FLEXDROID: Enforcing In-App Privilege Separation in Android

Jaebaek Seo*, Daehyeok Kim*, Donghyun Cho*,
Taesoo Kim†, Insik shin*

* KAIST

† Georgia Institute of Technology

3rd-party libraries become popular in Android

Ad, Analytics, Game engine, Billing, Social

3rd-party libraries become popular in Android

How can we trust them?

Ad, Analytics, Game engine, Billing, Social

Over half of 3rd party Android in-app ad libraries have privacy issues and possible security holes

TIME

Your Favorite Apps Know More About You Than You Realize

BLUEBOX

Bluebox Security Research on Top Travel Apps

On average, only 30% of code for the apps was created in-house. The remaining 70% was made up of third-party components and libraries that may introduce vulnerabilities that are unknown to the developer, creating a huge potential attack surface

In NDSS 16

The Price of Free: Privacy Leakage in Personalized Mobile In-Apps Ads

What Mobile Ads Know About Mobile Users

Free for All! Assessing User Data Exposure to Advertising Libraries on Android

In NDSS 16

The Price of Free: Privacy Leakage in Personalized Mobile In-Apps Ads

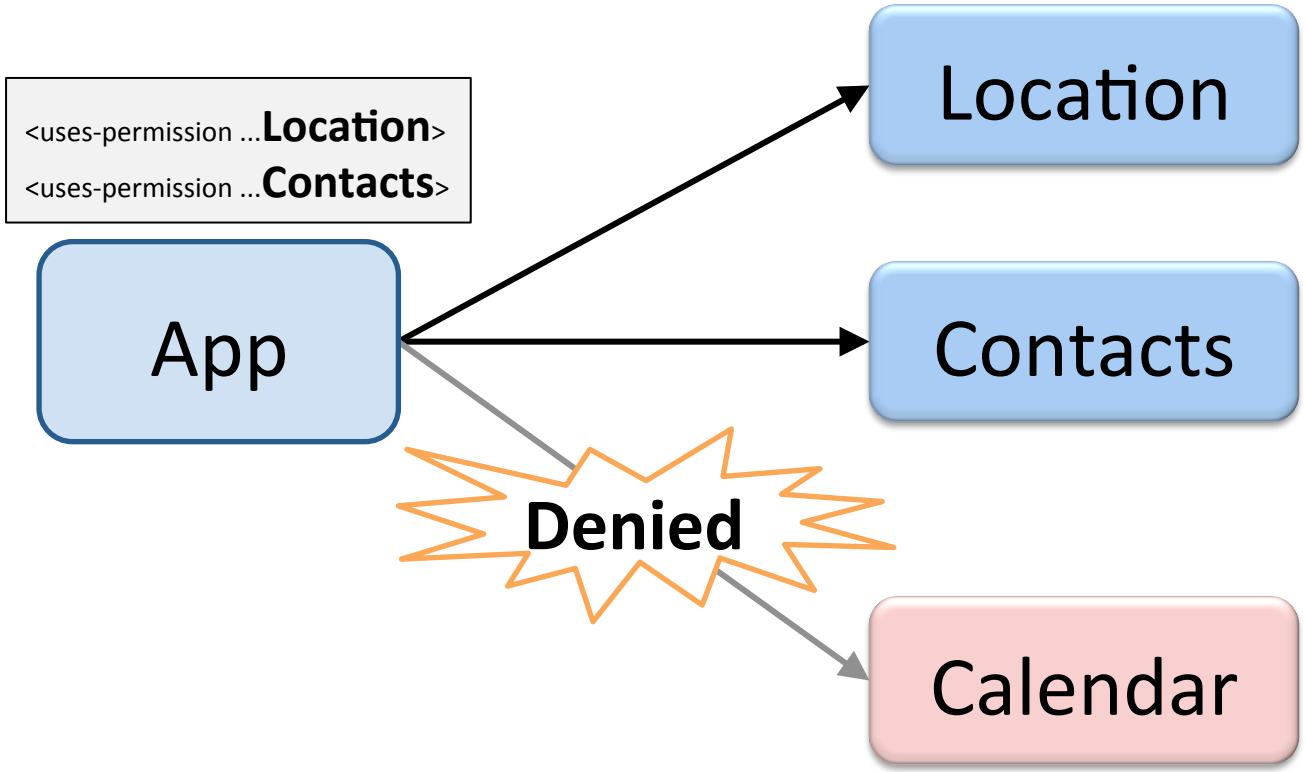
Fundamental problem in Android's permission system

