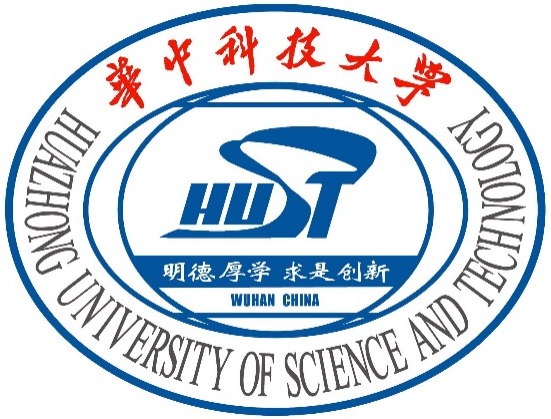
NativeSummary
Summarizing Native Binary Code for Inter-language Static Analysis of Android Apps

*Jikai Wang, Haoyu Wang**

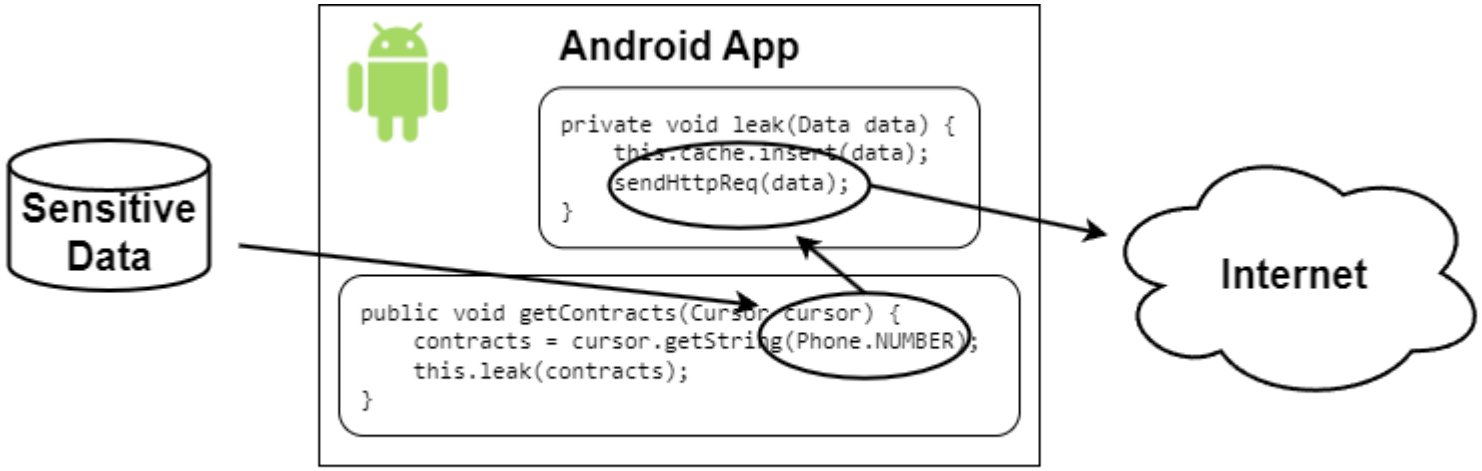
wangjikai@hust.edu.cn,

Huazhong University of Science and Technology



Android App Dataflow Analysis

- Android apps are everywhere
  - Malware Detection, Bug Finding
- Taint dataflow analysis is widely used
  - Tools: FlowDroid, IccTA, DroidSafe, AmanDroid……

