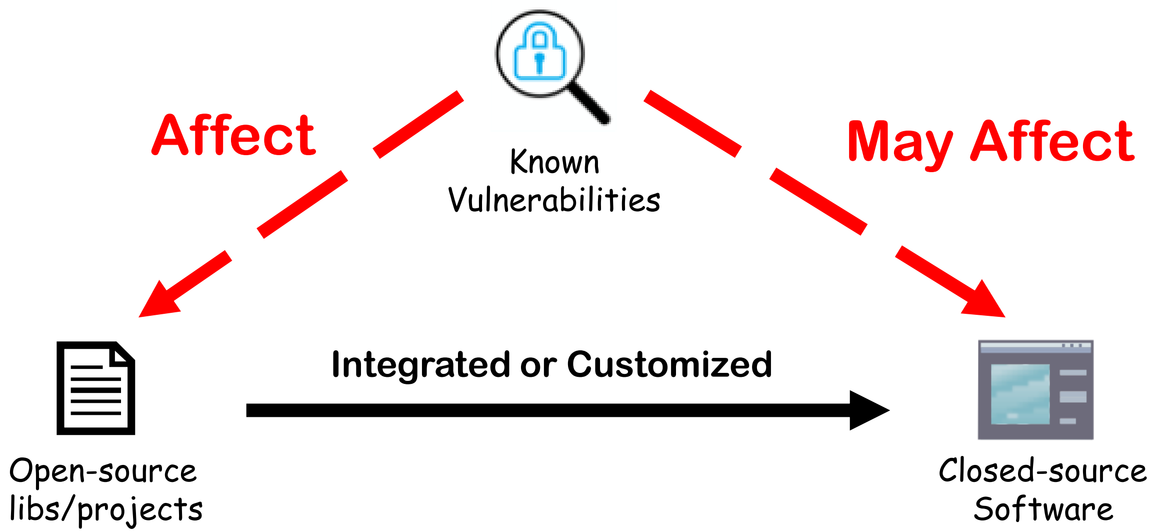
BScout: Direct Whole Patch Presence Test for Java Executables

Jiarun Dai†, Yuan Zhang†, Zheyue Jiang†, Yingtian Zhou†, Junyan Chen†, Xinyu Xing*, Xiaohan Zhang†, Xin Tan†, Min Yang†, Zhemin Yang†

†Fudan University
*Pennsylvania State University

Background

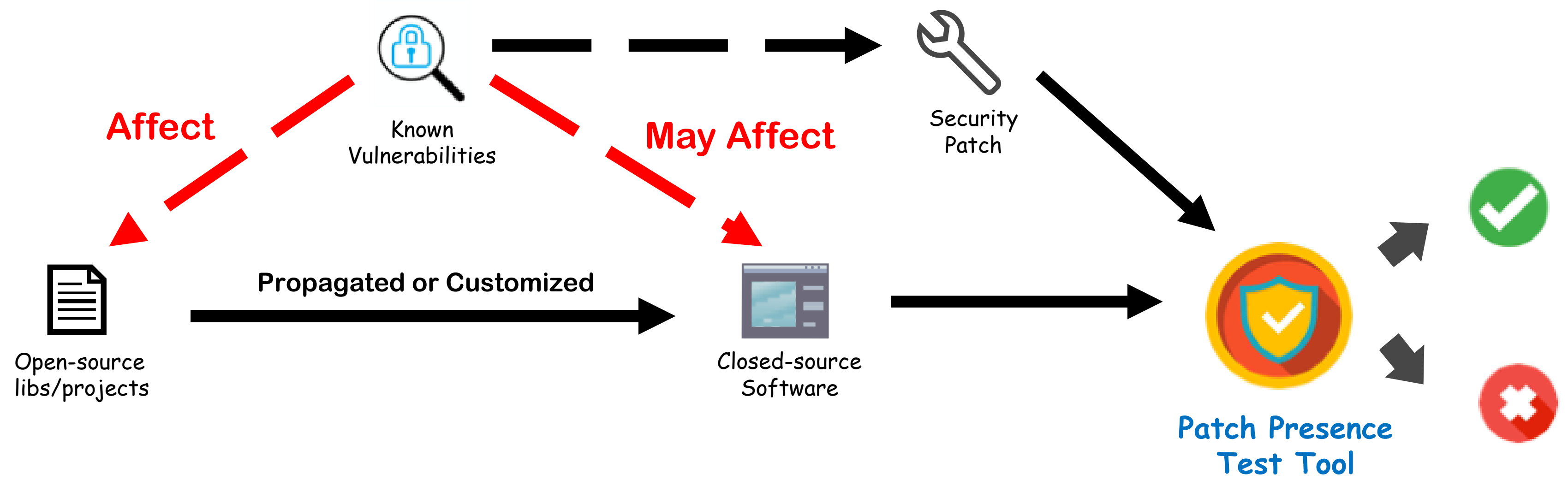
- Vulnerabilities in open-source libs/projects could be propagated to closed-source software



