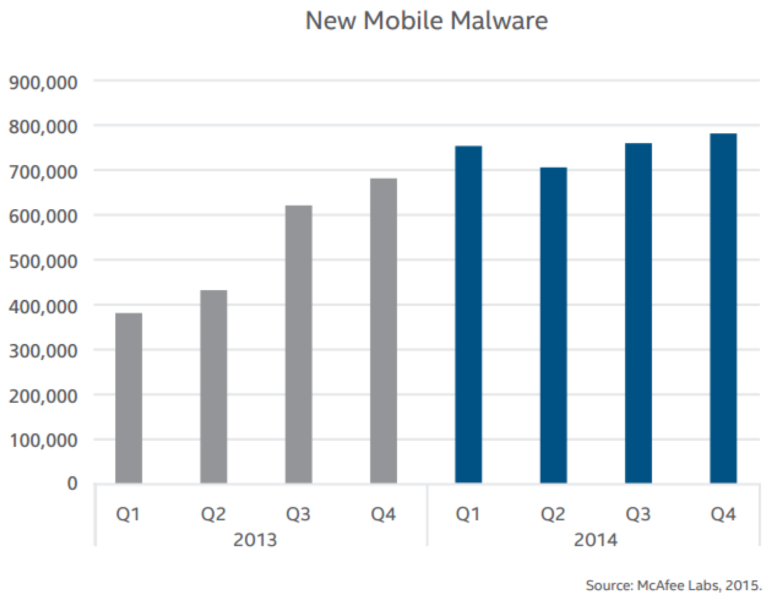
HotOS XV, Kartause Ittingen, Switzerland | May 20, 2015

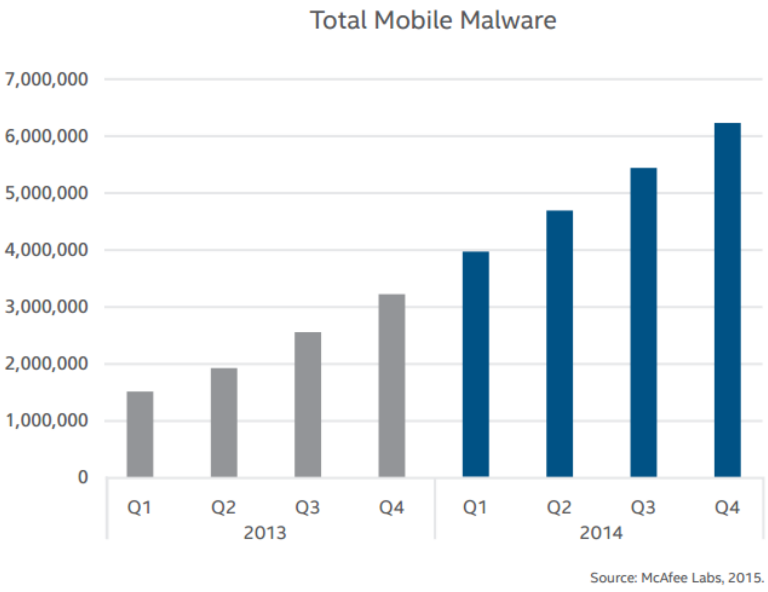
Practical Always-On Taint Tracking for Mobile Devices

Justin Paupore[*]
Earlence Fernandes[*]
Atul Prakash[*]
Sankardas Roy[†]
Xinming Ou[†]

[*]University of Michigan
[†]Kansas State University

Mobile Malware: A Growing Problem





Mobile Malware: A Growing Problem

- Most users get apps through centralized app stores
- App store vendors want to detect and remove malware

Example: Bouncer

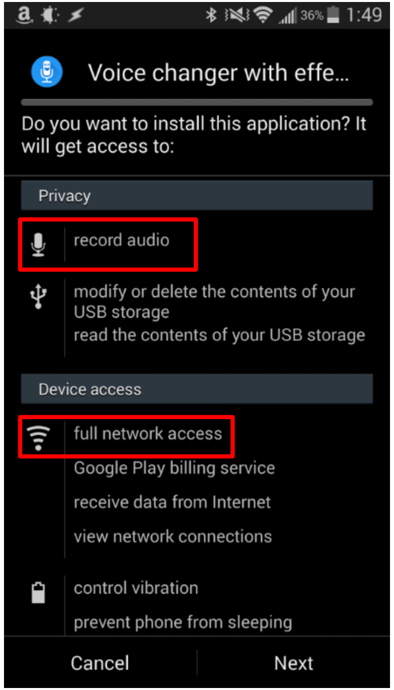
- Google Play malware detection engine
- Apps are scanned on submission
  - Static analysis
  - Dynamic sandboxing
- Problem: can be detected and evaded [Oberheide and Miller, SummerCon '12]

Better solution: on-device analysis

- Observe “in the wild” behavior
- Google already does this, to some degree
  - How? They’re not telling
  - All we know: **Not** a framework modification

What if we want more?

