JNFuzz-Droid: A Lightweight Fuzzing and Taint Analysis Framework for Android Native Code

Jianchao Cao, **Fan Guo**, Yanwen Qu

School of Computer Information and Engineering
Jiangxi Normal University

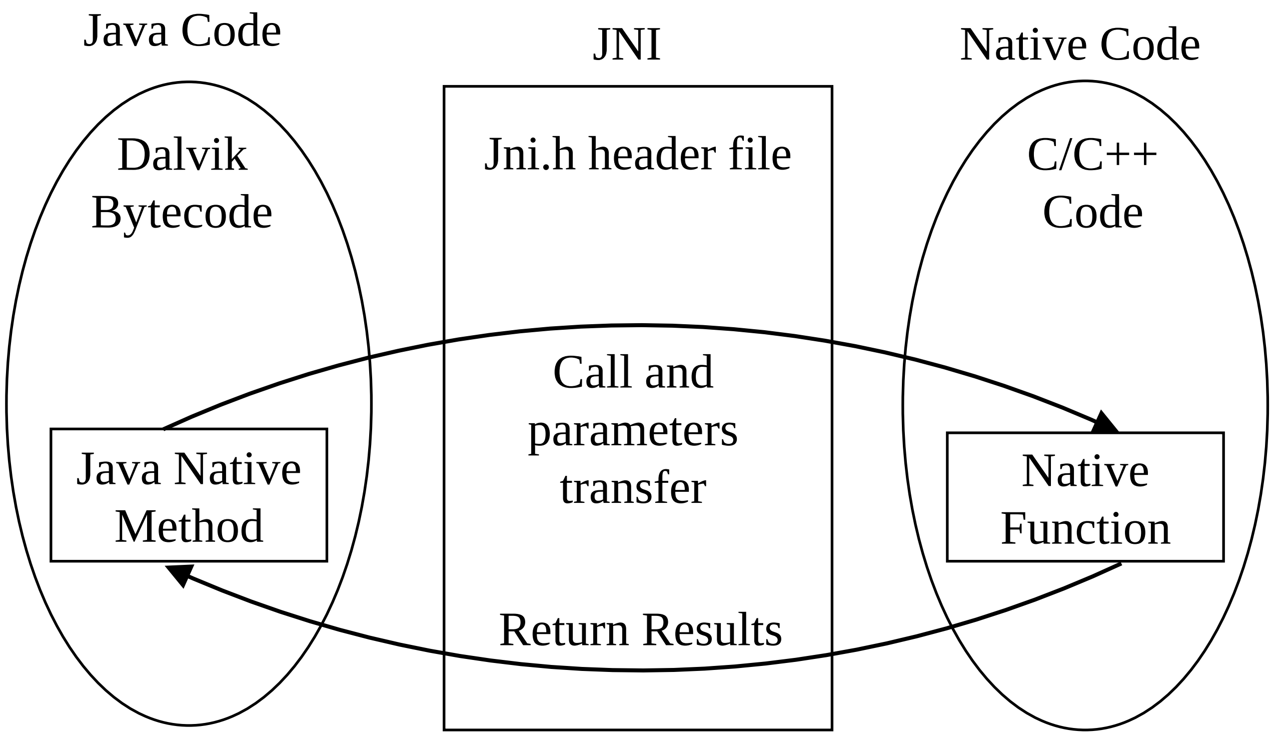
Outline

- Research Background: Advantage and disadvantage of native code
- Limitation of Current Techniques
- Our Approach: JNFuzz-Droid
- Experiments and Results

The **Native Development Kit (NDK)** is a set of tools that allows you to develop native code in Android applications using C and C++ code. The communication and interaction between native code and Java code is implemented through the **Java Native Interface (JNI)** .

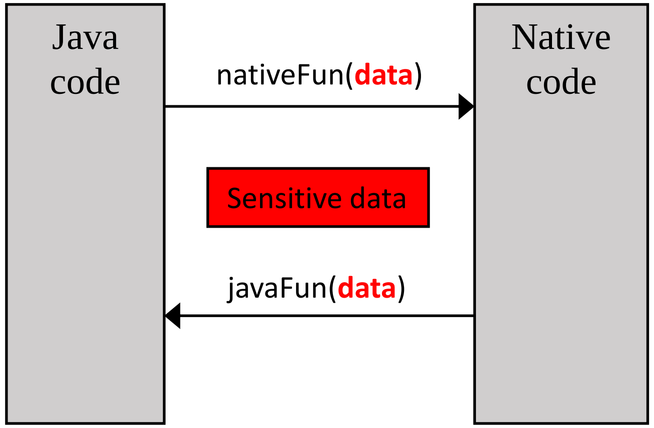
Benefits of using native code:

- Improving program performance
- Reusing existing C or C++ libraries
- Increasing complexity in decompilation
- …