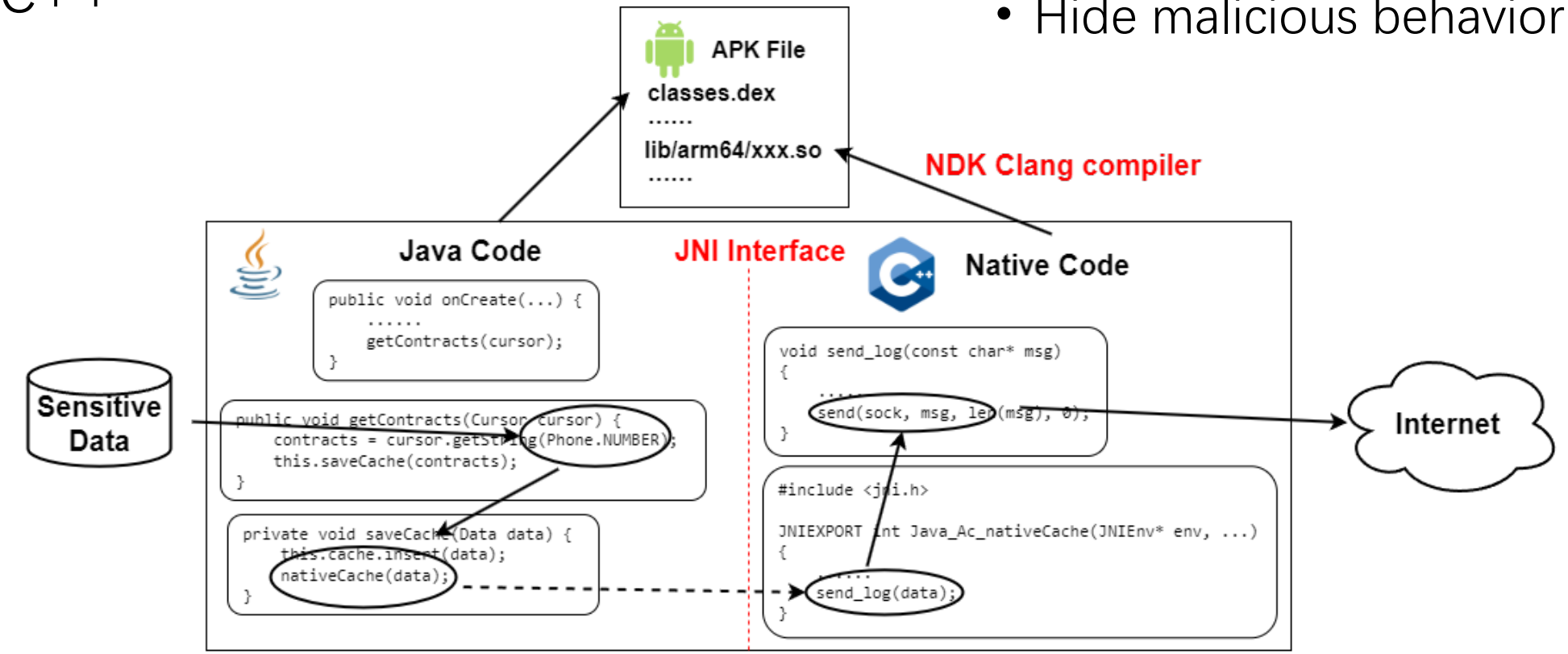


Sensitive data is leaked

Native Code in Android Apps

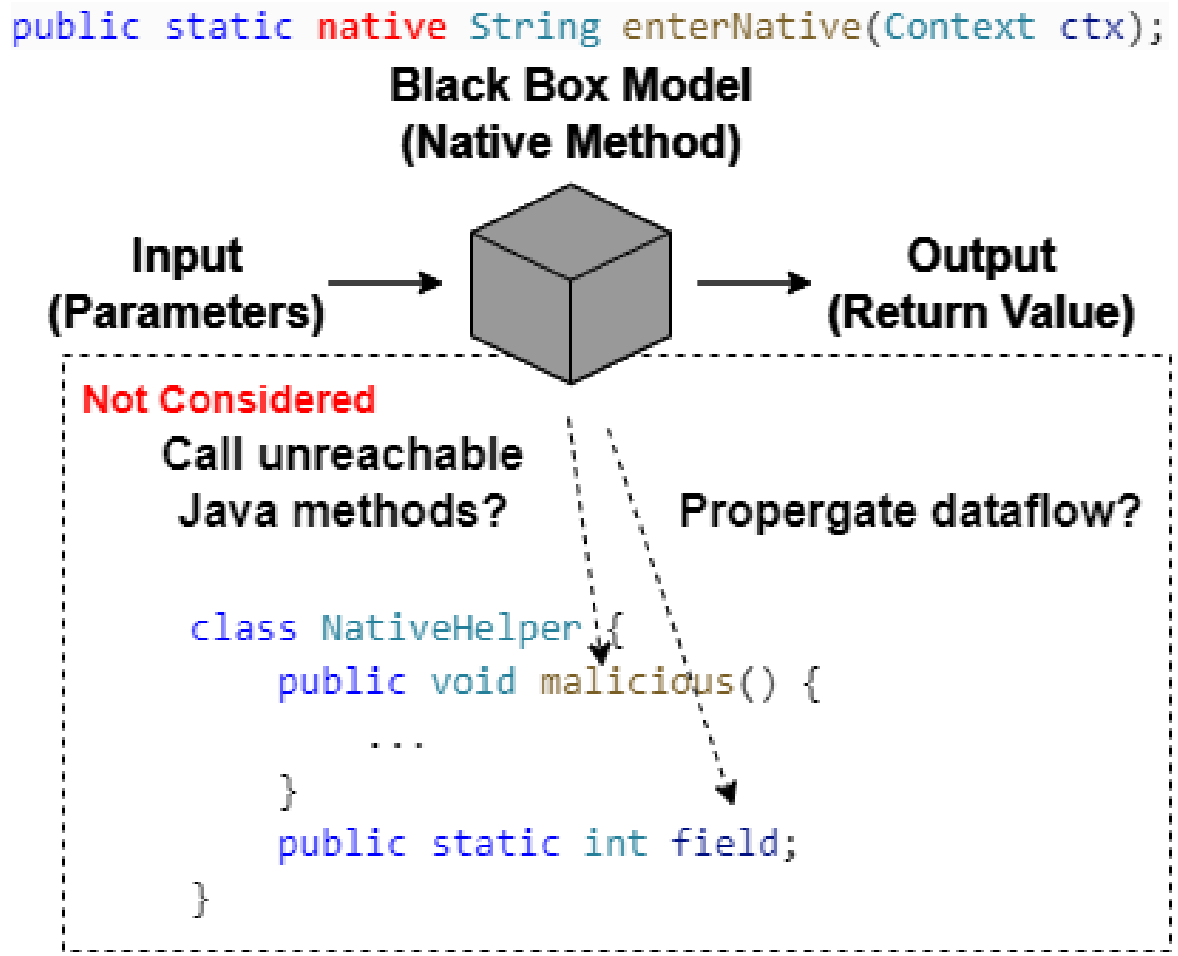
- NDK & JNI enables Android apps to use C/C++