- Inspecting permissions used isn't enough
- Nor is pure static analysis [Wang et al, SEC'13]
- Better idea: monitor how data is used at runtime
- Solution: Taint tracking!
  - As made famous on Android by TaintDroid [Enck et al, OSDI'10]



The Problem with TaintDroid

- Adds ~15% overhead to *all* Java code on device
  - … even trusted system processes
  - … even the 99% of code that never touches sensitive data [Wei and Lie, SPSM'14]
- Problem: latency-sensitive code (UI drawing, audio, games, …)

The Proposal

- Take advantage of mobile phone ecosystem
  - Push heavy static analysis to app store owner
  - Instrument app code during install
  - Use and abuse ASIC peripherals to accelerate tracking

Static analysis

- Runs in the cloud when an app is submitted
- Identifies:
  - Known-safe sections of app code
  - Minimal set of instructions to track for taint propagation
- Signed by store owner, delivered with app

Runtime requirements

- Need to know when identified instructions run, and propagate taint
- Traditionally done in-line
- Doesn't have to be! [ShadowReplica, Jee et al, CCS'13]

Runtime requirements

- For out-of-line propagation:
  - Enqueue events inline
  - Dequeue later/in parallel, and reconstruct flow
- Speed of FIFO enqueue critical
- With two things, enqueue becomes nearly free:
  - Ahead-of-time compilation
  - Embedded Trace Macrocell (ETM)

Ahead-of-time compilation

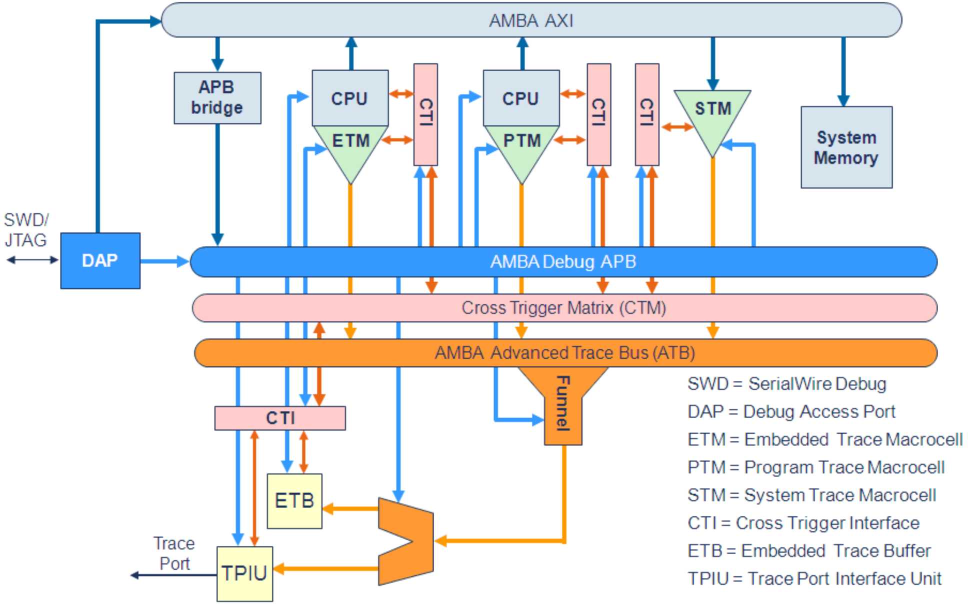
- Compile machine-code version of bytecode on-device
- Android example: Android Runtime (ART)
  - First included in 4.4, default in 5.0+
- Allows each bytecode instance to have independent machine code

Embedded Trace Macrocell (ETM)

- ARM hardware peripheral – part of CPU core
- Designed for full-speed program tracing, read out by JTAG
  - Can also be read out by CPU
- Included in nearly every ARM CPU in the past 10+ years (original spec released 1999)

Embedded Trace Macrocell (ETM)

