Tackling runtime-based obfuscation in Android with TIRO

Michelle Wong and David Lie

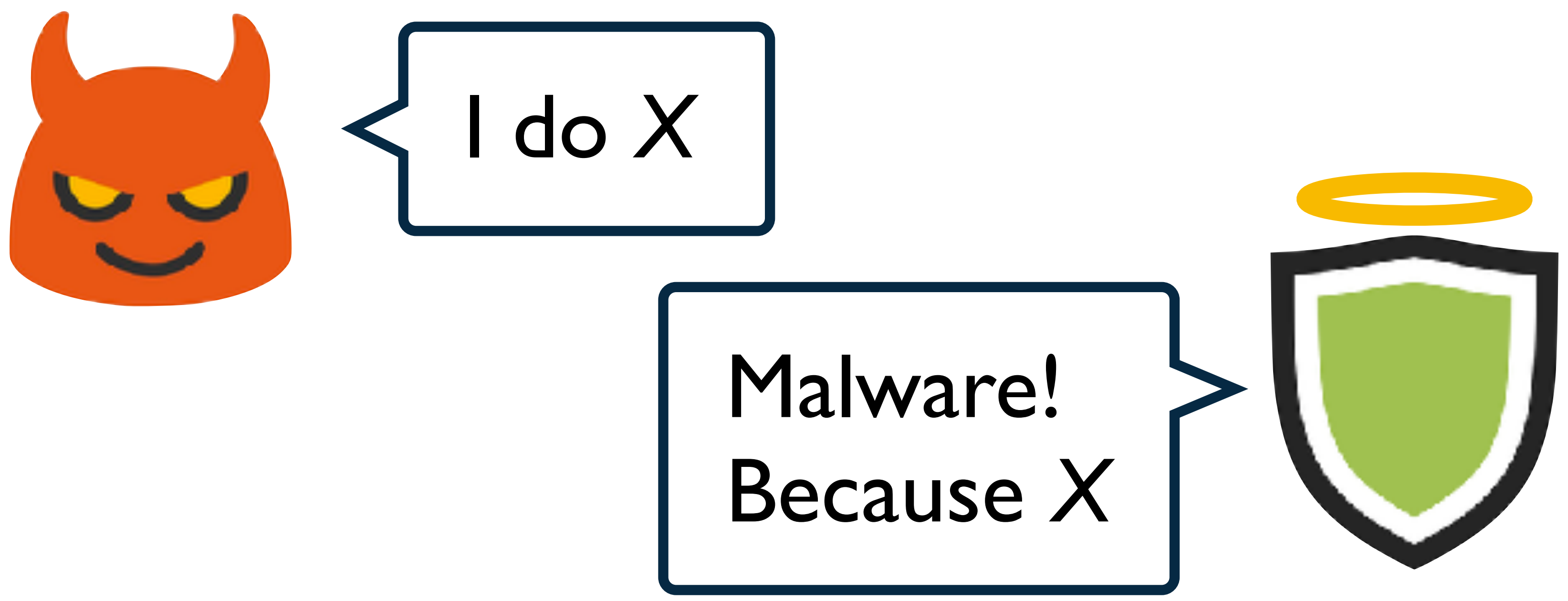
University of Toronto

Usenix Security 2018

The Edward S. Rogers Sr. Department
of Electrical & Computer Engineering
UNIVERSITY OF TORONTO

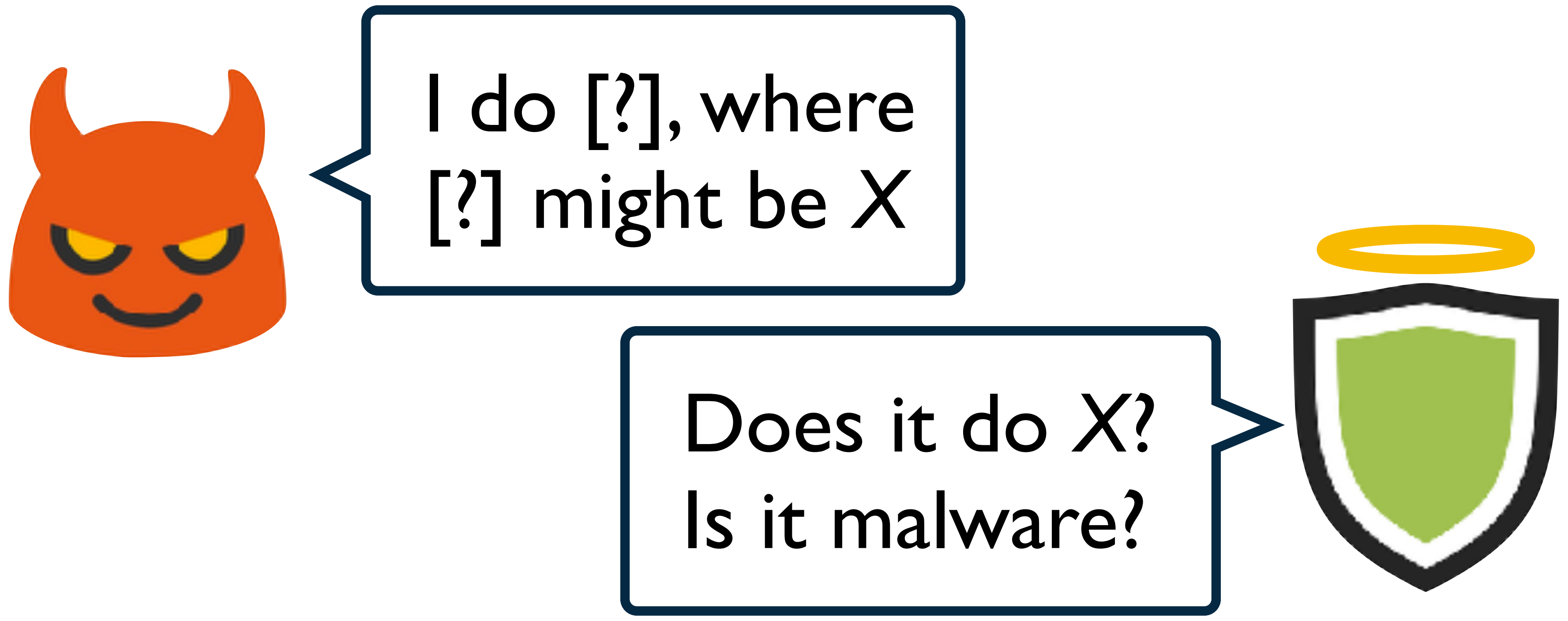
Android malware and analysis

- Mobile devices are a valuable target for malware developers
  - Access to sensitive information and functionality
- Arms race between malware developers and security analyzers