- Native code is widely used!
  - Prevalent in over 60% of Android apps
  - Hide malicious behavior in native code



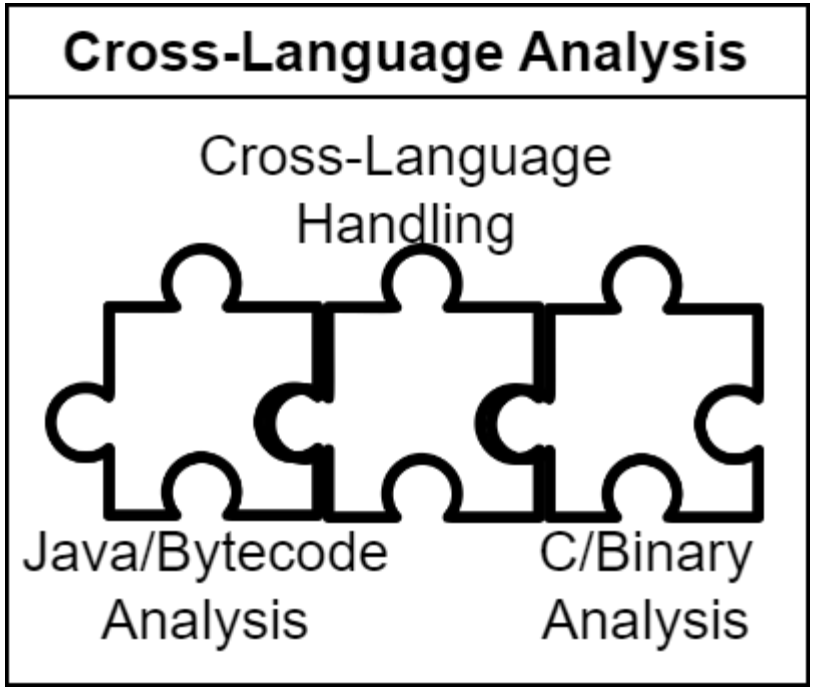
Cross-Language Analysis Needs to Improve

- Black box model
  - Incomplete call graph
  - Incorrect data flows
- Boost many existing analyses
  - App clone detection
  - Malicious behavior detection