Research Problem:
To check whether a given closed-source software is affected by known vulnerabilities

Patch Presence Test

- Check the presence of a patch in target software



Target Users

Government Agents

Enterprise Users

Third-party users of closed-source product

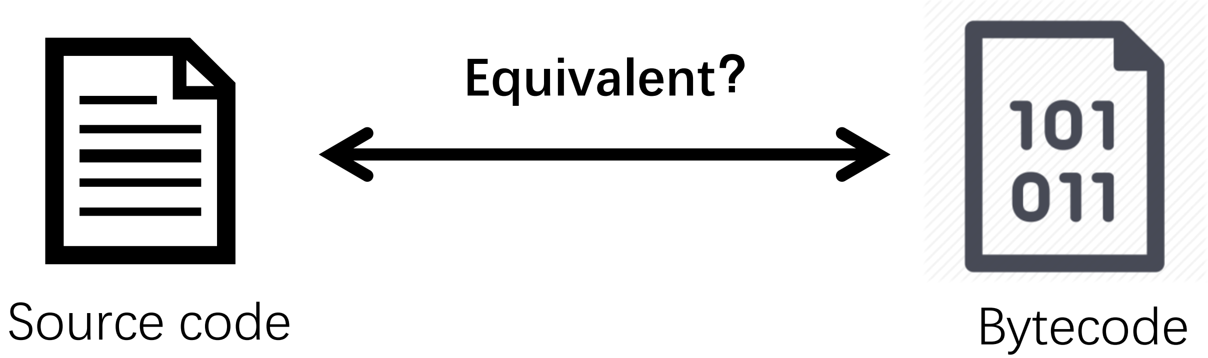
Security Companies

Developers &Testers

Security Analyst

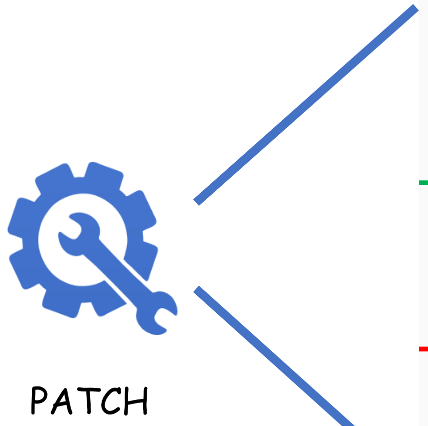
Challenges

- 1) Need to Perform Source-to-Binary Analysis
  - Different Language layers
  - *Therefore, we need to perform a hard cross-layer code equivalence test*



Challenges

- 2) Security patches may only introduce minor changes
  - Example: CVE-2018-9474, move only one line of code
  - *Thus, we need fine-grained detection methods*