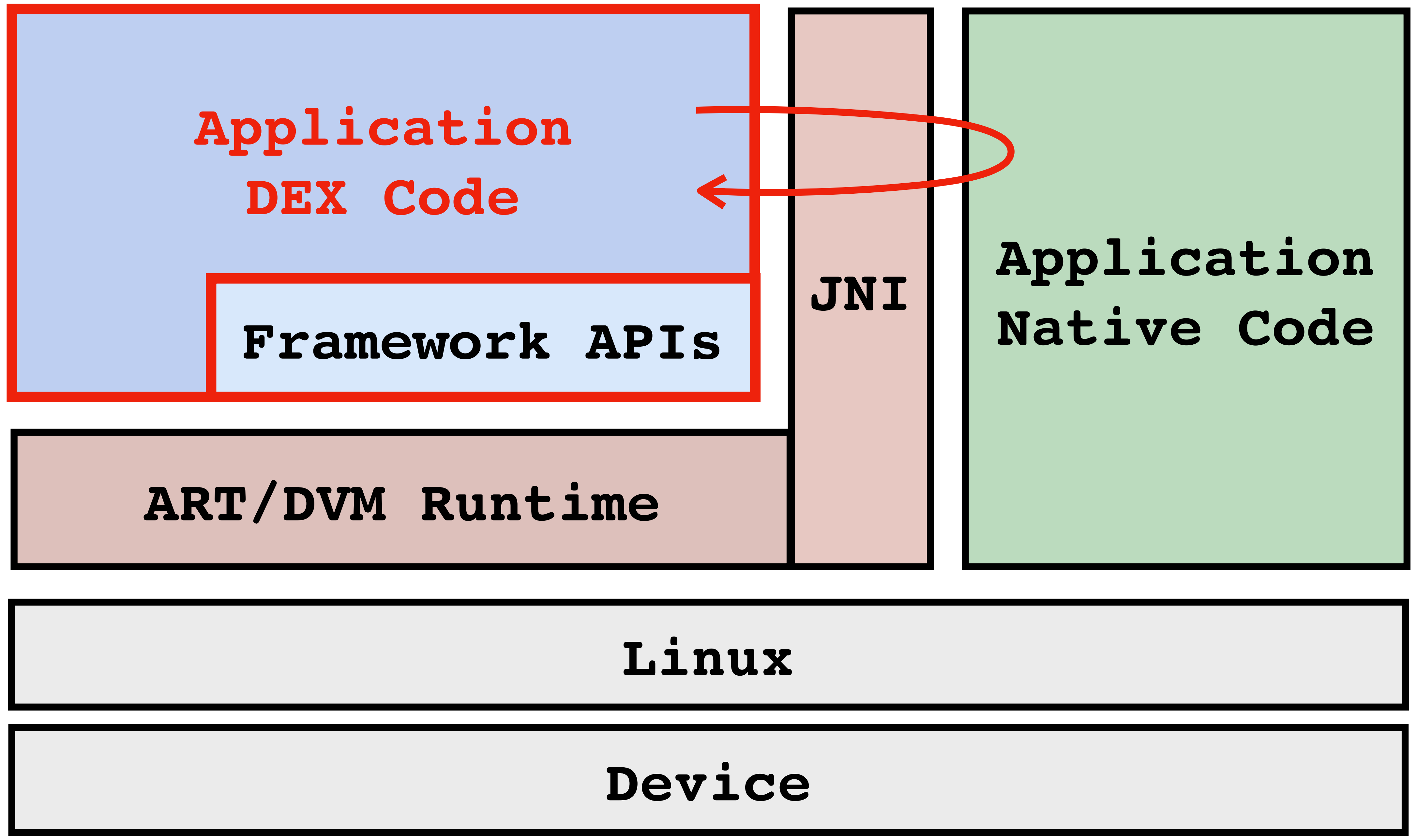


Java obfuscation

- Most Android applications written in Java
- Obfuscation using Java features
  - Reflection
  - Dynamic code loading
  - Native methods