V rs

Free for All! Assessing User Data Exposure to Advertising Libraries on Android

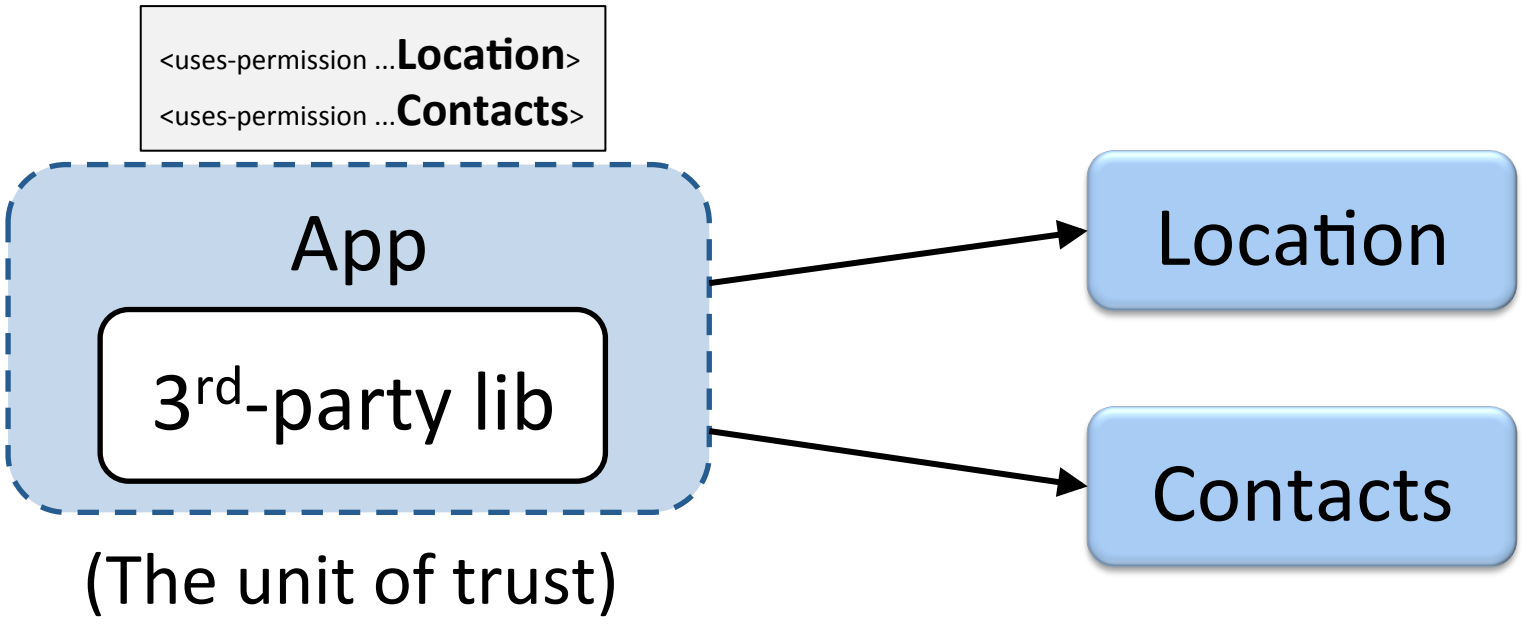
Problem: Android Permission System

- The unit of trust in Android: **Application**



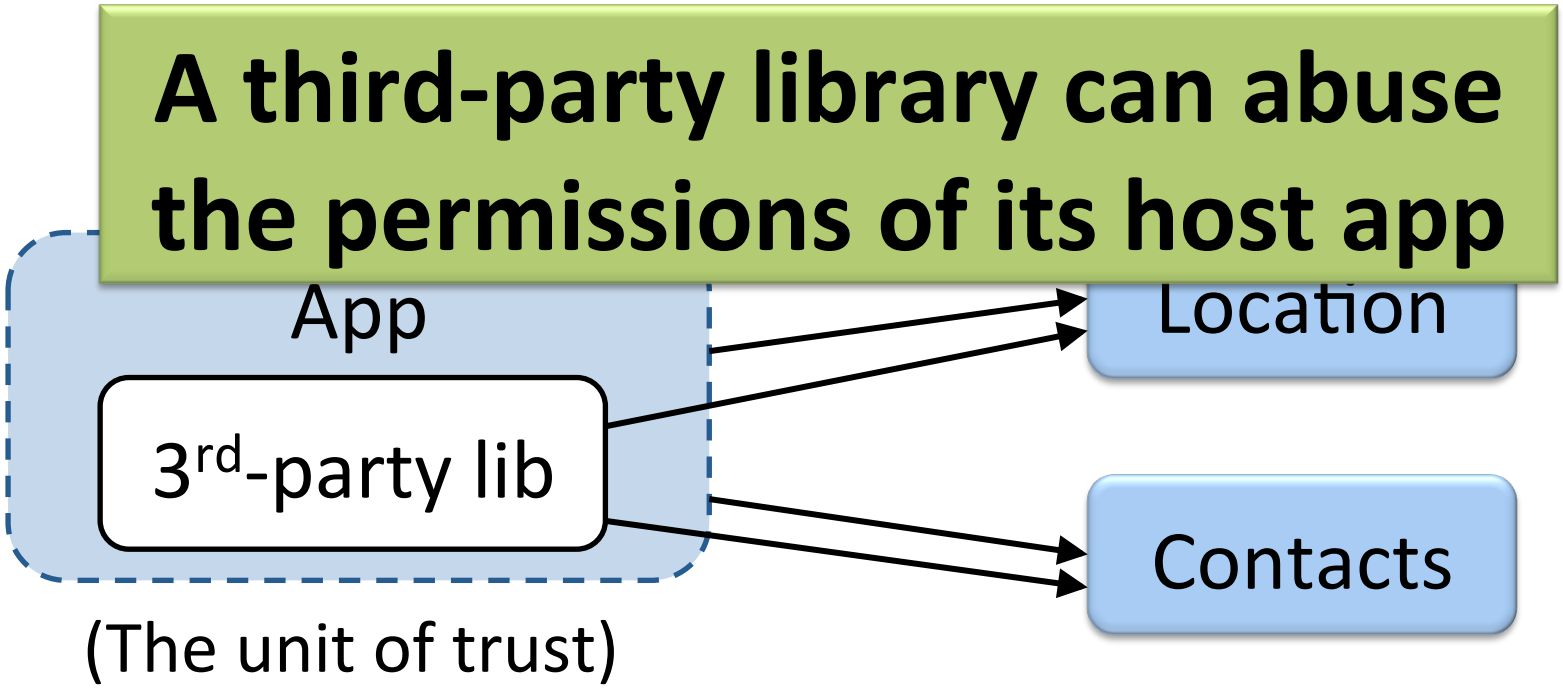
Problem: Android Permission System

- **Third-party library**: having the same access right as the host app



Problem: Android Permission System

- **Third-party library**: having the same access right as the host app

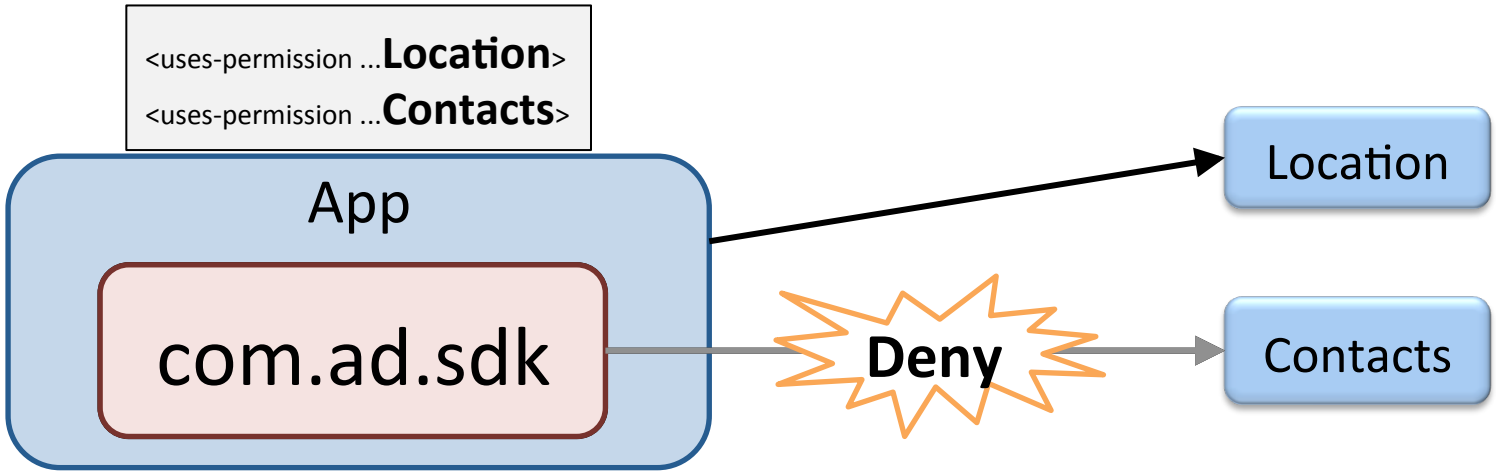


FLEXDROID

Goal: In-app privilege separation between a host application and its third-party libraries

Overview of FLEXDROID

Specifying the package name and its permissions in ***AndroidManifest.xml***




```
<flexdroid android:name="com.ad.sdk" >
    <allow ...Location>
</flexdroid>
```