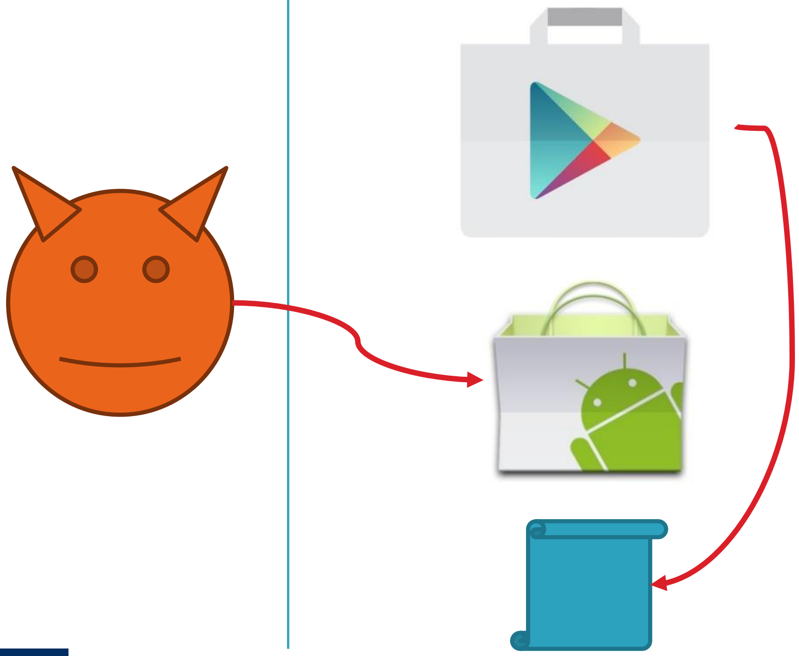
- One ETM per core
- Executed instructions logged to trace bus
  - PC, address, data
  - Filterable
- Trace buffer (ETB) captures events
- Buffer memory-mapped

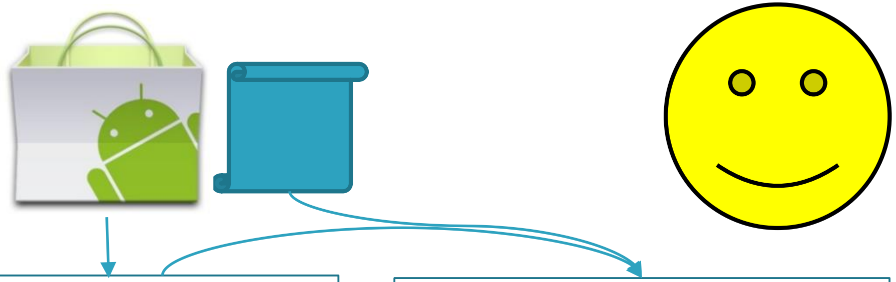


Using ETM as a FIFO

- During AOT compilation, emit *marker instructions*
  - Store to a designated “magic” address
  - NOP from app’s perspective
  - Value stored can encode payload
- At runtime:
  - Configure ETM filters to recognize “magic” address
  - Run app normally
  - ETM generates events when marker instructions executed
  - Read events from another core and reconstruct program flow