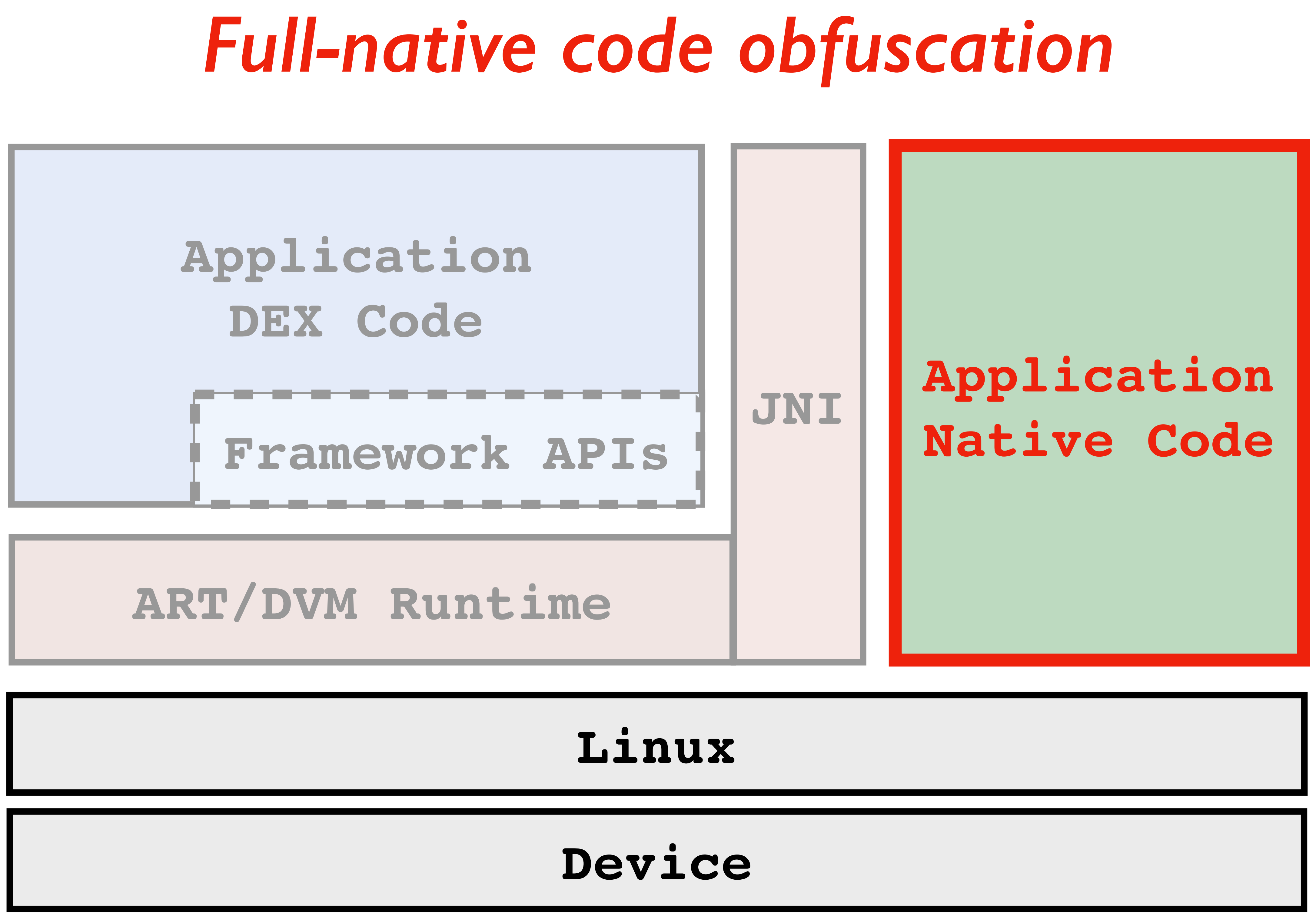


Language-based obfuscation

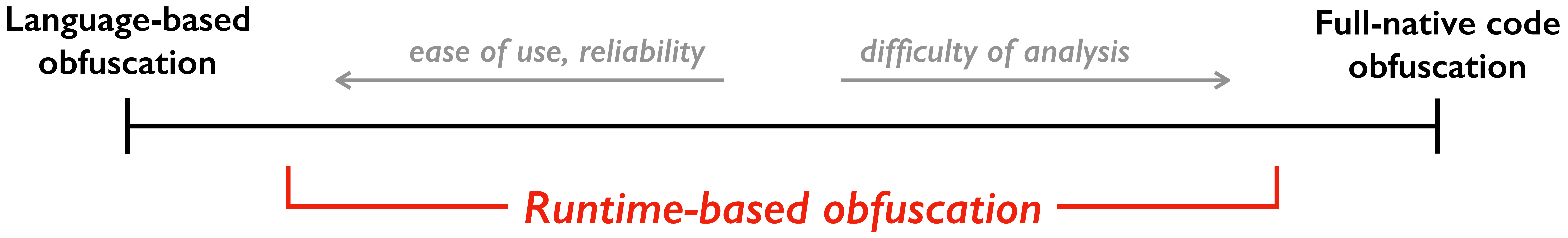


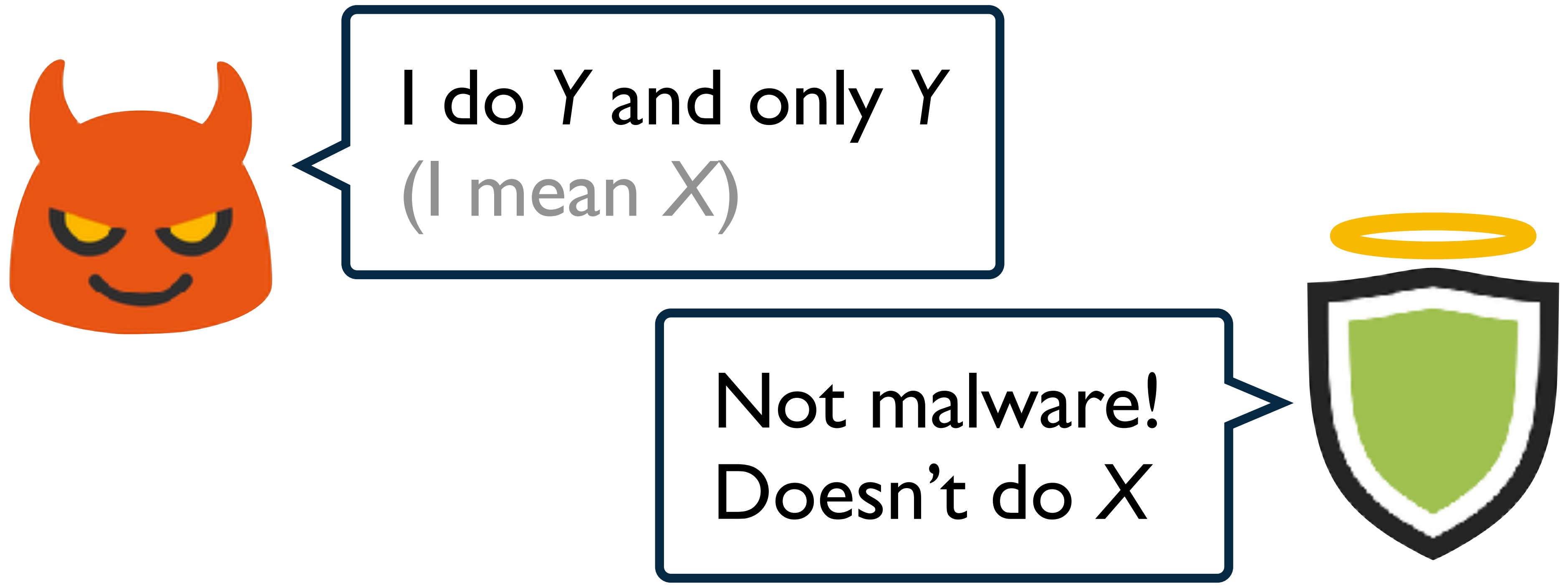
Native obfuscation

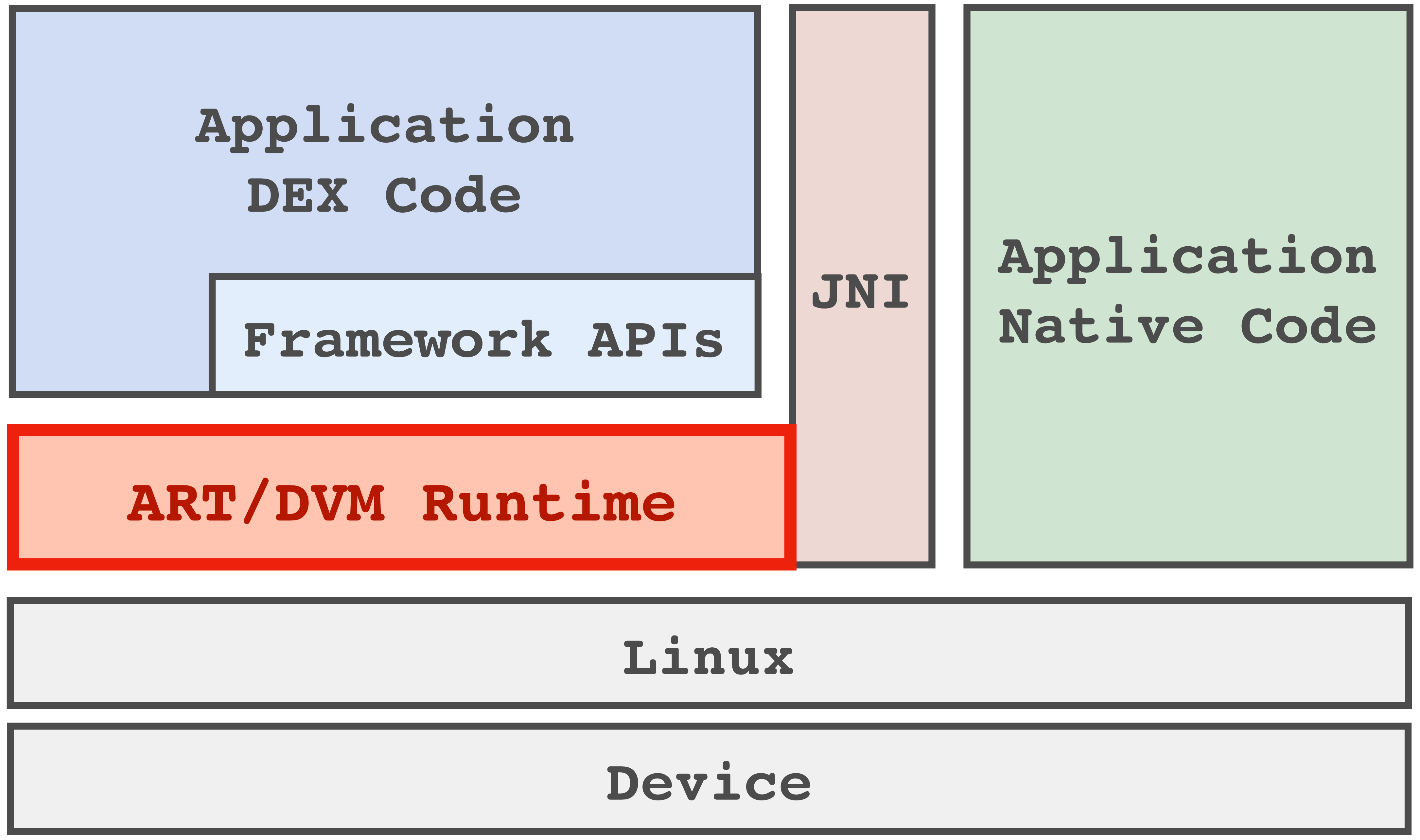
- Can avoid runtime entirely by using native code
  - No Java code or invocations to Java methods
- Seems very little malware do this
  - Framework APIs mostly in Java
  - Requires access to undocumented low-level interfaces of system services