Contributions

1. Report potential privacy threats of third-party libraries by analyzing 100,000 real-world Android apps

2. Provide an in-app privilege separation in Android
   - Supporting JNI, reflection, and multi-threading

3. Adopt a fault isolation using ARM Memory Domain to sandbox native code in Android

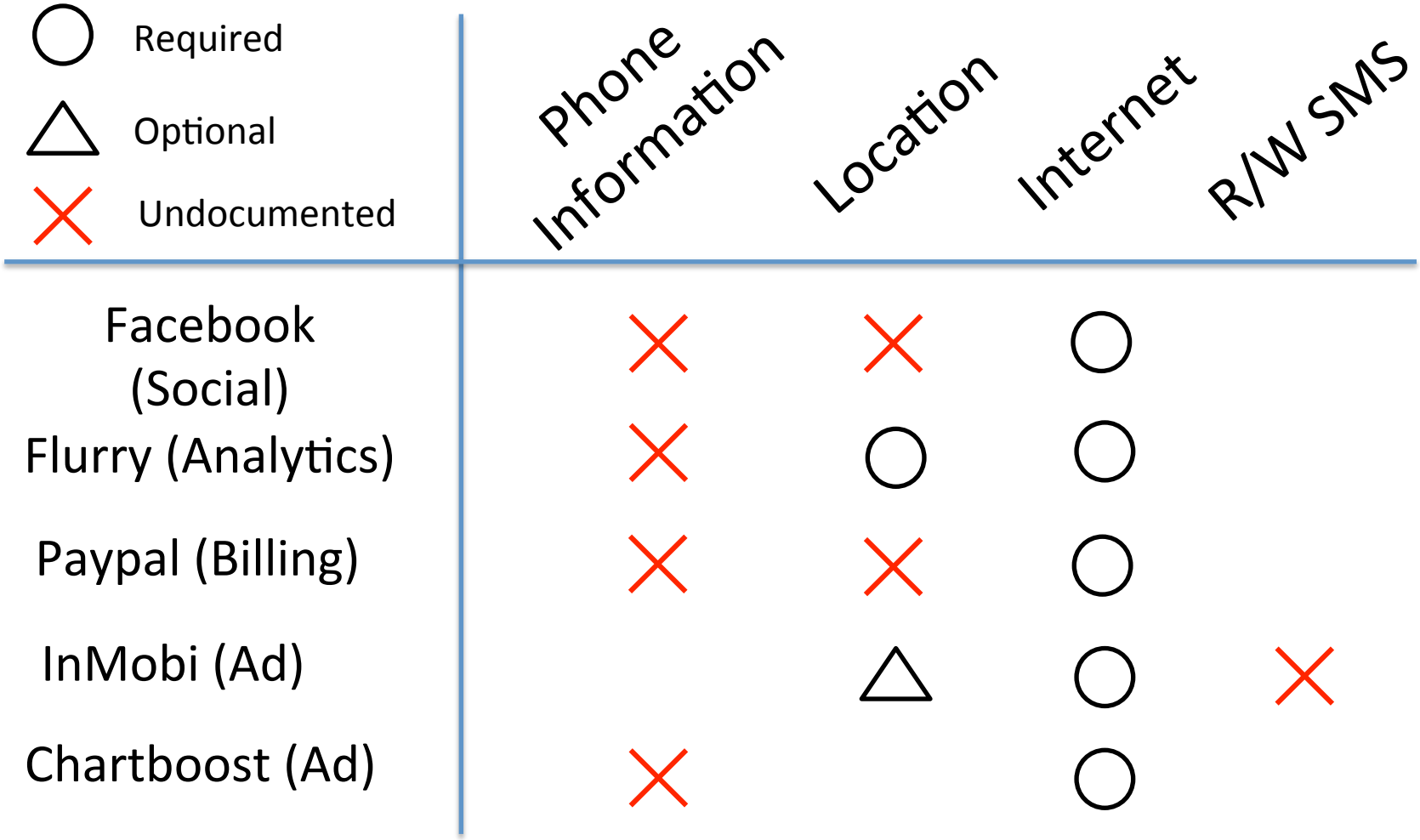
Investigating Real-world Threats

- Investigate 100,000 Android apps from Google Play using a static analysis

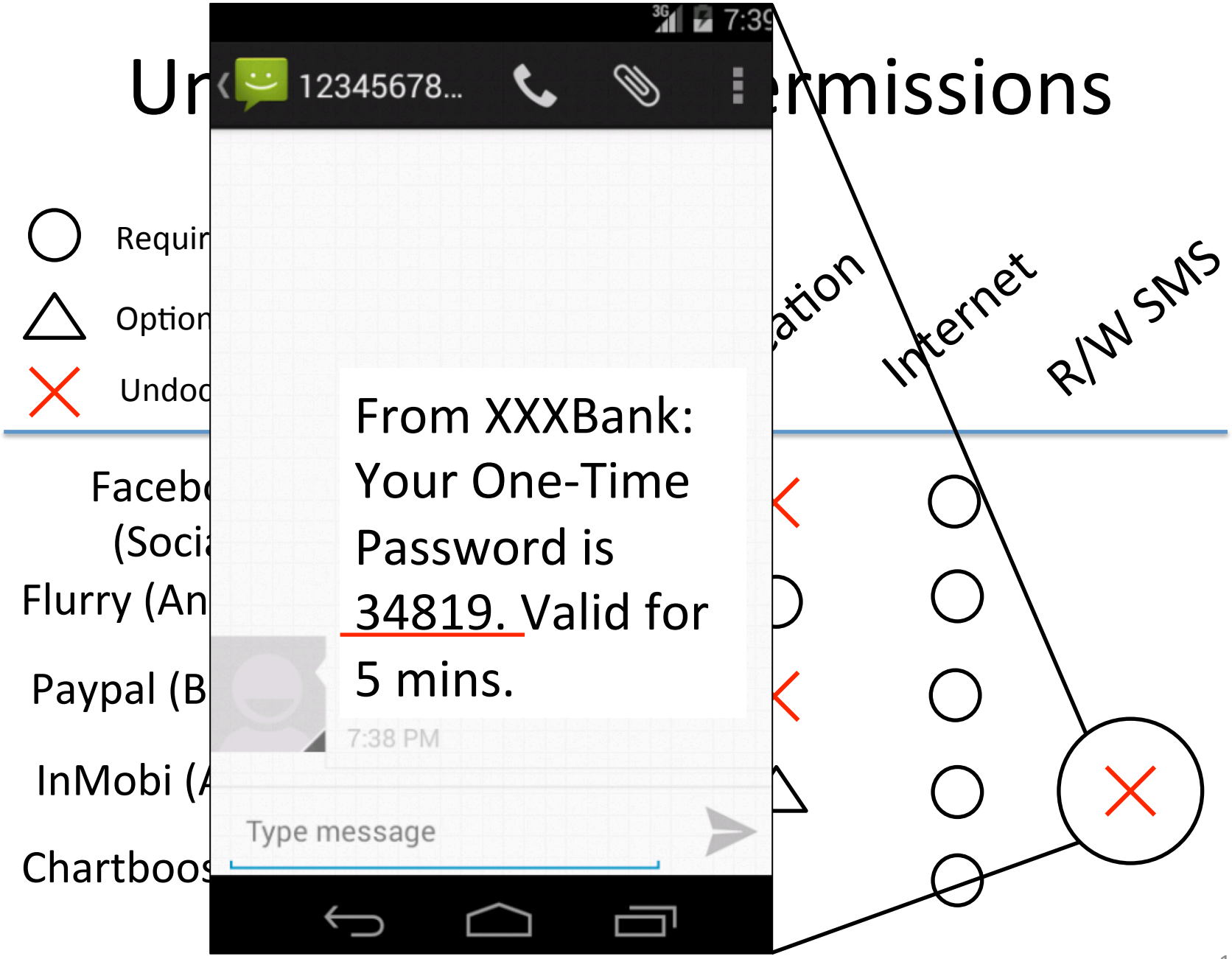
  Q1: How many third-party libraries use *undocumented* permissions?

  Q2: How many of them rely on dynamic code execution?

Undocumented Permissions



Ur ermissions



Analysis of Real-World Apps

- Control-flow and data dependency
  - Class Inheritance → **71.5%**

- Dynamic runtime behavior
  - Java Native Interface (JNI) → **17.1%**
  - Runtime class loading → **27.9%**
  - Reflection → **49.6%**

Challenges

- Control-flow and data dependency
  → Naïvely separating third-party libraries from the host app is not applicable

- Dynamic runtime behavior
  → Statically or dynamically detecting malicious behaviors introduces low accuracy

Threat Model

- Potentially malicious third-party libraries
  - Obfuscated code and logic
- Use of dynamic features
  (e.g., JNI, reflection, multi-threading)
- App developers specifying permissions of each third-party library

SYSTEM DESIGN

Key Idea

Adjusting permissions dynamically whenever an app requests a resource

Dynamic Permission Adjustment

When executing the **host application's code**

App Permissions 

Permissions of host application
- Location
- Contacts

Permissions of third-party library
- Location

Dynamic Permission Adjustment

When executing the **3rd-party lib's code**

App Permissions

Permissions of host application

- Location
- Contacts

Permissions of third-party library

- Location

Identification of Executed Code

1. Identify the principal using stack inspection
2. Apply the stack inspection to Android
3. Protect the integrity of call stack information against attacks via:
   - JNI
   - Reflection
   - Multi-threading

