TaintART: A Practical Multi-level Information-Flow Tracking System for Android RunTime

Mingshen Sun[*], Tao Wei[†], and John C.S. Lui[*]

[*] The Chinese University of Hong Kong
[†] Baidu X-Lab

October 25 @ CCS 2016

Introduction – Android Malware

- **Mobile devices become the biggest target among all threats**

Report

From 2004 to 2013 we detected nearly 200,000 samples of malicious mobile code. In 2014 there were 295,539 new programs, while the number was 884,774 in 2015. — Kaspersky [1]

[1] https://securelist.com/analysis/kaspersky-security-bulletin/73839/mobile-malware-evolution-2015/

- **Android malware samples accounted for *98%* of all mobile threats**

- **Trojan**
- **Spyware**
- **Phishing apps**
- **Ransomware**
- **Rootkit**
- **...**

[2]

[2] http://www.phonearena.com/news/Malware-on-Android---a-myth-or-real-threat_id37322

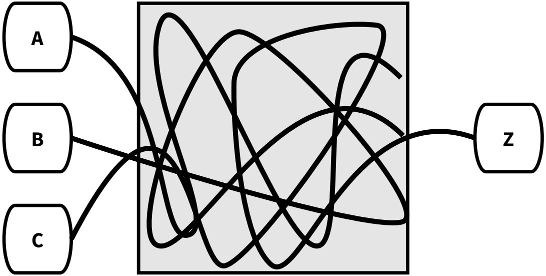
Rest of the talk…

1 Introduction to dynamic taint analysis

2 TaintART: A Practical Multi-level Information-Flow Tracking System for Android RunTime
- Introduction to dynamic taint analysis system
- TaintART
- Background of Dalvik and ART
- System Design of TaintART
- Implementation & Case Study
- Evaluation by macro/micro benchmark

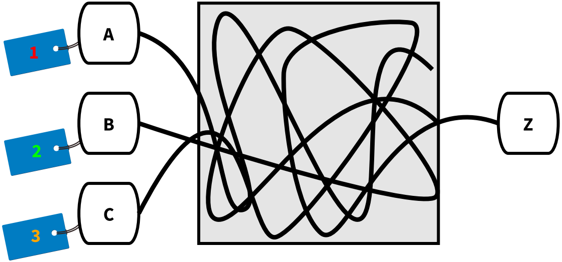
3 Summary

- **Dynamic taint analysis (aka. dynamic information-flow analysis)**



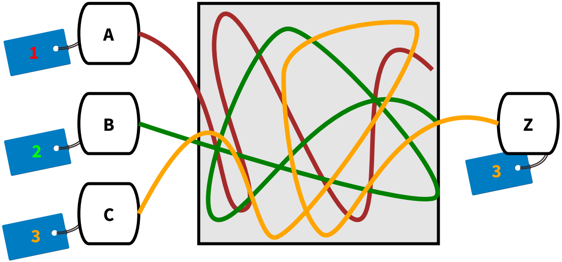
Introduction

- **Dynamic taint analysis (aka. dynamic information-flow analysis)**
  1. label (*taint*) sensitive data from certain functions (*source*)



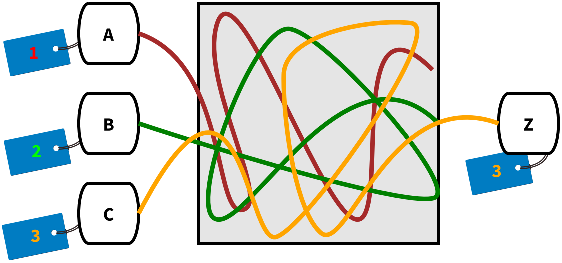
Introduction

- **Dynamic taint analysis (aka. dynamic information-flow analysis)**
  1. label (*taint*) sensitive data from certain functions (*source*)
  2. handle label transitions (*taint propagation*) between variables, files, and procedures at runtime
  3. a tainted label transmit out of the device through some functions (*sinks*)
  4. data leakage