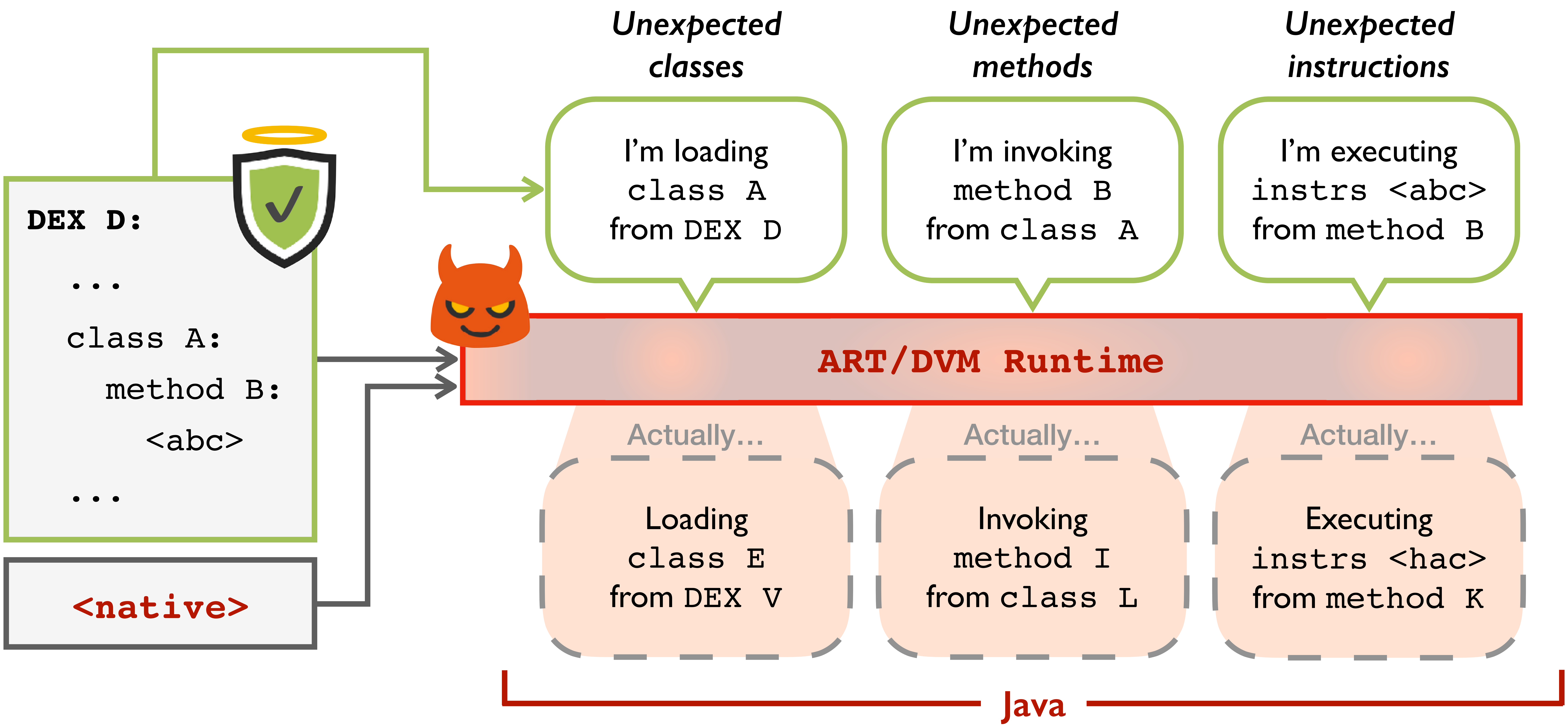
Obfuscation via runtime tampering





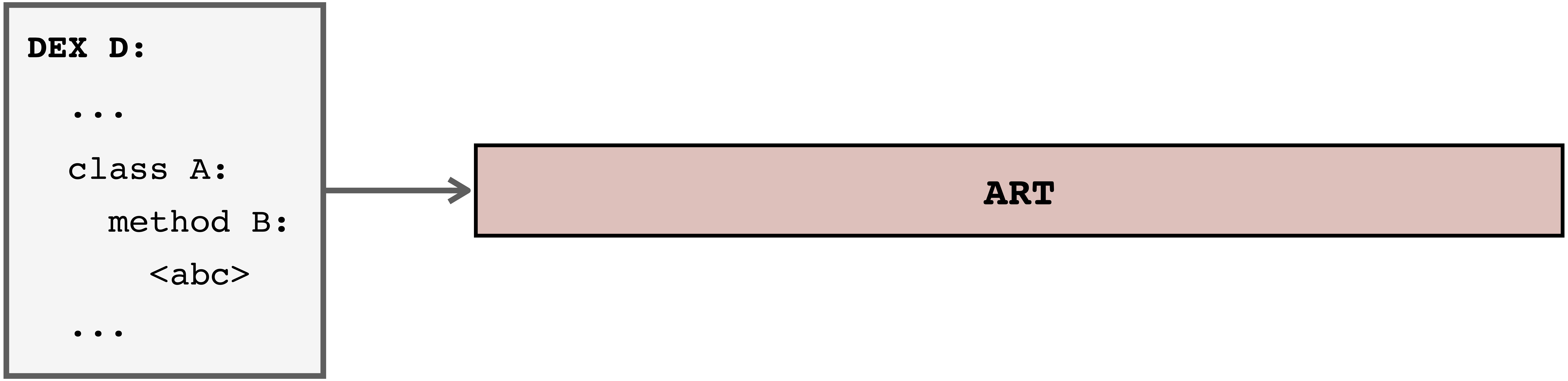


Unexpected code behavior

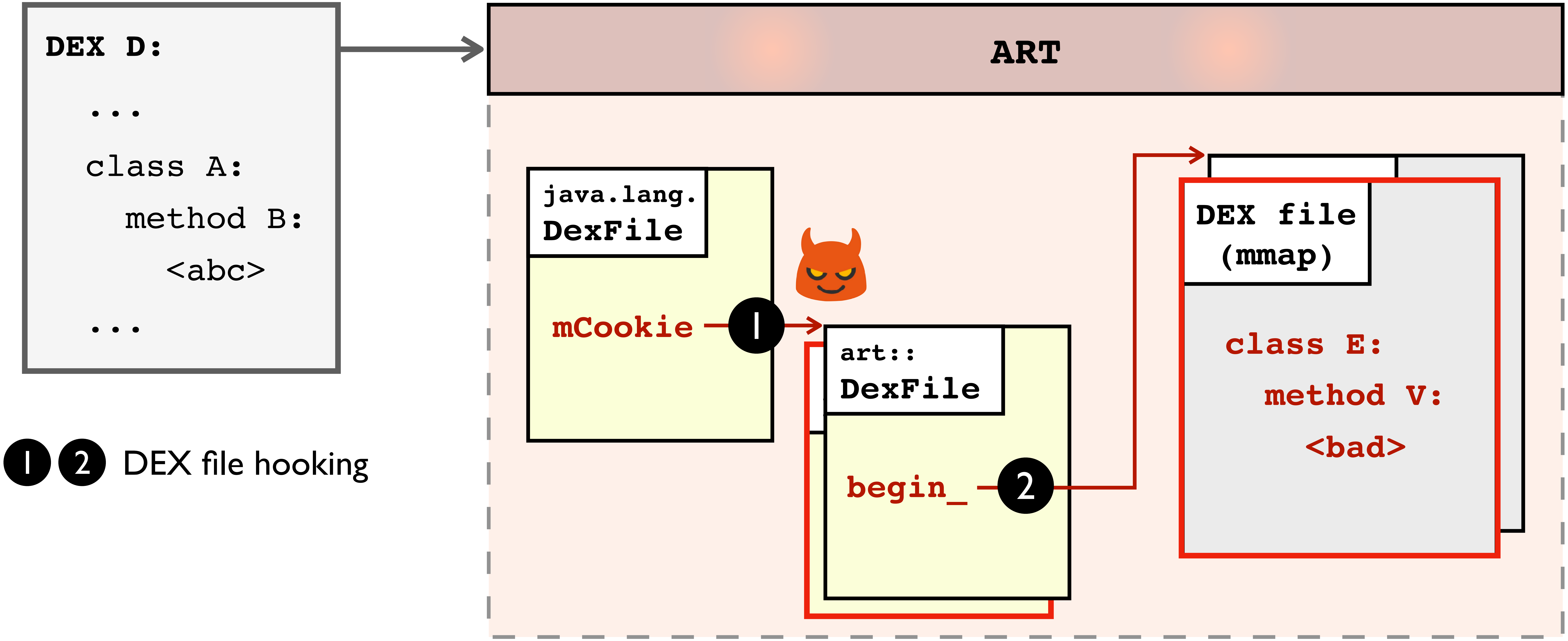


Android RunTime (ART)