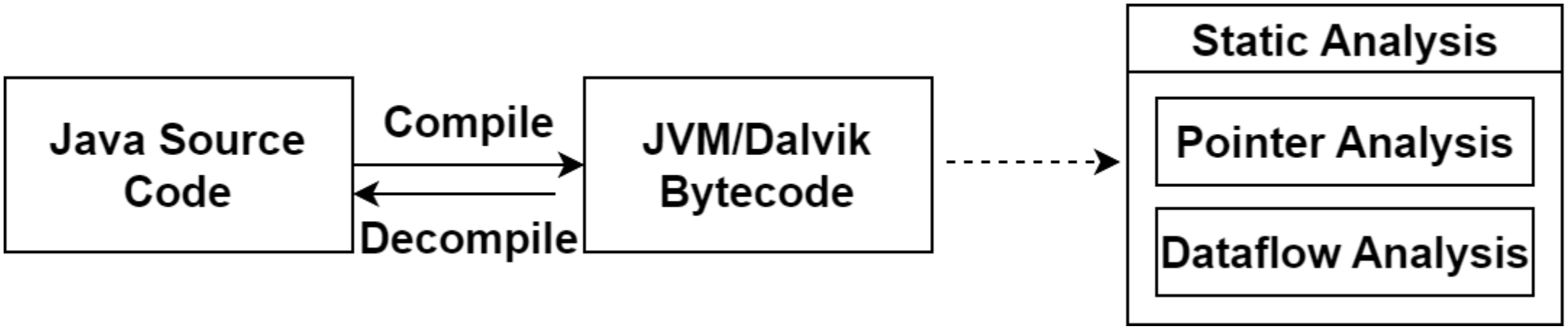
Towards a "Perfect" Cross-Language Analysis

1. Good Java (bytecode) analysis: ✓
2. Good C (binary code) analysis: **?**
3. Good interlanguage handling: **?**



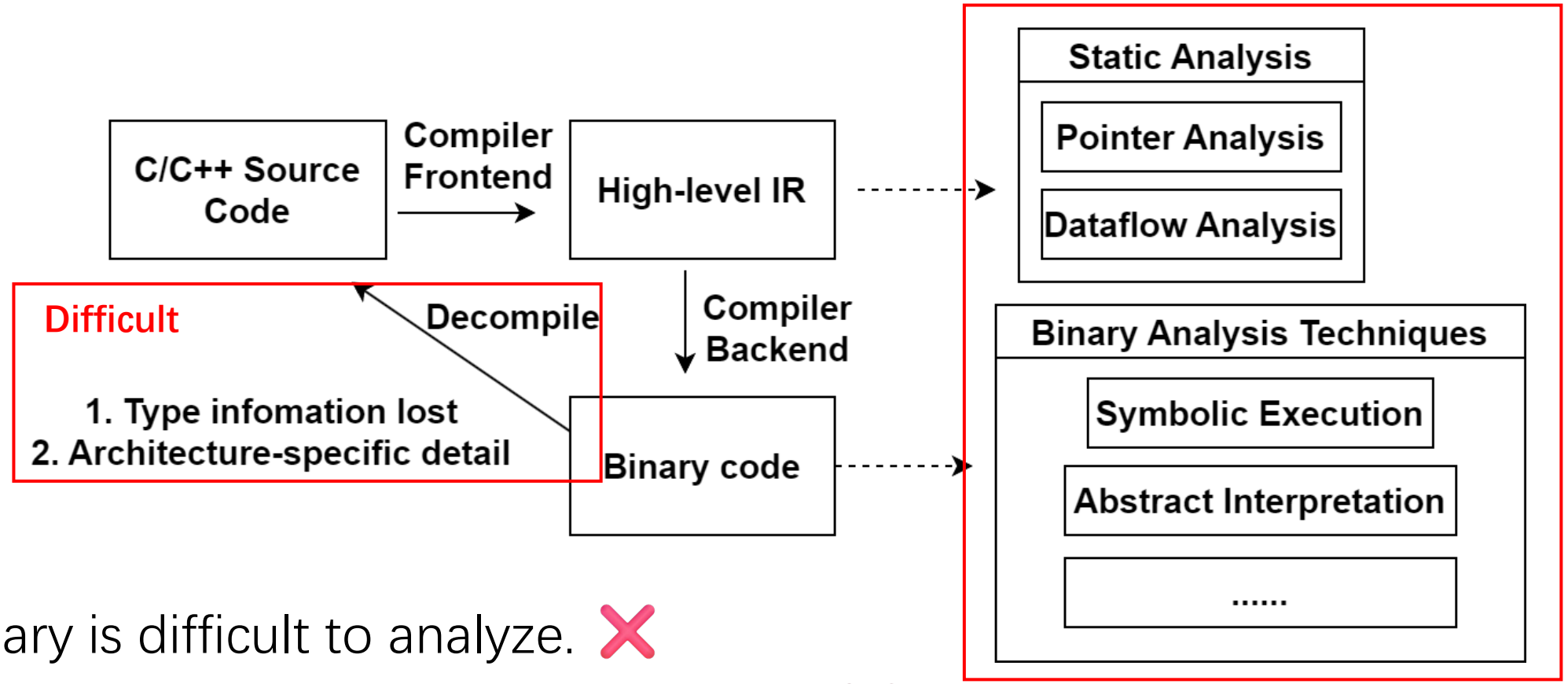
- Existing approaches are far from being perfect.