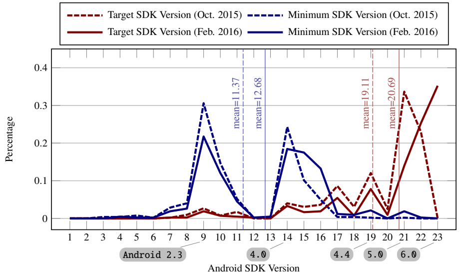
Introduction

- **Applications of dynamic taint analysis systems (aka. dynamic information-flow analysis)**
  1. attack detection and prevention
  2. information policy enforcement
  3. testing in software engineering
  4. data lifetime and scope analysis



Introduction

- **Current status of dynamic taint analysis tool for Android**
  - TaintDroid is a notable system released in 2010 by William Enck et al., and many systems are based on TaintDroid
  - TaintDroid was designed for VM-based system and implemented on legacy Android systems (2.1, 2.3, 4.1, and 4.3)
  - recent Android adopted ahead-of-time (AOT) compilation strategy and introduced new Android RunTime (ART) to replace Davlik VM
  - portability, compatibility, and performance issues

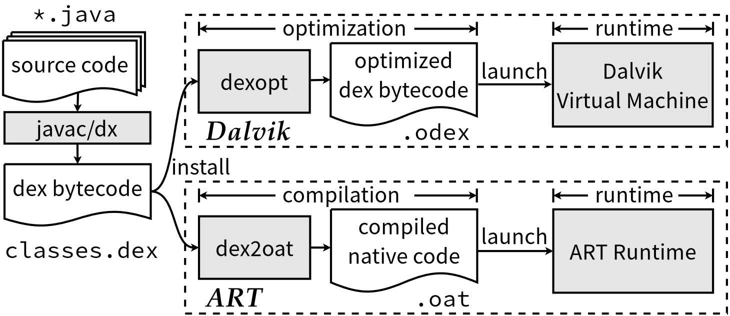


TaintART

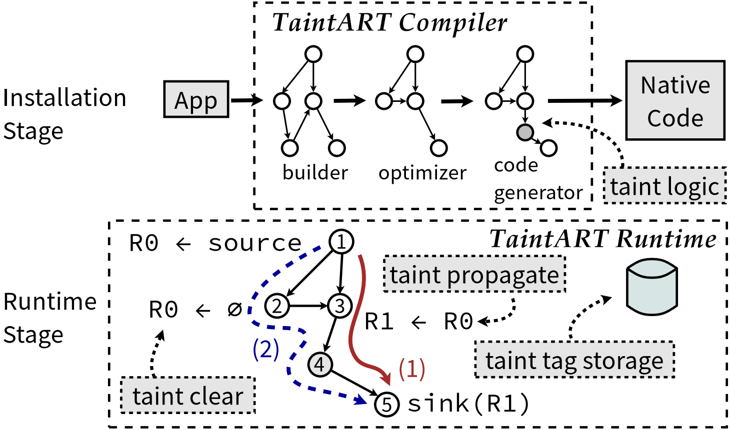
- **TaintART**
  - We design and implement TaintART, a dynamic information-flow tracking system which targets the latest Android runtime
  - TaintART introduces a multi-level taint label to tag the sensitive levels
  - TaintART instruments Android's compiler and utilizes processor registers for taint storage
  - TaintART only needs registers accesses and achieve faster taint propagation compared to TaintDroid

Background – Android App Environment

- **The Dalvik app environment**
  - source code -> dex bytecode -> optimized dex bytecode -> run
- **The ART app environment**
  - source code -> dex bytecode -> compiled native code -> run



- **The TaintART compiler in the installation stage**
- **The TaintART runtime in the runtime stage**



System Design – Taint Tag Storage

- **Taint tag storage**
  - The TaintART compiler will reserve registers for taint storage

	S15	…	S4	S3	S2	S1	S0	R15	…	R4	R3	R2	R1	R0
R5	0	…	0	0	0	1	1	0	…	1	0	0	1	0

floating-point registers   regular registers

- **Taint tag propagation (from R1 to R0)**
  1. mask
  2. shift
  3. merge


```
;;  clear dest bit                         (1)
BIC R5, R5, #1

;;  mask tainted bit                       (2)
AND R12, R5, #2

;;  shift right (or left)                  (3)
MOV R12, R12, LSR #1

;;  merge tainted bits                     (4)
ORR R5, R5, R12

ORR R5, R5, R12, LSR #1                    (3')
```

Figure: Taint tag propagates from R1 to R0.