- Investigated how code is loaded and executed within ART

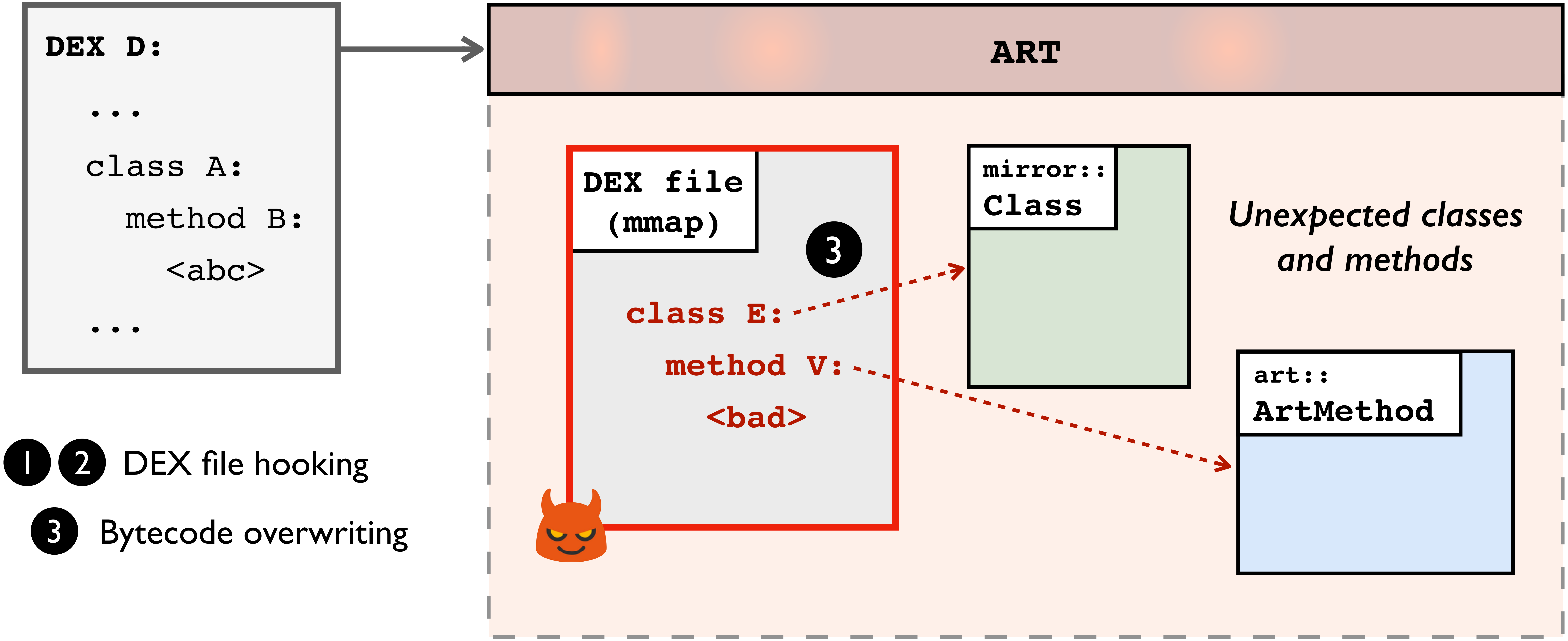


ART code loading



1 2 DEX file hooking

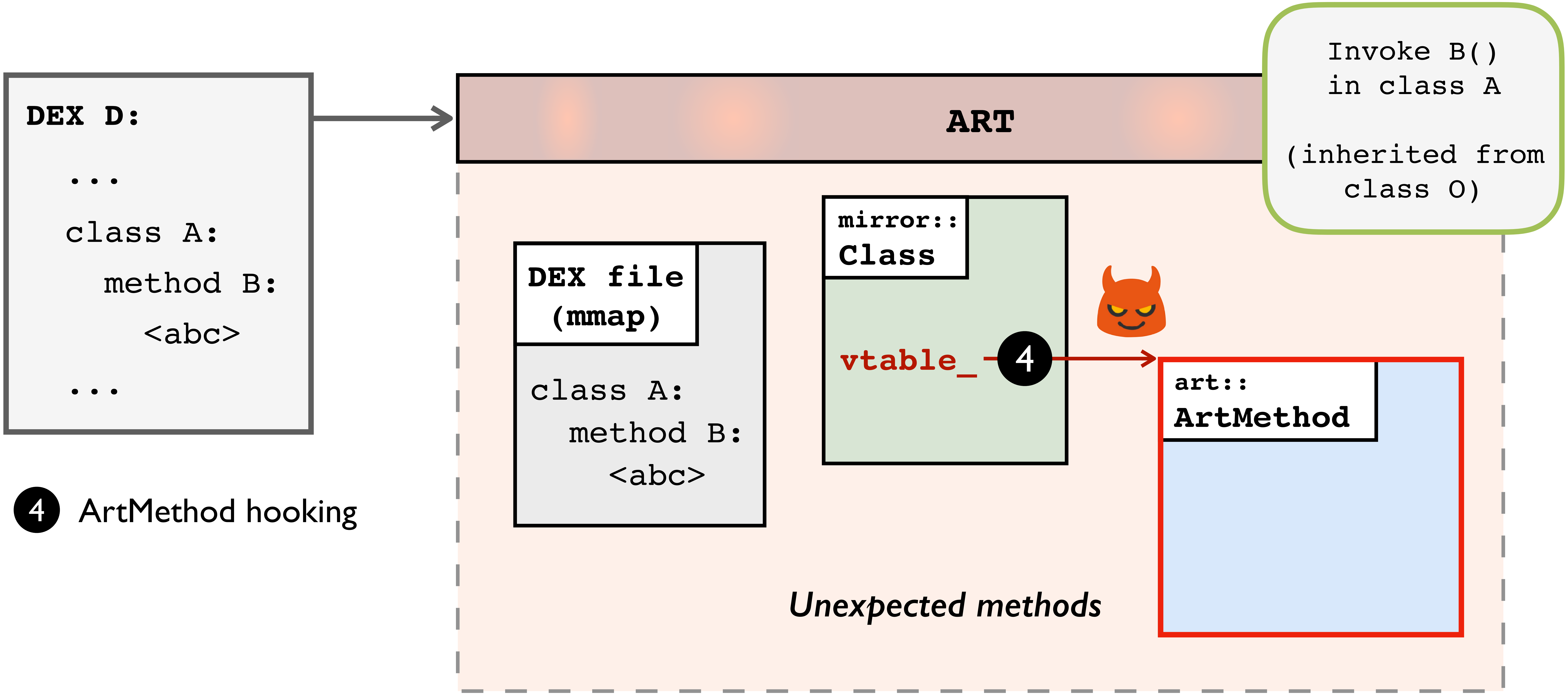
ART code loading



❶ ❷ DEX file hooking

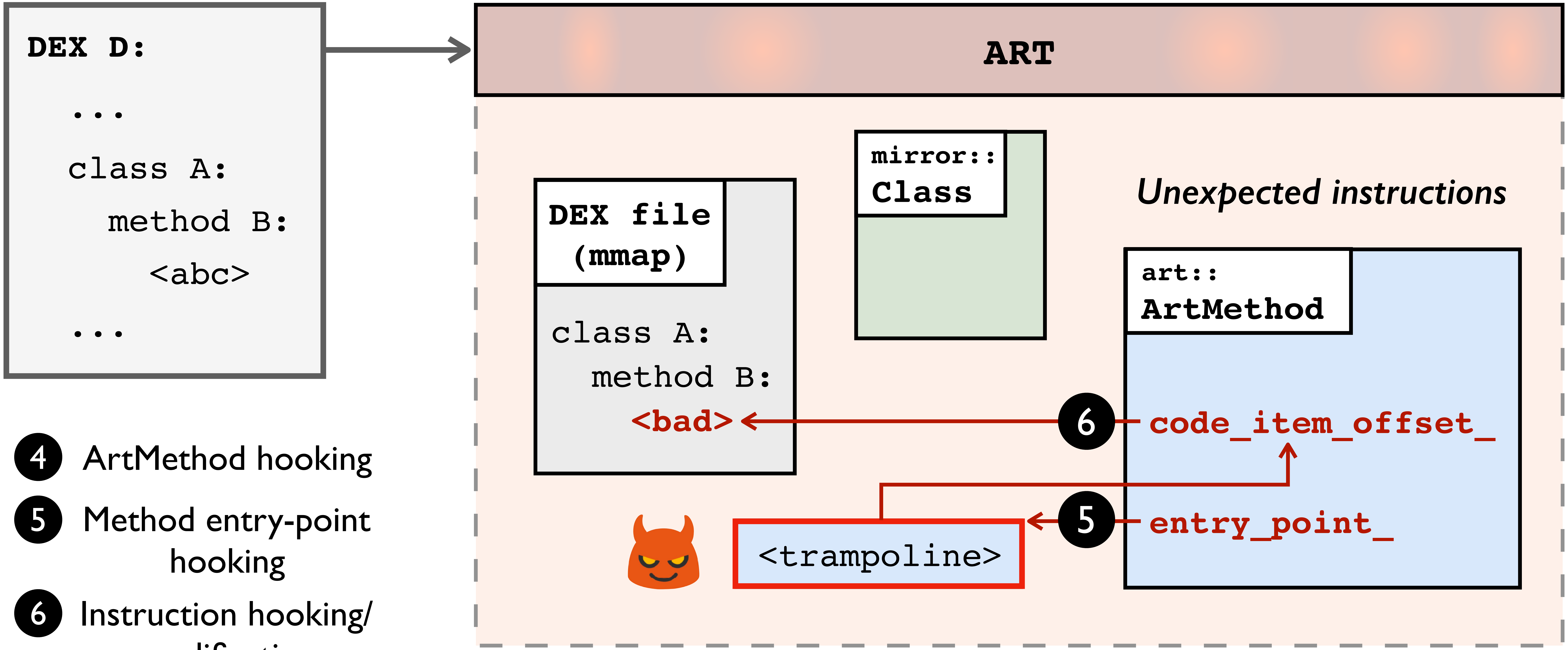
❸ Bytecode overwriting

ART code execution



4 ArtMethod hooking

ART code execution

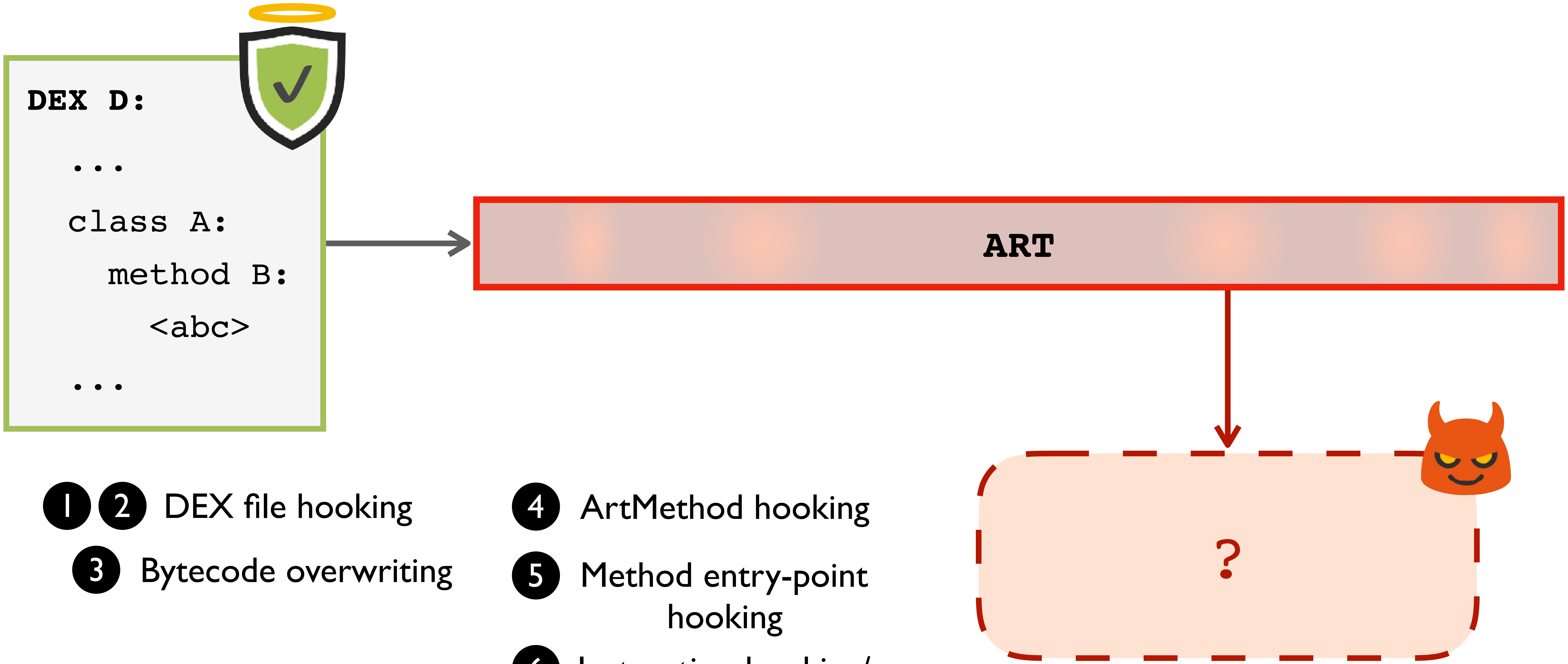


4 ArtMethod hooking

5 Method entry-point hooking

6 Instruction hooking/ modification