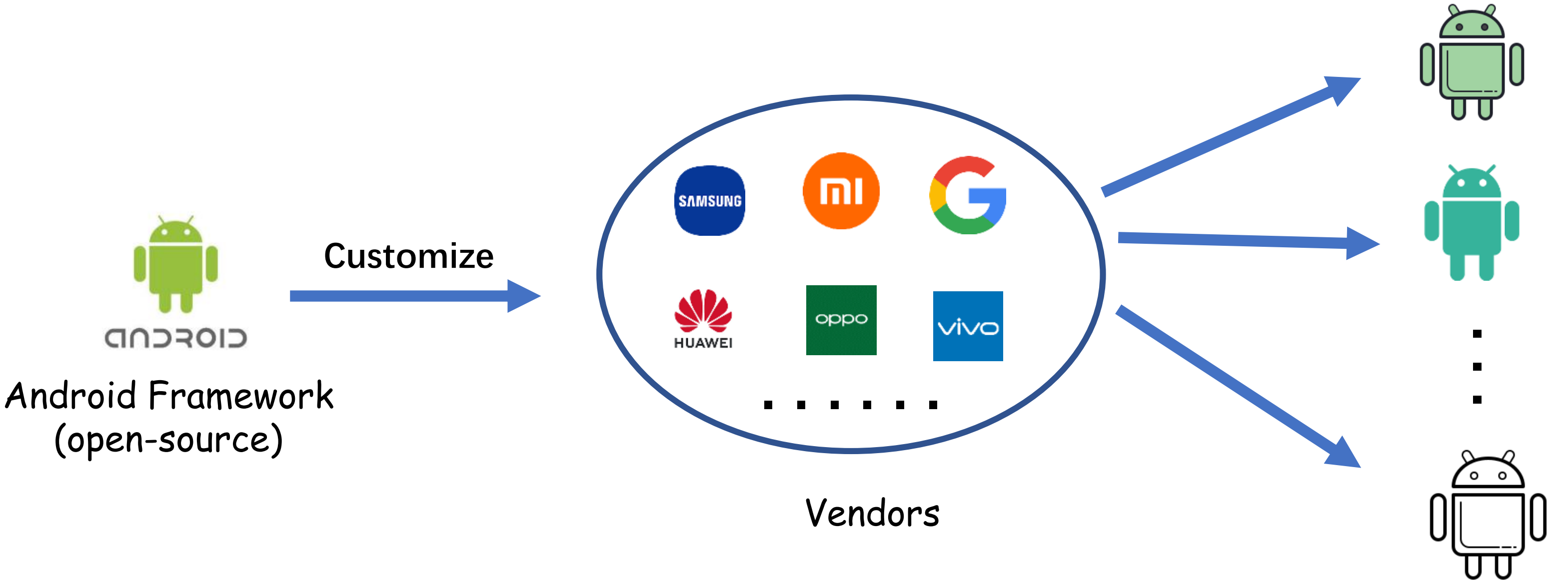

```
@@ -2510,10 +2510,10 @@ public class MediaPlayer extends PlayerBase
         @Override
         public void writeToParcel(Parcel dest, int flags) {
             dest.writeInt(mTrackType);
+            dest.writeString(mFormat.getString(MediaFormat.KEY_MIME));
             dest.writeString(getLanguage());

             if (mTrackType == MEDIA_TRACK_TYPE_SUBTITLE) {
-                dest.writeString(mFormat.getString(MediaFormat.KEY_MIME));
                 dest.writeInt(mFormat.getInteger(MediaFormat.KEY_IS_AUTOSELECT));
                 dest.writeInt(mFormat.getInteger(MediaFormat.KEY_IS_DEFAULT));
                 dest.writeInt(mFormat.getInteger(MediaFormat.KEY_IS_FORCED_SUBTITLE));
{
```

CVE-2018-9474

Challenges

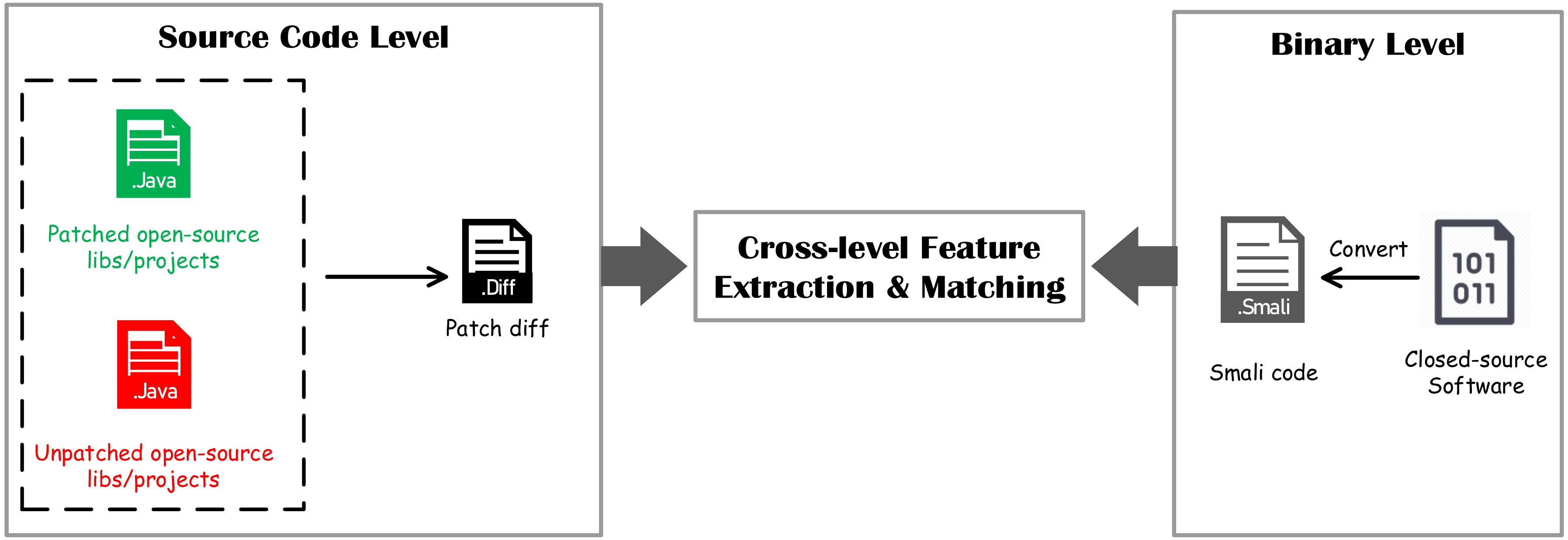
- 3) Test closed-source targets may be customized from open-source counterparts
  - Example: Android ROM, Linux Kernel
  - *Thus, patch presence test should be resilient to code customization*



