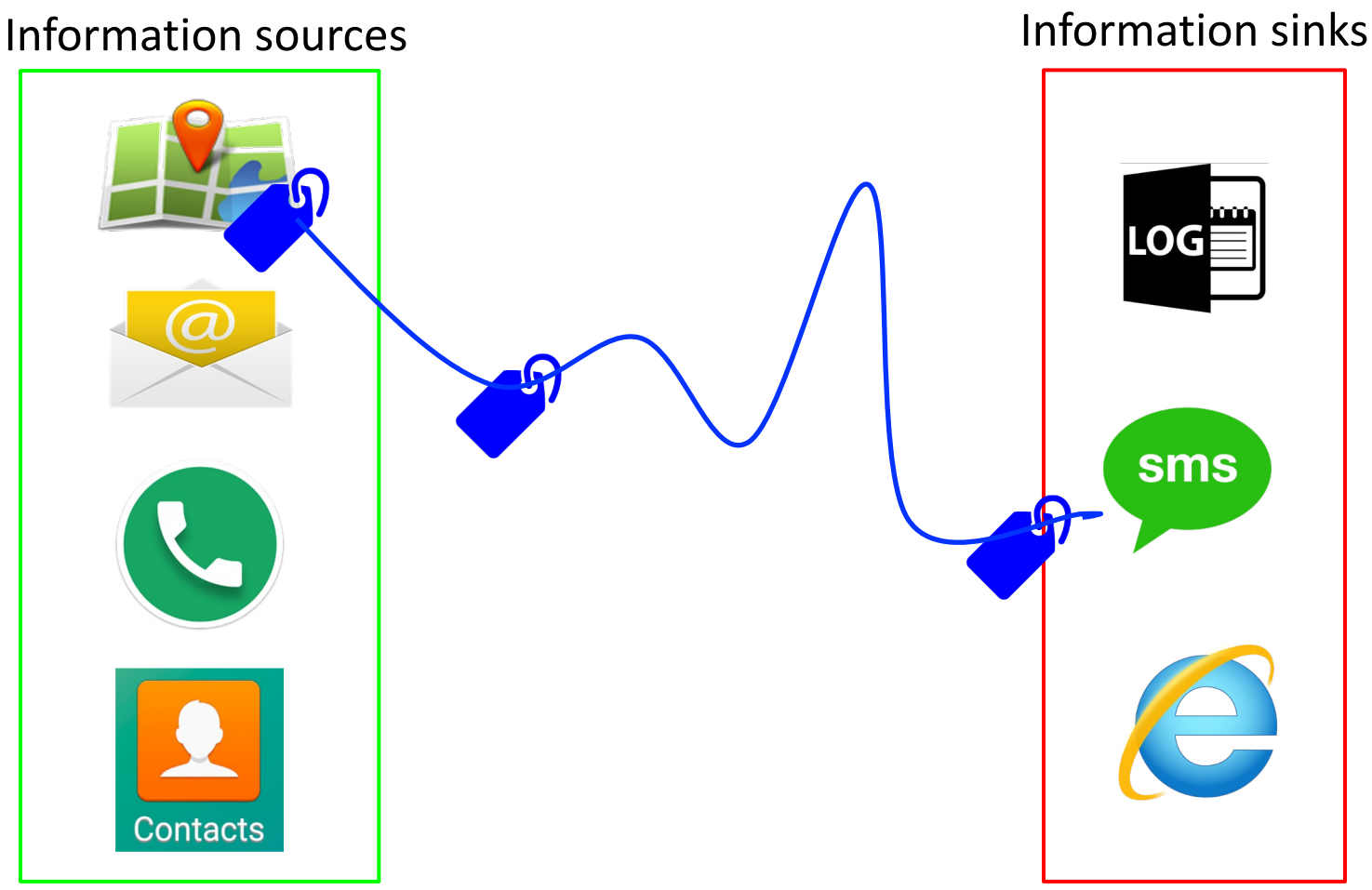
MANDOLINE



2. **Path-Aware Taint Analysis (current work)**

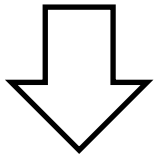
[2] Khaled Ahmed, Mieszko Lis, and Julia Rubin. *MANDOLINE: Dynamic Slicing of Android Applications with Trace-Based Alias Analysis*, ICST 2021, **Distinguished Paper Award**

Dynamic Taint Analysis



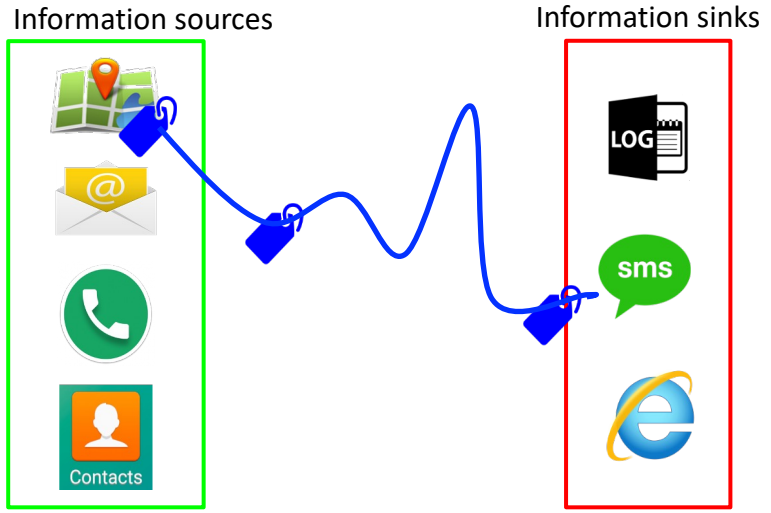
Existing Dynamic Analysis Techniques

- *TaintDroid*: An Information-flow Tracking System for Realtime Privacy Monitoring on Smartphones. William Enck et al., OSDI'10.
- *TaintART*: A Practical Multi-level Information-Flow Tracking System for Android RunTime. Mingshen Sun et al., CCS'16
- *TaintMan*: An ART-Compatible Dynamic Taint Analysis Framework on Unmodified and Non-Rooted Android Devices. Wei You et al., IEEE Trans. Dependable Secur. Comput. 17, 1 (Jan.-Feb. 2020), 209–222.



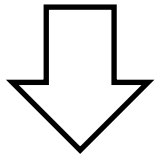
Efficient: Track a few bits per tainted variable

Limited # sources



Existing Dynamic Analysis Techniques

- *TaintDroid*: An information-flow tracking system for realtime privacy monitoring on smartphones. William Enck et al., OSDI'10.
- *TaintART*: A Practical Multi-level Information-Flow Tracking System for Android RunTime. Mingshen Sun et al., CCS'16
- *TaintMan*: An ART-Compatible Dynamic Taint Analysis Framework on Unmodified and Non-Rooted Android Devices. Wei You et al., IEEE Trans. Dependable Secur. Comput. 17, 1 (Jan.-Feb. 2020), 209–222.

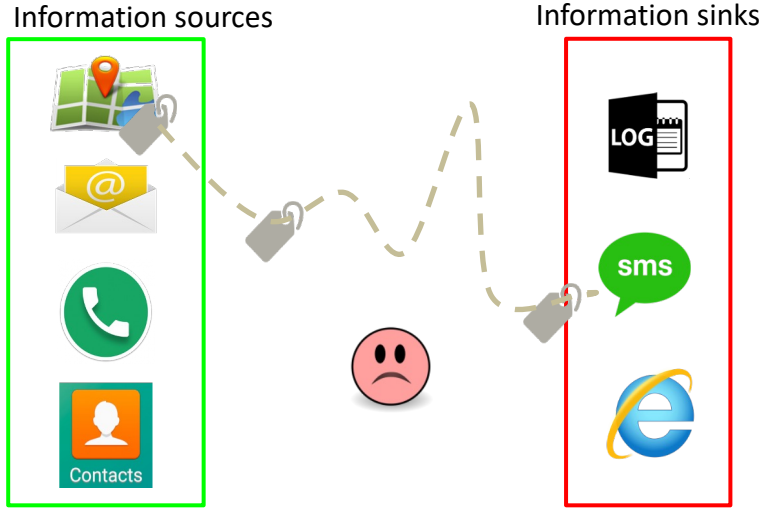


Efficient: Track a few bits per tainted variable

Limited # sources

Only report [source,sink] pairs, no paths

[User input, Internet]
[Location, Internet]



Existing Dynamic Analysis Techniques

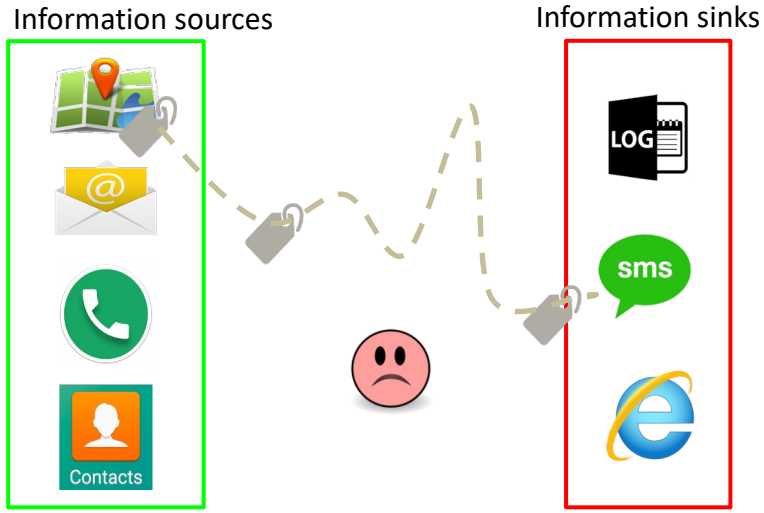
- *TaintDroid*: An information-flow tracking system for realtime privacy monitoring on smartphones. William Enck et al., OSDI'10.
- *TaintART*: A Practical Multi-level Information-Flow Tracking System for Android RunTime. Mingshen Sun et al., CCS'16
- *TaintMan*: An ART-Compatible Dynamic Taint Analysis Framework on Unmodified and Non-Rooted Android Devices. Wei You et al., IEEE Trans. Dependable Secur. Comput. 17, 1 (Jan.-Feb. 2020), 209–222.

Efficient: Track a few bits per tainted v

Limited # sources

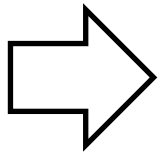
Only report [source,sink] pairs, no paths

[User input, Internet]
[Location, Internet]



Our Path-Aware Dynamic Taint Analysis for Android

```
 3   String getUserInfo() {
 4     String input1 = getUserInput();
 4T    int input1T = 0b0001;
 5     String input2 = getUserInput();
 5T    int input2T = 0b0001;
 6     Location loc = getLocation();
 6T    int locT = 0b0010;
 7     String size = input1.getLength();
 7T    int sizeT = input1T;
 8     String lang = loc.getLanguage();
 8T    int langT = locT;
 9     String data = size + lang;
 9T    int dataT = sizeT | langT;
10     … }
21   void sendInfo() {
22     String data = getInfo();
22T   int dataT = valT;
23T   if (dataT != 0){report(data, dataT);}
23     sendToInternet(data);                    //sink
24 }}
```

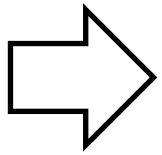


```
 3   String getUserInfo() {
 4     String input1 = getUserInput();
 4T    Taint input1T = injectT(4, null, null);
 5     String input2 = getUserInput();
 5T    Taint input2T = injectT(5, null, null);
 6     Location loc = getLocation();
 6T    Taint locT = injectT(6, null, null);
 7     String size = input1.getLength();
 7T    Taint sizeT = propagateT(7, input1T, null);
 8     String lang = loc.getLanguage();
 8T    Taint langT = propagateT(8, locT, null);
 9     String data = size + lang;
 9T    Taint dataT = propagateT(9, sizeT, langT);
10     … }
21   void sendInfo() {
22     String data = getInfo();
22T   int dataT = valT;
23T   checkT(dataT);
23     sendToInternet(data);                    //sink
24 }}
```

Our Path-Aware Dynamic Taint Analysis for Android

Taint marking
(one bit per taint)

```
 3   String getUserInfo() {
 4     String input1 = getUserInput();
 4T    int input1T = 0b0001;
 5     String input2 = getUserInput();
 5T    int input2T = 0b0001;
 6     Location loc = getLocation();
 6T    int locT = 0b0010;
 7     String size = input1.getLength();
 7T    int sizeT = input1T;
 8     String lang = loc.getLanguage();
 8T    int langT = locT;
 9     String data = size + lang;
 9T    int dataT = sizeT | langT;
10     … }
21   void sendInfo() {
22     String data = getInfo();
22T   int dataT = valT;
23T   if (dataT != 0){report(data, dataT);}
23     sendToInternet(data);                    //sink
24 }}
```

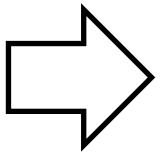


```
 3   String getUserInfo() {
 4     String input1 = getUserInput();
 4T    Taint input1T = injectT(4, null, null);
 5     String input2 = getUserInput();
 5T    Taint input2T = injectT(5, null, null);
 6     Location loc = getLocation();
 6T    Taint locT = injectT(6, null, null);
 7     String size = input1.getLength();
 7T    Taint sizeT = propagateT(7, input1T, null);
 8     String lang = loc.getLanguage();
 8T    Taint langT = propagateT(8, locT, null);
 9     String data = size + lang;
 9T    Taint dataT = propagateT(9, sizeT, langT);
10     … }
21   void sendInfo() {
22     String data = getInfo();
22T   int dataT = valT;
23T   checkT(dataT);
23     sendToInternet(data);                    //sink
24 }}
```