1. Good Java (bytecode) analysis



- Analysis-friendly bytecode ✅
  - Enables Large-scale analysis.
- Scalable distributive static analysis algorithm (IFDS/IDE) ✅

2. Good C (binary code) analysis? No



- Binary is difficult to analyze. ❌
- Different and Less scalable techniques ❌
- Better Way: Improve binary decompilation result and use source-level analysis

3. Good Interlanguage Handling? No

- "Heuristically" link missing call graph and dataflow ❌
  - Corner case:

Native function1: ContextInit

```
return (jlong) malloc(size);
```

Native function2: ContextUser

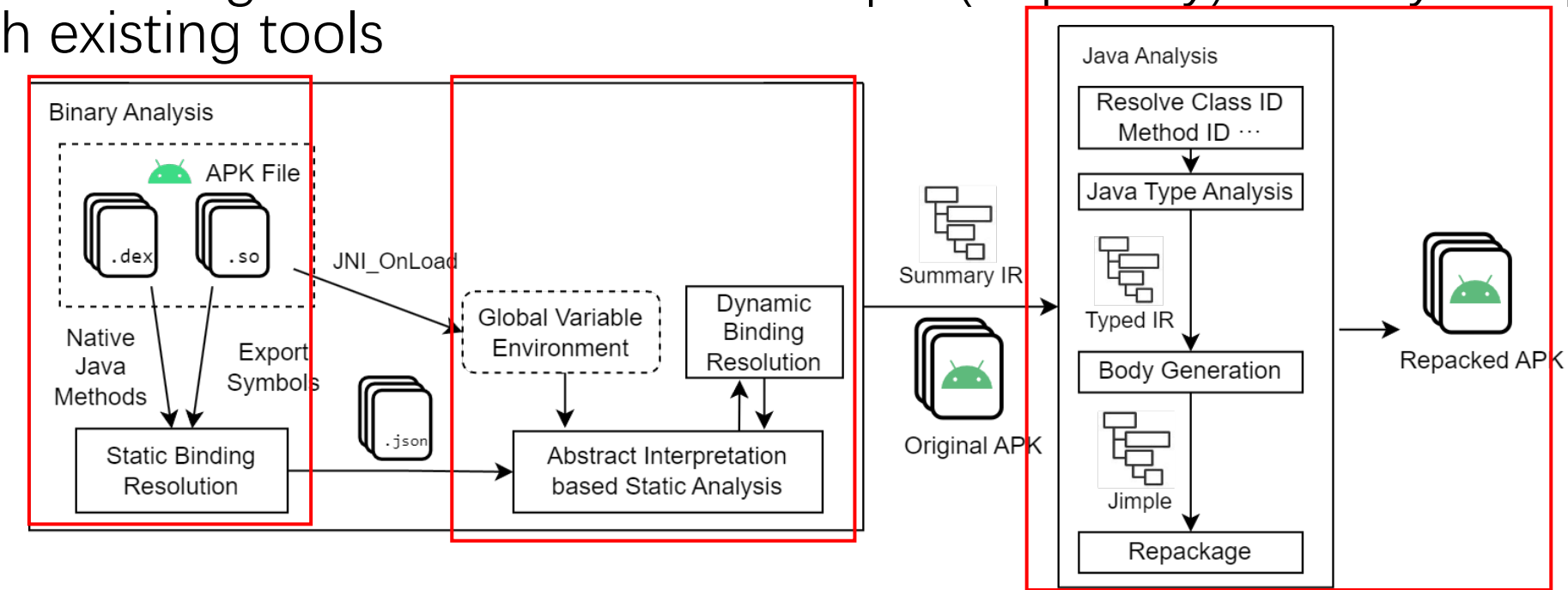
```
Arg: jlong handle
char* buffer = (char*) handle;
```

- Better Way: Use a new memory model* and design a new algorithm

Tan, Gang. "JNI Light: An operational model for the core JNI." *Asian Symposium on Programming Languages and Systems*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010.