BScout

- BScout: Direct Whole Patch Presence Test for *Java Executables*
  - Directly test presence of the whole patch in Java **Bytecode** from **Source**

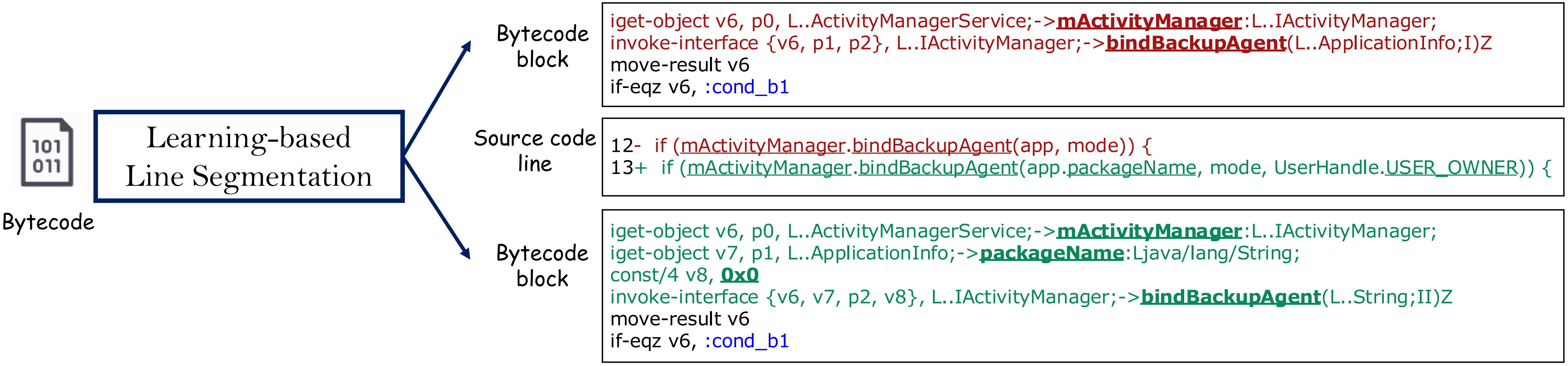
Key Idea-1: Cross-level features to bridge the gap between source and binary.



BScout

- **BScout**: Direct Whole Patch Presence Test for *Java Executables*
  - Directly test presence of the whole patch in Java **Bytecode** from **Source**

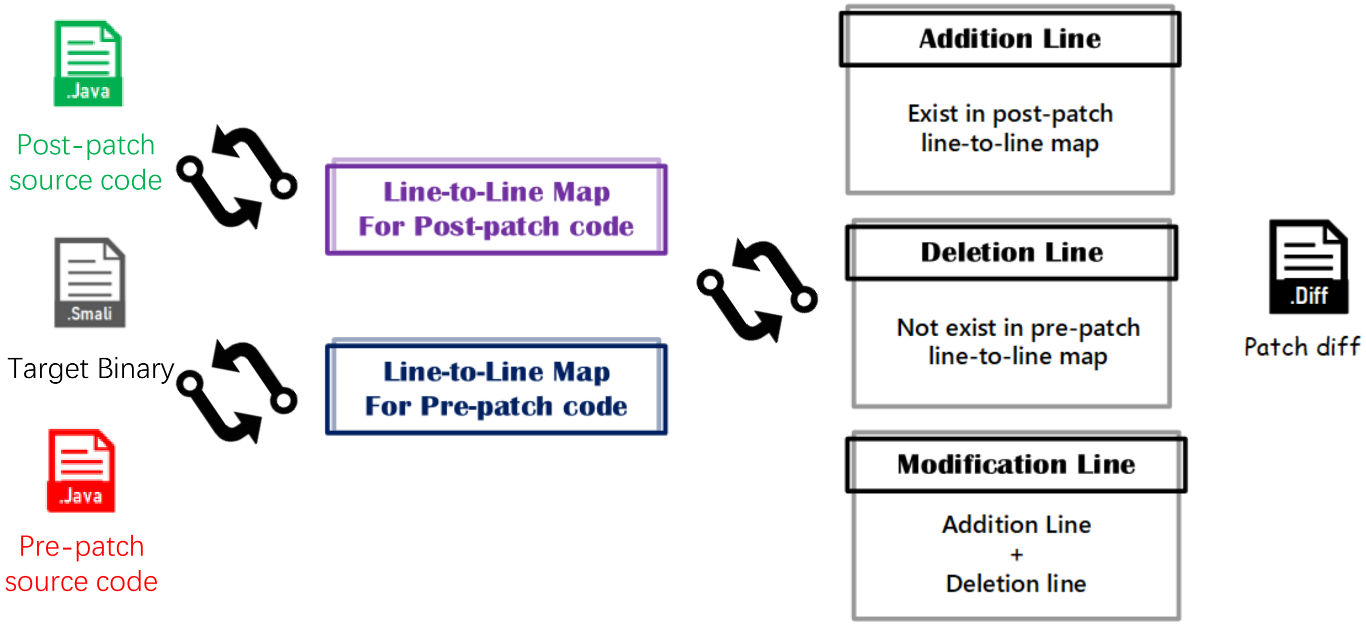
Key Idea-2: Perform fine-grained line-level matching techniques.