Our Path-Aware Dynamic Taint Analysis for Android

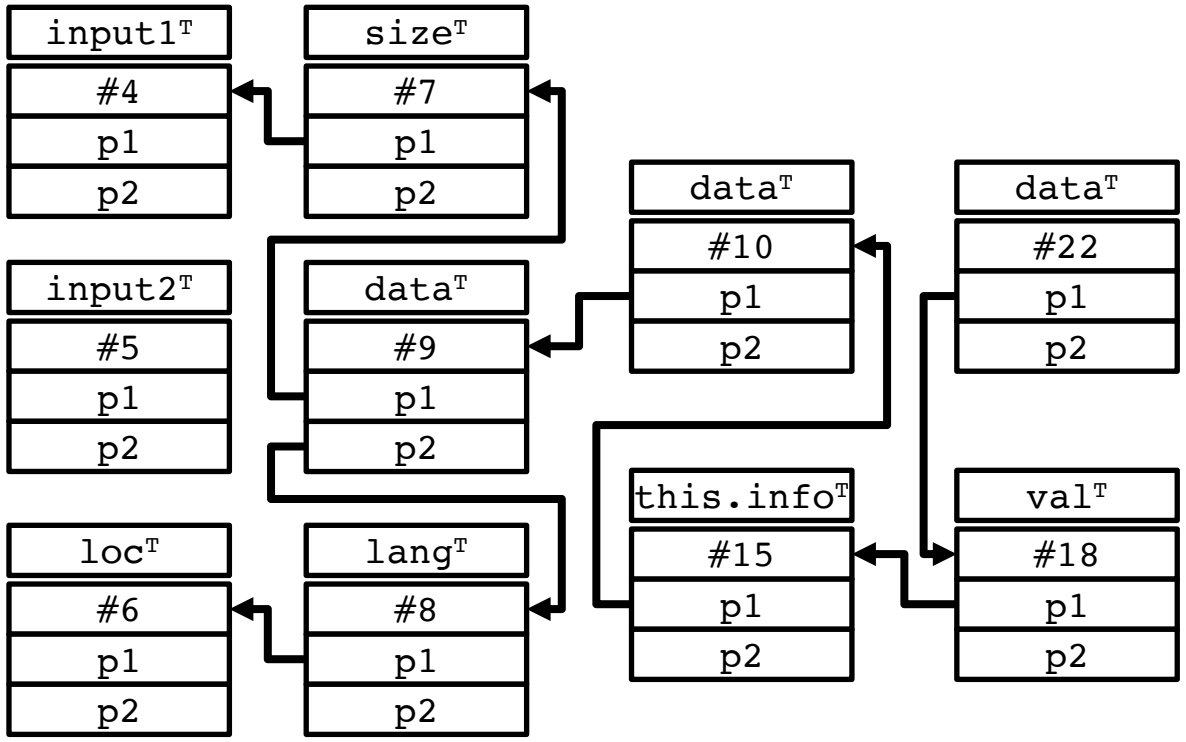
Taint marking
(one object per taint)

```
 3   String getUserInfo() {
 4     String input1 = getUserInput();
 4T    int input1T = 0b0001;
 5     String input2 = getUserInput();
 5T    int input2T = 0b0001;
 6     Location loc = getLocation();
 6T    int locT = 0b0010;
 7     String size = input1.getLength();
 7T    int sizeT = input1T;
 8     String lang = loc.getLanguage();
 8T    int langT = locT;
 9     String data = size + lang;
 9T    int dataT = sizeT | langT;
10     … }
21   void sendInfo() {
22     String data = getInfo();
22T   int dataT = valT;
23T   if (dataT != 0){report(data, dataT);}
23     sendToInternet(data);                    //sink
24 }}
```



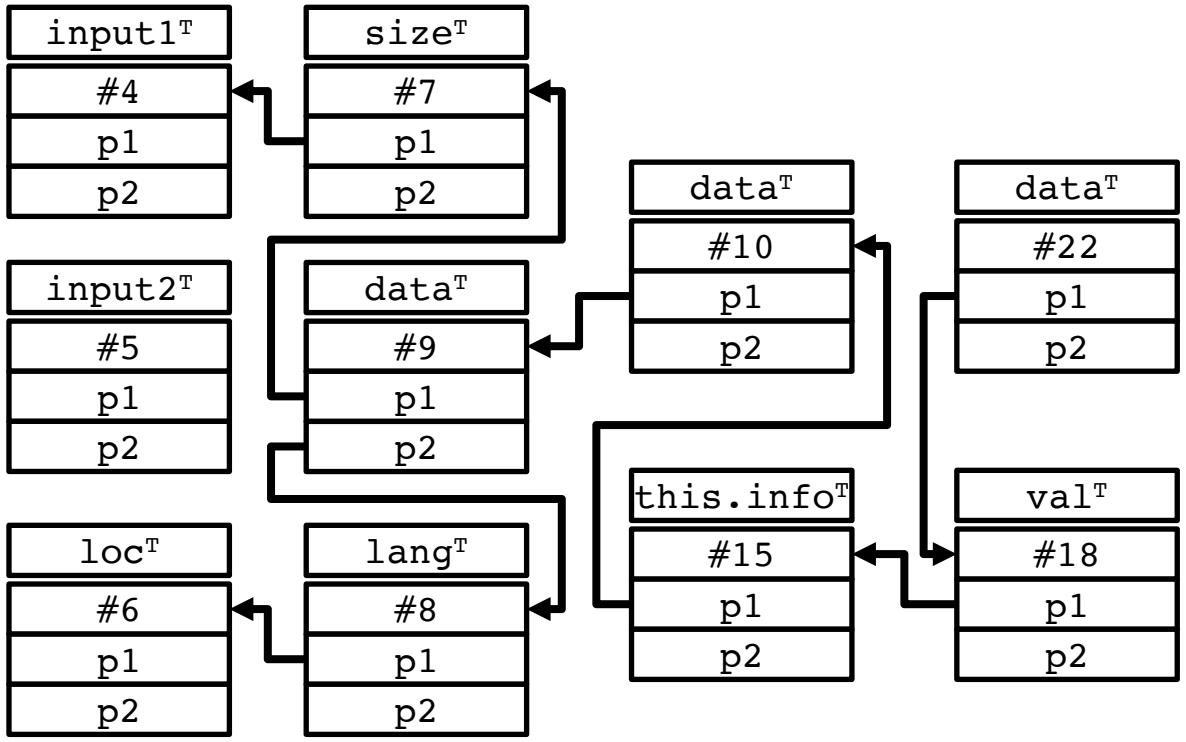
```
 3   String getUserInfo() {
 4     String input1 = getUserInput();
 4T    Taint input1T = injectT(4, null, null);
 5     String input2 = getUserInput();
 5T    Taint input2T = injectT(5, null, null);
 6     Location loc = getLocation();
 6T    Taint locT = injectT(6, null, null);
 7     String size = input1.getLength();
 7T    Taint sizeT = propagateT(7, input1T, null);
 8     String lang = loc.getLanguage();
 8T    Taint langT = propagateT(8, locT, null);
 9     String data = size + lang;
 9T    Taint dataT = propagateT(9, sizeT, langT);
10     … }
21   void sendInfo() {
22     String data = getInfo();
22T   int dataT = valT;
23T   checkT(dataT);
23     sendToInternet(data);                    //sink
24 }}
```

Our Path-Aware Dynamic Taint Analysis for Android



```
 3   String getUserInfo() {
 4     String input1 = getUserInput();
 4T    Taint input1T = injectT(4, null, null);
 5     String input2 = getUserInput();
 5T    Taint input2T = injectT(5, null, null);
 6     Location loc = getLocation();
 6T    Taint locT = injectT(6, null, null);
 7     String size = input1.getLength();
 7T    Taint sizeT = propagateT(7, input1T, null);
 8     String lang = loc.getLanguage();
 8T    Taint langT = propagateT(8, locT, null);
 9     String data = size + lang;
 9T    Taint dataT = propagateT(9, sizeT, langT);
10     … }
21   void sendInfo() {
22     String data = getInfo();
22T    int dataT = valT;
23T    checkT(dataT);
23     sendToInternet(data);                    //sink
24 }}
```

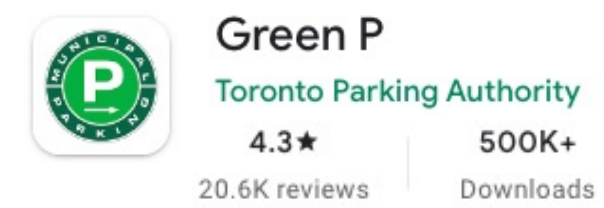
Our Path-Aware Dynamic Taint Analysis for Android

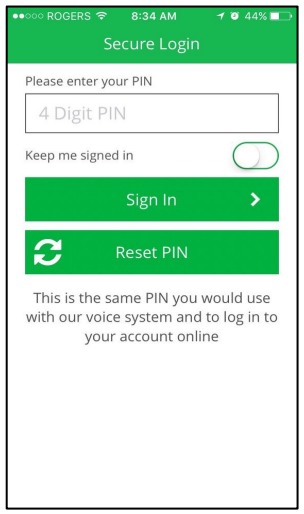


1. Tracks any number of taint sources
2. Reports all source to sink paths
3. Memory efficient: taints of out-of-scope objects are garbage collected

```
 3    String getUserInfo() {
 4      String input1 = getUserInput();
 4T     Taint input1T = injectT(4, null, null);
 5      String input2 = getUserInput();
 5T     Taint input2T = injectT(5, null, null);
 6      Location loc = getLocation();
 6T     Taint locT = injectT(6, null, null);
 7      String size = input1.getLength();
 7T     Taint sizeT = propagateT(7, input1T, null);
 8      String lang = loc.getLanguage();
 8T     Taint langT = propagateT(8, locT, null);
 9      String data = size + lang;
 9T     Taint dataT = propagateT(9, sizeT, langT);
10      … }
21    void sendInfo() {
22      String data = getInfo();
22T     int dataT = valT;
23T     checkT(dataT);
23      sendToInternet(data);                    //sink
24  }}
```

Utility: Sanitization on the Path

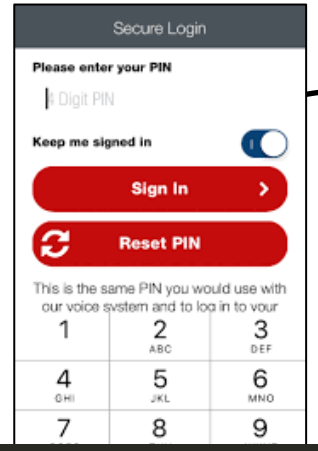




Encryption on path

```
String enc = RC4Encryption(pwd)
```

Internet



Hardcoded token

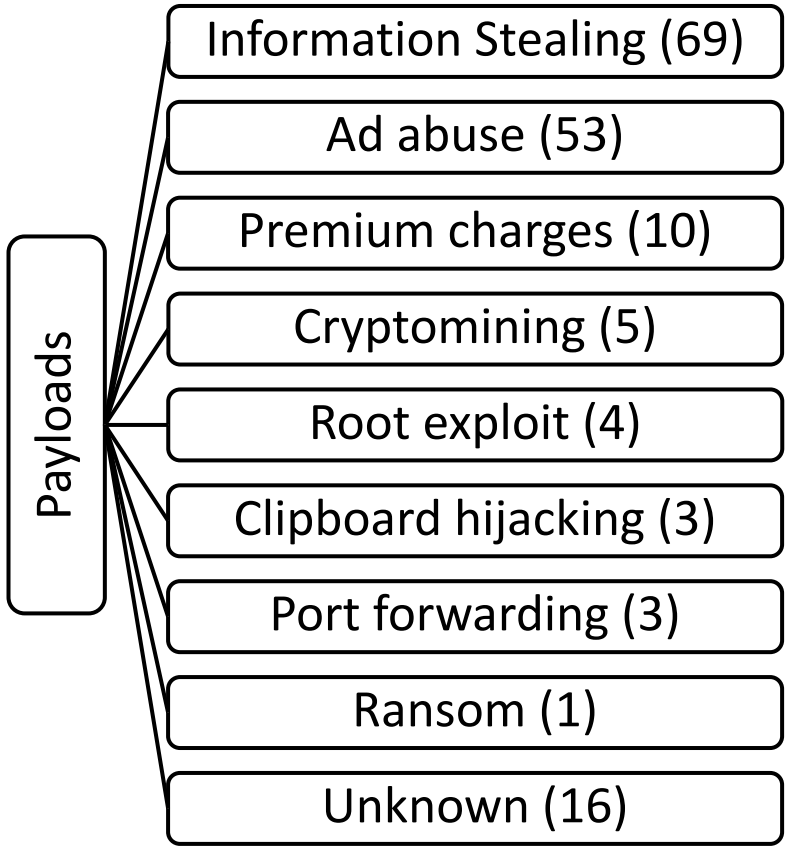
```
public final class AppData {
    private static final String PREFERENCE_NAME = "PassportParkingSettings";
    private static final String SECURE_KEY = "63fefc0690f44d74af5182ddceee21de";
```

```
String jwt = new JWTSigner(AppDate.SECURE_KEY).sign(pwd)
```