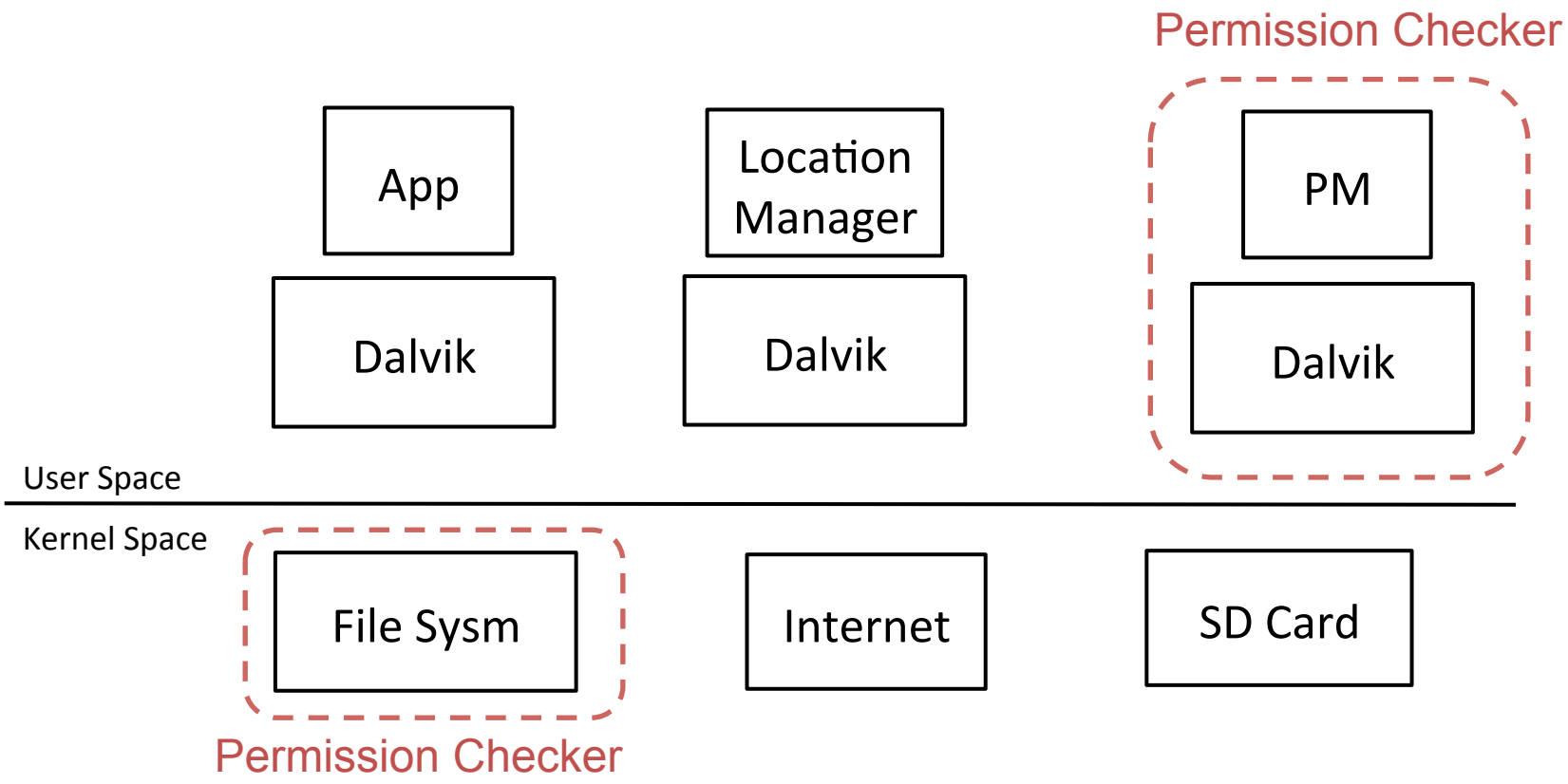
Stack Inspection in Security Context

Process of determining the permissions allowed to the current thread according to principals shown in the call stack

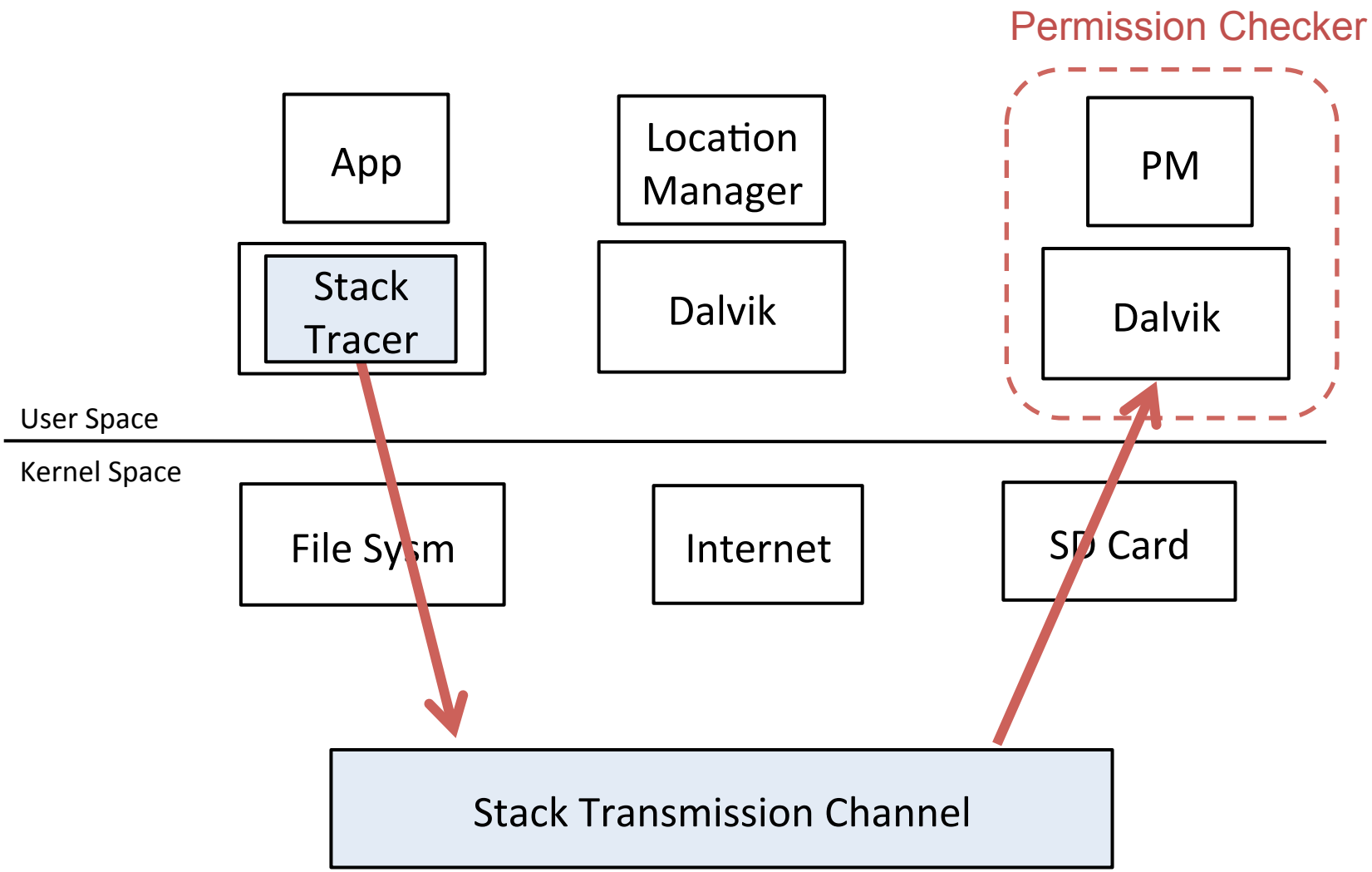
	P	Call stack
⬇	A	com.A.functionA
	B	com.B.functionB
	C	com.C.functionC

Perm = Perm(A)
∩ Perm(B)
∩ Perm(C)