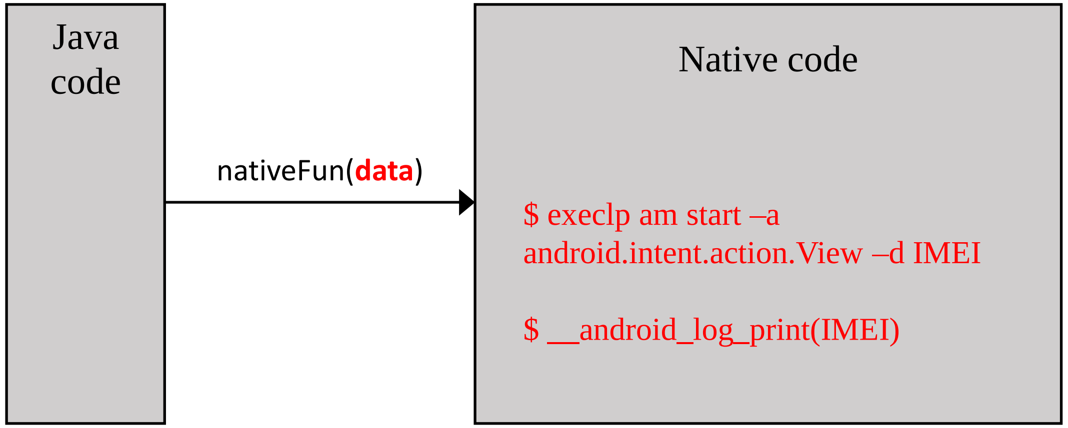


Potential Threat: Malicious App Developers Exploiting Native Code

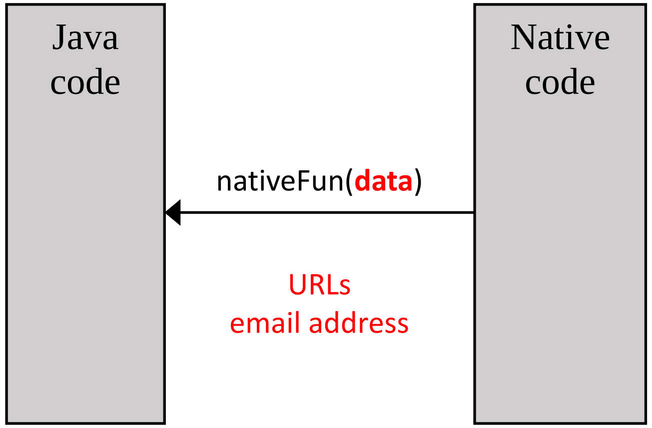
Inter-language Data Leakage



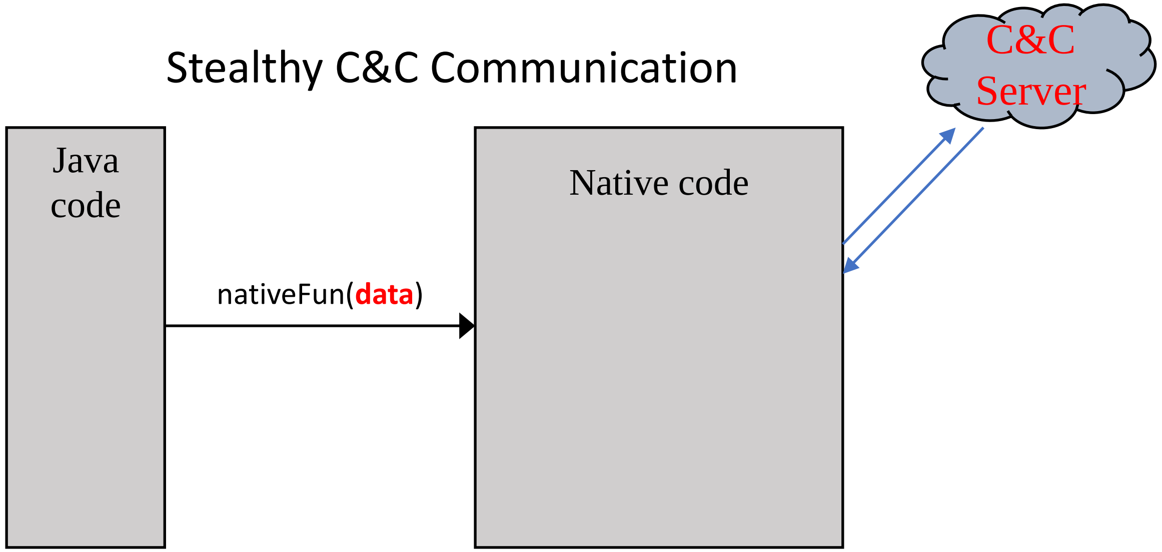
Stealthy Command Execution



Malicious Identity Hiding



Stealthy C&C Communication



Current State of Native Code Analysis Techniques

- Static Taint Analysis [JN-SAF, JuCify]
  - ➢ Using symbolic execution to build CFG/CG to detect leakage, but
  - ➢ Limit of path expolsion, SMT solving, unmodeled semantics, false positive, and so on
- Dynamic Taint Analysis [Malton, NDroid]
  - ➢ Using dynamic instrumentation tools to track the flow of sensitive data, very accurately, however
  - ➢ Limit of code coverage, hard to trigger native code (Human-based; Monkey)
- Fuzzing Testing [AFL++, LibFuzzer]
  - ➢ Using many random input to improve code coverage, while
  - ➢ Lack of consideration of JNIEnv* environment, and it mainly used for software defect dection, rather than taint analysis