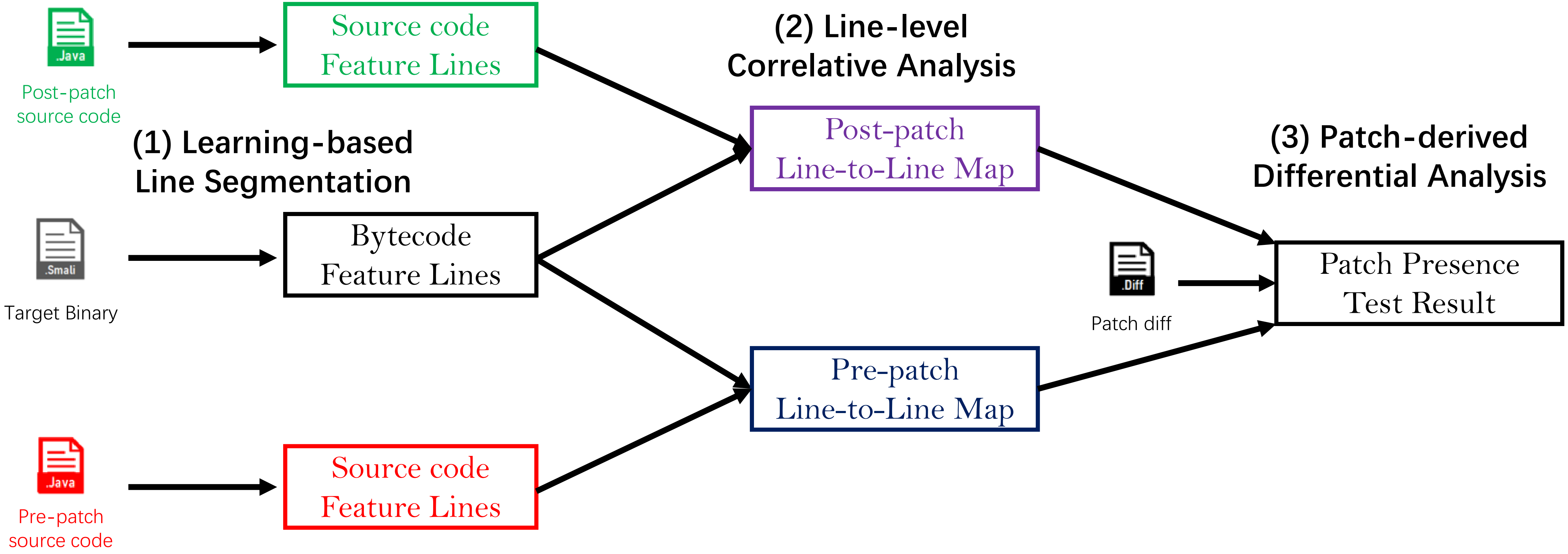
BScout

- **BS**cout: Direct Whole Patch Presence Test for *Java Executables*
  - Directly test presence of the whole patch in Java **Bytecode** from **Source**

Key Idea-3: Utilize pre-/post-patch code to neutralize customization.



