Malware Analysis

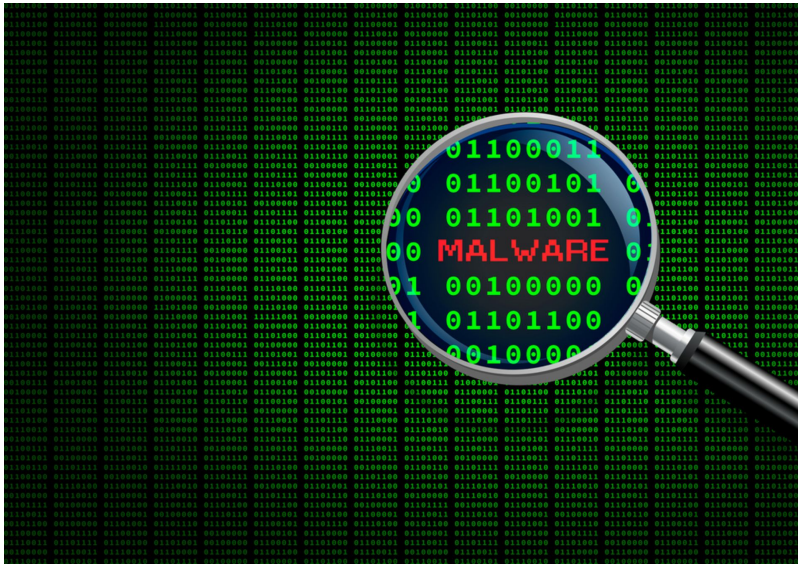
Yue Duan

Introduction

- Malware
  - software intentionally designed to cause damage
- Malware Detection techniques
  - Static analysis
  - Dynamic analysis

Introduction

- Static analysis
  - testing and evaluation of an application by examining the code without executing the application
  - Pros:
    - Good code coverage
    - Time efficiency
  - Cons:
    - False positives
    - Code obfuscation
    - Encryption



Introduction

- Dynamic analysis
  - testing and evaluation of an application during runtime
  - Pros:
    - Capture behaviors accurately
  - Cons:
    - Poor code coverage
    - High runtime overhead

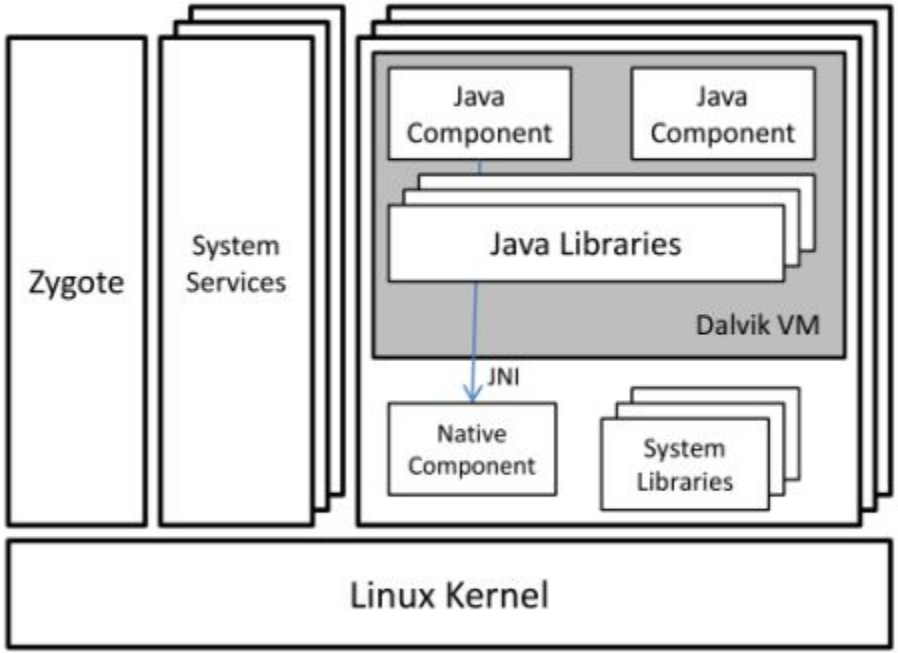
DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis

Lok Yan, Heng Yin

Syracuse University

Usenix Security 2012

Android



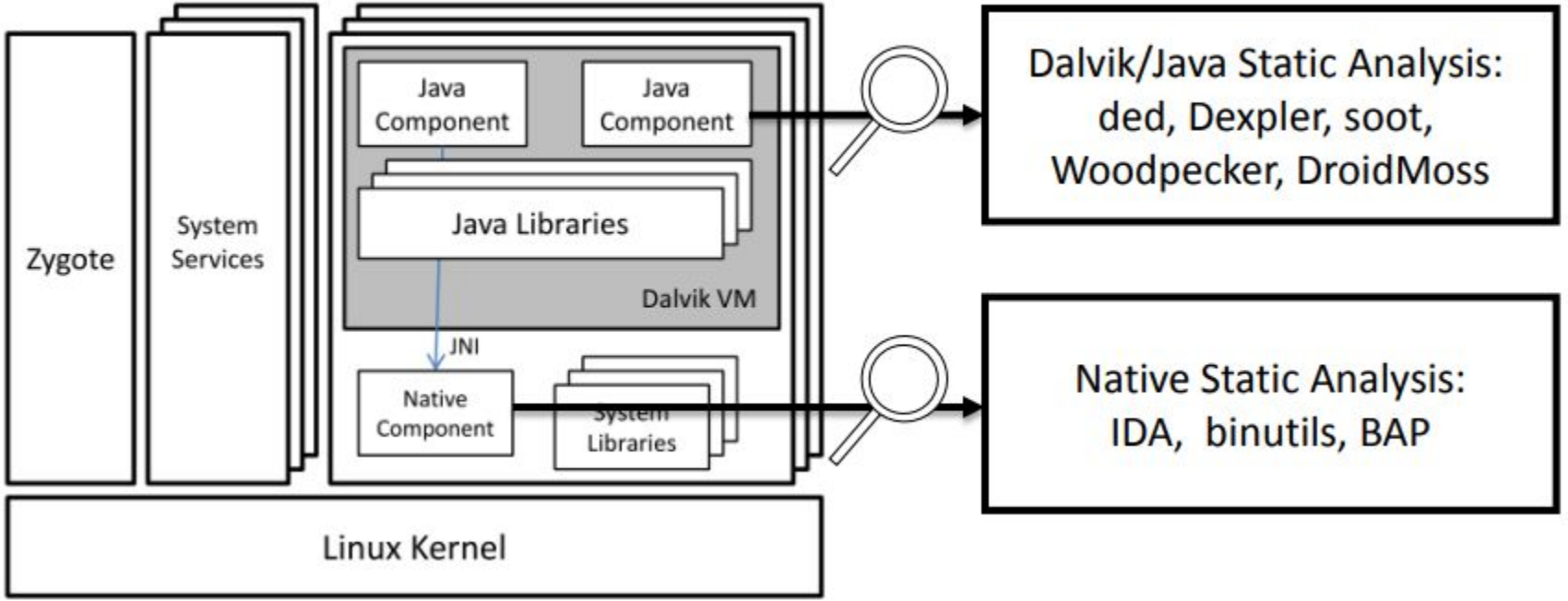
Java Components

Native Components

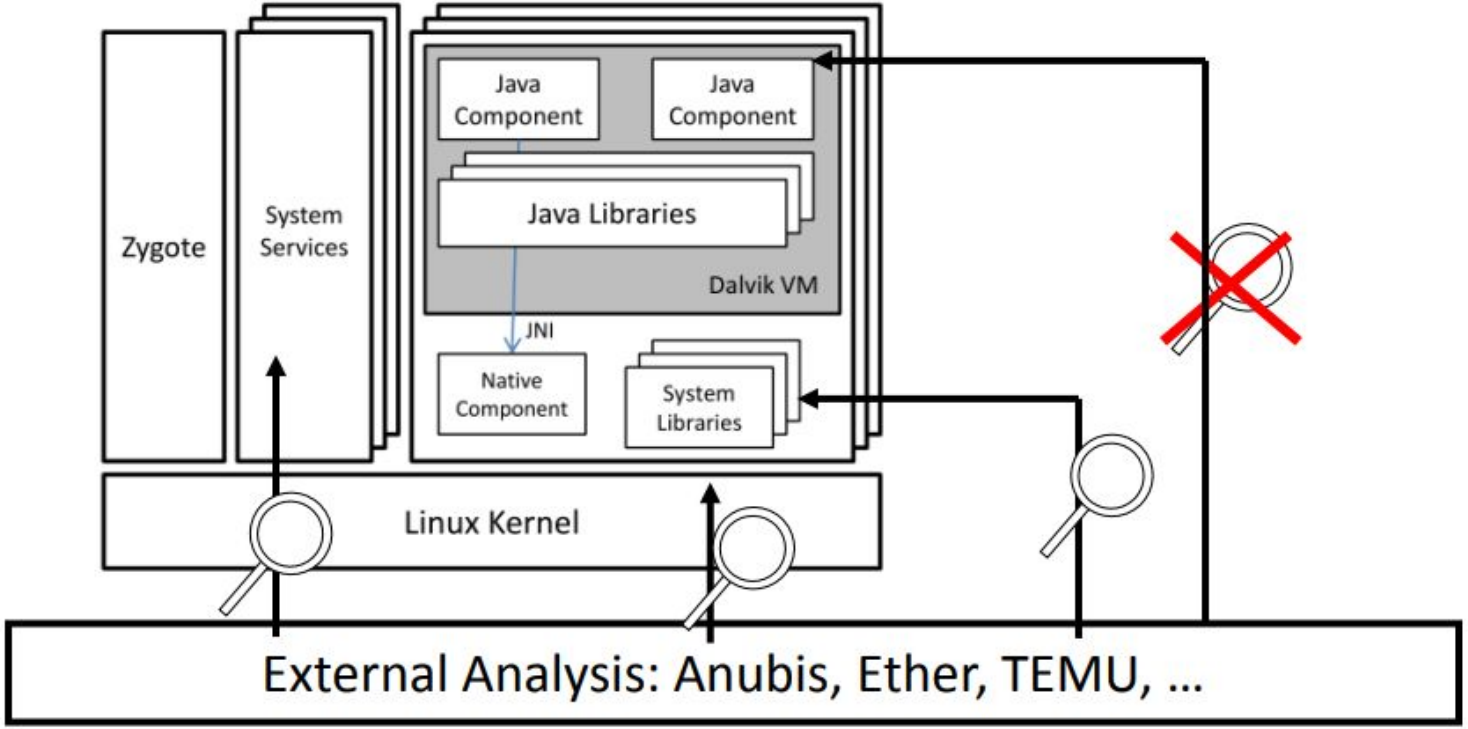
System Services

Apps

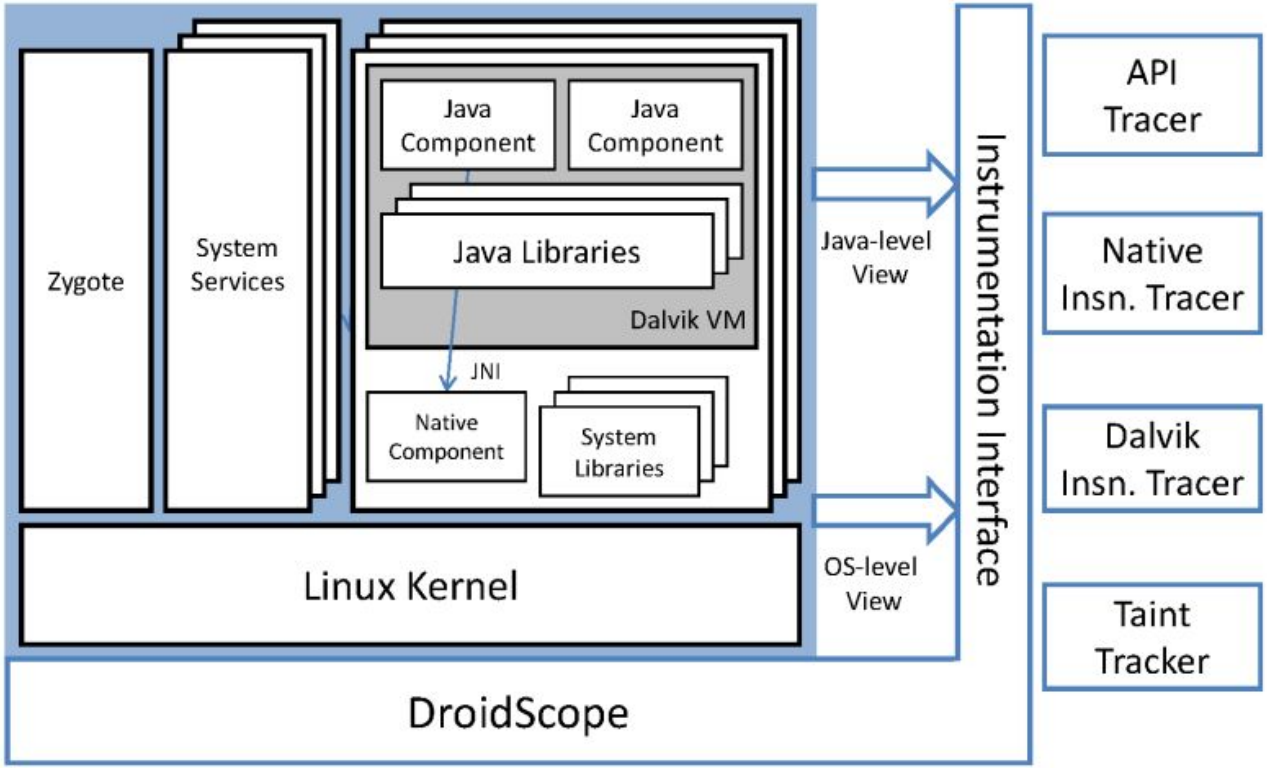
Motivation


Zygote
System Services
Java Component
Java Component
Java Libraries
Dalvik VM
JNI
Native Component
System Libraries
Linux Kernel
Dalvik/Java Static Analysis:
ded, Dexpler, soot,
Woodpecker, DroidMoss
Native Static Analysis:
IDA, binutils, BAP

Motivation


Zygote
System Services
Java Component
Java Component
Java Libraries
Dalvik VM
JNI
Native Component
System Libraries
Linux Kernel
External Analysis: Anubis, Ether, TEMU, …

DroidScope Overview



DroidScope Overview

- Dynamic binary instrumentation for Android
  - Leverage Android Emulator in SDK
  - No changes to Android Virtual Devices
  - External instrumentation
    - Linux context
    - Dalvik context
  - Extensible: plugin-support / event-based interface
  - Performance
    - Partial JIT support
    - Instrumentation optimization

Linux Context: Identify Apps

- Shadow task list
  - pid, tid, uid, gid, euid, egid, parent pid, pgd, comm
  - argv[0] : app name
- Shadow memory map
  - Address Space Layout Randomization (Ice Cream Sandwich)
  - Update on
    - fork, execve, clone, prctl and mmap2