Runtime state tampering in ART



❶ ❷ DEX file hooking

❸ Bytecode overwriting

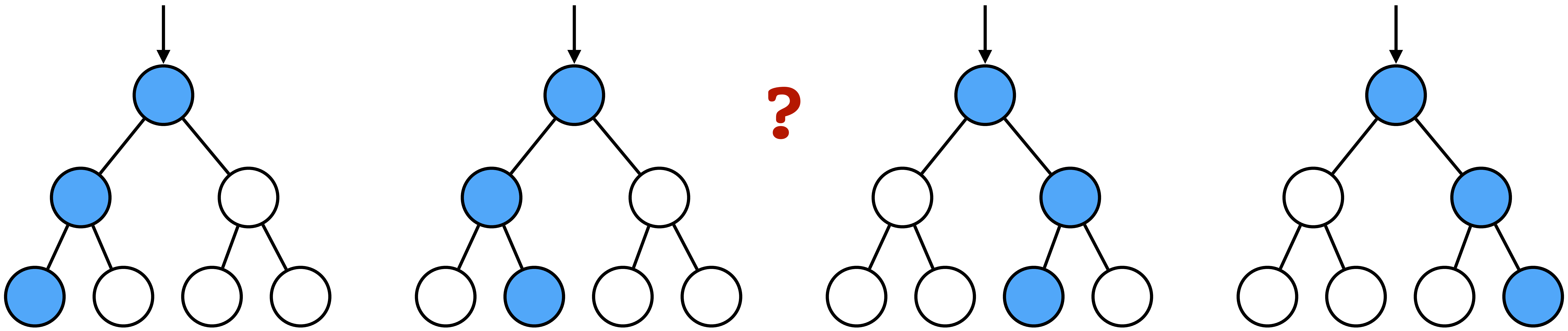
❹ ArtMethod hooking

❺ Method entry-point hooking

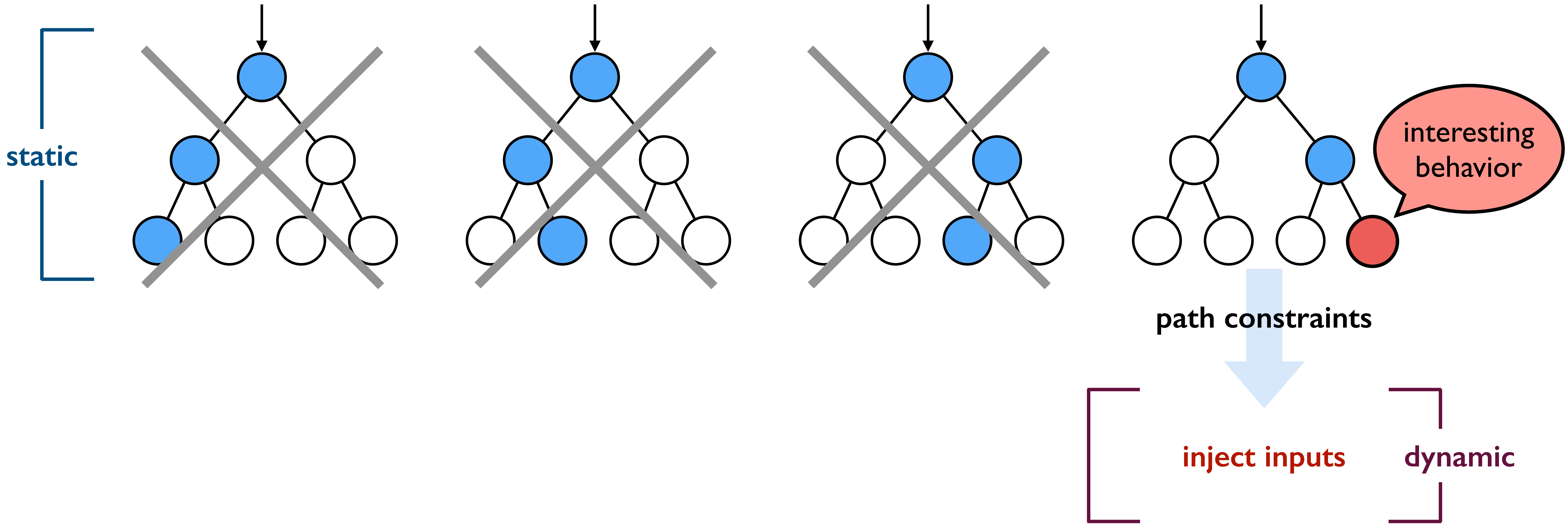
❻ Instruction hooking/ modification

Deobfuscation

- Unified framework to handle language-based and runtime-based obfuscation
- Pure static analysis: imprecise, no run-time information to deobfuscate
  - Reflection targets, dynamically loaded code, etc.
- Pure dynamic analysis: lack of code coverage