Workflow



.Java
Post-patch source code
Source code Feature Lines
(1) Learning-based Line Segmentation
.Smali
Target Binary
Bytecode Feature Lines
.Java
Pre-patch source code
Source code Feature Lines
(2) Line-level Correlative Analysis
Post-patch Line-to-Line Map
Pre-patch Line-to-Line Map
(3) Patch-derived Differential Analysis
.Diff
Patch diff
Patch Presence Test Result

Evaluation

Dataset of Android ROMs

150 Android Framework CVEs(**2015.8 ~ 2019.7**)

15 Android ROMs from **6** vendors

Google, Xiaomi, Meizu, Vivo, Oppo, Samsung

Dataset of Java Apps

15 CVEs of **11** popular 3rd-party libraries

261 Android Apps

28 Desktop/Server Apps

Tool	Android ROMs					Java Apps					Overall	
	TP	TN	FN	FP	Acc.	TP	TN	FN	FP	Acc.	Acc.	FPR
BScout	266	177	31	0	93.5%	286	410	5	0	99.3%	96.9%	0.0%

Empirical Study

- Understand Real-world Patch Application Practice

2,506 Android ROMs from **7** vendors
(Google, Xiaomi, Meizu, Huawei,
Oppo, Vivo, Samsung)

Vendor	Phone Models	Count	Versions	Build Time
Google	14	569	4.4.4-8.1.0	2014.06-2019.05
Samsung	24	468	5.0.0-8.1.0	2016.10-2018.09
Meizu	44	481	5.0.1-8.1.0	2015.06-2019.07
Xiaomi	45	464	4.4.4-8.1.0	2016.02-2019.08
Oppo	31	281	4.4.4-8.1.0	2014.11-2019.08
Vivo	46	152	5.0.2-8.1.0	2015.11-2019.05
Huawei	31	91	6.0.0-7.0.0	2016.01-2017.10

150 Android Framework CVEs
2015.8 ~ 2019.7

Android Version	# of Affected CVEs
Android 4.*	40
Android 5.*	69
Android 6.*	95
Android 7.*	92
Android 8.*	50
Android 9.*	26
Total	150[1]

Empirical Study

- **Q1**: What is the average lag for different vendors to apply a patch?

Table 4: The average patch lag for different vendors.

Vendor	# of Selected Phone Models	# of ROMs per Model	Average Patch Lag per Model (day)
Google	12	20 ~ 77	-65 ~ -21.47
Samsung	16	10 ~ 54	38.16 ~ 412
Xiaomi	33	10 ~ 31	70.07 ~ 449.25
Meizu	25	10 ~ 29	85 ~ 411
Vivo	2	10 ~ 12	186.35 ~ 194.58
Oppo	10	11 ~ 24	62.71 ~ 368.89
Huawei	1	10	65.55

Take-away:

(1) Google proactively applies patches before announcing the vulnerabilities to the public.

(2) Third-party device manufactures apply patches relatively slowly.

Empirical Study

- **Q2**: Do vendors patch vulnerabilities for one model but ignore another?

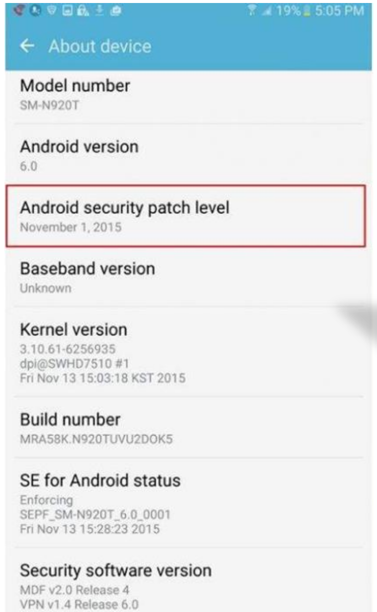
Vendor	# of Ill-managed CVEs	# of Ill-managed Models
Google	24	12
Samsung	76	25
Meizu	93	43
Xiaomi	75	43
Oppo	63	20
Vivo	41	35
Huawei	33	32

Ill-managed Model: models forgotten by vendors to patch some vulnerabilities
Ill-managed CVE: vulnerabilities forgotten by vendors to be patched

Take-away:
All vendors (including Google) have ever patched a vulnerability on one model but forgot to patch the same vulnerability on another model.

Empirical Study

- **Q3**: Do vendors correctly set security patch level?



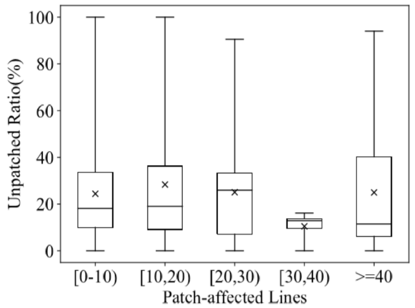
Vendor	# of Negligent ROMs	# of Diligent ROMs	# of Prudent ROMs
Google	112	182	185
Samsung	376	12	66
Meizu	412	6	5
Xiaomi	448	2	8
Oppo	173	19	24
Vivo	139	2	6
Huawei	89	0	2