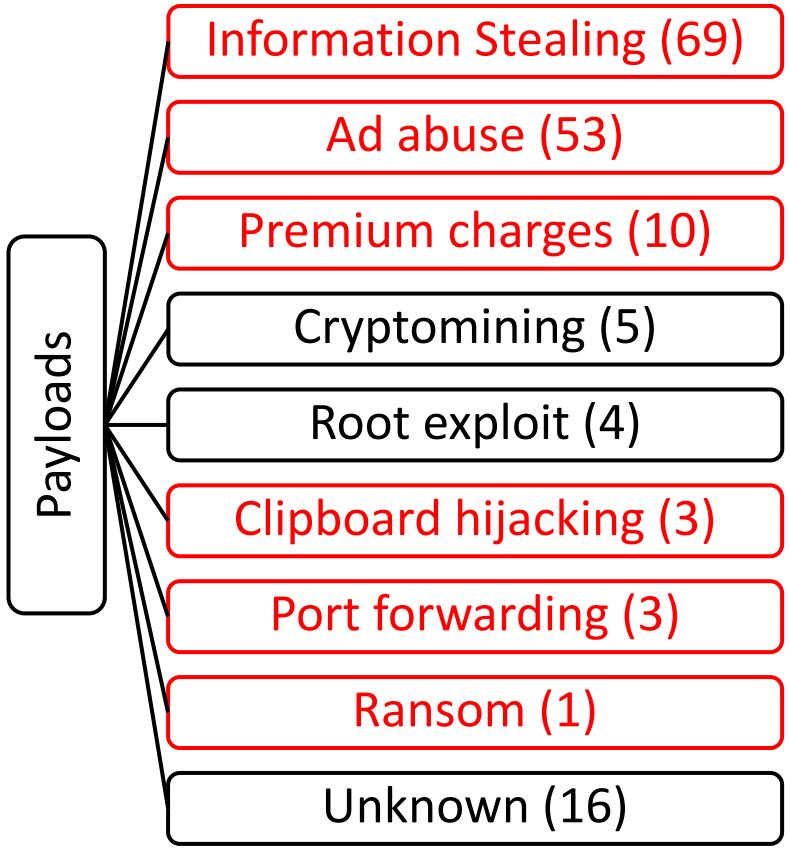
Internet

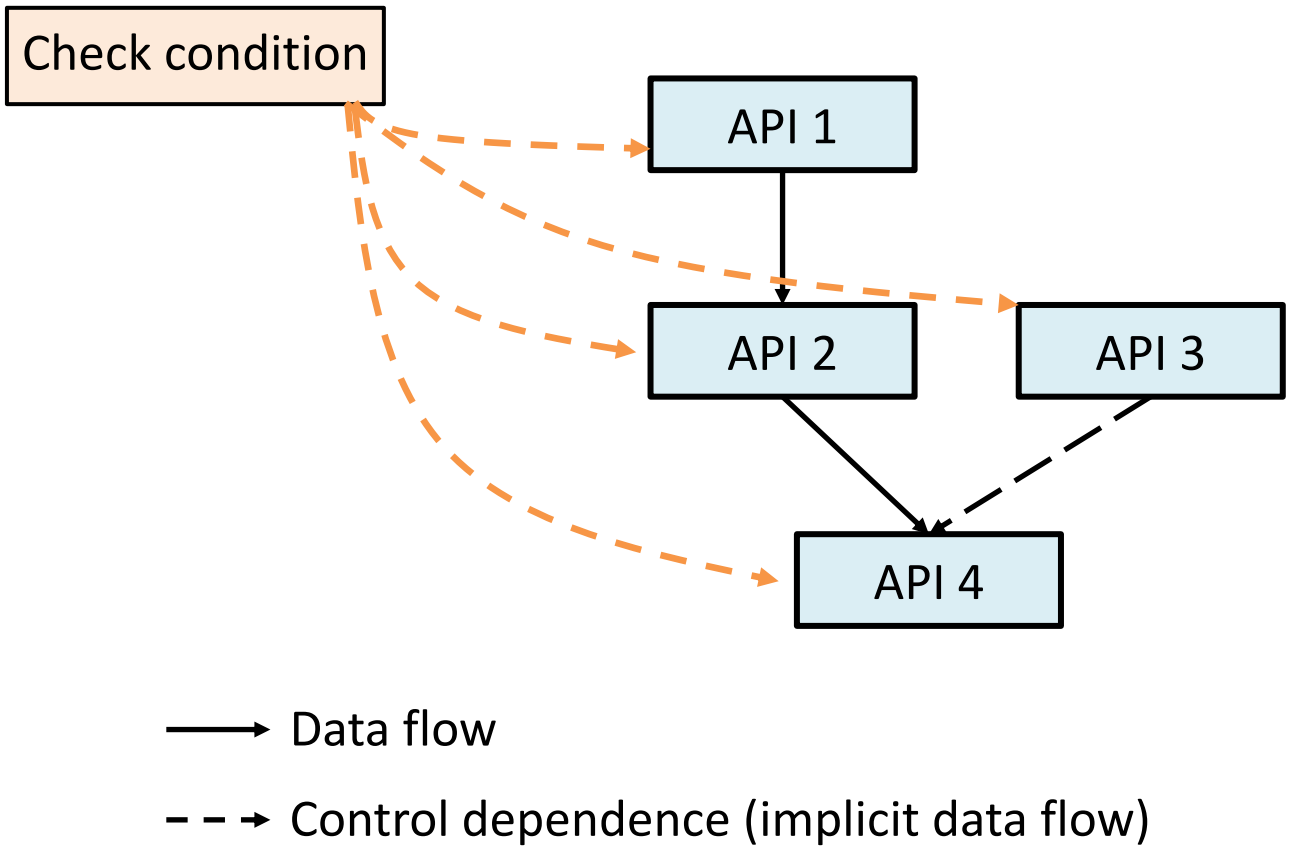
[3] Junbin Zhang, Yingying Wang, Lina Qiu, and Julia Rubin, *Analyzing Android Taint Analysis Tools: FlowDroid, Amandroid, and DroidSafe*, TSE 2021.

Taint Analysis For Malware Detection



Taint Analysis For Malware Detection

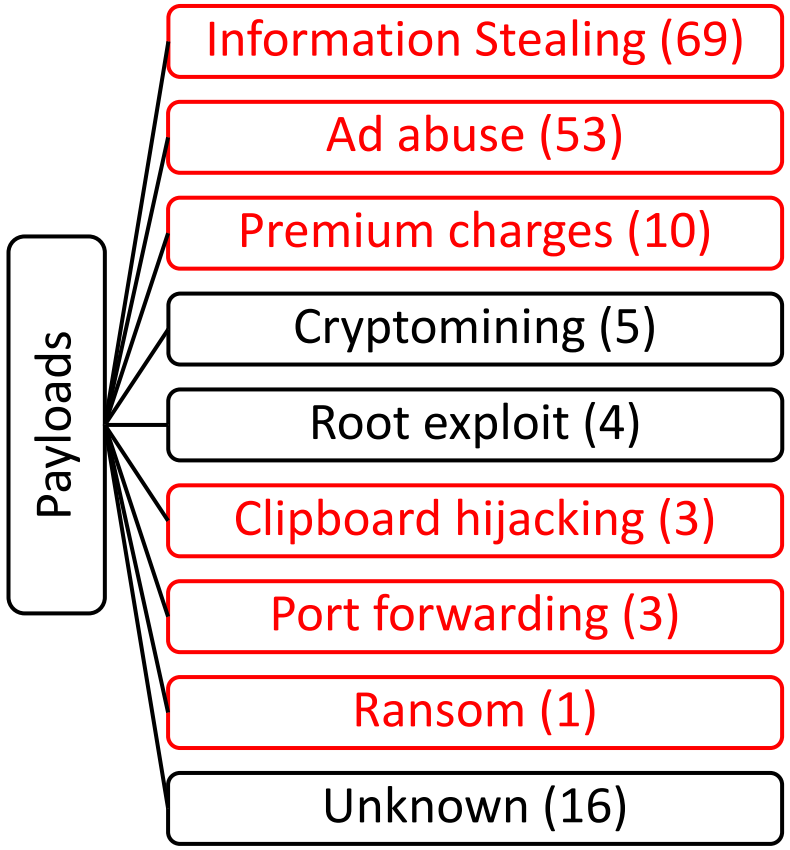




Payload: Data flow and control dependence graph

Check condition: control-dominate one or more API of the payload

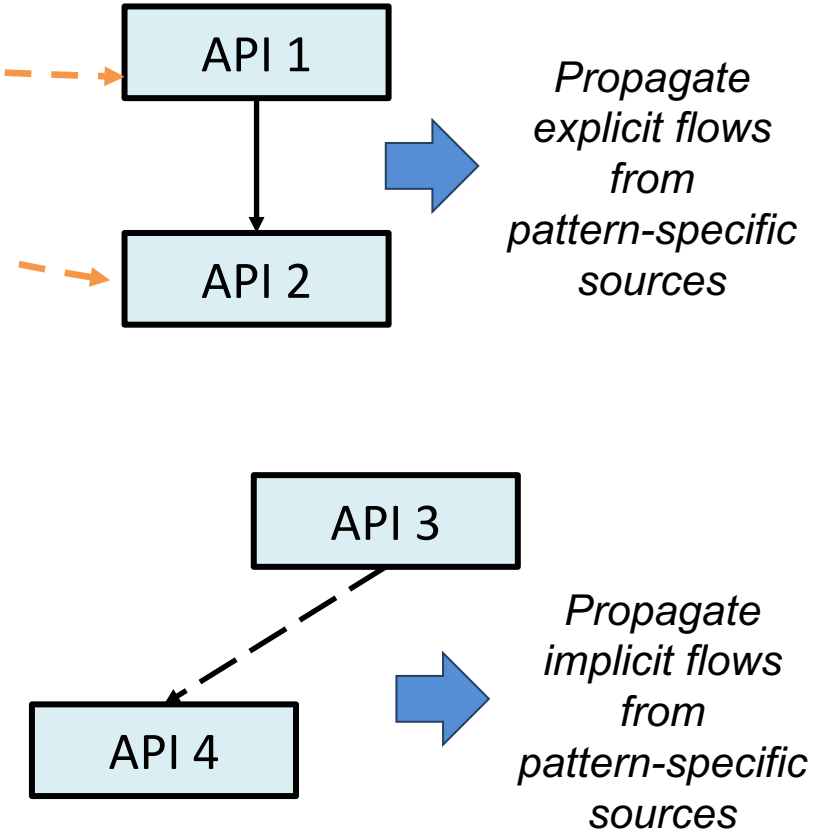
Taint Propagation



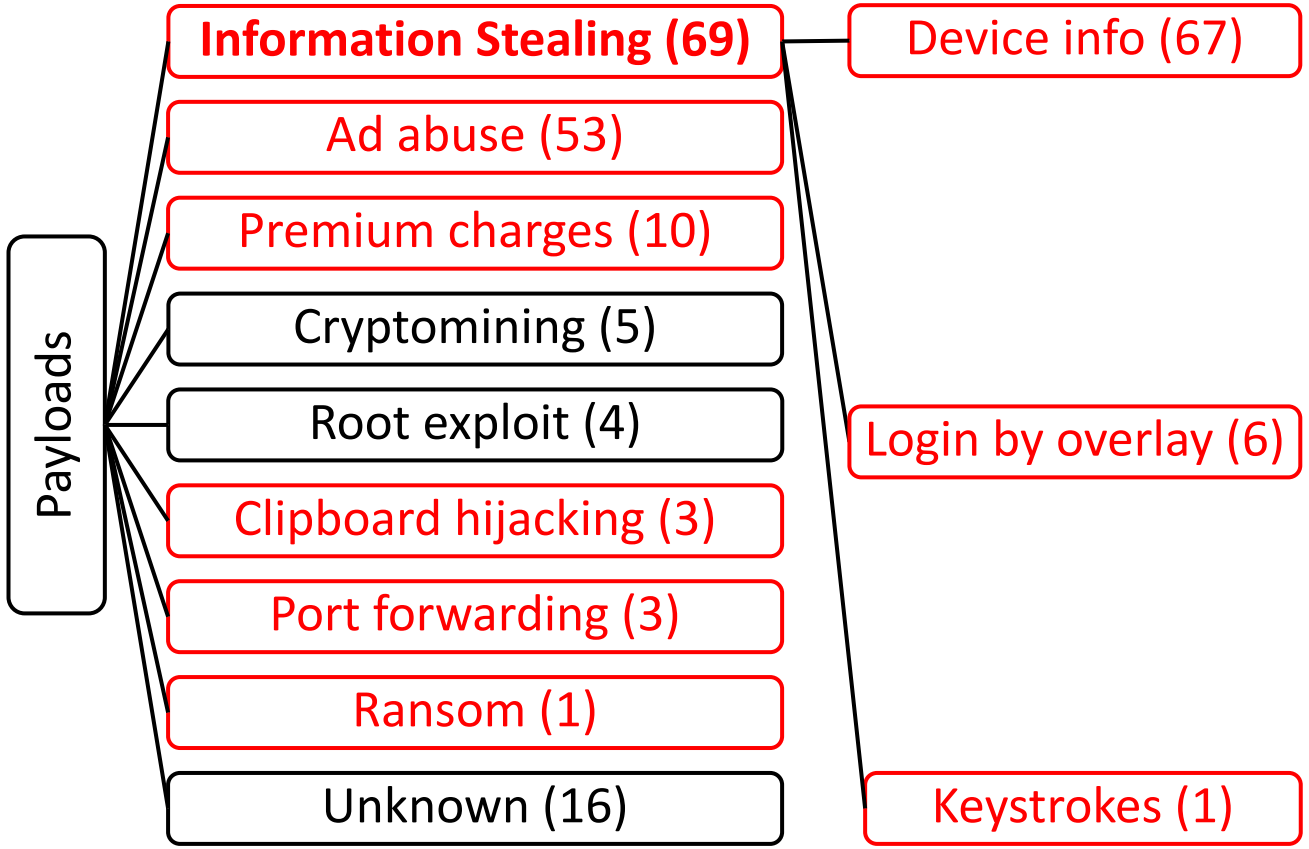
Check condition

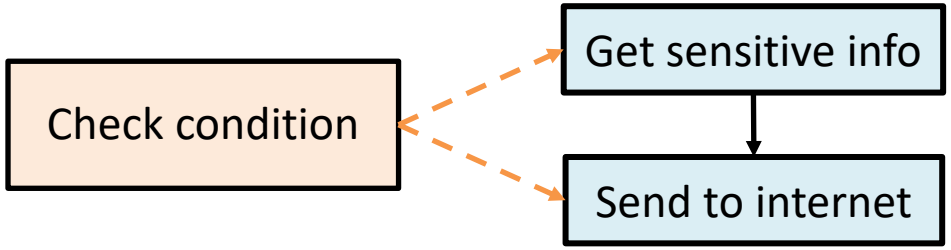
Propagate implicit flows from

Check conditions	#	%	
External server	79	75.2	
Internet	78	74.3	
SMS	2	1.9	
Device	82	78.1	
Software specs	62	59.0	
Network	43	41.0	
→ Hardware specs	28	26.7	
→ Sensors	4	3.8	
Environment	41	39.0	
Time	38	36.2	
Location	6	5.7	
Application	42	40.0	
Permissions	31	29.5	
Data format	10	9.5	
→ Probability	8	7.6	
Version	1	1.0	