A Motivation Example

```java
/*** Java World ***/
public class MainActivity extends AppCompatActivity {
  protected void onCreate(Bundle savedInstanceState) {
    TelephonyManager tel = (TelephonyManager) getSystemService
(TELEPHONY_SERVICE);
    String imei = tel.getDeviceId(); // Source
    ...
  }
}
```

```java
/*** Java World ***/
public class utilActivity extends AppCompatActivity {
  static {
   System.loadLibrary("native-lib");
  }
  public static native void fun1(String obj, int length);
  protected void onCreate(Bundle savedInstanceState) {
    ...
    Button button = (Button) findViewById(R.id.register);
    button.setOnClickListener(new View.OnClickListener() {
      public void onClick(View arg0) {
        TextView tv = findViewById(R.id.password);
        fun1(imei, tv.getText().length());
      }
    });
  }
}
```

native-lib.cpp $\xrightarrow{\text{compile to}}$ libnative-lib.so

```cpp
/*** C/C++ World ***/
C1.  void leaksensitive(const char *mess, int len) {
C2.    if (sqrt(5 + len) >= 4) {
C3.      LOGE("%s", mess); //Sink
C4.    }
C5.  }
     //Static register
C6.  extern "C" JNIEXPORT void JNICALL
C7.  Java_com_test_example_utilActivity_fun1(JNIEnv *env, jclass clazz,
C8.  jstring data, jint length) {
C9.    // TODO: implement fun1()
C10.   const char *mess = env->GetStringUTFChars(data, 0);
C11.   leaksensitive(mess, length);
C12.   env->ReleaseStringUTFChars(data, mess);
C13.   return;
C14. }
```

Limitation of current techniques

- Dynamic analysis hard to automatically identify and satisfy the execution requirements of native methods.
- Static analysis explore path via symbolic execution, but cannot store the return value of the sqrt function (in line C2), causing the analysis interrupt.
- Fuzzing techniques cannot fuzz native functions, and it used for software defect detection, rather than taint anlaysis.

A Motivation Example

```java
/*** Java World ***/
public class MainActivity extends AppCompatActivity {
  protected void onCreate(Bundle savedInstanceState) {
    TelephonyManager tel = (TelephonyManager) getSystemService
(TELEPHONY_SERVICE);
    String imei = tel.getDeviceId(); // Source
    ...
  }
}
```

```java
/*** Java World ***/
public class utilActivity extends AppCompatActivity {
  static {
   System.loadLibrary("native-lib");
  }
  public static native void fun1(String obj, int length);
  protected void onCreate(Bundle savedInstanceState) {
    ...
    Button button = (Button) findViewById(R.id.register);
    button.setOnClickListener(new View.OnClickListener() {
      public void onClick(View arg0) {
        TextView tv = findViewById(R.id.password);
        fun1(imei, tv.getText().length());
      }
    });
  }
}
```

native-lib.cpp $\xrightarrow{\text{compile to}}$ libnative-lib.so

```cpp
/*** C/C++ World ***/
void leaksensitive(const char *mess, int len) {
  if (sqrt(5 + len) >= 4) {
    LOGE("%s", mess); //Sink
  }
}
//Static register
extern "C" JNIEXPORT void JNICALL
Java_com_test_example_utilActivity_fun1(JNIEnv *env, jclass clazz,
jstring data, jint length) {
  // TODO: implement fun1()
  const char *mess = env->GetStringUTFChars(data, 0);
  leaksensitive(mess, length);
  env->ReleaseStringUTFChars(data, mess);
  return;
}
```

Our Approach

1. Locate the target native method by static taint analysis.
2. Obtain the native functions corresponding to the native method.
3. Fuzz the native function to explore path.
4. Track the sensitive data flow in the native code.

Core Challenge

- How to fuzz the native function?
- How to track the sensitive data flow in the native code?

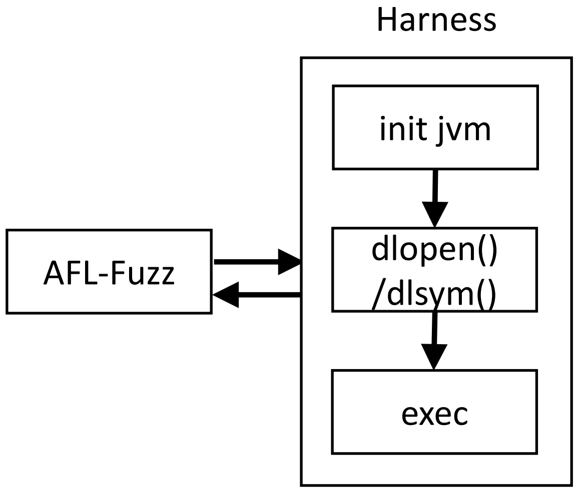
Fuzz the native functions

*/*** C/C++ code***/*

void JNICALL Java_com_test_example_utilActivity_fun1(JNIEnv *env, jclass clazz, jstring data, jint length)