1) **Negligent ROMs:** some unpatched vulnerabilities at a lower patch level
2) **Diligent ROMs**: all vulnerabilities before declared patch level patched
3) **Prudent ROMs:** even patch some vulnerabilities at a higher patch level

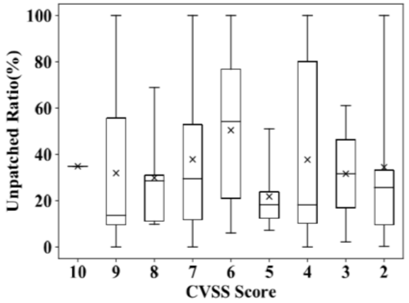
Take-away:
Every vendor including Google inevitably over-claims the security patch level in some of their devices.

Empirical Study

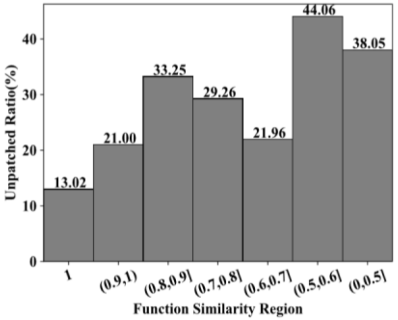
- **Q4**: Which factors affect the application ratio for a security patch?



Complexity of a security patch?



Severity of a vulnerability?



Code Customization?

Take-away:
It seems vulnerability severity and patch complexity are not considered by vendors in applying patches, but code customization is an obvious obstacle.

THANKS
Q&A

jrdai14@fudan.edu.cn

Evaluation

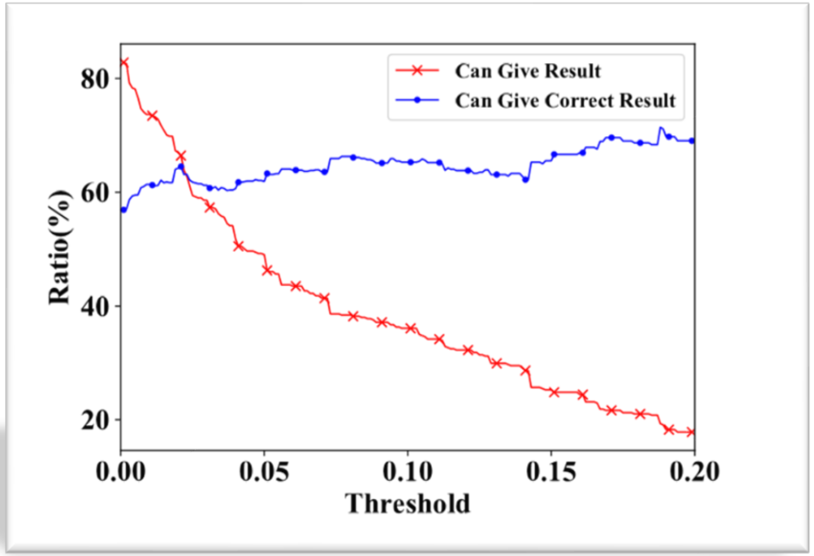
- Comparison with Function-level Similarity
  - Leverage **Centroid**[1] to measure function-level similarity
  - Pre-patch and post-patch versions of 471 patch-related functions

When ***|α – β| > threshold****,* *How to determine the threshold ?*

if ***α > β****, then patch is absent, otherwise, patch is present*

Detection Strategy

When the threshold increases,
1) the ratio of **can give result** drop dramatically,
2) while the ratio of **can give correct result** does not increase significantly



[1] Achieving Accuracy and Scalability Simultaneously in Detecting Application Clones on Android Markets. In ICSE'14.

Evaluation

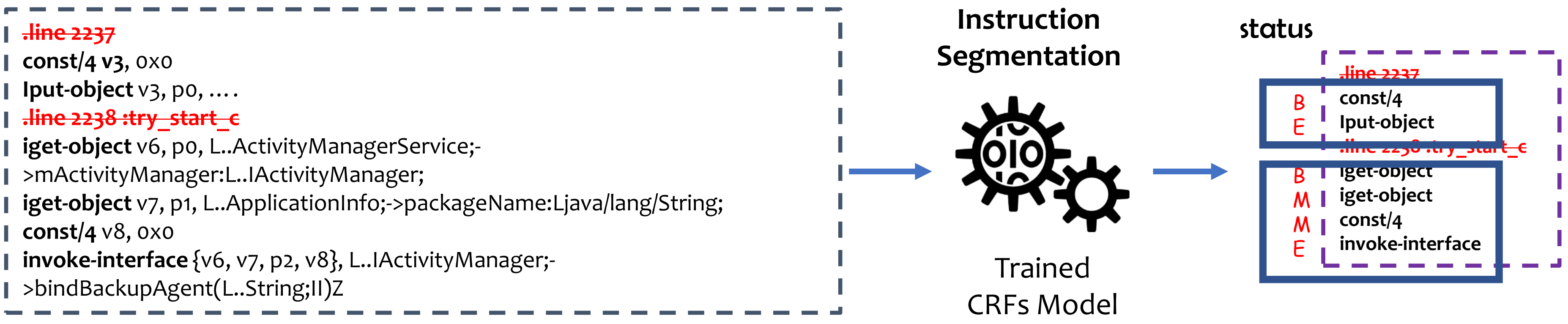
- Comparison with Version Pinning
  - Leverage **LibScout**[1] and **OSSPolice**[2] to do version pinning
  - Reference Set: 215 unique ROMs compiled from different tags and branches of AOSP