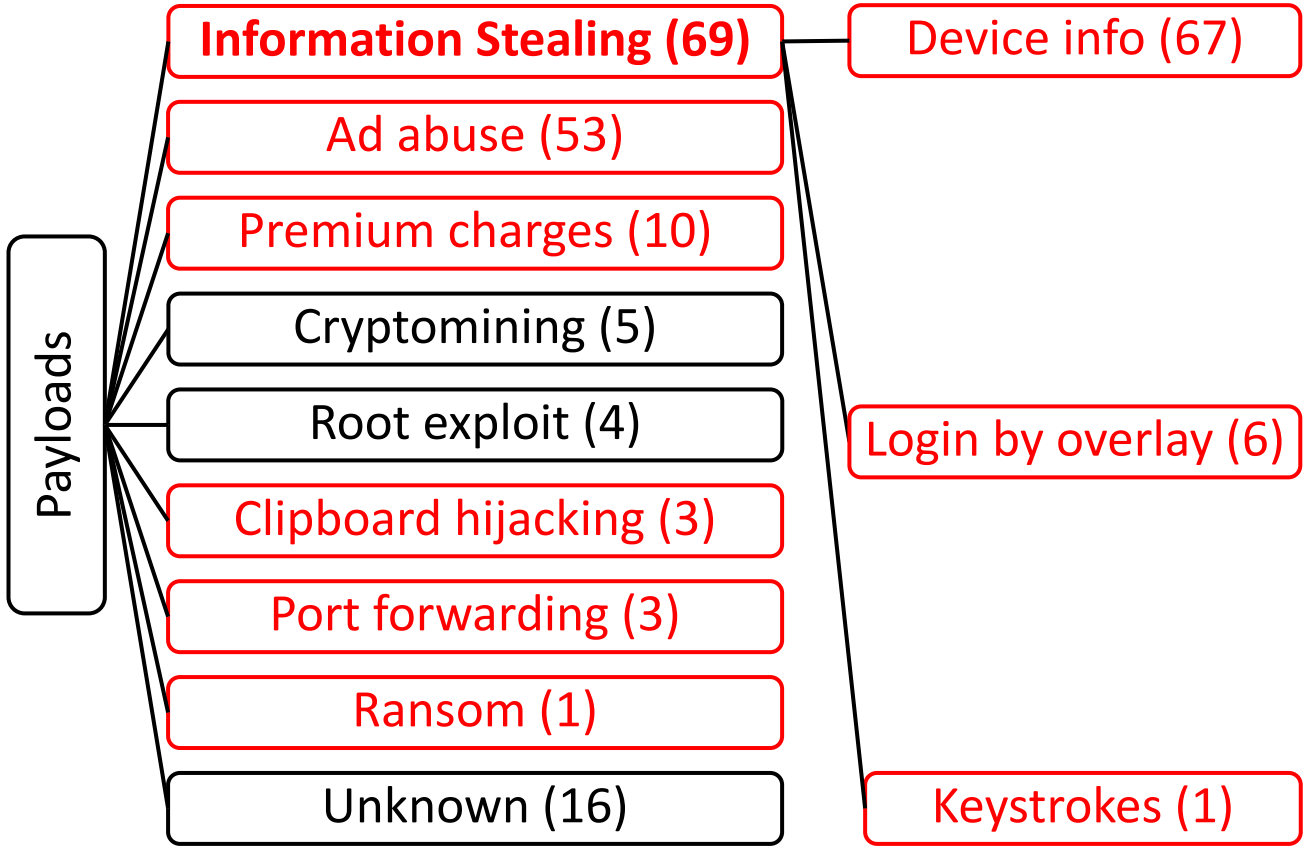


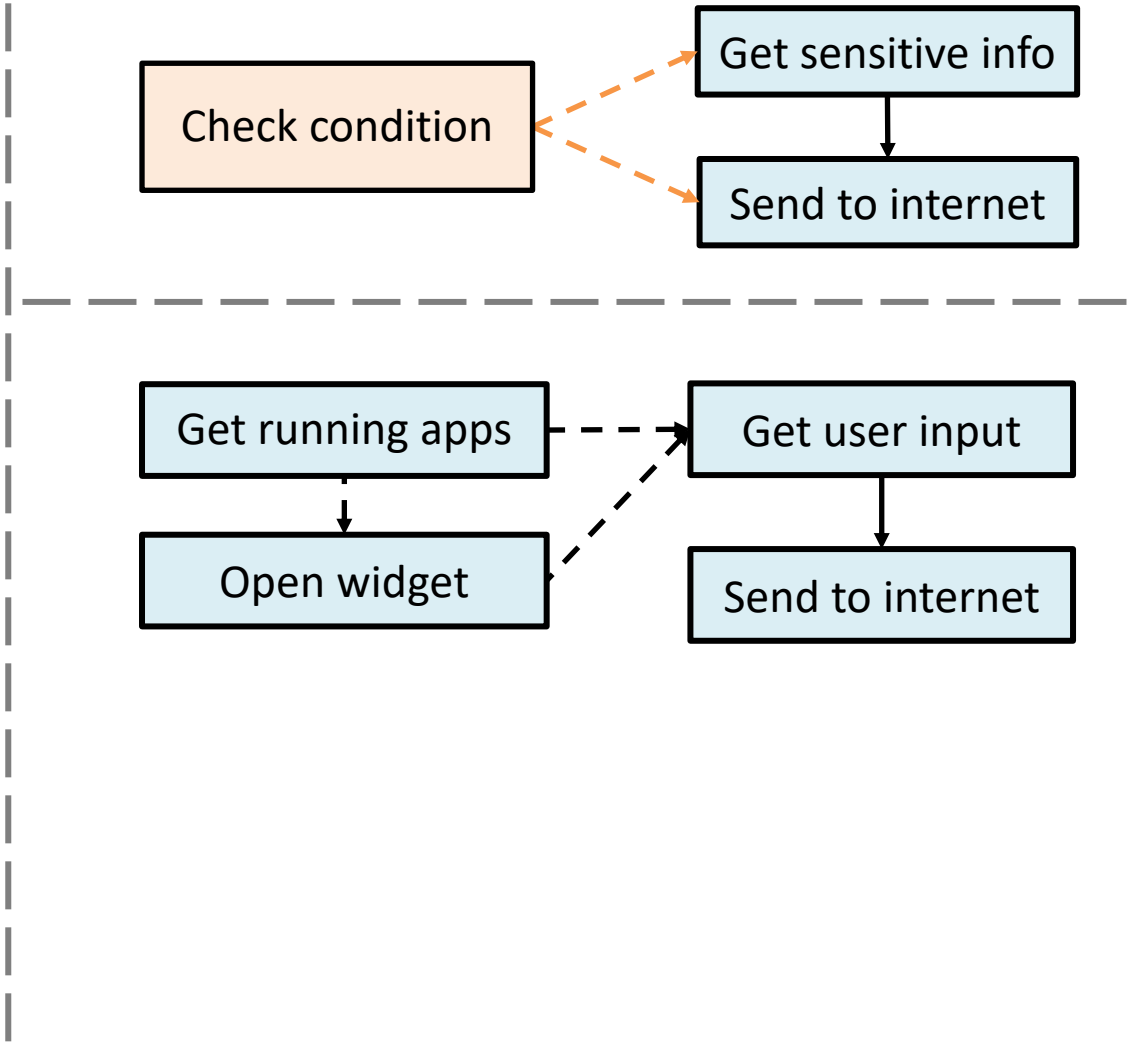
Pattern Example: Information Stealing



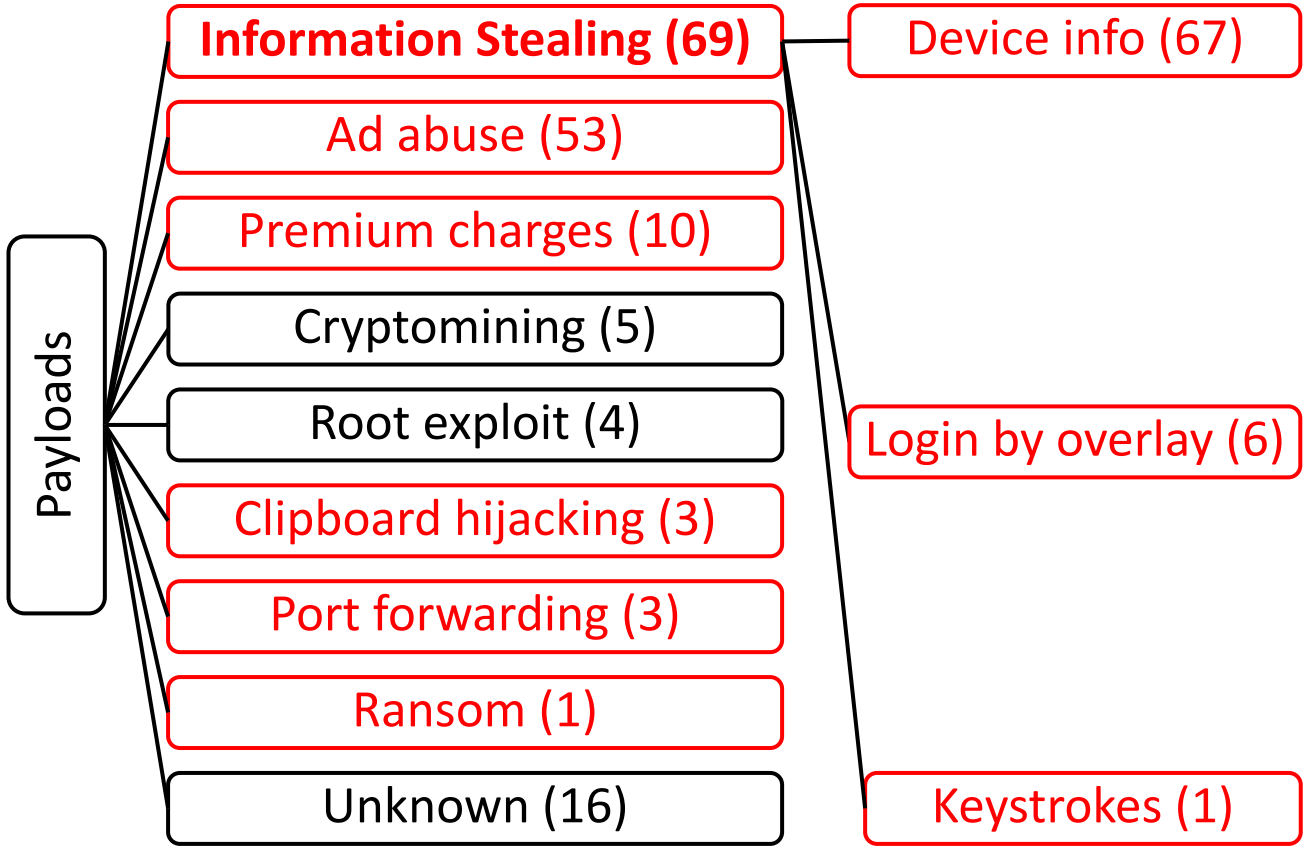


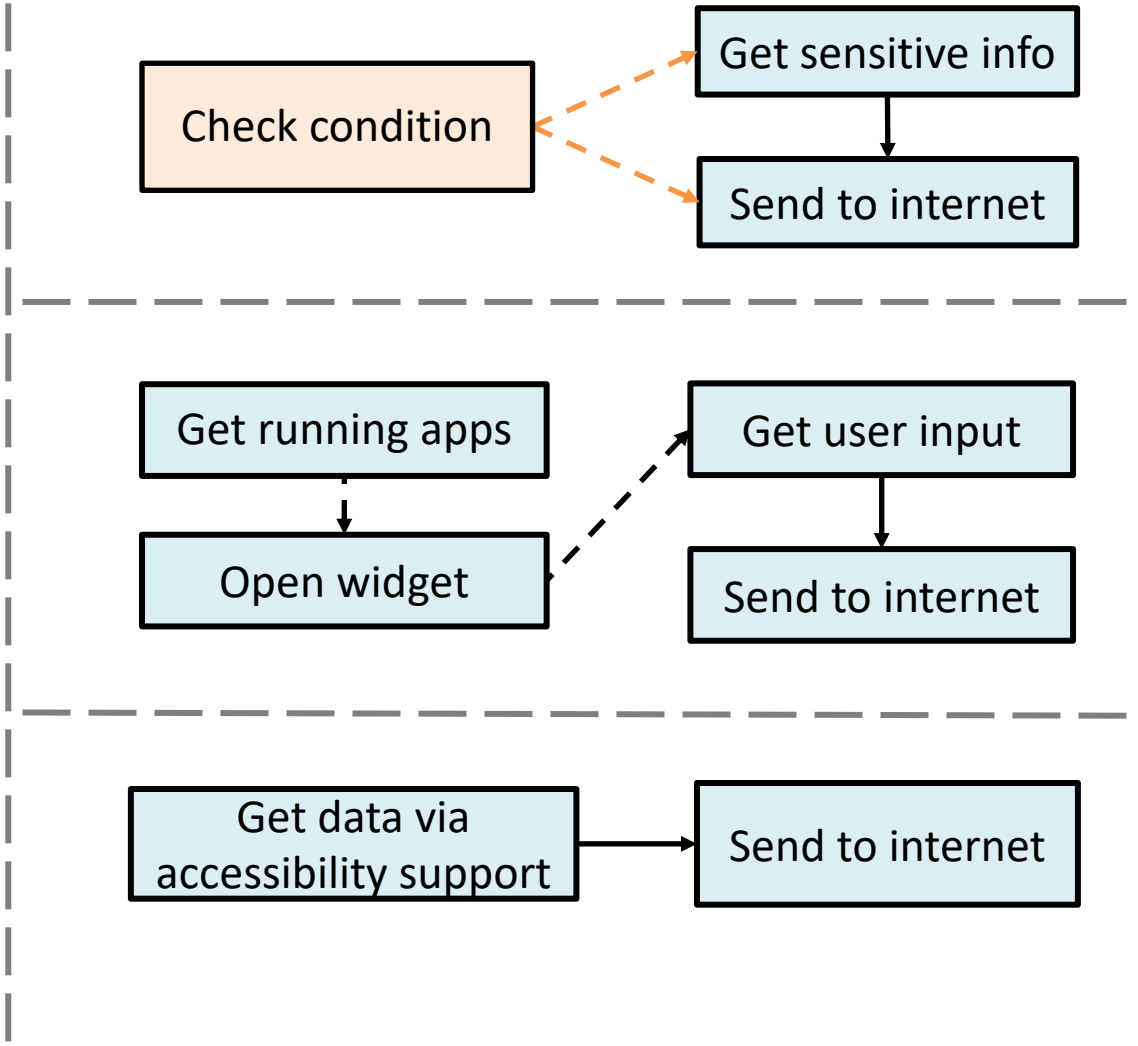
Pattern Example: Information Stealing



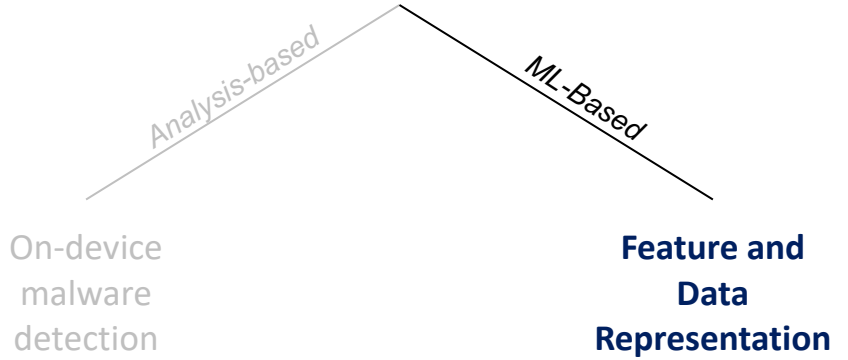


Pattern Example: Information Stealing





Our Current Work



- Reproduced high performance (F1-measure = 96%) but…
- Learned “unreliable” correlations,
  e.g., that benign apps use analytics and crash logging libraries