Example

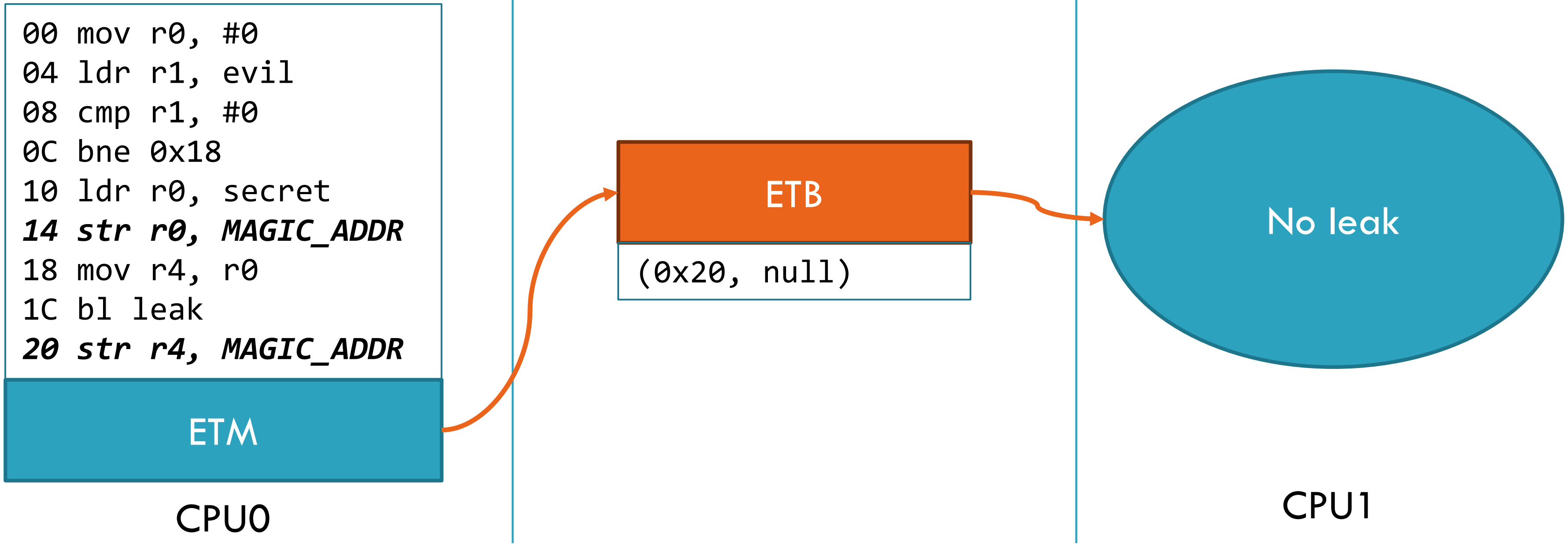




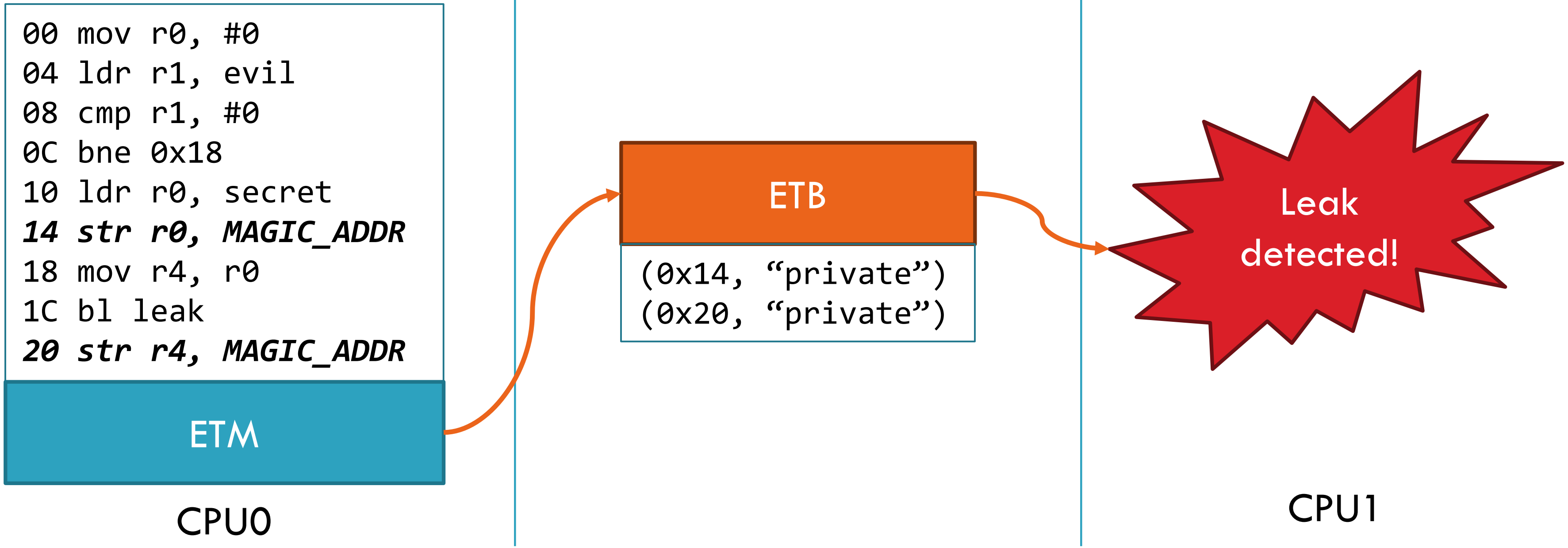
```
x = null;
if (evil) {
  x = secret;
}
leak(x);
```

```
00 mov r0, #0
04 ldr r1, evil
08 cmp r1, #0
0C bne 0x1C
10 ldr r0, secret
14 str r0, MAGIC_ADDR
18 mov r4, r0
1C bl leak
20 str r4, MAGIC_ADDR
```

Example


00 mov r0, #0
04 ldr r1, evil
08 cmp r1, #0
0C bne 0x18
10 ldr r0, secret
14 str r0, MAGIC_ADDR
18 mov r4, r0
1C bl leak
20 str r4, MAGIC_ADDR
ETM
CPU0
ETB
(0x20, null)
No leak
CPU1

Example


00 mov r0, #0
04 ldr r1, evil
08 cmp r1, #0
0C bne 0x18
10 ldr r0, secret
14 str r0, MAGIC_ADDR
18 mov r4, r0
1C bl leak
20 str r4, MAGIC_ADDR
ETM
CPU0
ETB
(0x14, “private”)
(0x20, “private”)
Leak detected!
CPU1

Design Benefits

- Minimal overhead [~O(1 store)] for instructions that need tracking
- Zero overhead for instructions that don't
- Easily enabled/disabled on the fly

Conclusion

- Taint tracking on ARM smartphones can be performed with low latency cost
- Allows in-the-field usage information to be collected and fed back to app store owners, without unduly burdening the user

THANK YOU!

QUESTIONS?

Justin Paupore <jpaupore@umich.edu>