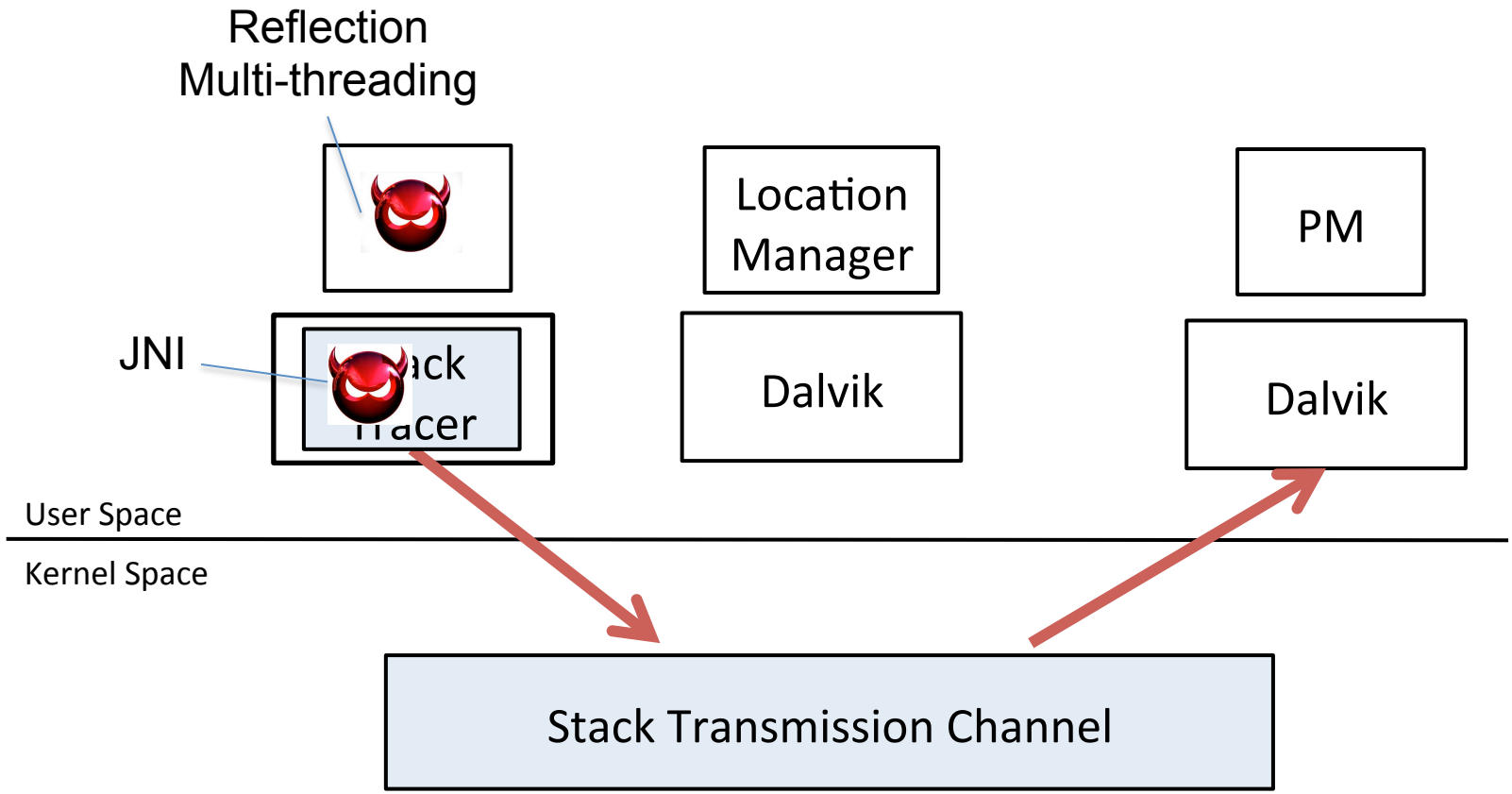
Inter-process Stack Inspection



Inter-process Stack Inspection



Potential Attack Surface



Potential Attack Surface

- Compromising stack tracer ← **JNI**
- Manipulating Dalvik call stack ← **JNI, Reflection, Multi-threading**
- Hijacking the control data
  e.g., code injection on Dalvik functions, manipulating code pointers ← **JNI**

Protecting Integrity of Call Stack

✓ JNI Sandbox

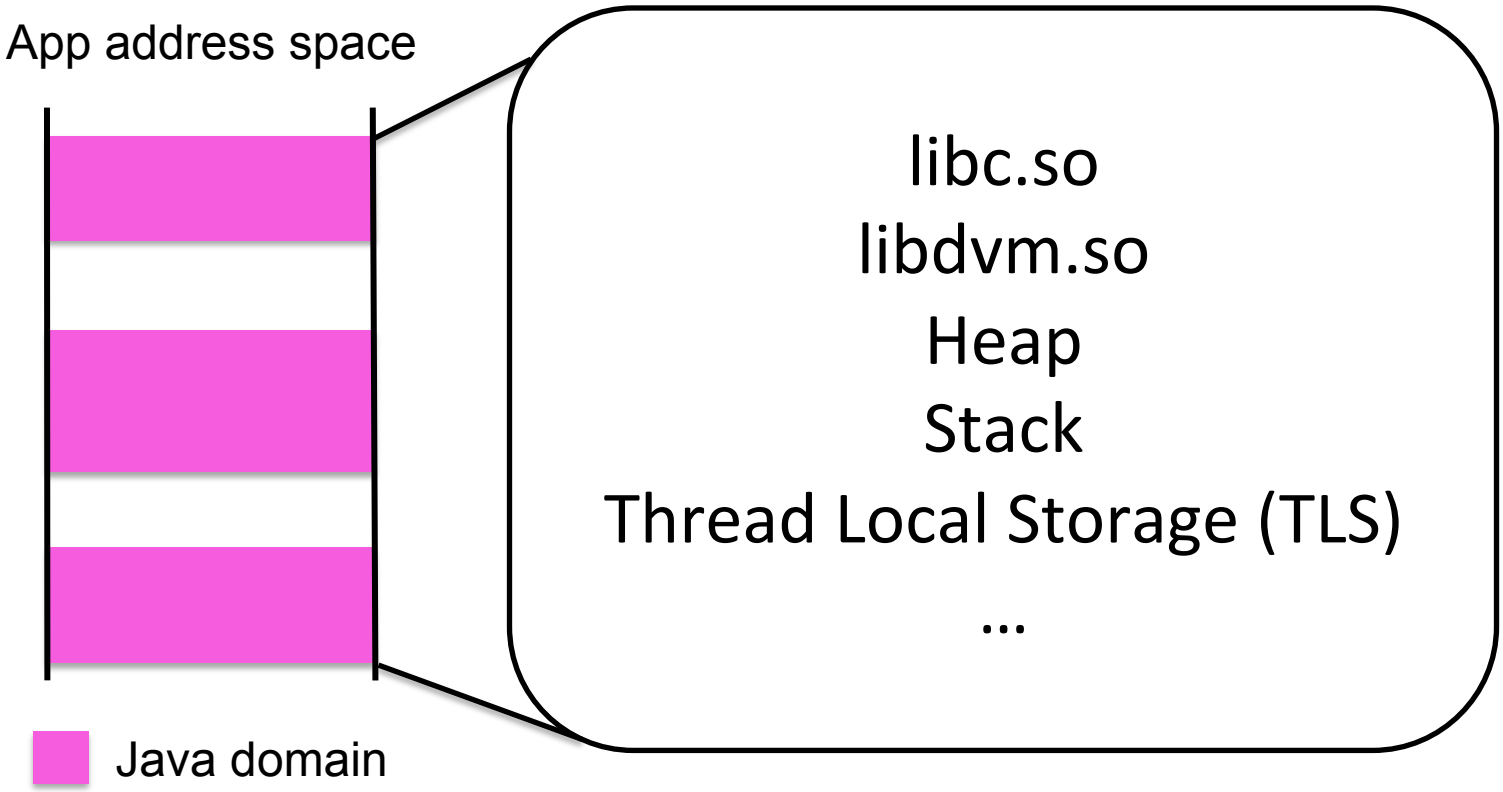
✓ Defense mechanism against attacks via reflection