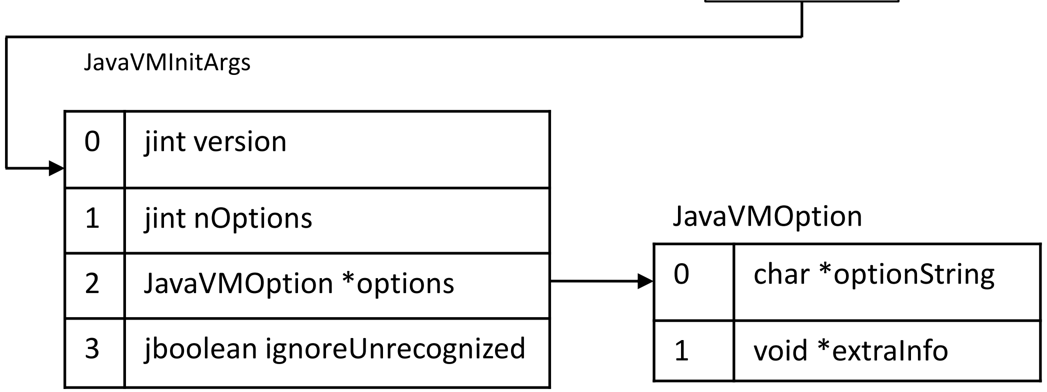
*/*** Java code***/*

static native void fun1(string obj, int length);



Obatin JNIEnv* pointer type (init JVM)

jint JNI_CreateJavaVM(JavaVM **p_vm, void **p_env, void *vm_args);



Obatin other JNI type

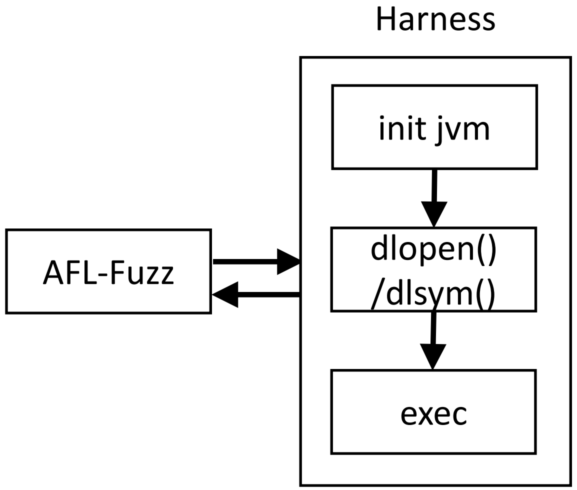
Java Type	Native Type
boolean	jboolean
byte	jbyte
char	jchar
short	jshort
int	jint
long	jlong
float	jfloat
double	jdouble
void	void