System Design – Taint Propagation Logic

- **Taint propagation logic**
  - classes of instructions (instruction type and related locations)
  - e.g., move, boolean not, add, etc.
  - the location is an abstraction over the potential registers containing variables or constants
- **Method invocation taint propagation**
- **Binder IPC & native code taint propagation**

Table 1: Descriptions of multi-level aware taint propagation logic.

HInstruction (Location)	Semantic	Taint Propagation Logic Description
HParallelMove(dest, src)	dest ← src	Set dest taint to src taint, if src is constant then clear dest taint
HUnaryOperation(out, in) HBooleanNot, HNeg, HNot	out ← in	Set out taint to in taint, unary operations ∈ {!, -, ~}
HBinaryOperation(out, first, second) HAdd, HSub, HMul, HDiv, HRem, HShl, HShr, HAnd, HOr, HXor	out ← first ⊗ second	Set out taint to max(first taint, second taint), ⊗ ∈ {+, -, *, /, %, <<, >>, &, |, ^}
HArrayGet(out, obj, index)	out ← obj[index]	Set out taint to obj taint
HArraySet(value, obj, index)	obj[index] ← value	Set obj taint to value taint
HStaticFieldGet(out, base, offset)	out ← base[offset]	Set out taint to base[offset] field taint
HStaticFieldSet(value, base, offset)	base[offset] ← value	Set base[offset] field taint to value taint
HInstanceFieldGet(out, base, offset)	out ← base[offset]	Set out taint to base[offset] field taint
HInstanceFieldSet(value, base, offset)	base[offset] ← value	Set base[offset] field taint to value taint

Implementation & Case Study

- **Taint sources and privacy leakage levels**
  - **four levels**: no leakage, device identity, sensor data & location data leakage, and sensitive content
  - **classes or services**: Telephony Manger, Sensor Manger, Location Manger, Content Resolver, File, Camera, and MediaRecorder
- **Case study for privacy tracking**
  - analysis popular apps at runtime
  - tracking data flows
  - Taobao <u>leaks device identity, sensor data and location data at runtime</u>

Table 2: Taint Sources and Privacy Leakage Levels

Level	Leaked Data	Source	Class/Service
0 (00)	No Leakage	N/A	N/A
1 (01)	Device Identity	IMSI	`TelephonyManager`
		IMEI	`TelephonyManager`
		ICCID	`TelephonyManager`
		SN	`TelephonyManager`
2 (10)	Sensor Data	Accelerometer	`SensorManager`
		Rotation	`SensorManager`
	Location Data	GPS Location	`LocationManager`
		Last Seen Location	`LocationManager`
		Network Location	`LocationManager`
3 (11)	Sensitive Content	SMS	`ContentResolver`
		MMS	`ContentResolver`
		Contacts	`ContentResolver`
		Call log	`ContentResolver`
		File content	`File`
		Camera	`Camera`
		Microphone	`MediaRecorder`

Evaluation – Macrobenchmarks

- **Macrobenchmarks**
  - app launch time: <u>6%</u>
  - app installation time: <u>12%</u>
  - contacts read/write: <u>20%/12%</u>

Table 5: Macrobenchmark results.

Macrobenchmark Name (ms)	Original (with Optimizing Backend)	TaintART
App Launch Time	348.2	370.3
App Installation Time	1680.5	1886.3
Contacts Read/Write	7.0/9538.5	8.4/9655.2

- **Compiler microbenchmark: compilation time**
  - 80 built-in apps in AOSP project
  - 19.9% overhead

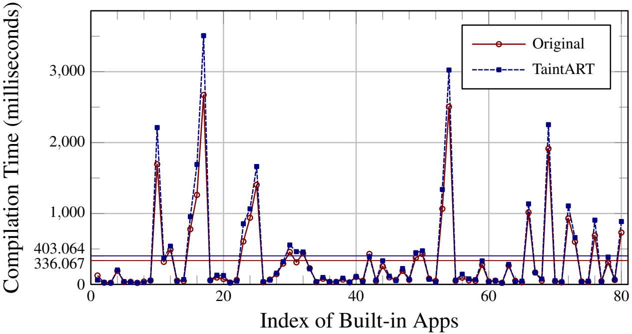


Figure: Comparison of compilation time.