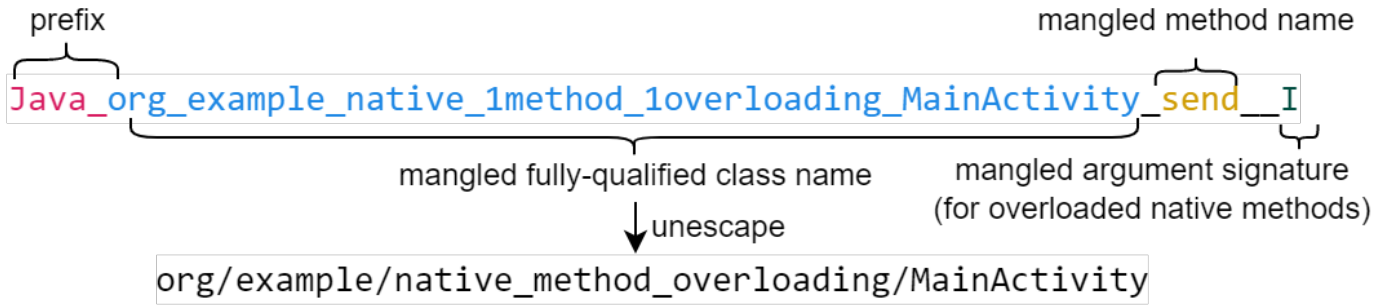
Overview of Our Approach

- Try to link dataflows by generating a function body for native functions.
  1. **Find the entry points**: Binding resolution
  2. **Get native dataflows**: Binary Analysis. Output Summary IR
  3. **Link the dataflows**: Convert Summary IR to a function body
- Repack the generated code to new apk. (hopefully) Directly compatible with existing tools



Phase 1: Find native entry points

- JNI (Java Native Interface) Registration: Find the entry points of native code
  - Direct binding using export symbols
    - Resolve according to the specifications