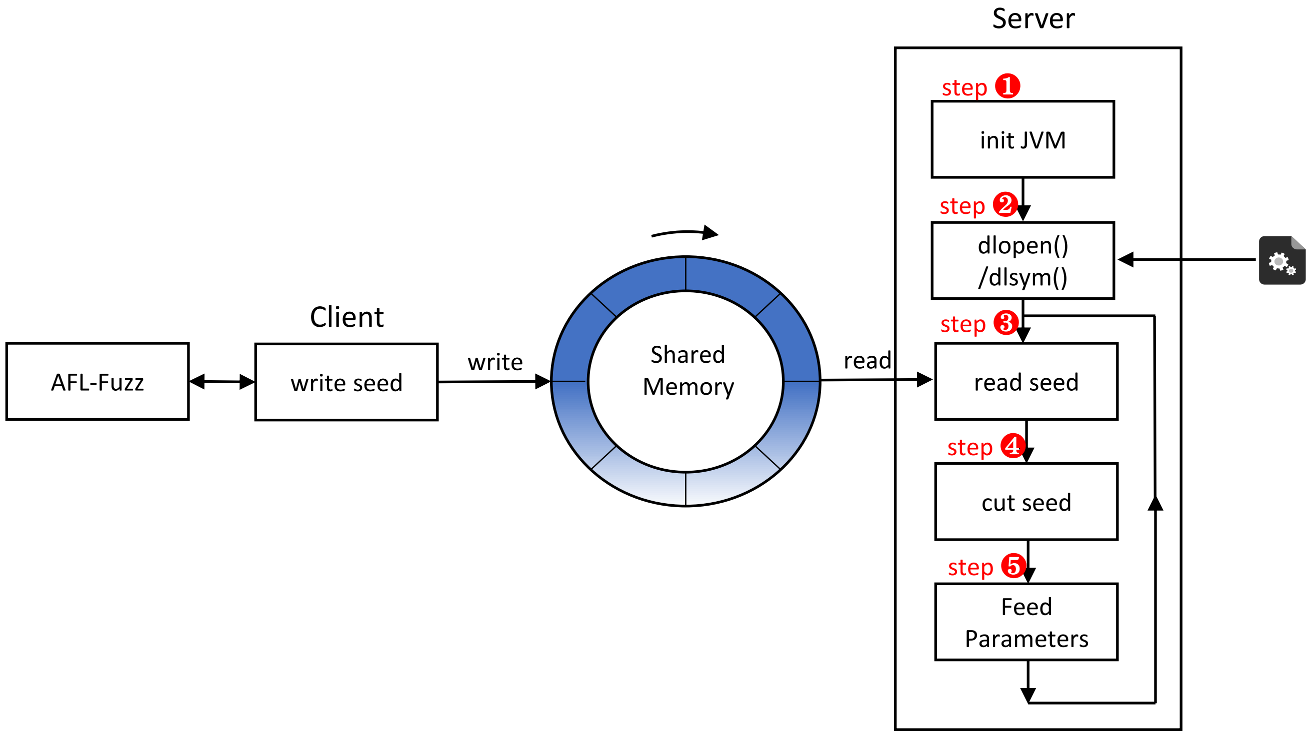
Primitive Types

- jobject (all Java objects)
  - jclass (java.lang.Class objects)
  - jstring (java.lang.String objects)
  - jarray (arrays)
    - jobjectArray (object arrays)
    - jbooleanArray (boolean arrays)
    - jbyteArray (byte arrays)
    - jcharArray (char arrays)
    - jshortArray (short arrays)
    - jintArray (int arrays)
    - jlongArray (long arrays)
    - jfloatArray (float arrays)
    - jdoubleArray (double arrays)
  - jthrowable (java.lang.Throwable objects)

Reference Types

Fuzz the native functions





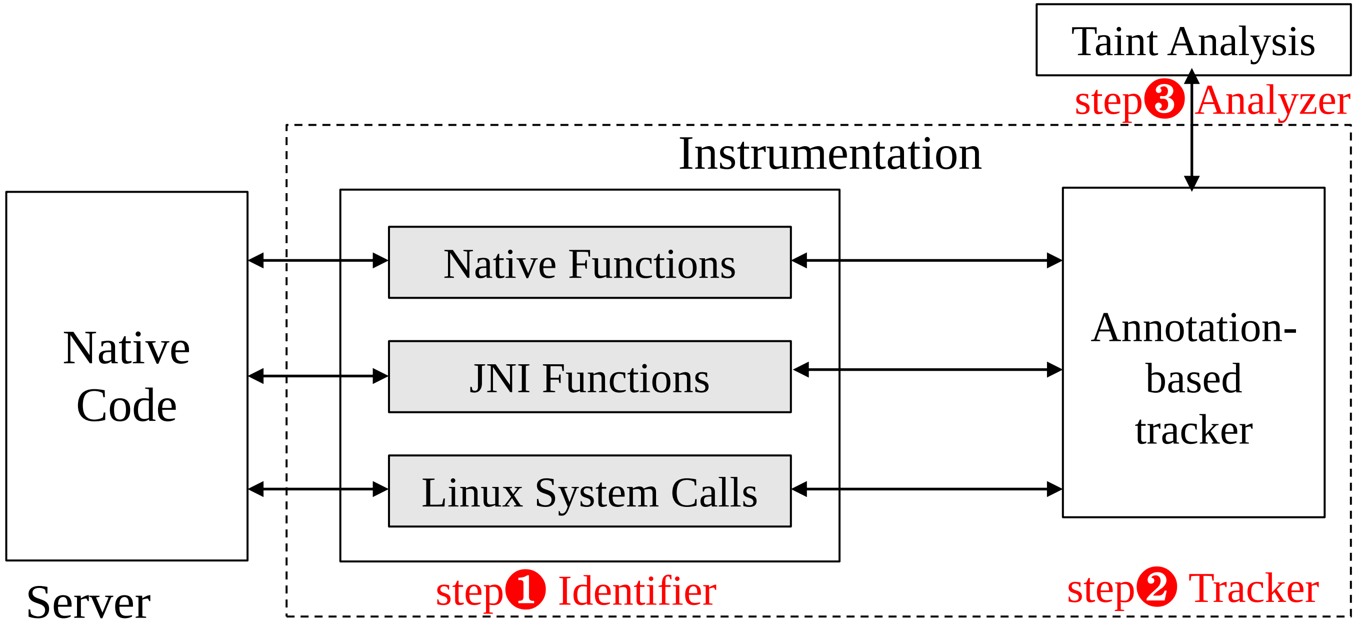
shortcoming:

- initialization JVM is time-consuming
- Determining the address of the native function via dlopen/dlsym is time-consuming
- Dynamic taint analysis for track the sensitive data flow is time-consuming