Java/Dalvik View

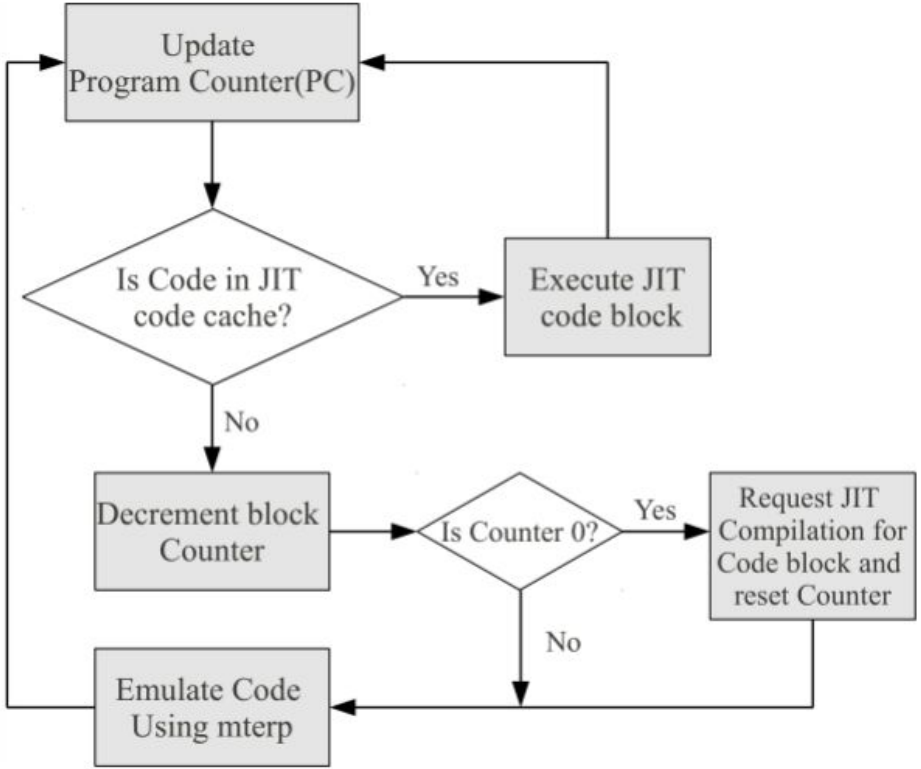
- Dalvik virtual machine
  - register machine (all on stack)
  - 256 opcodes
  - saved state, glue, pointed to by ARM R6, on stack in x86
- mterp
  - offset-addressing: fetch opcode then jump to

    (dvmAsmInstructionStart + opcode * 64)
- Which Dalvik opcode?
  - Locate dvmAsmInstructionStart in shadow memory map
  - Calculate opcode = (R15 - dvmAsmInstructionStart) / 64.

Just In Time (JIT) Compiler

- Designed to boost performance
- Triggered by counter
  - mterp is always the default
- Trace based
  - Multiple basic blocks
  - Multiple exits or chaining cells
  - Complicates external introspection
  - Complicates instrumentation

Disable JIT


Update
Program Counter(PC)
Is Code in JIT
code cache?
Yes
Execute JIT
code block
No
Decrement block
Counter
Is Counter 0?
Yes
Request JIT
Compilation for
Code block and
reset Counter
No
Emulate Code
Using mterp

Dynamic Instrumentation

- Event based interface
  - Execution: e.g. native and Dalvik instructions
  - Status: updated shadow task list
- Query and Set, e.g. interpret and change cpu state
- Performance
  - Example: Native instructions vs. Dalvik instructions
  - Instrumentation Optimization

Dynamic Instrumentation

	NativeAPI	LinuxAPI	DalvikAPI
Events	instruction begin/end	context switch	Dalvik instruction begin
	register read/write	system call	method begin
	memory read/write	task begin/end	
	block begin/end	task updated	
		memory map updated	
Query & Set	memory read/write	query symbol database	query symbol database
	memory r/w with pgd	get current context	interpret Java object
	register read/write	get task list	get/set DVM state
	taint set/check		taint set/check objects
			disable JIT

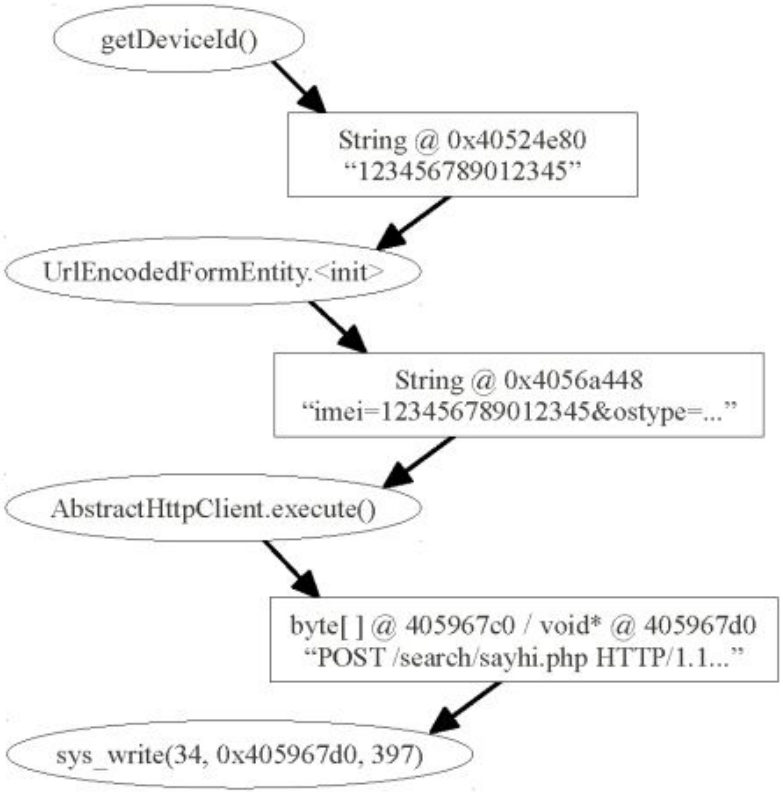
Example: Dalvik Instruction Tracer

```
1. void opcode_callback(uint32_t opcode) {
2.   printf("[%x] %s\n", GET_RPC, opcodeToStr(opcode));
3. }
4.
5. void module_callback(int pid) {
6.   if (bInitialized || (getIBase(pid) == 0))
7.     return;
8.
9.
10.  getModAddr("dfk@classes.dex", &startAddr, &endAddr);
11.  addDisableJITRange(pid, startAddr, endAddr);
12.  disableJITInit(getGetCodeAddrAddress(pid));
13.  addMterpOpcodesRange(pid, startAddr, endAddr);
14.  dalvikMterpInit(getIBase(pid));
15.  registerDalvikInsnBeginCb(&opcode_callback);
16.  bInitialized = 1;
17. }
18.
19. void _init() {
20.  setTargetByName("com.andhuhu.fengyinchuanshuo");
21.  registerTargetModulesUpdatedCb(&module_callback);
22. }
```