- Defense mechanism against attacks via multi-threading

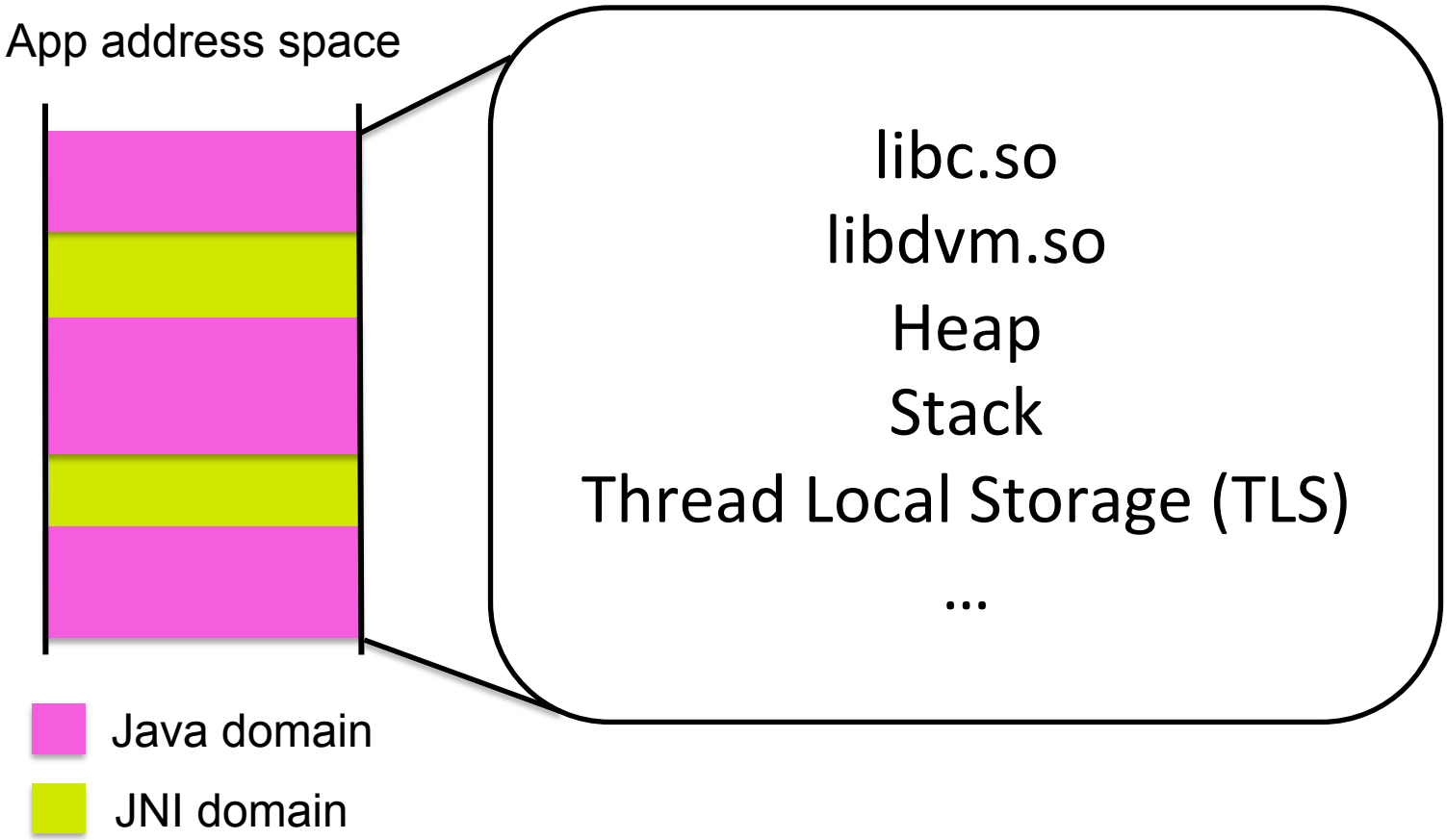
JNI Sandbox

- Inspired by ARMlock (CCS'14), applying ***Fault Isolation*** using ***ARM Memory Domain*** to Android
- **Key Idea**
  - Regard JNI code of 3rd-party libraries as potentially malicious code
  - Run JNI in an isolated and restricted memory domain