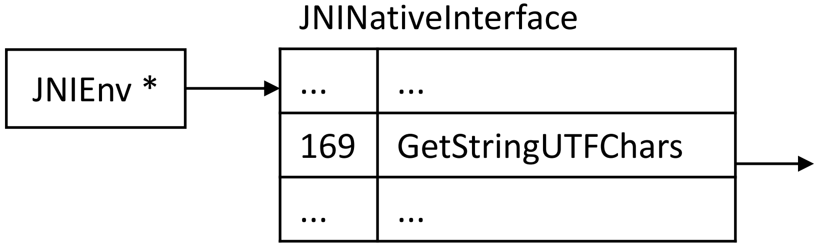
benefit

- ➢ initialization JVM, Determing the address of the native function, Dynamic taint analysis only once
- ➢ 0.75 exec/sec V.S. 148.2 exec/sec 197 times

Track sensitive data flow in native code



- Identify and recognize the Native functions、JNI Function、Linux System Calls
- Design custom annotation to implement the data flow tracker
- Analyze the (taint) data flow



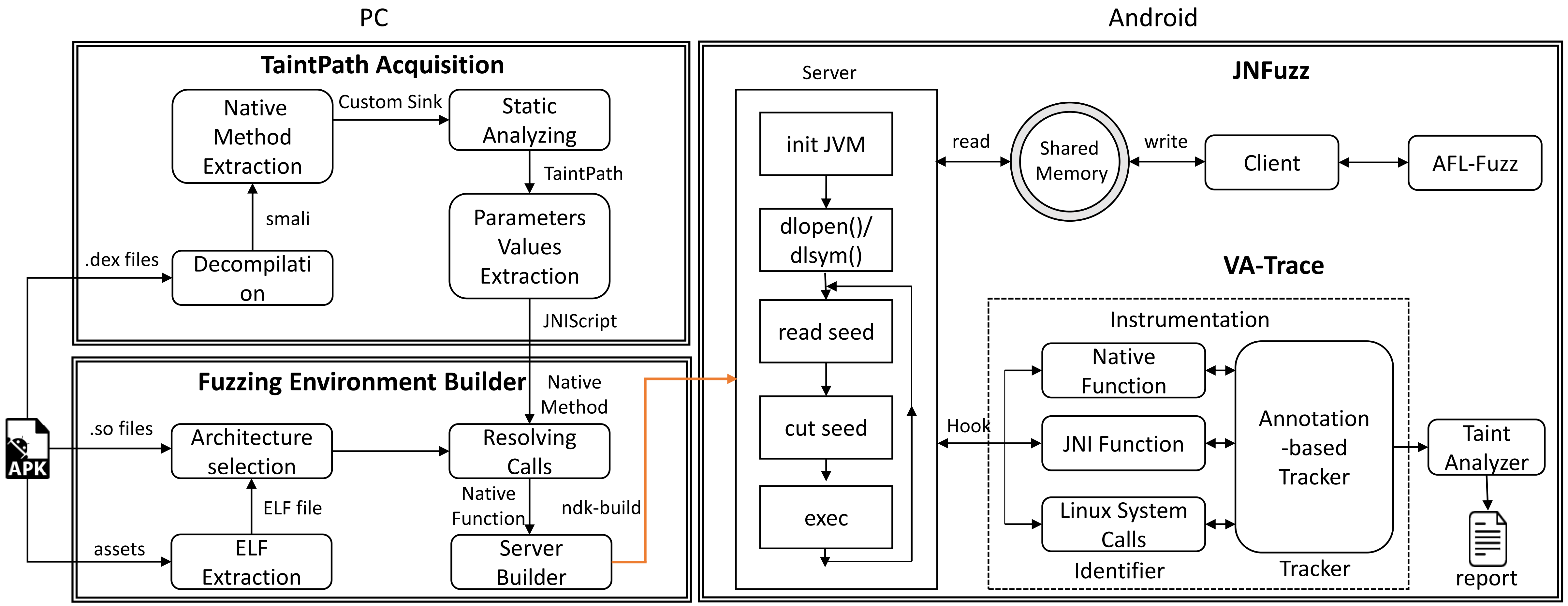

```
Intercepter.attach(JNIEnv* + 0x2A4){
        TaintAnnotation
        Value: arg1.value() -> ret.value();
        Addr: arg1 -> ret;
}
```

Track sensitive data flow in native code

List of Library Calls for Taint Propagation

<table>
<tr><th>Category</th><th>Library Calls</th></tr>
<tr><td>JNINativeInterface</td><td>**Call<type>Method(v)***, FindClass, ...</td></tr>
<tr><td>Print & Log</td><td>**__android_log_print***, **(f|s|sn|vsn|vs|vf)printf***, ...</td></tr>
<tr><td>Network</td><td>gethostbyname, recv(from), connect, **send(to)***, listen,
socket, bind, accept, select, **write***, ...</td></tr>
<tr><td>File Operation</td><td>**(f)write***, (f)open, (f)read, **fput(c|s)***, fdopen</td></tr>
<tr><td>Dynamic Loading</td><td>dl(m)open, dl(v)sym</td></tr>
<tr><td>Monitoring</td><td>inotify_init, inotify_add_watch, inotify_rm_watch</td></tr>
<tr><td>Execution</td><td>**execcl(p|e)***, **execv(p|e)***, system, popen</td></tr>
<tr><td>Process Management</td><td>kill, ptrace, fork, get(e)uid, get(e)gid, get(p)pid, ...</td></tr>
<tr><td>Time</td><td>clock_gettime, gettimeofday, ...</td></tr>
<tr><td>File Permission</td><td>(f|l)chmod, (f|l)chown, access</td></tr>
<tr><td>Other</td><td>strlen, str(n|case|ncase)cmp, str(n)cpy, strstr, str(r)chr,
strcat, strdup, strto(u)l, strtod, memcpy, malloc, memset,
memcmp, memchr, memmove, atoi, atol, atoll</td></tr>
</table>