Evaluation – Microbenchmarks

- **Compiler microbenchmark: instruction overhead**
  - 21% overhead in total
  - 0.8% overhead only for memory-related instructions

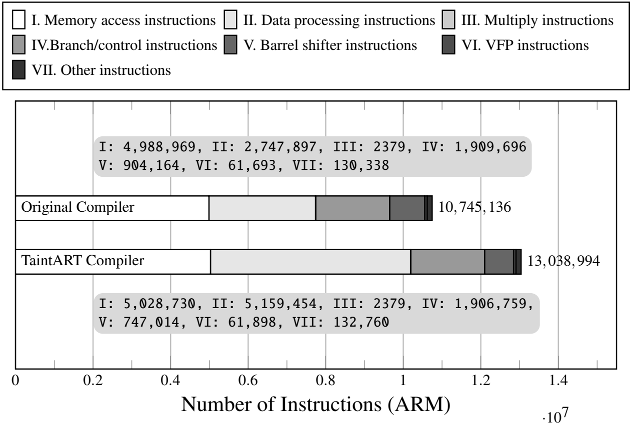


Figure: Comparison of instruction numbers for different types.

Evaluation – Microbenchmarks

- **Java microbenchmark**
  - CaffeineMark 3.0
    - 14% overhead overall
    - 99.8% more scores compared to the legacy Dalvik environment

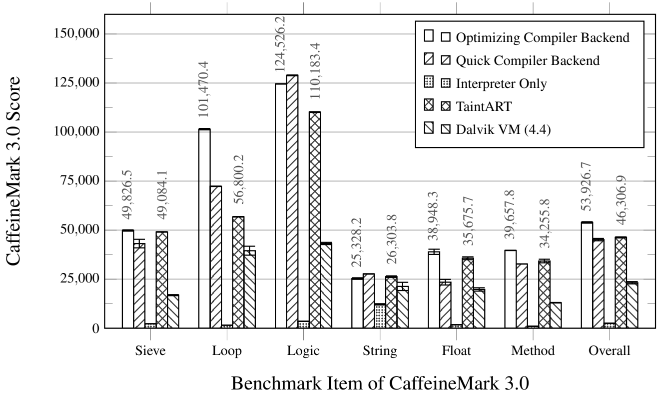


Figure: CaffeineMark 3.0 Java microbenchmark.

Evaluation – Others

- **Memory microbenchmark: 0.4%**
- **IPC microbenchmark: 4%**
- **Compatibility evaluation**

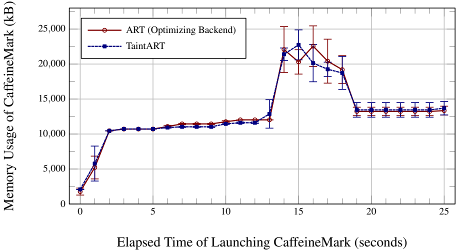


Table 6: IPC Throughput Benchmark (10,000 pairs of messages).

Macrobenchmark Name	Original	TaintART	Overhead
Execution Time	2987 ms	3117 ms	4.35 %
Memory (client)	51 572 kB	53 170 kB	3.10 %
Memory (server)	38 812 kB	39 689 kB	2.26 %

Summary

- **Tracking information flows**
  - dynamic taint analysis for new Android RunTime
  - register-based and compiler instrumentation
  - evaluate in micro/macro benchmark

Thank you!

Thank you.

Question?

Backup Slides for TaintART – Taint Tag Storage

- **Taint tag spilling**
  - the register allocator will temporarily store extra variables in the main memory