Tool	Cannot Give Results		Can Give Results					
	Count	Ratio	TP	TN	FP	FN	Acc.	FPR
LibScout	455	96.0%	12	1	0	6	68.4%	0%
OSSPolice	5	1.1%	69	168	6	226	50.5%	3.5%

[1] Reliable Third Party Library Detection in Android and its Security Applications. In CCS'16
[2] Identifying Open-Source License Violation and 1-day Security Risk at Large Scale. In CCS'17

Learning-based Line Segmentation

- Split raw continuous .smali instructions into ***segments***
  - *Segment: instructions generated from same Java source line*
  - Conditional Random Fields
    - Sequence Labeling
    - **S**: single instruction, **B**: begin of a segment, **M**: middle of a segment, **E**: end of a segment



Line-level Correlative Analysis

- Construct function-level line-to-line map between source code and bytecode
    - 1) Determine whether a source code line and a bytecode block are equivalent

Bytecode block

```
iget-object v6, p0, L..ActivityManagerService;->mActivityManager:L..IActivityManager;
invoke-interface {v6, p1, p2}, L..IActivityManager;->bindBackupAgent(L..ApplicationInfo;I)Z
move-result v6
if-eqz v6, :cond_b1
```

Source code line

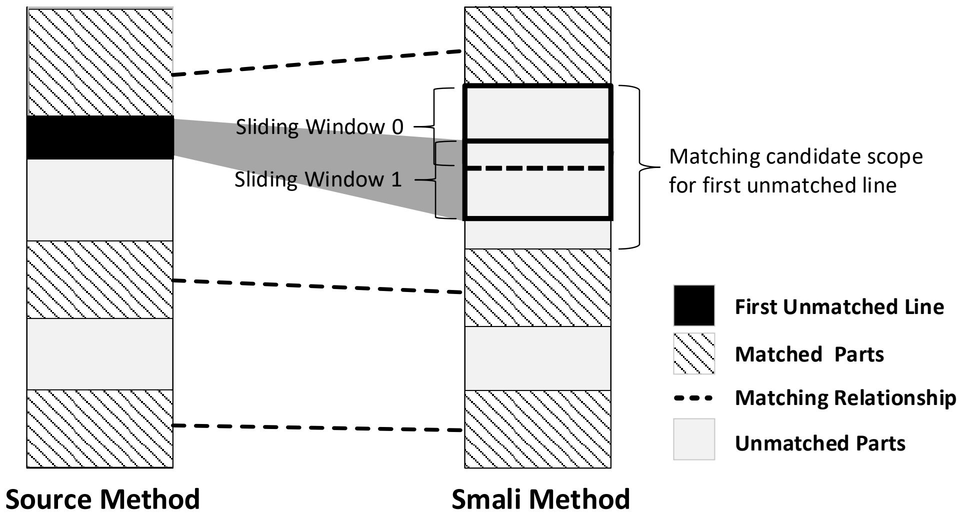
```
12-  if (mActivityManager.bindBackupAgent(app, mode)) {
13+   if (mActivityManager.bindBackupAgent(app.packageName, mode, UserHandle.USER_OWNER)) {
```

Bytecode block

```
iget-object v6, p0, L..ActivityManagerService;->mActivityManager:L..IActivityManager;
iget-object v7, p1, L..ApplicationInfo;->packageName:Ljava/lang/String;
const/4 v8, 0x0
invoke-interface {v6, v7, p2, v8}, L..IActivityManager;->bindBackupAgent(L..String;II)Z
move-result v6
if-eqz v6, :cond_b1
```

Line-level Correlative Analysis

- Construct function-level line-to-line map between source code and bytecode
  - 2) Map all source code lines and instruction blocks in the scope of the whole method



Patch-derived Differential Analysis

- Use following presence test strategies for a patch