Our Current Work

Analysis-based

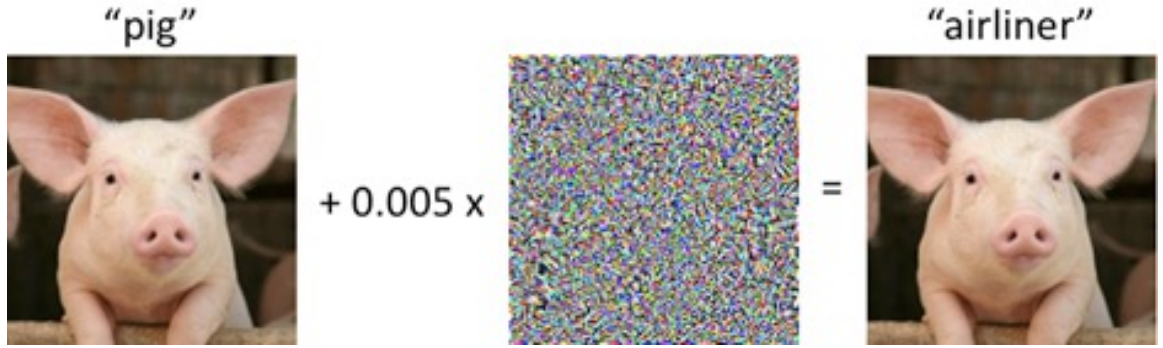
ML-Based

On-device malware detection

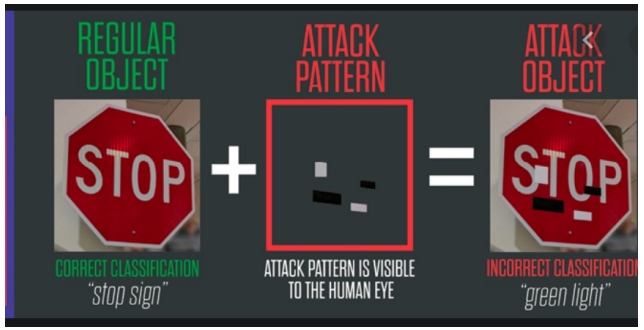
Feature and Data Representation

- Reproduced high performance (F1-measure = 96%) but…
- Learned “unreliable” correlations,
  e.g., that benign apps use analytics and crash logging libraries

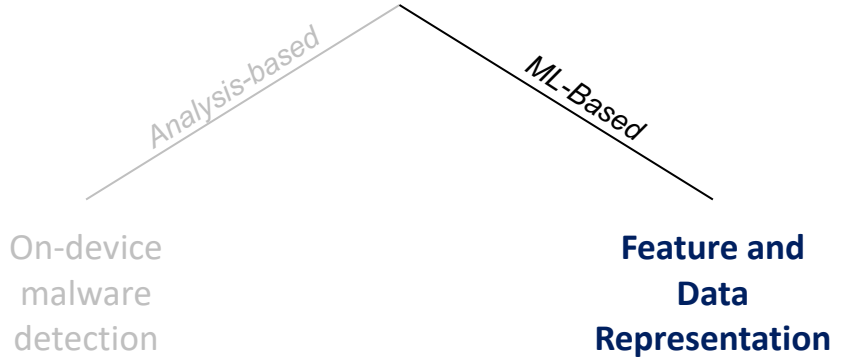
Adversarial robustness, evasion attack



Not random!

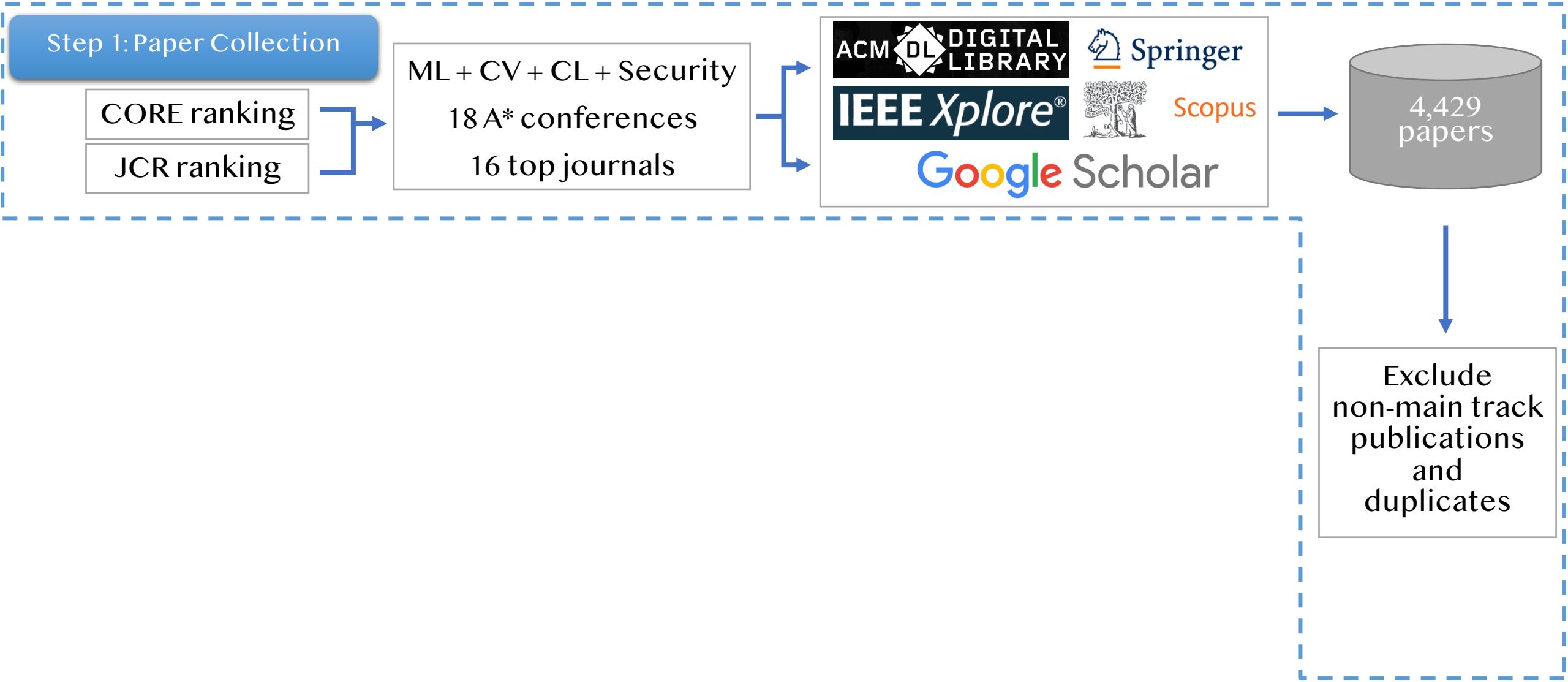


Our Current Work

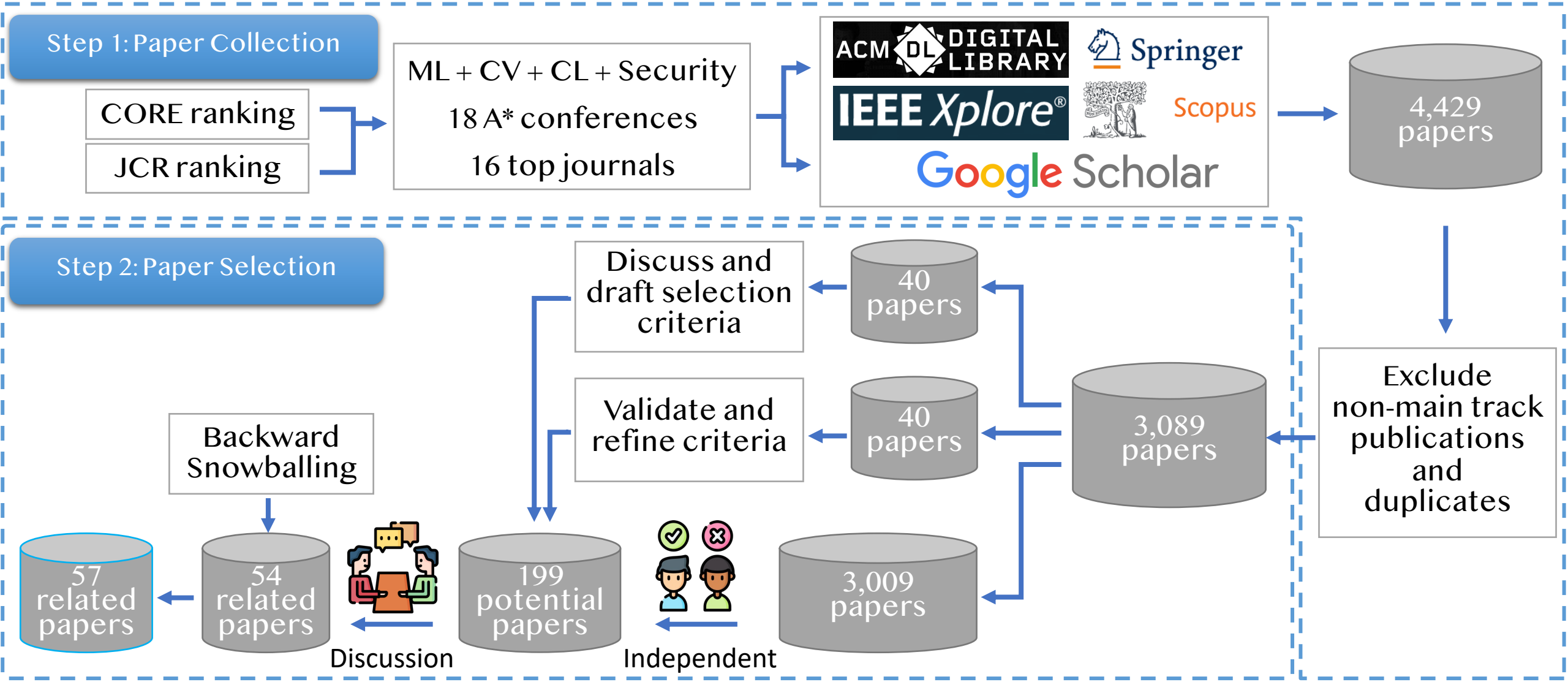


- *How to represent software in terms of features?*
- *How to select samples for training?*
- *Which properties of data influence ML reliability?*
- *What learning paradigms are most applicable?*

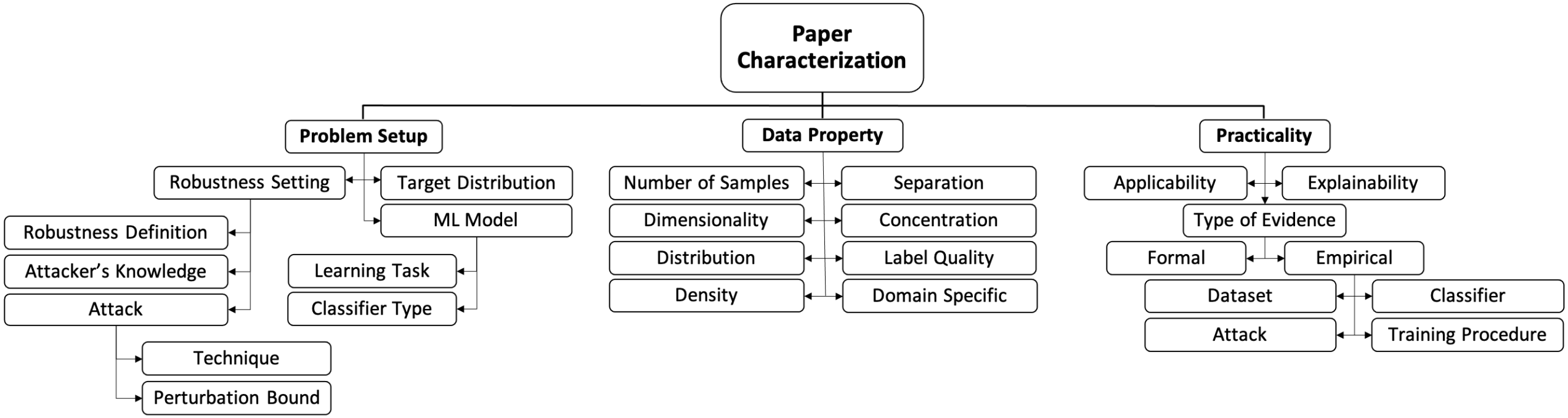
A Survey on the Effects of Data on Adversarial Robustness