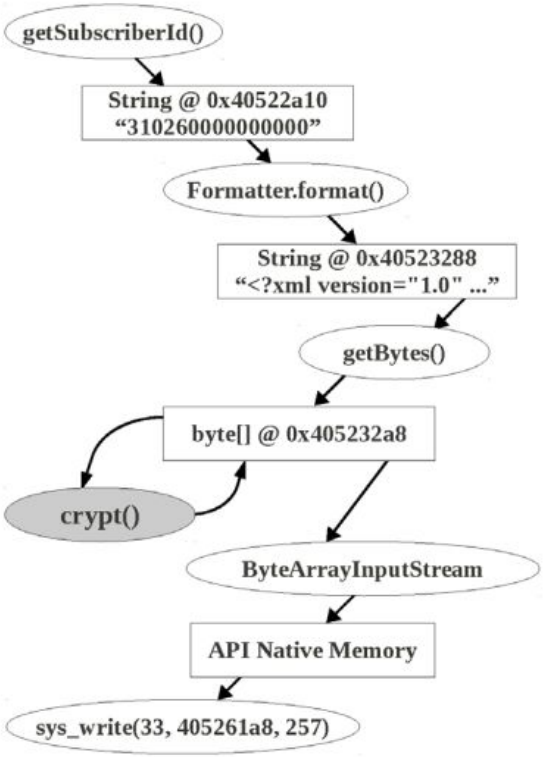
Usage Evaluation

- Use DroidScope to analyze real world malware
  - API Tracer
  - Dalvik Instruction Tracer
  - Taint Tracker – taint IMEI/IMSI @ move_result_object after getIMEI/getIMSI
- Analyze included exploits
  - Removed patches in Gingerbread
  - Intercept system calls
  - Native instruction tracer