JNFuzz-Droid workflow



Experiments and Results

RQ1: What is the current state of native library usage in real-world apps?

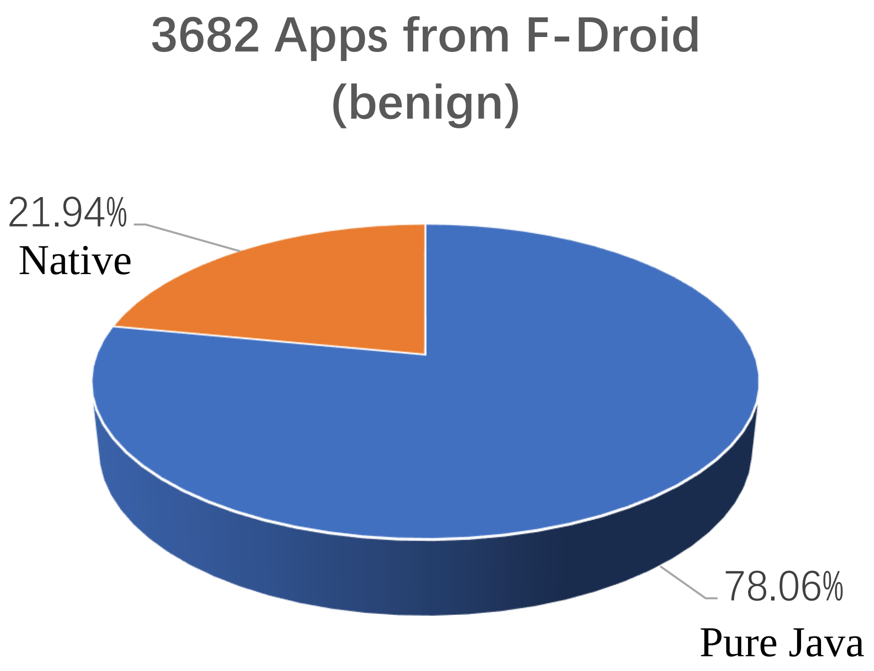
RQ2: How does JNFuzz-Droid perform on benchmark apps?

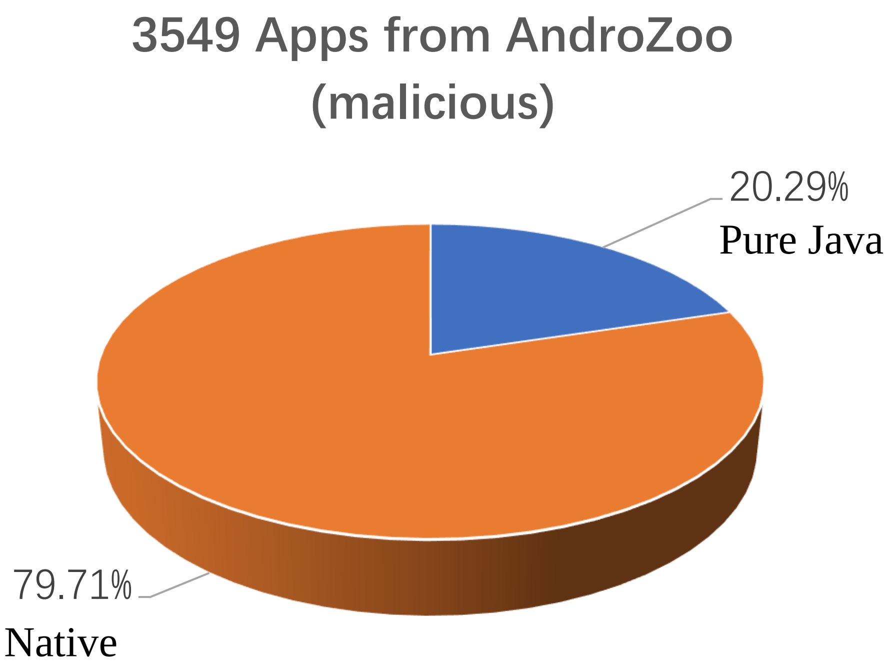
RQ3: How does JNFuzz-Droid perform in real-world apps?