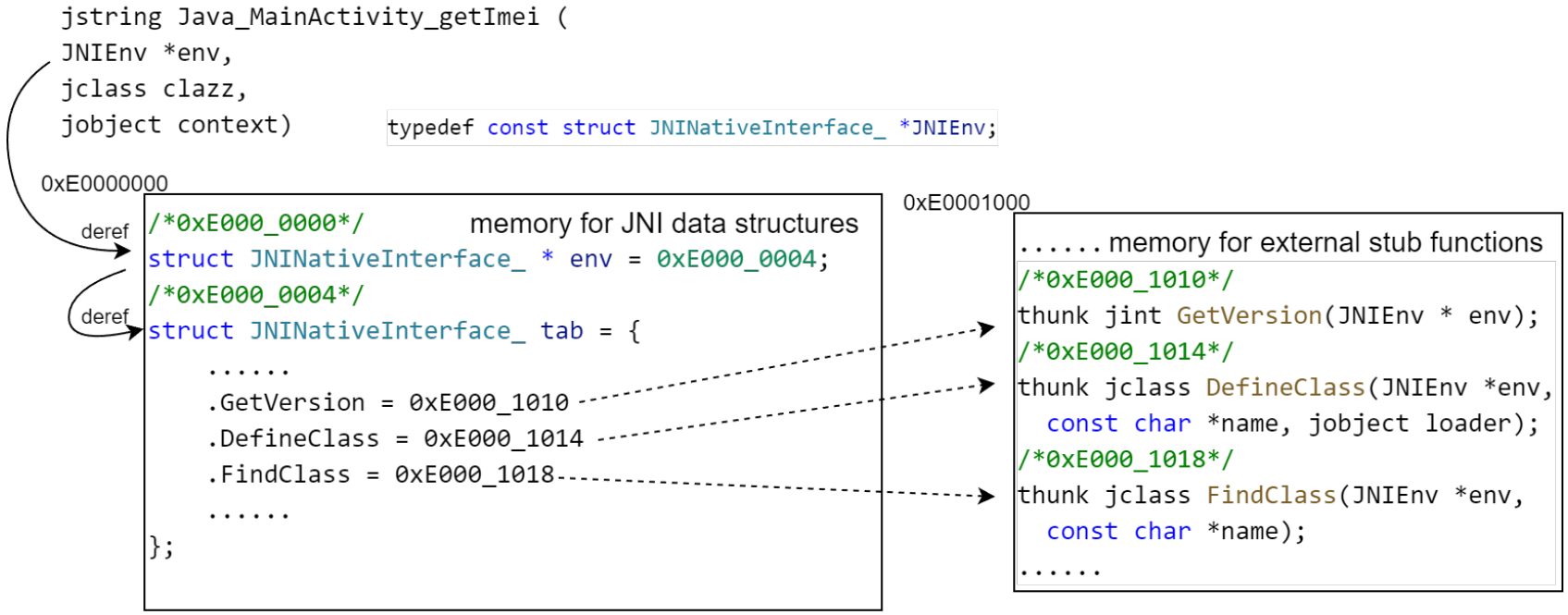


  - Dynamic registration using JNI API.
    - Use Binary analysis to find the dataflow

```
jint JNI_OnLoad(JavaVM *vm, void *reserved) {
  env = ...; // omit version check and GetEnv call
  clazz = env->FindClass(className);
  env->RegisterNatives(clazz, gMethods, numMethods);
}
static JNINativeMethod gMethods[] = {
 {"send",    "(Ljava/lang/String;)V",  (void *) native_send},
 {"sendFoo", "(ILjava/lang/String;)V", (void *) native_sendFoo},
 {"sendBar", "(DLjava/lang/String;)V", (void *) native_sendBar},
};
```

Phase 2: Get the native dataflow

- Dataflow analyzer: Abstract-Interpretation over machine code
  - Handle complex assembly code features like callee-saved regs
- Prepare JNIEnv / Prepare models for JNI Functions



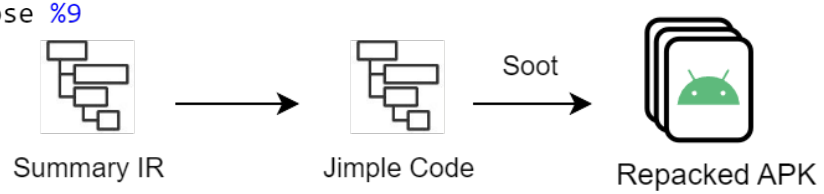
https://github.com/KeenSecurityLab/BinAbsInspector

Phase 3: Java Code Generation & APK Repacking

- Convert native-to-java calls
- Handle native-specific functions: create new native methods.
  - Try to infer function signature according its usage.
- Generate java bytecode using Soot. Rewrite dex files using dexlib2.

Generated Summary IR

```
I1  define jstring @MainActivity.enterNative
I2  (JNIEnv *env, jclass clazz, jobject a1){
I3    ...
I4    %7 = Call CallObjectMethod null, %6, %5 // getImei()
I5    %8 = Call GetStringUTFChars null, %7, long 0
I6    %9 = Call open char* "/mnt/sdcard/info.txt", long 65, long 418
I7    %10 = Call write %9, %8, long 20
I8    %11 = Call close %9
I9    ret %7
I10 }
```



Generated Java Bytecode

```
G1  public class NativeSummaryFuncs {
G2    public static native int open(String s,long j,long j2);
G3    public static native int write(int i,String s,long j);
G4  }
G5  public class MainActivity extends Activity {
G6    ... // omit other methods
G7    public static String enterNative(Context ctx) {
G8      String s = ctx.getSystemService("phone").getImei();
G9      int $1 = NativeSummaryFuncs.open("/sdcard/info.txt",65,418);
G10     NativeSummaryFuncs.write($1, s, 20); // SINK
G11     return s;
G12   }
G13 }
```