Droid Kung Fu: TaintTracker



DroidDream: TaintTracker

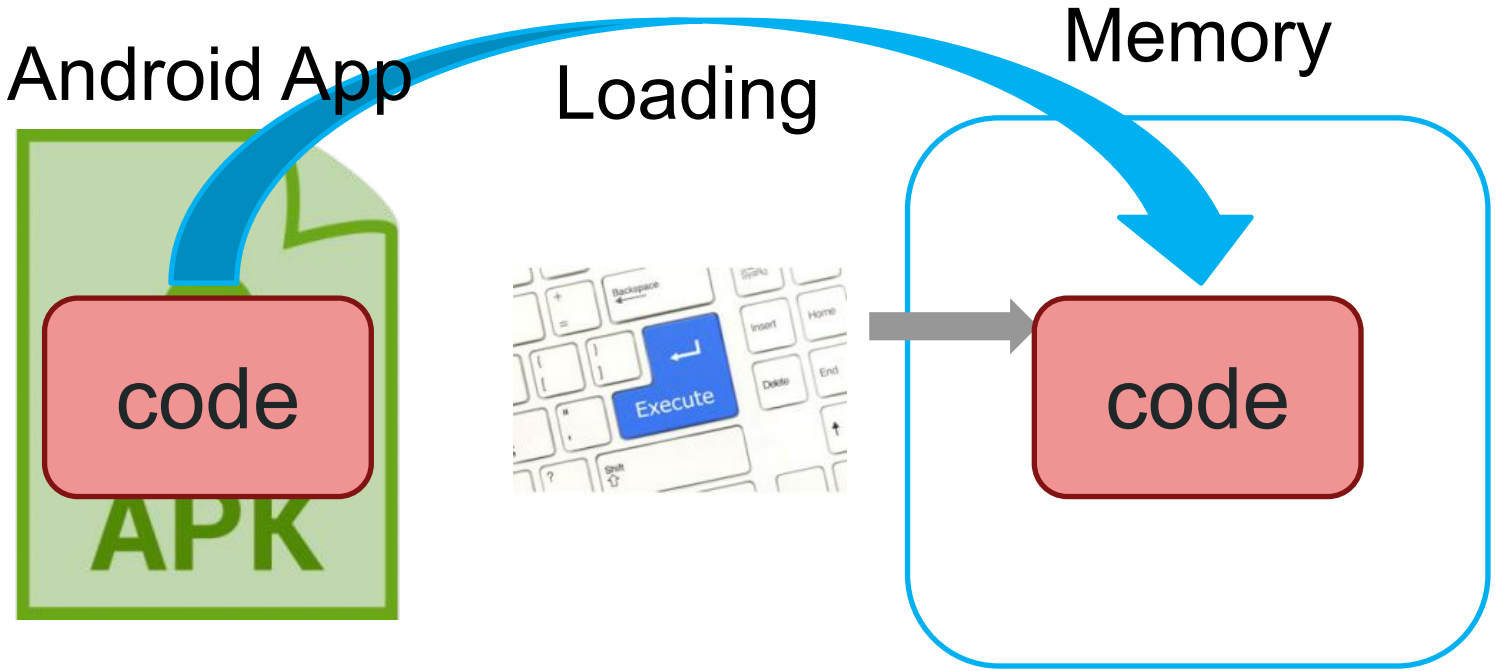


DeepVMUnpack: Neural Network-based Semantic Recovery from VM-Protected Android Apps for Malware Detection

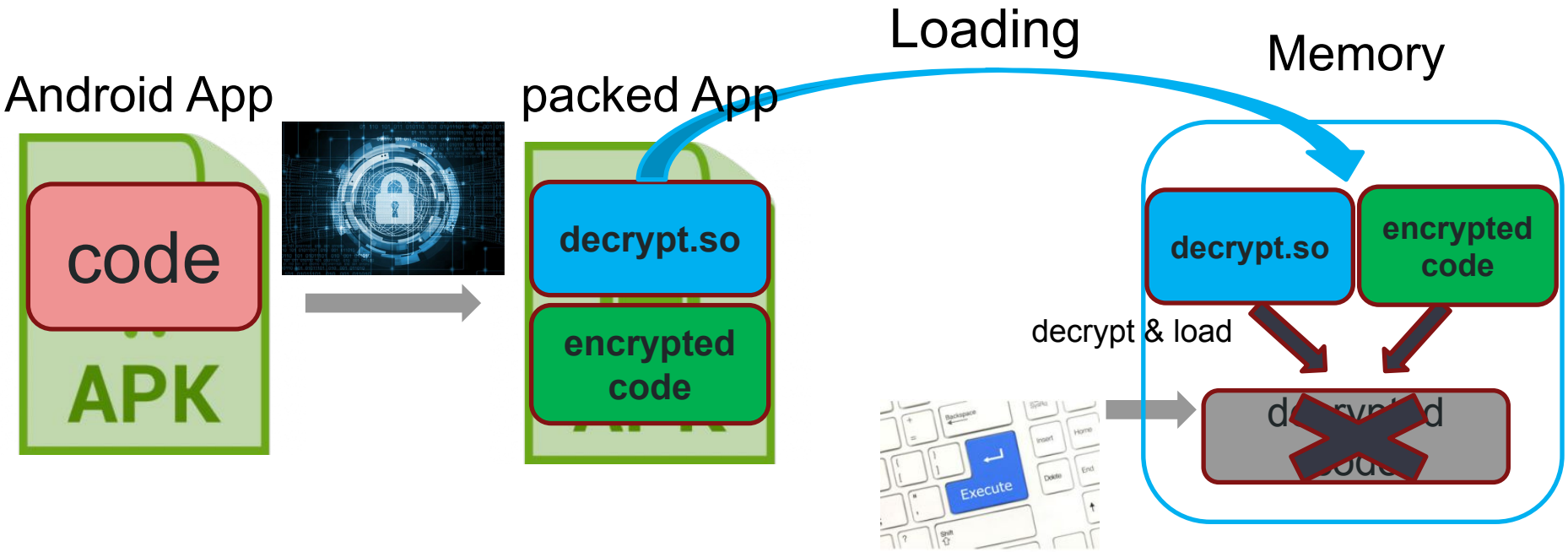
Xin Zhao, Mu Zhang, Yue Duan, Fengyuan Xu