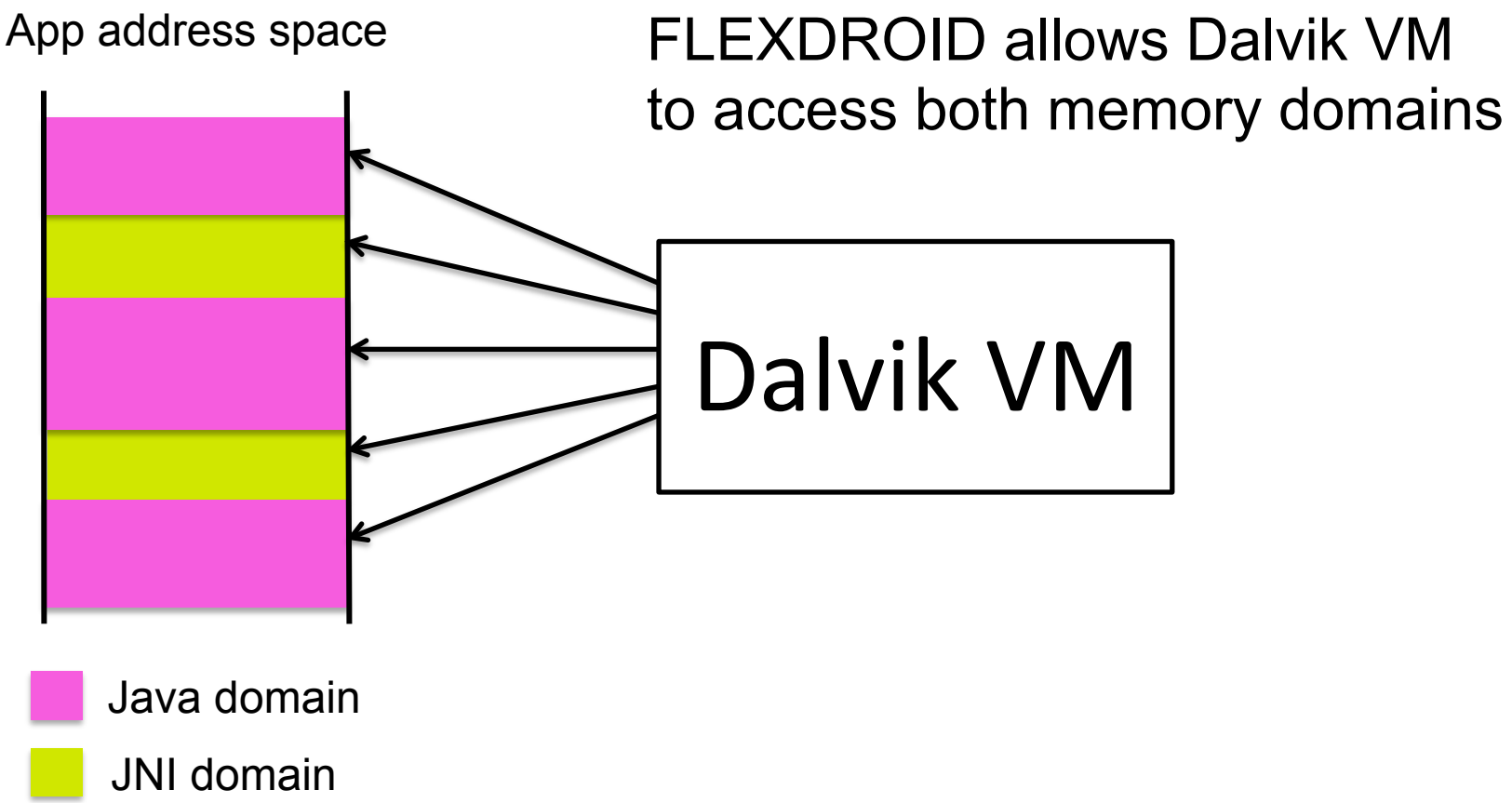
Fault Isolation using ARM Memory Domain



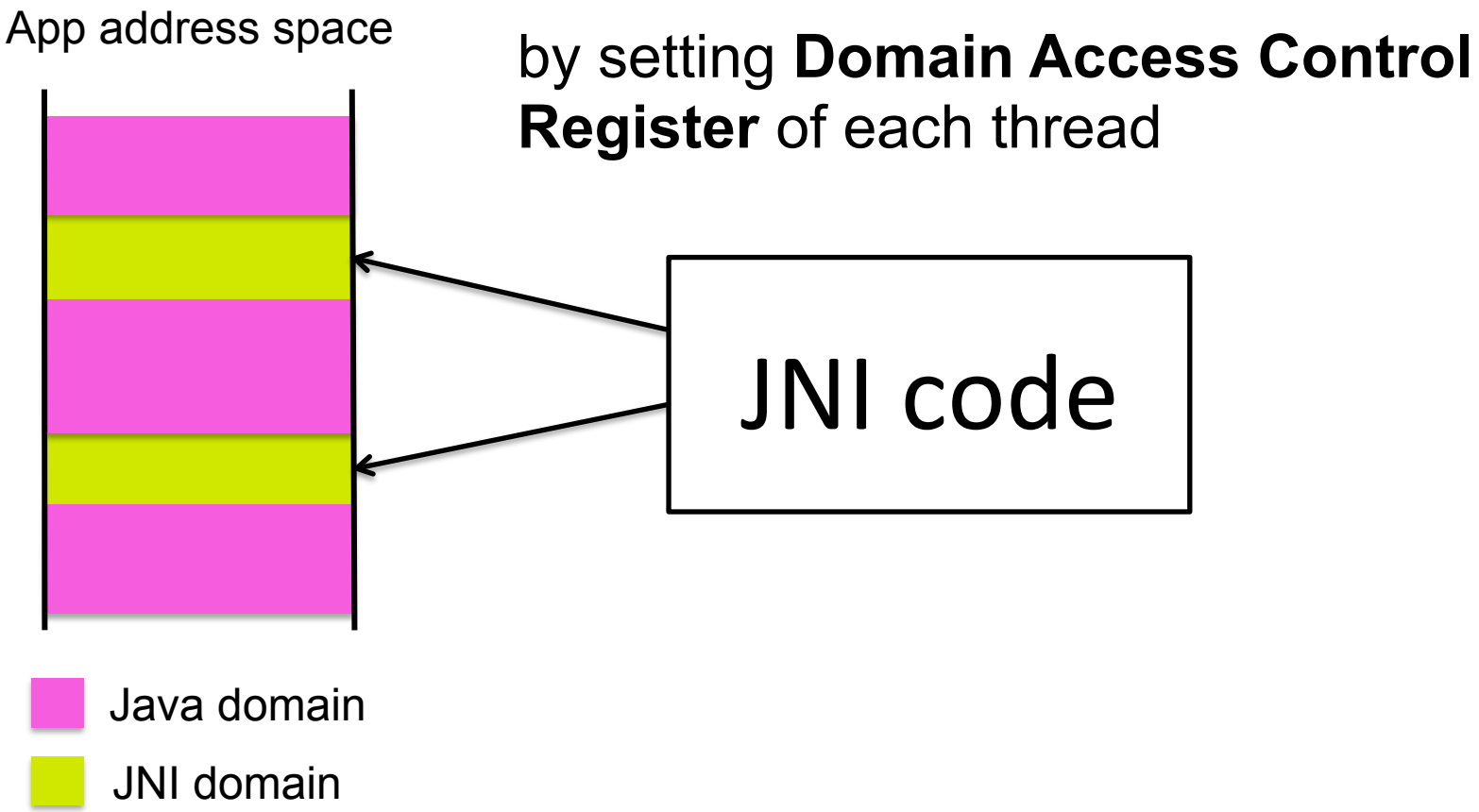
Fault Isolation using ARM Memory Domain



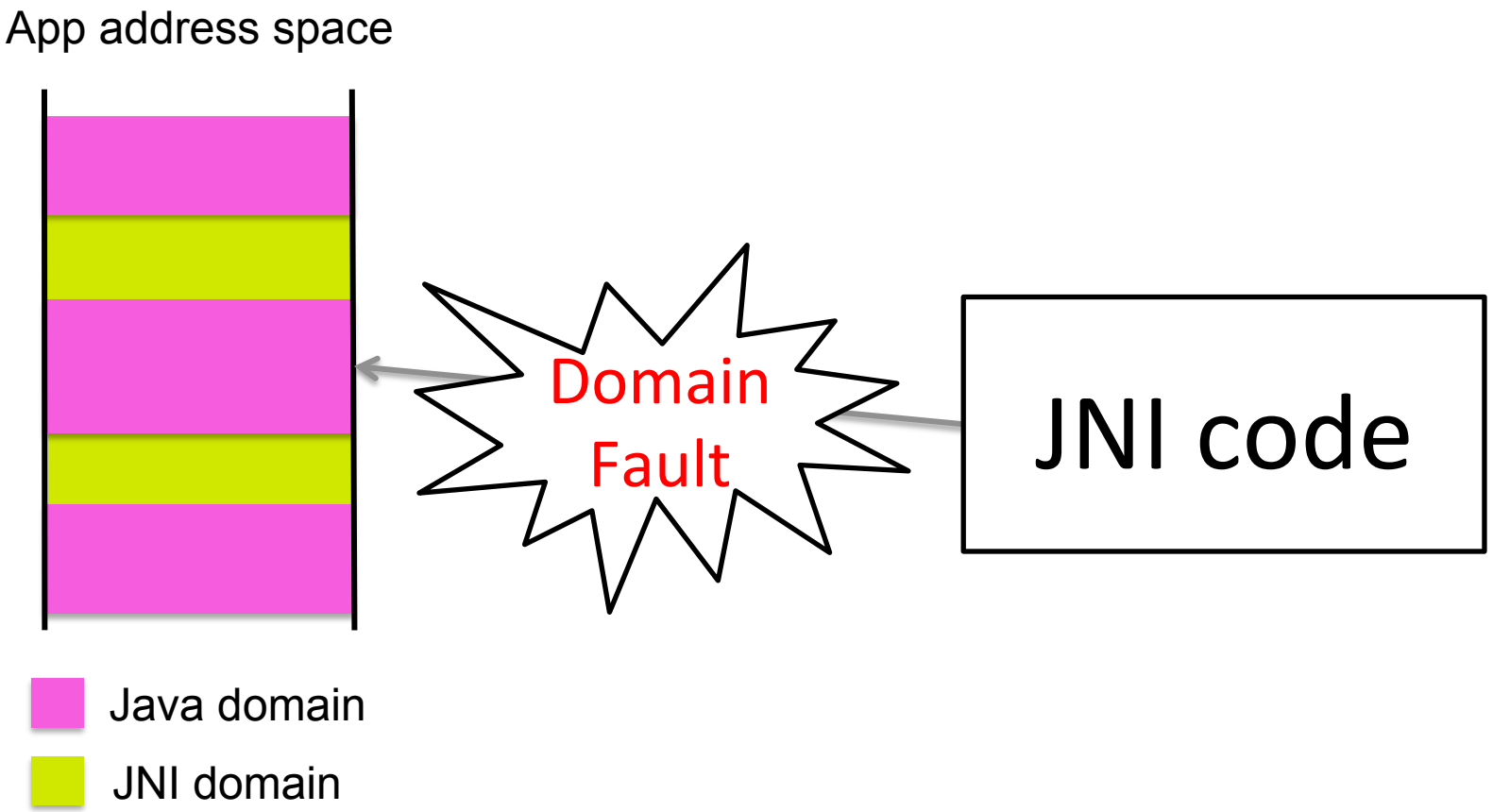
Fault Isolation using ARM Memory Domain



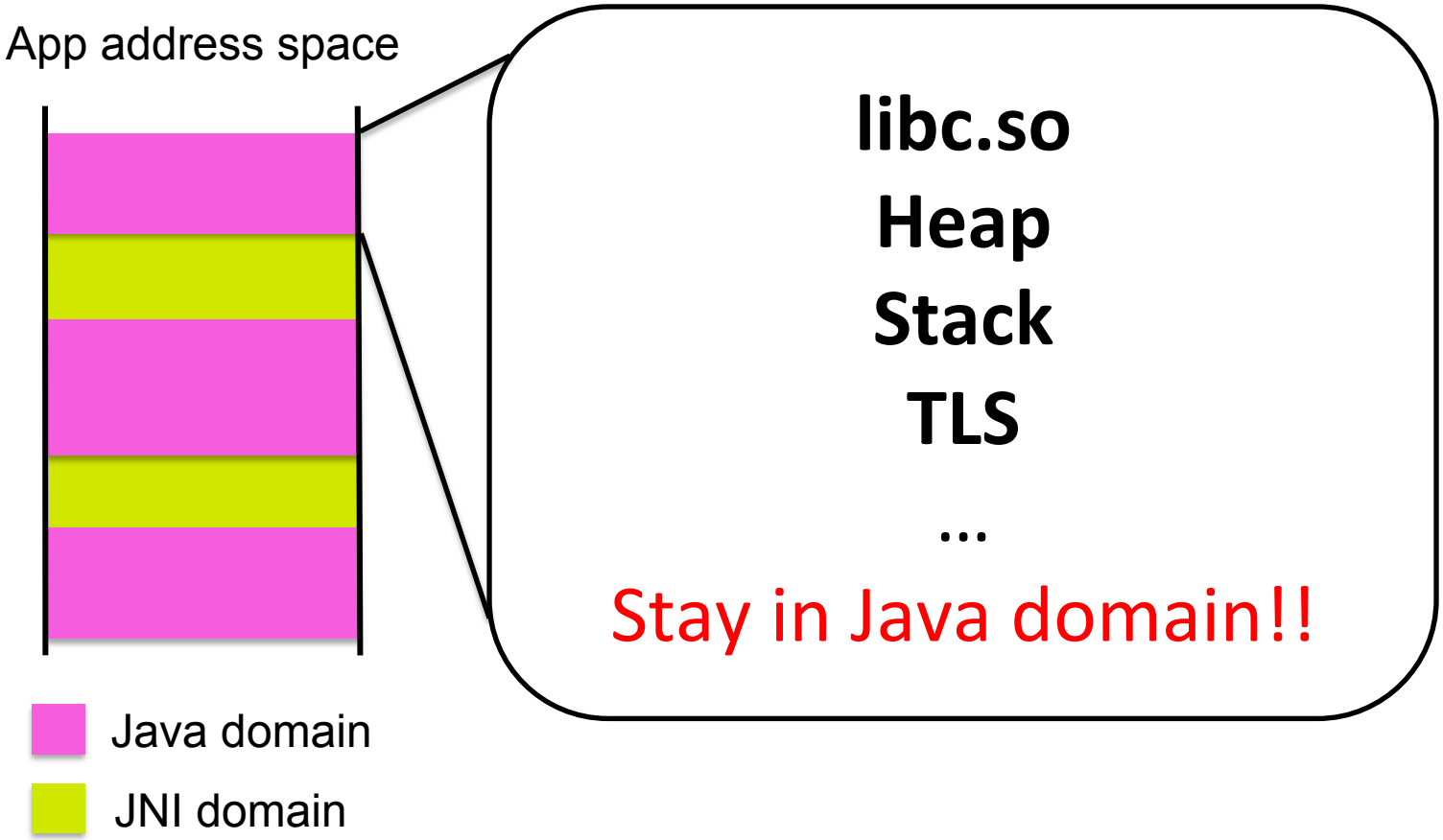
Fault Isolation using ARM Memory Domain



Fault Isolation using ARM Memory Domain



Memory and Shared Libraries for JNI



Memory and Shared Libraries for JNI

- Shared libraries (e.g., libc.so), heap, stack and TLS are **in Java domain**
  - JNI cannot access them

→ FLEXDROID provides JNI with **independent** shared libraries, heap, stack and TLS

Defense against Reflection

- **Problem**: A third-party library can dynamically generate a class with the package name of its host application

Defense against Reflection

- **Problem**: A third-party library can dynamically generate a class with the package name of its host application

```
package com.malicious.lib
class A
  method launch_attack
    generateClass("com.host.B")
    generateClass("com.host.B", "malFunction")
    loadClass("com.host.B")
    com.host.C.setCallback(new com.host.B())
  end method
end class
```

	P	Call stack
↓	H	com.host.C.runCallback
	L	com.host.B.malFunc

Defense against Reflection

- **Problem**: A third-party library can dynamically generate a class with the package name of its host application

```
package com.malicious.lib
class A
  method launch_attack
    generateClass("com.host.B")
    generateClass("com.host.B", "malFunction")
    loadClass("com.host.B")
    com.host.C.setCallback(new com.host.B())
  end method
end class
```

	P	Call stack
↓	H	com.host.C.runCallback
	L	com.host.B.malFunc
		↳ **com.malicious.lib**

FLEXDROID maintains the information of **class loader**

Implementation

- Android 4.4.4 Kitkat / Linux 3.4.0

	# of Files	Insertion (LoC)	Deletion (LoC)
Kernel	28	1831	25
Android Framework	46	1466	77
Dalvik VM	24	6081	22
Bionic	23	2827	70
Others	12	95	24
Total	**133**	**12300**	**218**

EVALUATION

Overview

- How effective is FLEXDROID's policy to restrict third-party libraries?
- How easy is it to adopt FLEXDROID's policy to existing Android apps?
- How much performance overhead does FLEXDROID impose when adopted?

Blocking Permissions with FLEXDROID

- Choosing 8 third-party libraries from real-world apps
- Repackaging their host applications with FLEXDROID policy
  - No permission given to third-party libraries