Evaluation - Hand-Crafted Benchmarks

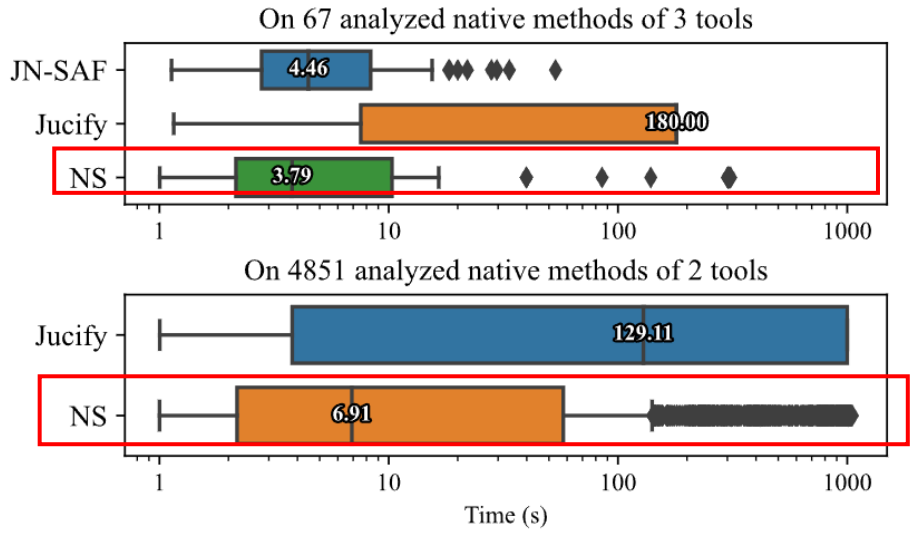
- Repackage can support different tools
- 16% Higher precision on all hand-crafted benchmarks
  - Handle more complex cases from real-world

	NS+FD	NS+AS	NS+AM	JS	JU
Sum and Precision					
◯, higher is better	35	34	30	22	9
×, lower is better	8	8	13	21	34
Percentage p = ◯/(◯+×)	81.4%	81.0%	69.8%	51.2%	20.9%

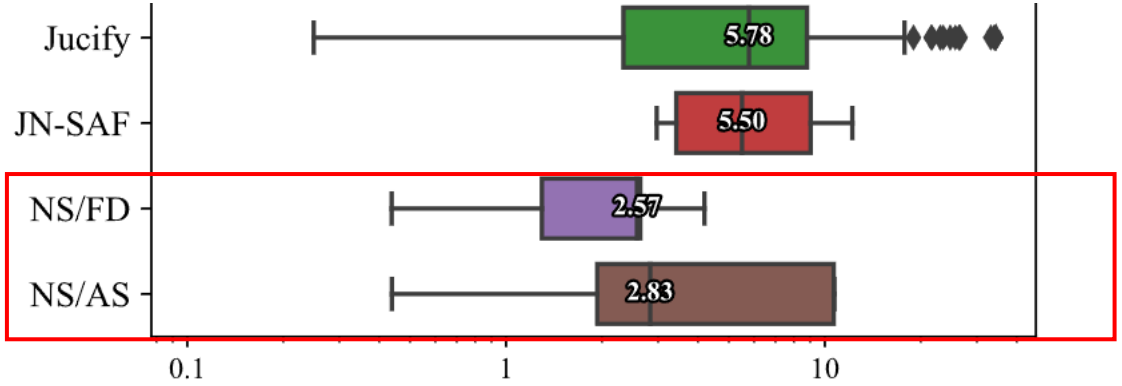
FD = FlowDroid, AS = AppShark, AM = Amandroid, JS = JN-SAF, JU = Jucify, NS = NativeSummary. Some apps contain two leaks.

Evaluation – Real-World Apps

- On average 15% faster, 50% less memory consumption than SOTA



Time Usage: Lower is better



Memory Usage: Lower is better

Evaluation – Real-World Apps

- Analyze much more native methods, discover more native-to-java edges
  - The only tool that is ready for real-world app analysis

Tools	#Total Apps	#Success App	#Total Flows	#Native Methods Analyzed	#Native Methods Succeeded	#Native Edges Created	#Native Related Flows
				In 271 Apps (Intersection of Successful Apps) / In All Apps			
NS+FD	2255	1360	1203775	8521 / 60227	6690 / 44993	247 / 7111	187 / 2239
NS+AS		1567	2483215				425 / 4572
JN-SAF		744	3832	43 / 212	42/177	5 / 85	6 / 26
Jucify		1111	718687	850 / 20462	0 / 2303	0 / 130	2 / 11

FD = FlowDroid, AS = AppShark, NS = NativeSummary

More is better

Takeaway

- What is the problem?
  - Dataflows/Call edges from native code is not considered
  - Existing approaches lack scalability, compatibility and precision
- How does it perform?
  - Compatible to existing bytecode analysis tools
  - Better robustness, better precision, better scalability
- What we do?
  - Inter-language static analysis by linking missing call edges and dataflows in the native code of android apps
  - Repackage to a new APK to maximize compatibility
- More Questions
  - warrenwjk@gmail.com
  - Fully open source:

github.com/security-pride/NativeSummary