To be submitted

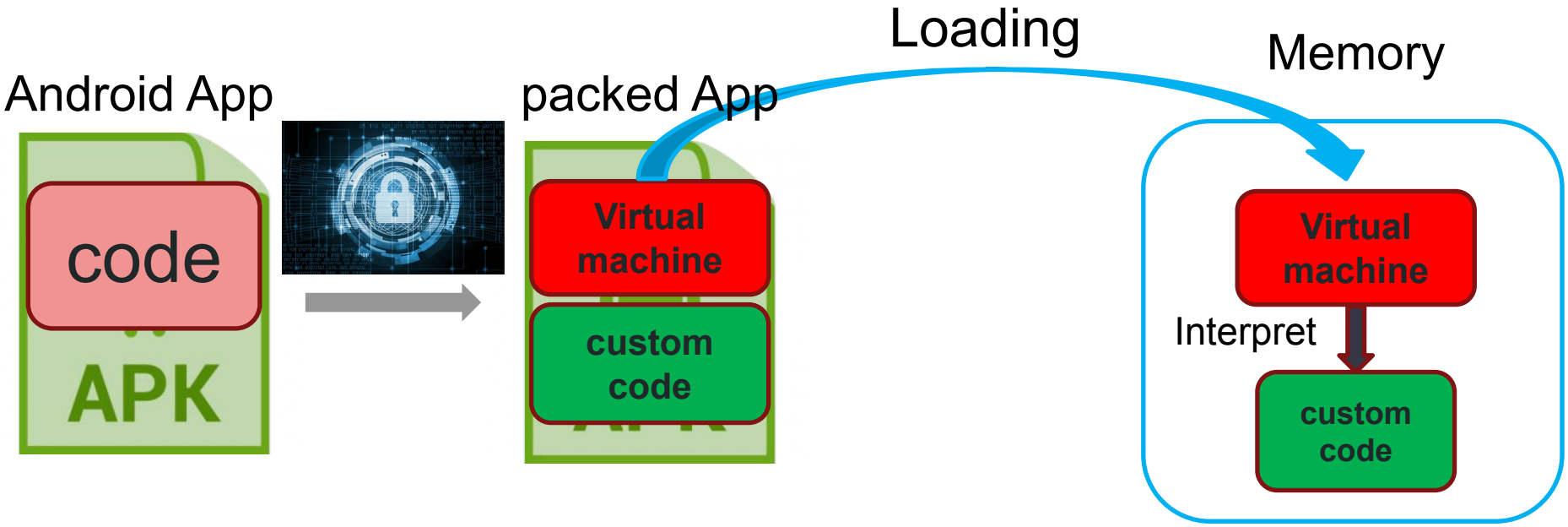
Android packing



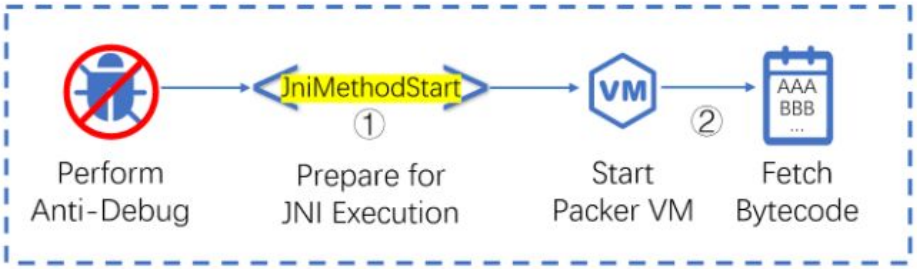
Android packing

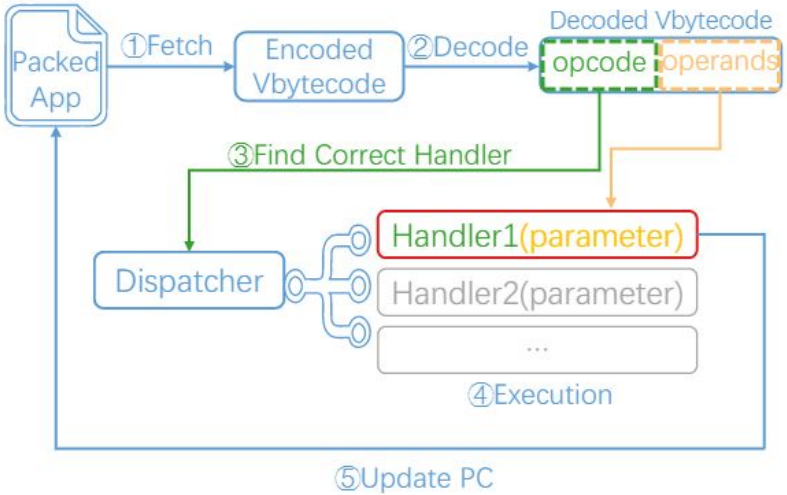


Android VM-packing

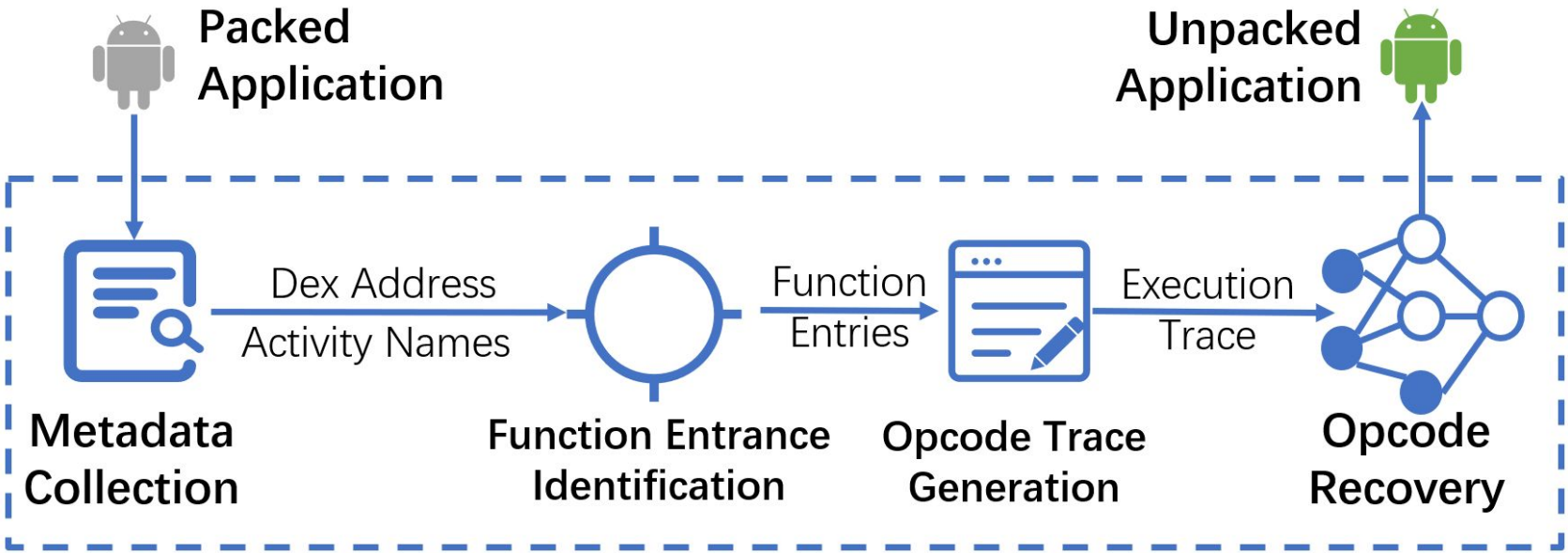


VM Execution

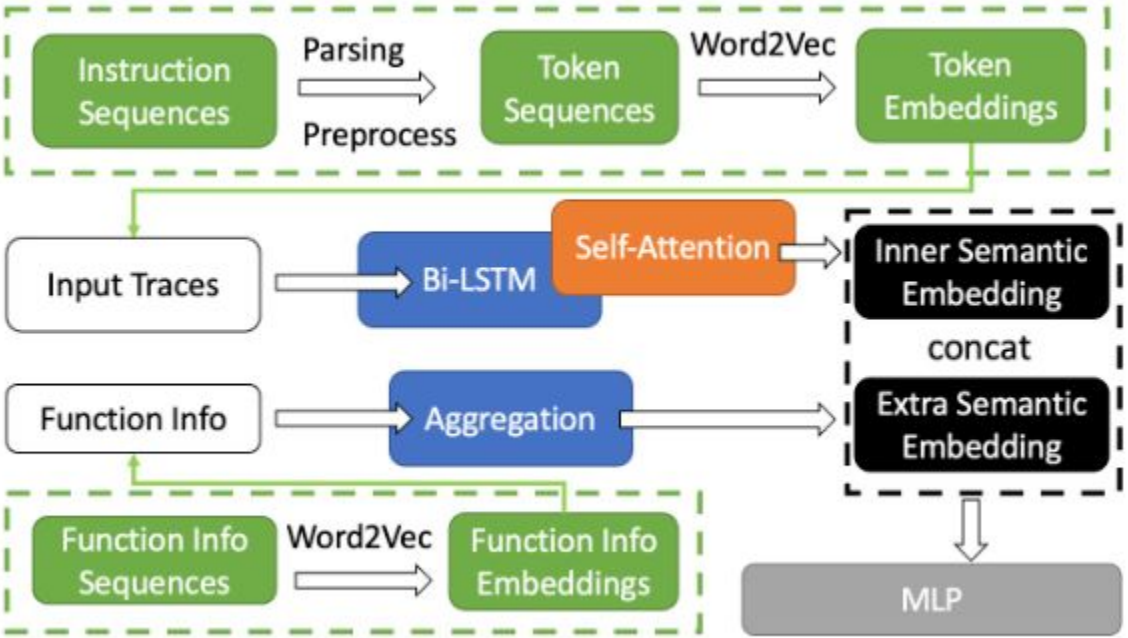




DeepVMUnpack Overview



Deep Learning Model



Evaluation

Bytecode	Succ	Fail	Accuracy	Bytecode	Succ	Fail	Accuracy	Bytecode	Succ	Fail	Accuracy
0x13	106	0	1.0	0x14	23	17	0.57	0x71	47	36	0.56
0x12	429	160	0.72	0x15	54	0	1.0	0x6f	150	5	0.96
0x6e	615	193	0.76	0x22	78	2	0.97	0x70	48	2	0.96
0xc	358	0	1.0	0x38	39	2	0.95	0x1a	167	0	1.0
0x54	34	2	0.94	0x37	2	0	1.0	0x19	9	0	1.0
0x62	28	0	1.0	0x1f	55	90	0.37	0x8	8	0	1.0
0x16	18	0	1.0	0x72	4	0	1.0	0x2	2	0	1.0
0x5b	60	11	0.84	0x75	9	0	1.0	0x74	1	0	1.0
0x18	4	0	1.0	0xb	2	0	1.0	0x8a	1	1	0.5
0x76	1	0	1.0	0xcd	1	0	1.0	average	2353	538	0.81

	F1-Score	Accuracy	Precision	Recall	False Positive	Succ	Total
Parema	0.0000	0.0000	0.0000	0.0000	1.0000	0	62
DEEPVMUNPACK	0.9026	0.8226	1.0000	0.8226	0.1774	51	62

Thank you!

Question?