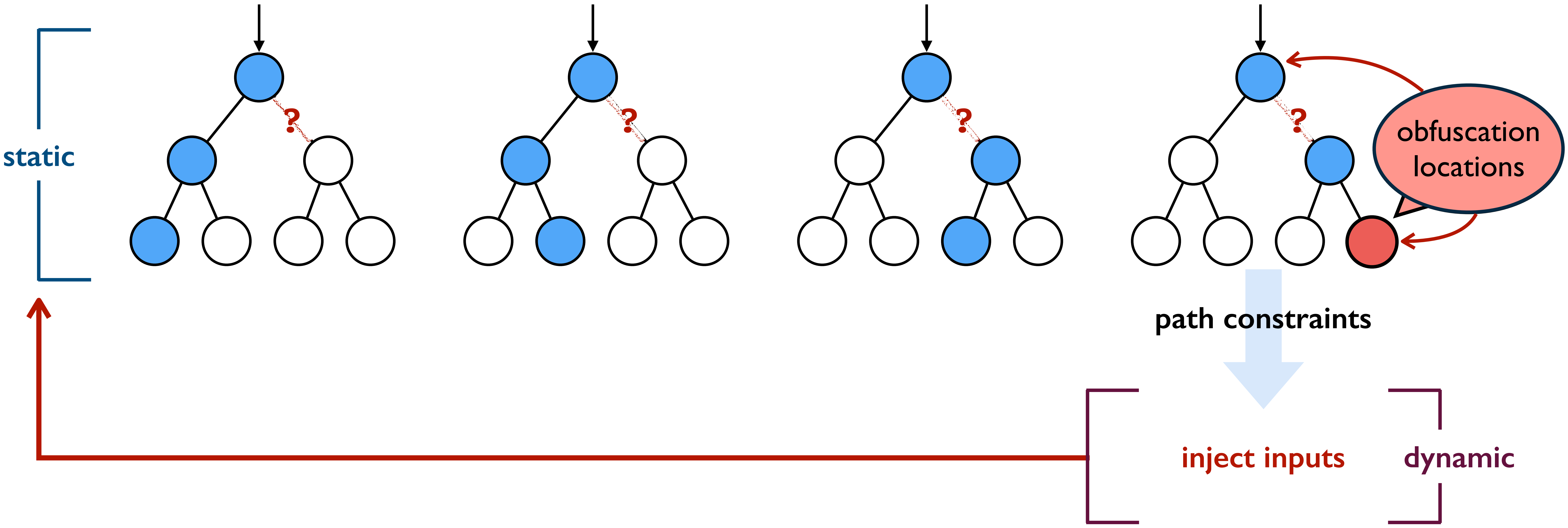


Targeted execution

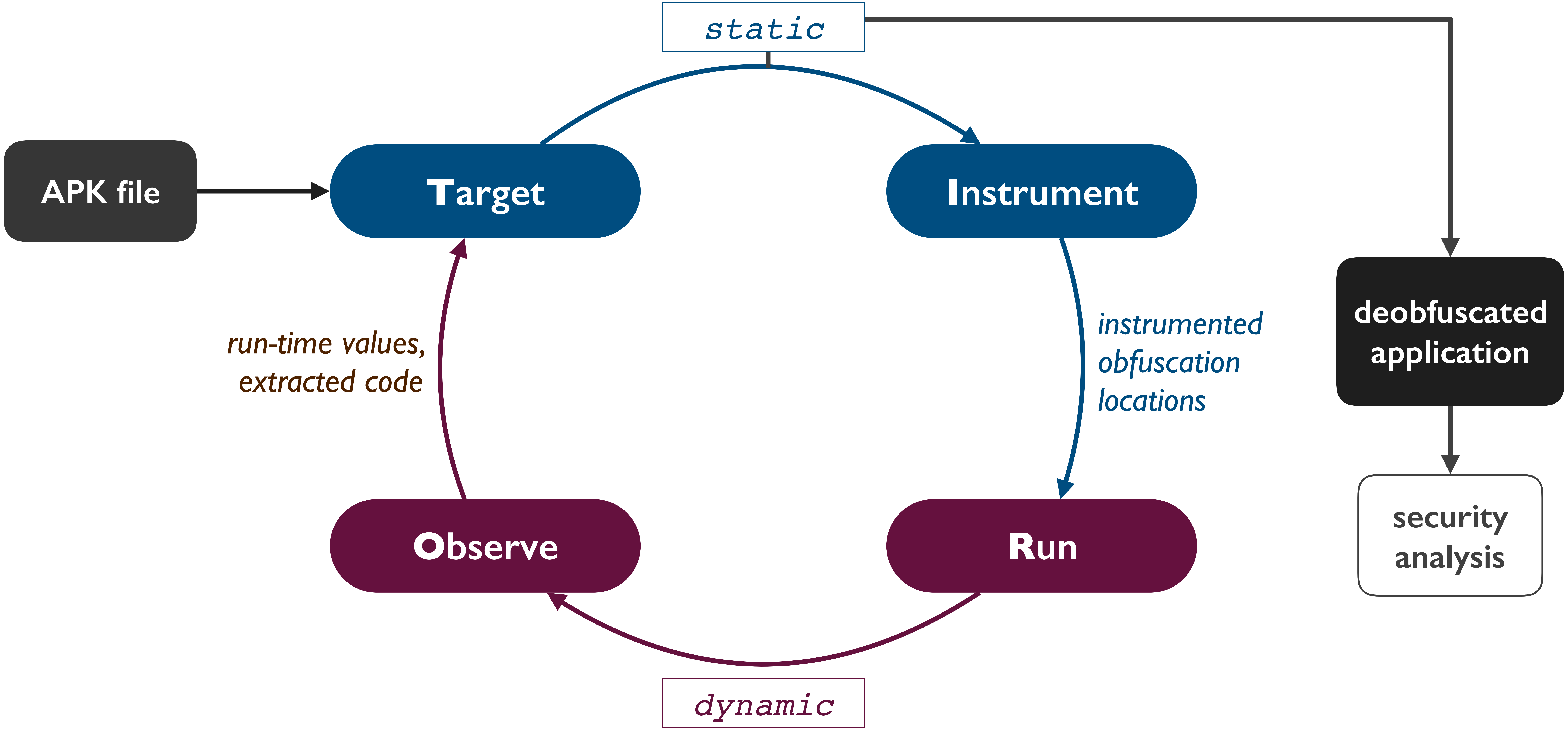


[1] Wong, M.Y., and Lie ,D. IntelliDroid: A targeted input generator for the dynamic analysis of Android malware. In *Proceedings of the Annual Symposium on Network and Distributed System Security* (NDSS), 2016.

Dealing with obfuscation



TIRO: A hybrid iterative deobfuscator



Target – Instrument – Run – Observe

Reflection

```
onCreate() {
    ...
    Method method = klass.getMethod(decrypt(“wzjg…”));
    method.invoke(receiver, args);
```

Target (Reflection)
`onCreate() → … → Method::invoke()`

Target

- Identify obfuscation locations
- Extract call paths and constraints

Target – Instrument – Run – Observe

Target → **Instrument**

- Instrument obfuscation location
- Report dynamic values and code

```
onCreate() {
    ...
  7  Method method = klass.getMethod(decrypt("wzjg…"));
  8  method.invoke(receiver, args);
```

Target (Reflection)
`onCreate() → … → Method::invoke()`

Instrument
`log(..., method.getName())`

Target – Instrument – Run – Observe

Target → **Instrument** → **Run**

- Generate inputs from targeting
- Inject inputs to run obfuscation locations

```
onCreate() {
    ...
  7  Method method = klass.getMethod(decrypt("wzjg…"));
  8  method.invoke(receiver, args);
```

Run
Log: refl,onCreate,8,"foo"

Instrument
log(..., method.getName())

Target – Instrument – Run – Observe

Target → **Instrument** → **Run** → **Observe**

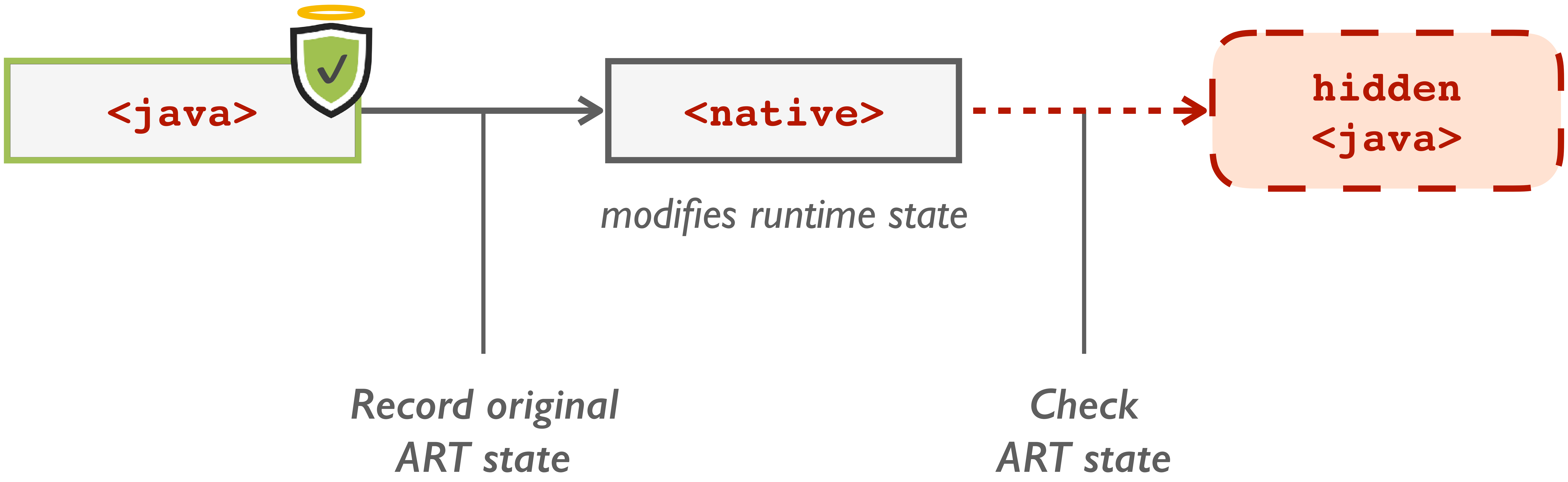
- Monitor deobfuscation log
- Extract dynamic values and code

```
onCreate() {
    ...
  7  Method method = klass.getMethod(decrypt("wzjg…"));
  8  method.invoke(receiver, args);
```

Run
Log: refl,onCreate,8,"foo"

Observe
onCreate() → foo()