Android Native library Statistics

Dataset:

- The 3682 Android apps from **F-Droid** as a sample of **benign** apps
- The 3549 Android apps from **AndroZoo** as a sample of **malicious** apps





Dataset:

- The 3682 Android apps from **F-Droid** as a sample of **benign** apps
- The 3549 Android apps from **AndroZoo** as a sample of **malicious** apps

F-Droid

1. Has Native Method: 21.9%
2. Has .so File: 21.7%
3. Has ELF in asset: 1.1%
4. Has Encrypted zip: -
5. Has Native Activity: 0.1%
6. Total Native Methods: 127.3k

   157.5/native_method_app

AndroZoo

1. Has Native Method: 79.7%
2. Has .so File: 23.3%
3. Has ELF in asset: 21.7%
4. Has Encrypted zip: 0.5%
5. Has Native Activity: -
6. Total Native Methods: 330k

   116.8/native_method_app

Native-code Benchmark Test

Covering:
- NativeFlowBench
- BenchApps
- DroidBench3.0

Configuration:
- Fuzzing time budget : 1 min

JNFuzz-Droid:
- Precision:100%
- Recall:68.4%
- F1-score:81.2%

O = True Positive, * = False Positive, X = False Negative

App Name	JN-SAF	JuCify	JNFuzz-Droid
Part A: JN-SAF NativeFlowBench			
native_source_clean			
native_leak	O	X	O
native_leak_array	O	X	O
native_leak_dynamic_register	O	X	O
native_dynamic_register_multiple	O	X	O
native_noleak			
native_method_overloading	X	X	O
native_multiple_interactions	O	X	X
native_multiple_libraries	O	X	O
native_complexdata	O	X	X
native_complexdata_stringop			
native_set_field_from_arg	OO	OX	XX
native_set_field_from_arg_field	OO	XX	XX
icc_nativetojava	O	X	O
Part B: JuCify BenchApps			
leaker_imei	X	O	O
leaker_string		*	
proxy	X	O	O
proxy_double	X	O	O
Part C: DroidBench3.0 Native			
NativeIDFunction	X	X	O
SinkInNativeCode	X	O	O
SinkInNativeLibCode	X	X	O
Sum, Precision and Recall			
O, higher is better	12	5	13
*, lower is better	0	1	0
X, lower is better	7	14	6
Precision $p = O/(O + *)$	100%	83.3%	100%
Recall $r = O/(O + X)$	63.2%	26.3%	68.4%
F_1-score $= 2pr/(p + r)$	77.4%	40.0%	81.2%

New Native-code Benchmark Test

CCBench:
- 23 hand crafted apps
- Eeah app test data leakage of the native world in common scenarios

Covering:
- Inter-language control flow analysis challenges
- Linux System library calls modeling
- ARM architecture and threading issues
- Native method overloading and others

Configuration:
- Fuzzing time budget: 1 min

JNFuzz-Droid:
- Precision:100%
- Recall:100%
- F1-score:100%

O = True Positive, * = False Positive, X = False Negative

App Name	JN-SAF	JuCify	JNFuzz-Droid
Part A: Inter-language Control Flow			
explosion_path	X	O	O
smt	*		
switch	*		
while	X	O	O
math_library	X	X	O
condition	X	O	O
weak	X	O	O
Part B: Linux System Call			
atoll	X	X	O
strcpy	X	O	O
strcpy1	X	X	O
tcp_client	XX	OX	OO
udp_client	XX	OX	OO
Part C: Architecture and Thread			
armeabi	O	O	O
armeabi-v7a	O	O	O
arm64-v8a	X	O	O
thread_leak	O	O	O
thread_in_leak	O	O	O
thread_noleak			
Part D: Overloading and Misc			
native_method_overloading	O	X	O
native_method_overloading1	X	O	O
global_imei	O	O	O
interrupt_cfg	X	O	O
log_noleak		*	
Sum, Precision and Recall			
O, higher is better	6	15	21
*, lower is better	2	1	0
X, lower is better	15	6	0
Precision p = O/(O + *)	75.0%	93.8%	100%
Recall r = O/(O + X)	28.6%	71.4%	100%
F_1-score = 2pr/(p + r)	41.4%	81.1%	100%

Real World Applications

Dataset:

- Rondomly selected 2500 malware apps from **AndroZoo**, 1596 of which contain native code.
- **Android Botnets** dataset, contain 1929 apps and 591 of which contain native code.

Configuration:

- Fuzzing time budget: 10 mins

Result:

- **30 apps** utilize classic encryption to encrypt sensitive data in native code
- **12 apps** leak sensitive data over network connections after being uninstalled
- **12 apps** use hard-code to hide the storage address to receive sensitive data, such as *URL*, *email*, etc
- **one app** writes sensitive data *IMEI* to the *LOG* function
- **one app** use the *execlp* command in the native function to leak sensitive data

Conclusion

1、JNFuzz-Droid is a lightweight automated fuzzing and taint analysis framework for native code of Android apps.

2、It can quickly locate the Android native code to which sensitive data is passed and automatically analyze and discover data leaks or transmission issues in the native code.

3、Open-source Plan:

JNFuzz-Droid, CCBench and results at https://github.com/cjc-github/JNFuzz-Droid.