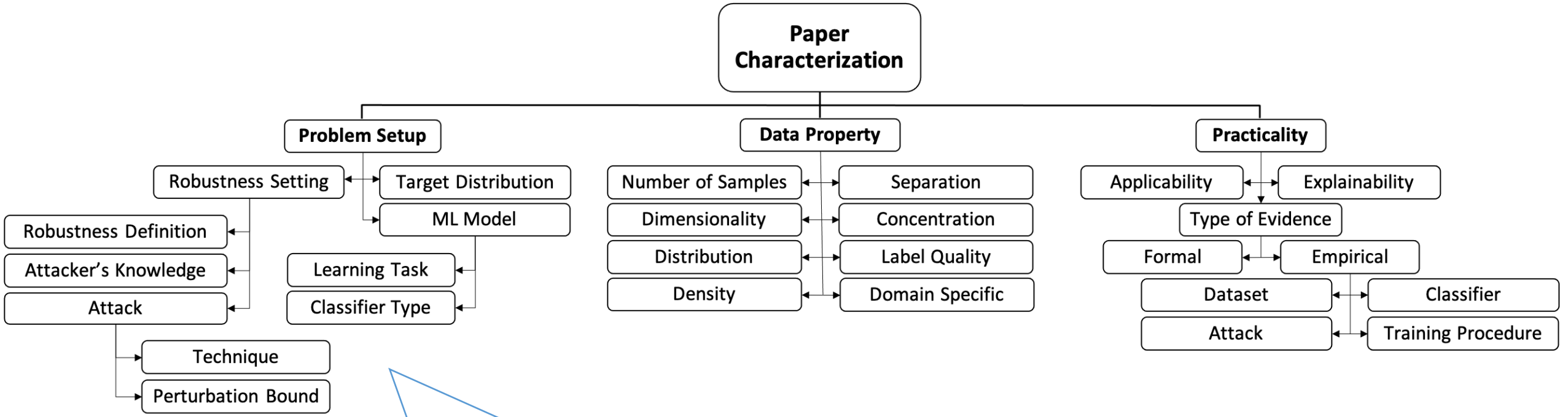
A Survey on the Effects of Data on Adversarial Robustness



Survey: Categorization



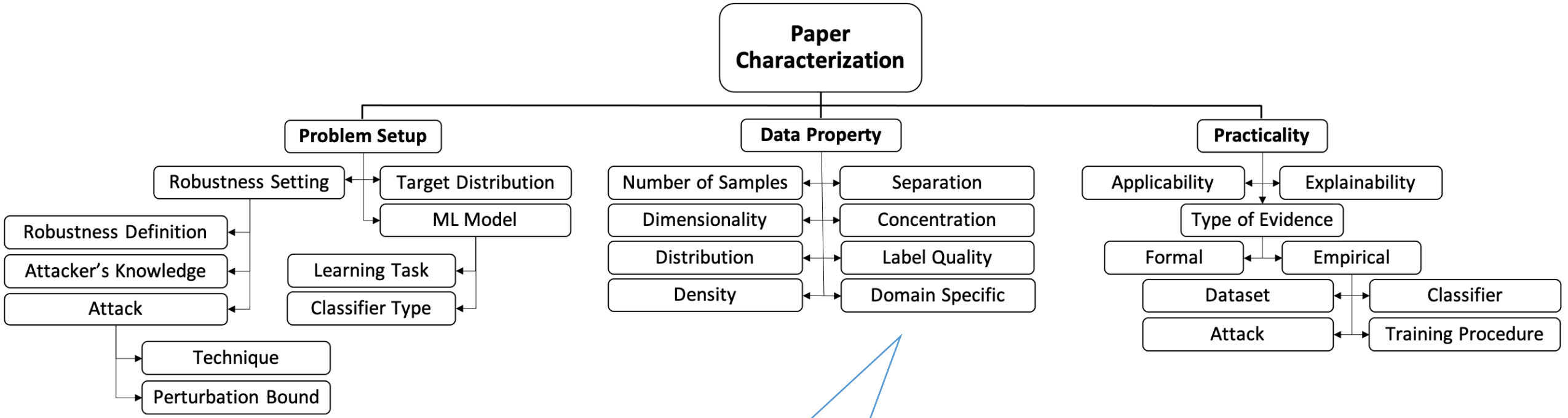
Survey: Categorization



Scope/assumptions, e.g.,

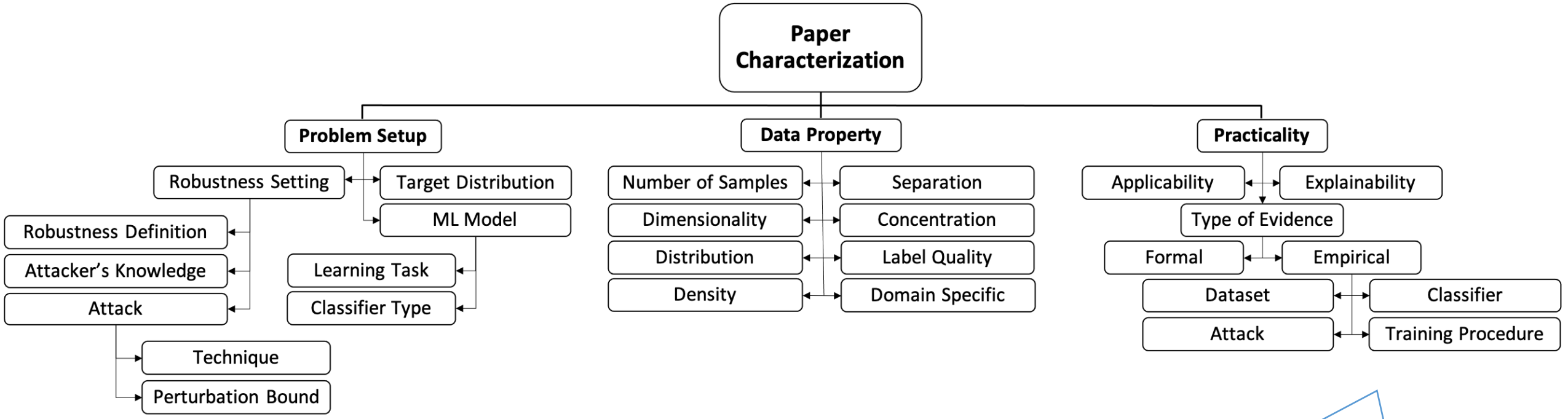
- Type of learning problem (classification / regression)
- Learning algorithm
- Attack considered
- Attacker knowledge
- Measurement of robustness

Survey: Categorization



Data properties correlated with adversarial robustness

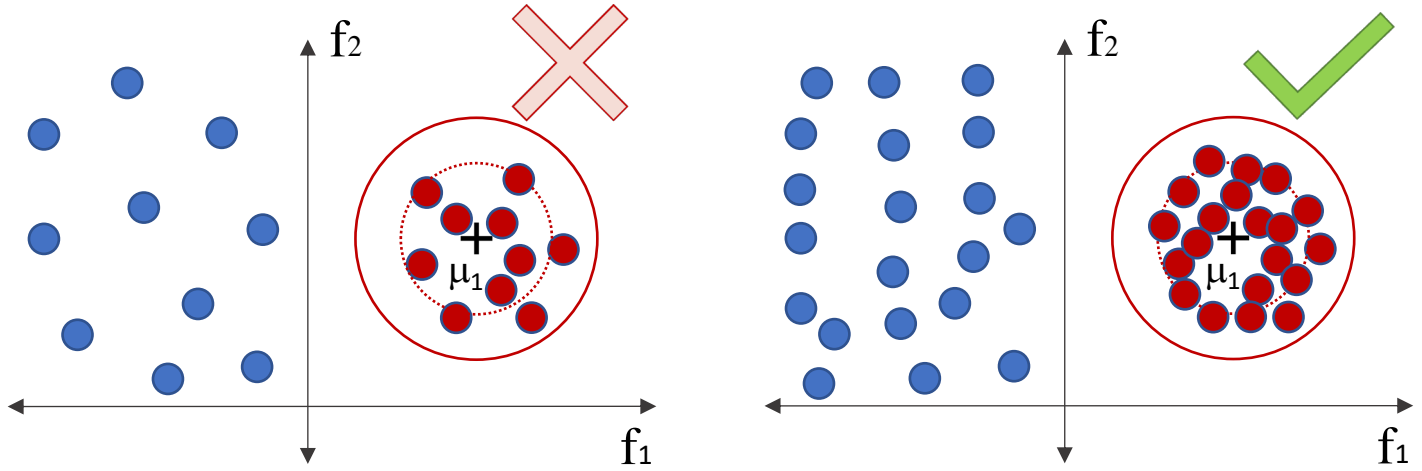
Survey: Categorization



Applicability of conclusions, e.g.,

- Actionable metrics or techniques
- Formal vs. empirical analysis (and under which setup)
- Statement vs. explanation of phenomenon

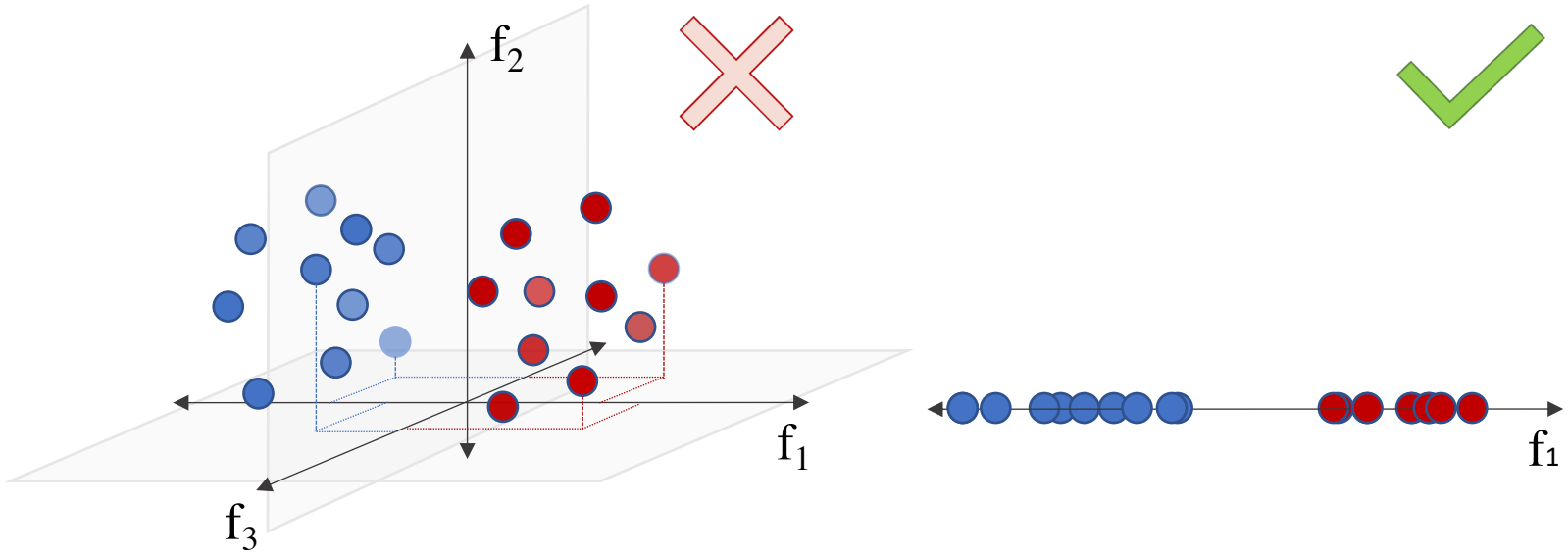
Survey: Results



10/57

	Number of samples	Dimensionality	Distribution	Density	Separation	Concentration	Label quality	Domain specific
What it is?	The amount of (labeled) training samples							
Main results	• (Much) more samples required for robust than standard generalization • Amount depends on data distribution, dimensionality • Unlabeled or generated data can be utilized • More difficult to achieve robustness on imbalanced data							

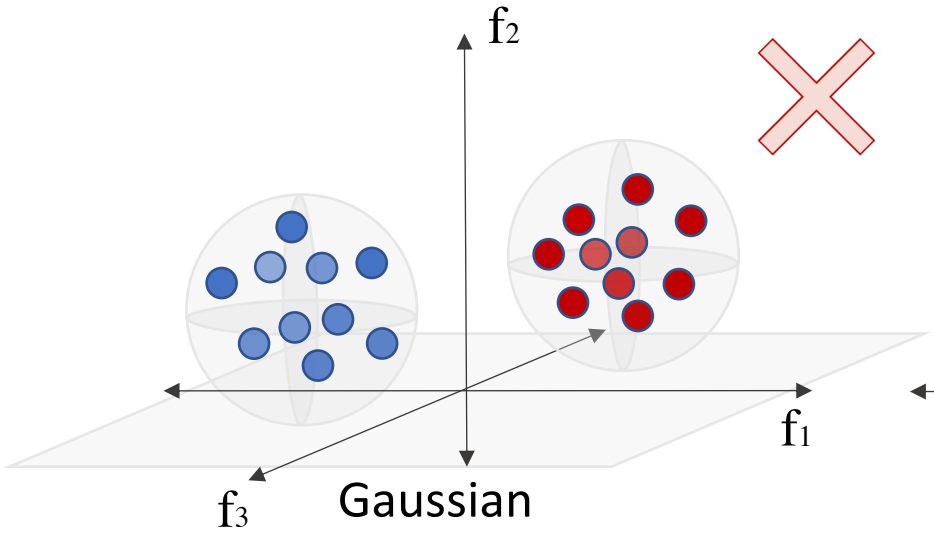
Survey: Results

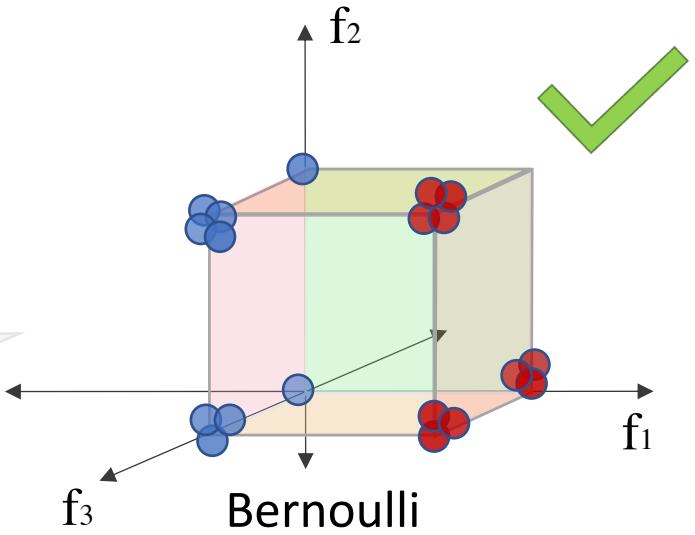


10/57 17/57

	Number of samples	**Dimensionality**	Distribution	Density	Separation	Concentration	Label quality	Domain specific
What it is?	The number of features used to represent samples							
Main results	• High dimensionality leads to higher adversarial risk, greater difficulty to defend • Certain models (e.g., BNNs) are less influenced							