Handling runtime-based obfuscation



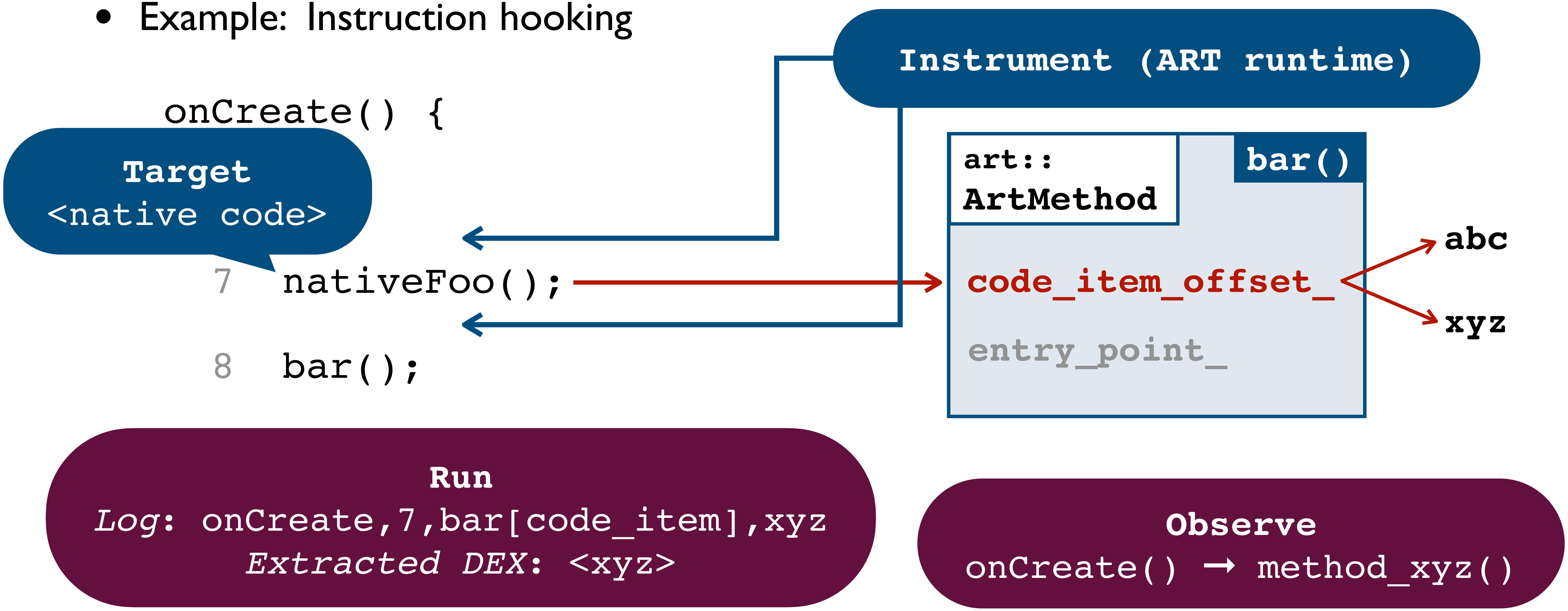
Runtime-based deobfuscation

- Example: Instruction hooking

```
onCreate() {
    ...
    nativeFoo();
    bar();
    ...
}
```

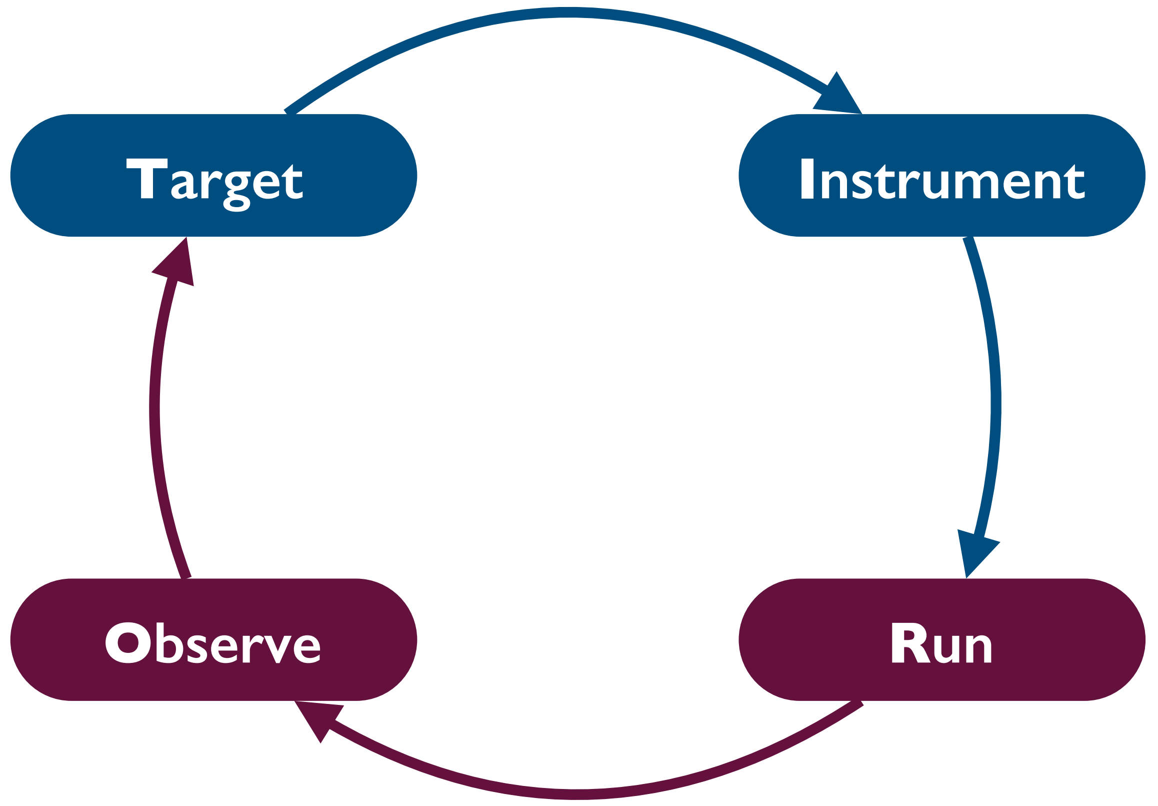
Runtime-based deobfuscation

- Example: Instruction hooking



Iterative deobfuscation

- Example: 2nd iteration




```
onCreate() {
    ...
    nativeFoo();
    bar();
    ...
}
```

```
method_xyz() {
  Method method =
      klass.getMethod(decode("vbs…"));
  method.invoke(receiver, args);
}
```

Target (Reflection) …

Implementation

- Static: Soot framework[2] for analysis and instrumentation
- Dynamic:
    - Modified AOSP with instrumented ART runtime
        - Android 4.4, 5.0, 6.0
    - Monitoring process to parse deobfuscation log and extract bytecode

[2] Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P., and Sundaresan, V. Soot - a Java bytecode optimization framework. In *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research* (1999), CASCON '99, IBM Press, p. 13.

Evaluation

- Ability to detect and deobfuscate techniques in modern Android malware
- Investigate use of language-based and runtime-based obfuscation in malware
- Deobfuscation performance (in paper)

TIRO: Detection and deobfuscation

- Labeled obfuscated samples, categorized by obfuscator/packer

	Language-based			Runtime-based					TIRO	Sensitive APIs	
	Reflection	Dynamic loading	Native methods	DEX file hooking	Class data overwriting	ArtMethod hooking	Instruction hooking	Instruction overwriting	Iterations	Before TIRO	After TIRO
aliprotect	●	●	●	●	●				3	0	44
baiduprotect	●	●	●	●	●				2	1	2
dexprotector	●	●	●						4	0	80
ijiamipacker	●	●	●	●	●	●	●	●	2	1	93
naga_pha	●	●	●	●	●	●	●	●	2	0	6
qihoopacker	●	●	●	●					2	3	217
secshell	●	●	●	●	●				2	200	287
⋮		⋮				⋮			⋮	⋮	⋮
	100%			53%					2.3	+30	

Obfuscation usage in malware

- Obfuscated malware samples from VirusTotal

Language-based		Runtime-based	
Reflection	58.5%	DEX file hooking	64.0%
Dynamic loading	79.9%	Class data overwriting	0.7%
Direct invocation	52.2%	ArtMethod hooking	0.5%
Reflected invocation	0.1%	Method entry-point hooking	0.3%
Native invocation	49.2%	Instruction hooking	33.7%
Native methods	96.8%	Instruction overwriting	0.1%

80%

Conclusion

- New category of obfuscation techniques in Android:

  runtime-based obfuscation

- TIRO: A hybrid iterative deobfuscation framework
  - Handles both language-based and runtime-based techniques
  - Deobfuscates modern malware and uncovers sensitive behaviors
- 80% of samples from VirusTotal dataset use runtime-based obfuscation