→ **Denying** all accesses to resources from third-party libraries

Blocking Permissions with FLEXDROID

- Choosing 8 third-party libraries from real-world apps
-

FLEXDROID can block permission abuses of 3rd-party libs

→ **Denying** all accesses to resources from third-party libraries

Blocking Permissions with FLEXDROID

- By modifying only ***AndroidManifest.xml***

```
<flexdroid
android:name="com.ebay.redlasersdk">
<!-- no permission -->
</flexdroid>
```

→ **Easy** to adopt FLEXDROID's policy

Backward Compatibility

- Run 32 popular apps from Google Play without any modification in FLEXDROID
- Check to see if each of them crashes during the execution

→ 27 of 32 apps run as normal

Other apps crashed due to JNI sandbox

→ FLEXDROID has a high backward compatibility

Performance Evaluation

- Environment setting
  - Nexus 5
  - Turning on all cores with maximum CPU frequency
- Micro-benchmark
- Macro-benchmark
  - K-9 email app

Micro-benchmark Result

Main factors of performance overheads

1. Inter-process stack inspection

   → *438 ~ 594 μs*

2. Sandbox switch

   (i.e., switch to JNI domain / Java domain)

   → *89 μs*

Macro-benchmark Result

In the experiment using K-9 email app

1. Launching the app

   → *1.55 %*

2. Send an email

   → *1.13 %*

Macro-benchmark Result

In the experiment using K-9 email app

1. Launching the app

   → *1.55 %*

2. Send an email

   → *1.13 %*

FLEXDROID incurs reasonable performance overheads

Conclusion

- **Problem**: Privacy threats from 3rd-party libraries
- **FLEXDROID**: Extension of Android permission system
  - Supporting in-app privilege separation
  - Resistant against attacks via JNI, reflection and multi-threading
  - Showing reasonable performance overheads

Thank you!

BACKUP SLIDE

Backward Compatibility Issues

- 5 crashed apps
  - Waze Social GPS Map & Travel → Pthread / TLS
  - Uber → mmap()
  - Adobe Acrobat Reader → free()
  - Facebook
  - UC Browser

} Many JNI libraries (29 and 20, respectively) → too complicated to manually analyze them

Previous Works

- AdRisk (Wisec' 12)
  - Report private threats from ad libraries
- AdSplit (Usenix Sec' 12) / AdDroid (AsiaCCS' 12)
  - Separate an ad library from its host app
- NativeGuard (WiSec' 14)
  - Separate a library written in native code from its host app
- Compac (CODASPY' 14)
  - Suggest an idea similar to inter-process stack inspection