Survey: Results

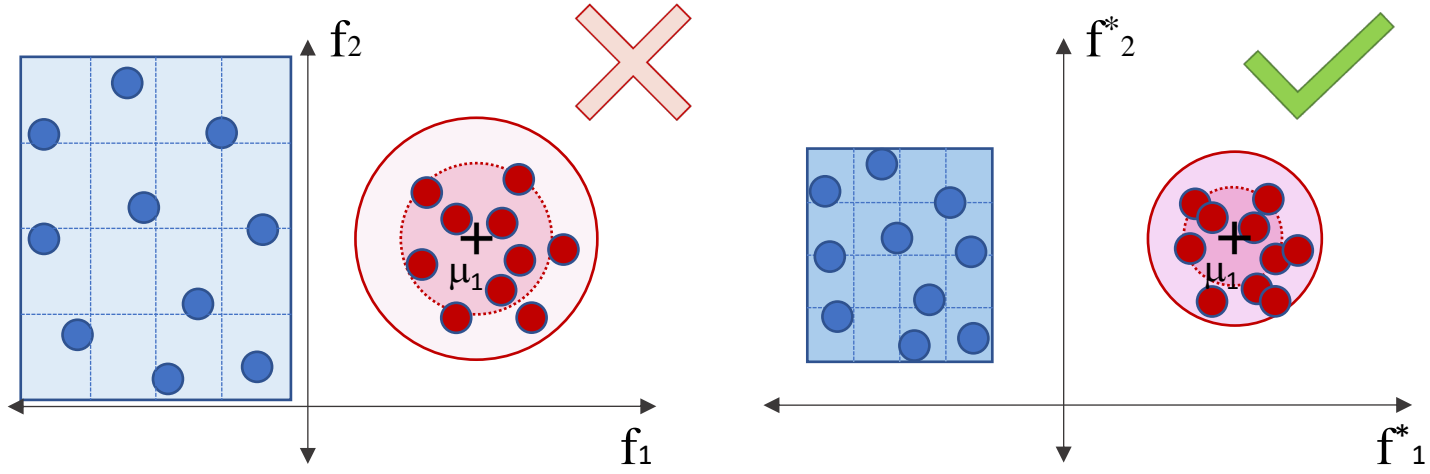




10/57 17/57 8/57

	Number of samples	Dimensionality	**Distribution**	Density	Separation	Concentration	Label quality	Domain specific
What it is?	Types and properties of distributions							
Main results	• Certain types of distributions are more optimal for training robust models • e.g., Bernoulli is more optimal than Gaussian • Symmetric distributions are more robust • Transforming feature distributions can improve robustness							

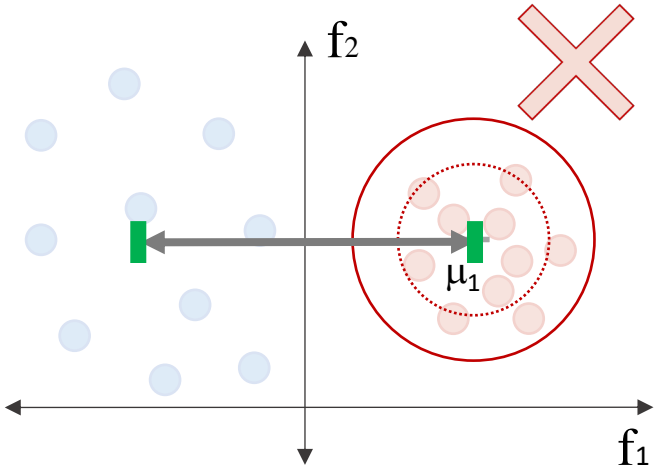
Survey: Results

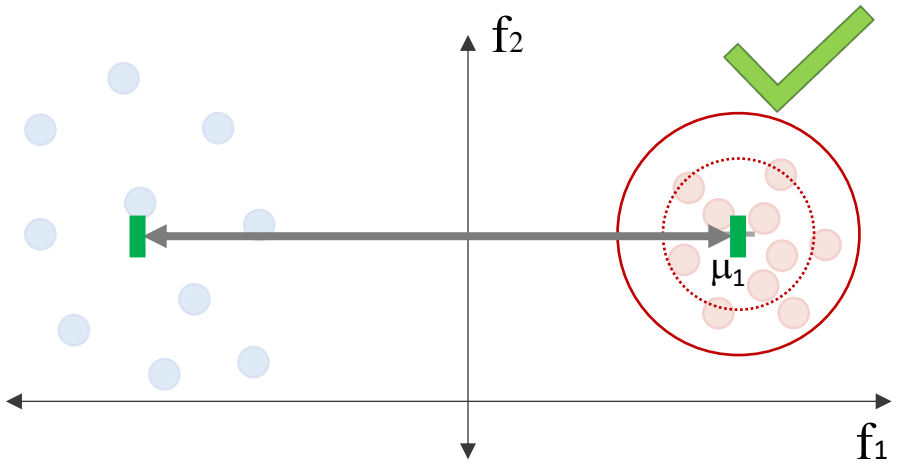


	Number of samples (10/57)	Dimensionality (17/57)	Distribution (8/57)	**Density** (5/57)	Separation	Concentration	Label quality	Domain specific
What it is?	Closeness of samples (from the same class) in bounded regions							

Main results	• High class density leads to lower adversarial vulnerability
	• Adversarial examples are commonly found in low-density regions
	  • Attack: picking samples from such regions
	  • Defense: projecting samples to regions with higher density							

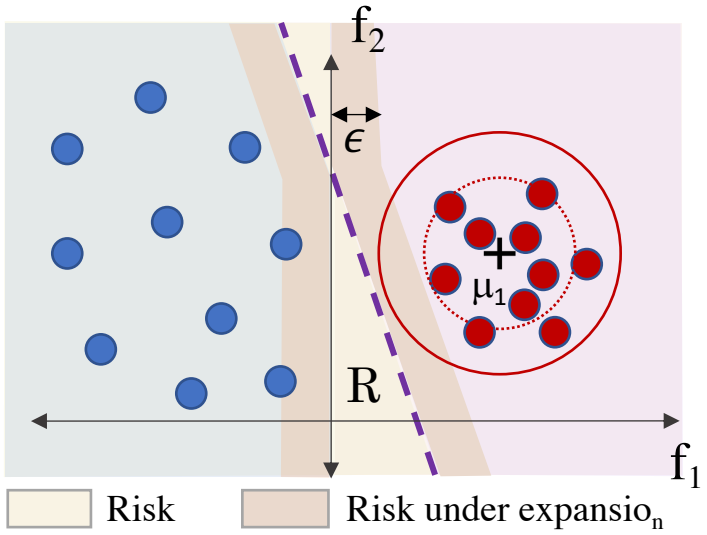
Survey: Results





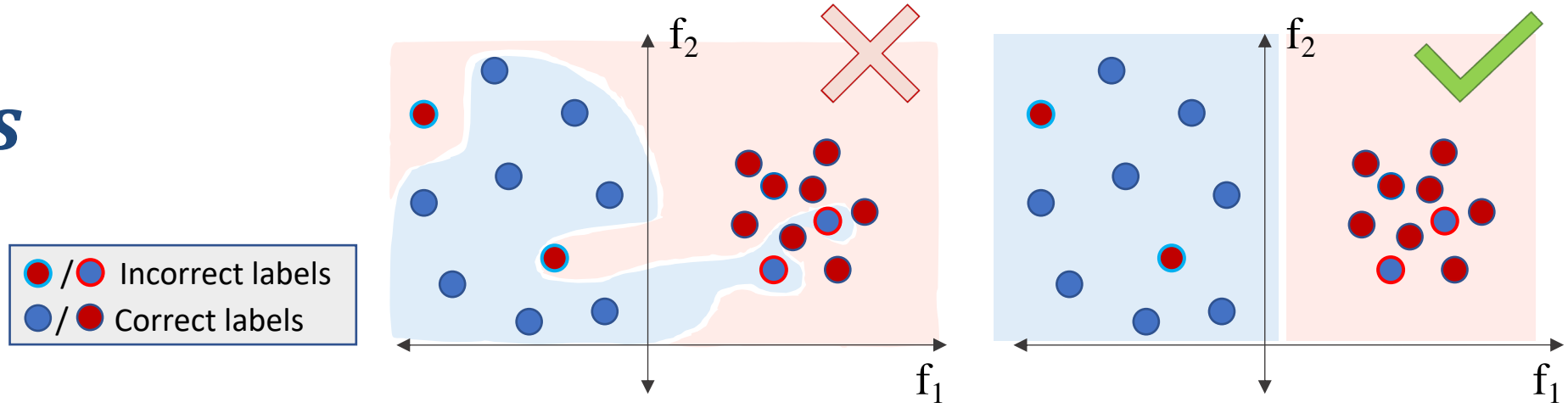
	Number of samples (10/57)	Dimensionality (17/57)	Distribution (8/57)	Density (5/57)	**Separation** (13/57)	Concentration	Label quality	Domain specific
What it is?	Distances between samples from different classes to each other (inter-class distance)							
Main results	• Larger separation leads to lower adversarial vulnerability • Distance metrics, e.g., optimal transport, characterize the lower bound • Large separation makes some local classifiers (e.g., 1-NN) naturally robust • Increasing the separation of internal (latent) features improves robustness							

Survey: Results



	10/57 Number of samples	17/57 Dimensionality	8/57 Distribution	5/57 Density	13/57 Separation	6/57 **Concentration**	Label quality	Domain specific
What it is?	Minimum value of function for any set, after expanding by ϵ. I.e., minimum value of the classification error function after expanding by an adversarial error ϵ							
Main results	• High concentration leads to higher adversarial vulnerability • The effect of high concentration is worsened by high dimensionality							

Survey: Results



	10/57 Number of samples	17/57 Dimensionality	8/57 Distribution	5/57 Density	13/57 Separation	6/57 Concentration	2/57 **Label quality**	Domain specific
What it is?	Presence of inaccurate labels (label noise). Coarse labels							
Main results	• High label noise correlates to higher adversarial vulnerability • More refined labels may result in more optimal feature representations for robustness • e.g., “Cat”, “Dog” instead of “Animal” • Training with labels from multiple tasks improves model robustness							

Survey: Results

	10/57 Number of samples	17/57 Dimensionality	8/57 Distribution	5/57 Density	13/57 Separation	6/57 Concentration	2/57 Label quality	4/57 **Domain specific**
What it is?	For image classification, image *frequency* refers to the intensity of pixel value changes							

Main results	• Diverse distribution of frequencies in image data correlate to lower adversarial risk
	• Adversarial images may arise due to:
	  • ML models utilize more features as they can "see" larger range of frequencies
	  • ML models pay insufficient attention to phase information in images							

Observation

Most of the work: image datasets →

Data from different domains exhibit different properties, attack constraints may differ

Proofs on synthetic data models → Real data may not satisfy such conditions

Few consider inter-dependence of data properties →
Overlook the influence from different factors

Multiple metrics for measuring data properties → May show conflicting trends

Uninvestigated data properties, e.g., outlier samples, small disjoint clusters

Same properties influence robustness and accuracy → which have trade-offs and which don't

Observation

Most of the work: image datasets →

Software/Malware

Data from different domains exhibit different properties, attack constraints may differ

Proofs on synthetic data models → Real data may not satisfy such conditions

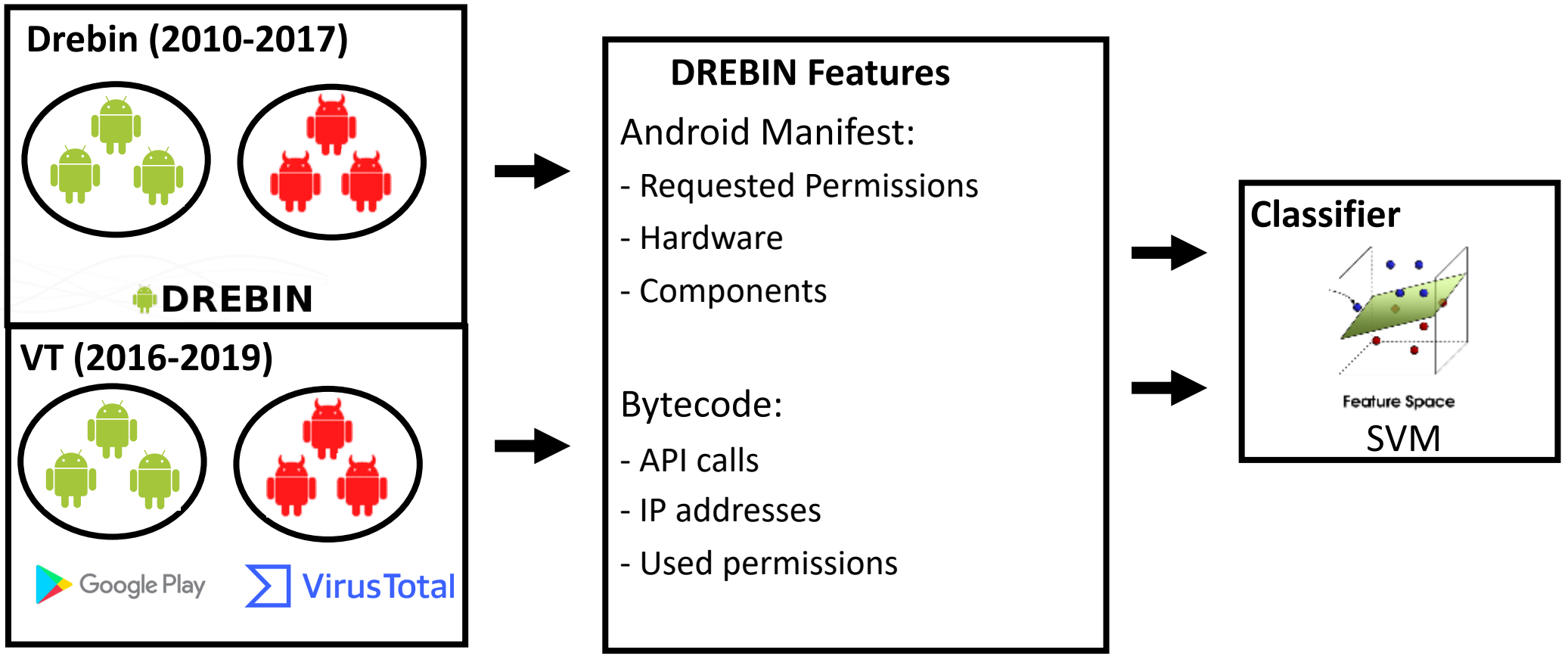
Few consider inter-dependence of data properties →
Overlook the influence from different factors

Multiple metrics for measuring data properties → May show conflicting trends

Uninvestigated data properties, e.g., outlier samples, small disjoint clusters

Same properties influence robustness and accuracy → which have trade-offs and which don't

Experiment: Datasets

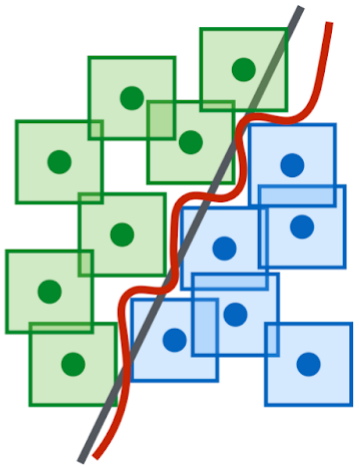


Time aware,
50% training, 50% testing

Experiment: Adversarial Training and Attacks

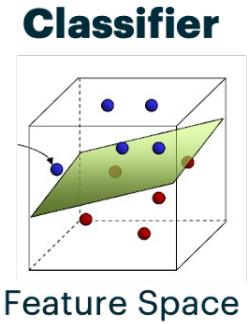
Adversarially trained model

- Key idea: teach the classifier to defend against attacks during training



White-box adversarial attack

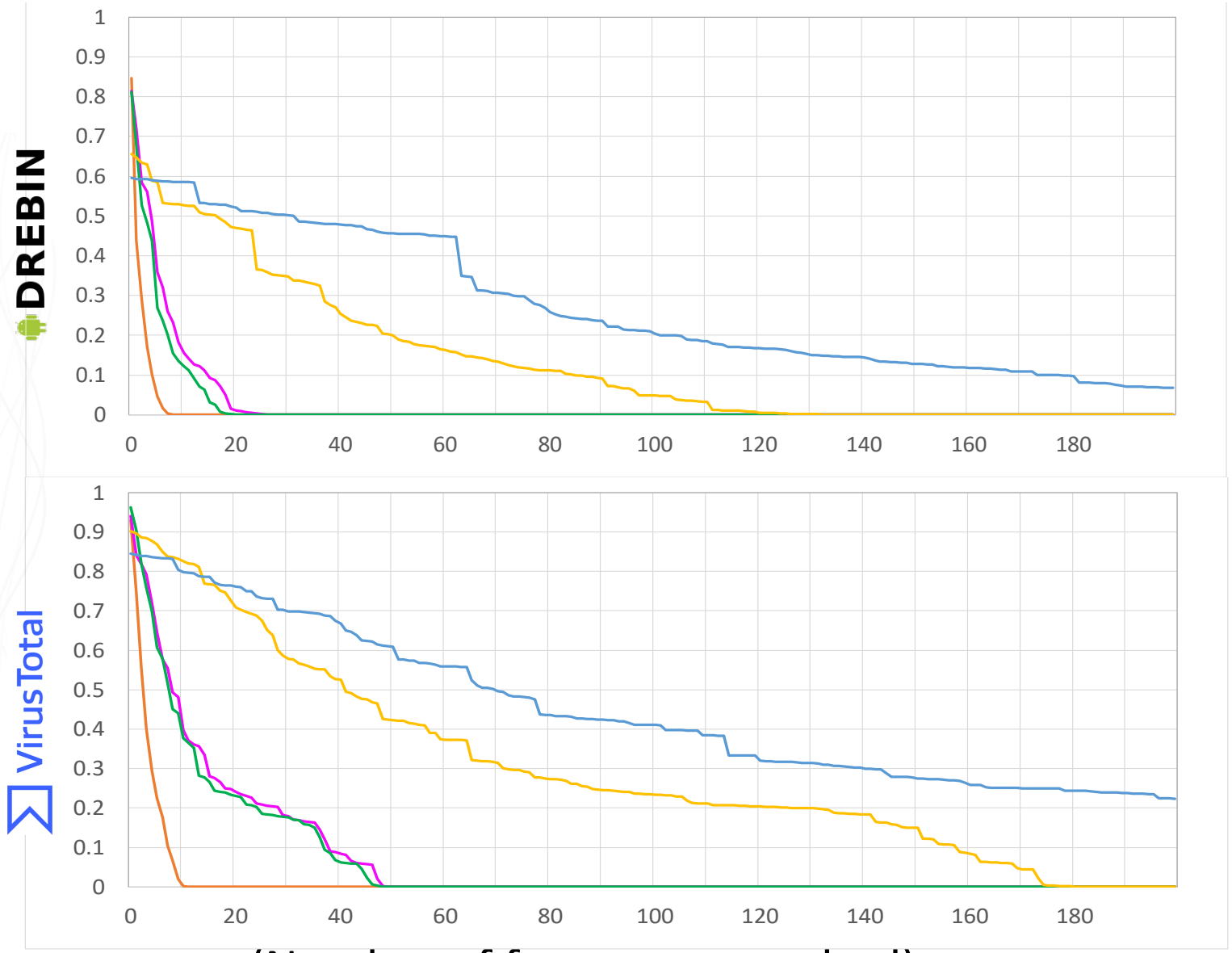
- ☑ **Has access to model**
- ☑ **Feature Knowledge**



→ Top Benign Features

→ Inject Features into Malware

Experiment: Results (TP at 1% FP)

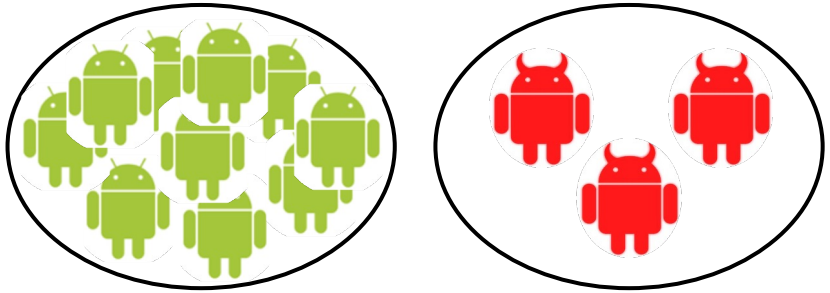


Experiment: Data Properties

Dataset	Number of samples ↑	Dimensionality ↓	Intrinsic Dimensionality ↓	Density ↑	Separation ↑	Concentration ↓
Drebin	5414	19793	1072.51	0.77	38.02	0.1167
VT	13094	66893	200.17	1.62	114.79	0.0638

Future Work

- Additional metrics and interpretations of the results
- Different feature combinations
- Different sample combinations (i.e., how to select training data)
- Different learning paradigms?



(Additional) Lab Research Areas

Mobile Software

malware detection,
efficient testing and analysis


Khaled Ahmed


Michael Cao


Faridah Akinotcho


Yingying Wang

Trustworthy ML

reliability, data quality,
explainability


Gabriella Xiong


Michael Tegegn


Jaskeerat Sarin


Shubraneel Pal

Cloud-based Software

security, performance,
architecture, quality


John Ahn


Saurav Banna

Collaborative SE

reuse, integration and
upgrade management


Evelien Boerstra

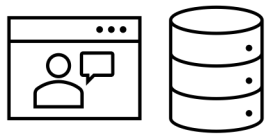

Yingying Wang

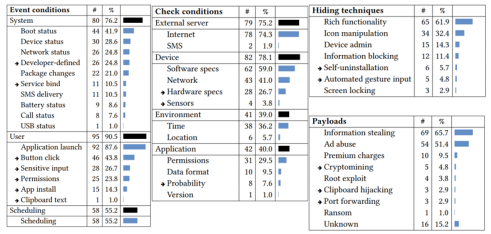

Sahar Badihi

Summary

Contact info: mjulia@ece.ubc.ca

Evasive Malware



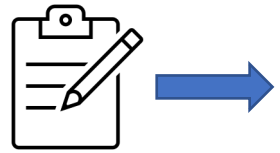


Event conditions	#	%
System	80	76.2
Boot status	44	41.9
Device status	30	28.6
Network status	26	24.8
⬥ Developer-defined	26	24.8
Package changes	22	21.0
⬥ Service bind	11	10.5
SMS delivery	11	10.5
Battery status	9	8.6
Call status	8	7.6
USB status	1	1.0
User	95	90.5
Application launch	92	87.6
⬥ Button click	46	43.8
⬥ Sensitive input	28	26.7
⬥ Permissions	25	23.8
⬥ App install	15	14.3
⬥ Clipboard text	1	1.0
Scheduling	58	55.2
Scheduling	58	55.2

Check conditions	#	%
External server	79	75.2
Internet	78	74.3
SMS	2	1.9
Device	82	78.1
Software specs	62	59.0
Network	43	41.0
⬥ Hardware specs	28	26.7
⬥ Sensors	4	3.8
Environment	41	39.0
Time	38	36.2
Location	6	5.7
Application	42	40.0
Permissions	31	29.5
Data format	10	9.5
⬥ Probability	8	7.6
Version	1	1.0

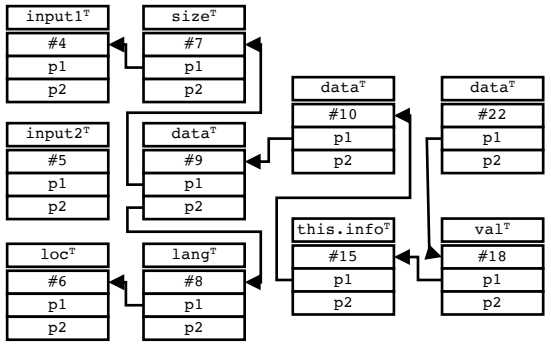
Hiding techniques	#	%
Rich functionality	65	61.9
Icon manipulation	34	32.4
Device admin	15	14.3
Information blocking	12	11.4
⬥ Self-uninstallation	6	5.7
⬥ Automated gesture input	5	4.8
Screen locking	3	2.9

Payloads	#	%
Information stealing	69	65.7
Ad abuse	54	51.4
Premium charges	10	9.5
⬥ Cryptomining	5	4.8
Root exploit	4	3.8
⬥ Clipboard hijacking	3	2.9
⬥ Port forwarding	3	2.9
Ransom	1	1.0
Unknown	16	15.2



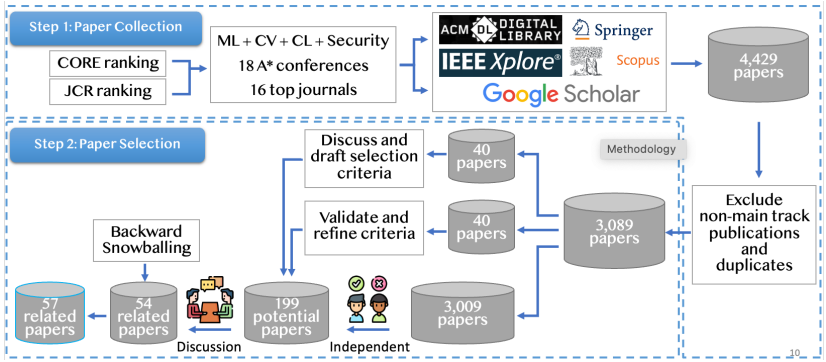
Static Analysis	*Cannot reason about dynamically-loaded and obfuscated code Hard to scale for real apps*
Dynamic Analysis	*Cannot reach hard-to-trigger code*
ML-based Detection	*Not reliable and not explainable*

Path-aware Dynamic Taint Analysis

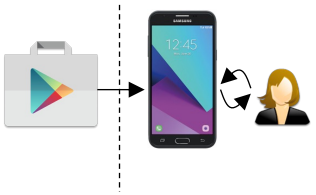


1. Tracks any number of taint sources
2. Reports all source to sink paths
3. Memory efficient: taints of out-of-scope objects are garbage collected

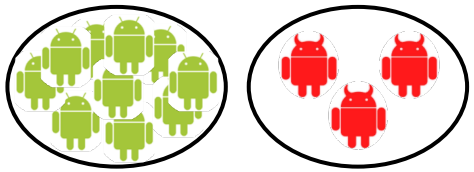
Data Properties for ML Robustness



Current work



On device detection



Feature, data, and learning representation