Whole-System Dynamic Binary Analysis
Continued

Repeatable Reverse Engineering for the Greater Good with PANDA

Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, Ryan Whelan
PANDA’s Workflow

Fig. 1: Replay-based Reverse Engineering Workflow. PANDA’s ability to record and replay whole system executions is the foundation of its use in reverse engineering. One captures a recording and then iteratively builds one or more plugins that perform dynamic analyses.

Record&Replay Internals

Fig. 2: PANDA Non-determinism log
Log sizes for various replays

<table>
<thead>
<tr>
<th>Replay</th>
<th>Instructions</th>
<th>Log size</th>
<th>Instr/byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>freebsdboot.rr</td>
<td>9.3B</td>
<td>533MB</td>
<td>17</td>
</tr>
<tr>
<td>spotify.rr</td>
<td>12B</td>
<td>229MB</td>
<td>52</td>
</tr>
<tr>
<td>haikuurl.rr</td>
<td>8.6B</td>
<td>119MB</td>
<td>72</td>
</tr>
<tr>
<td>carberp1.rr</td>
<td>9.1B</td>
<td>43MB</td>
<td>212</td>
</tr>
<tr>
<td>win7nessl.rr</td>
<td>8.6B</td>
<td>9.4MB</td>
<td>915</td>
</tr>
<tr>
<td>Starcraft.rr</td>
<td>60M</td>
<td>1.8MB</td>
<td>33</td>
</tr>
</tbody>
</table>

TABLE I: ND log sizes for various replays

Record and Replay Runtime Overheads

<table>
<thead>
<tr>
<th>Environment</th>
<th>Time in sec</th>
<th>Slowdown wrt Qemu 2.1.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Qemu 2.1.0</td>
<td>35.6</td>
<td>1.0</td>
</tr>
<tr>
<td>PANDA</td>
<td>37.2</td>
<td>1.05</td>
</tr>
<tr>
<td>PANDA+record</td>
<td>66</td>
<td>1.85</td>
</tr>
<tr>
<td>PANDA+replay</td>
<td>127</td>
<td>3.57</td>
</tr>
</tbody>
</table>

TABLE II: PANDA, record, and replay slowdowns
PANDA execution and instrumentation

PANDA Plugins

- Tappan Zee (North) Bridge
- System Calls
- Shadow Callstack
- Taint Analysis
- Scissors
DroidScope:
Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis

Lok Yan
Heng Yin
August 10, 2012

Android

System Services  Java Components

Apps

Native Components
Motivation: Static Analysis

Dalvik/Java Static Analysis: ded, Dexpler, soot, Woodpecker, DroidMoss

Native Static Analysis: IDA, binutils, BAP
Motivation: Dynamic Analysis

Android Analysis: TaintDroid, DroidRanger

System Calls

logcat, adb

Motivation: Dynamic Analysis

External Analysis: Anubis, Ether, TEMU, ...
Goals

• 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
Roadmap

- External instrumentation
  - Linux context
  - Dalvik context
- Extensible: plugin-support / event-based interface
- Evaluation
  - Performance
  - Usage

Linux Context: Identify App(s)

- Shadow task list

```
+----------------+-----------------+----------------+-----------------+-----------------+-----------------+
| PID | TID | Parent PID | UID | GID | EUID | EGID | Parent PID | PGID | Comm          |
+----------------+-----------------+----------------+-----------------+-----------------+-----------------+
| 123 | 321 | 322 | 324 | 326 | 327 | 328 | 329 | 330 | con.android.media |
| 333 | 334 | 335 | 336 | 337 | 338 | 339 | 340 | 341 | con.android.calculator2 |
+----------------+-----------------+----------------+-----------------+-----------------+-----------------+
```
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 
    \( (dvmAsmInstructionStart + \text{opcode} \times 64) \)
  - dvmAsmSisterStart for emulation overflow
- Which Dalvik opcode?
  1. Locate dvmAsmInstructionStart in shadow memory map
  2. Calculate opcode = \((R15 - \text{dvmAsmInstructionStart}) / 64\).

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
Disabling JIT

Update Program Counter(PC)

dvmGetCodeAddr(PC) != NULL

Roadmap

☑ External instrumentation
  – Linux context
  – Dalvik context

➢ Extensible: plugin-support / event-based interface

• Evaluation
  – Performance
  – Usage
Instrumentation Design

- 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

Update PC

- inCache?
  - yes
  - (un)registerCallback
    - needFlush?
      - yes
        - flushType
          - invalidateBlock(s)
          - flushCache
      - no
        - Translate
          - Execute
  - no
    - Translate
      - Execute

- no
  - Translate
    - Execute
## Dalvik Instruction Tracer (Example)

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

---

### Dalvik Instruction Tracer

<table>
<thead>
<tr>
<th>NativeAPI</th>
<th>LinuxAPI</th>
<th>DalvikAPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction begin/end</td>
<td>context switch</td>
<td>Dalvik instruction begin</td>
</tr>
<tr>
<td>register read/write</td>
<td>system call</td>
<td>method begin</td>
</tr>
<tr>
<td>memory read/write</td>
<td>task begin/end</td>
<td></td>
</tr>
<tr>
<td>block begin/end</td>
<td>task updated</td>
<td></td>
</tr>
<tr>
<td></td>
<td>memory map updated</td>
<td></td>
</tr>
<tr>
<td>memory read/write</td>
<td>query symbol database</td>
<td></td>
</tr>
<tr>
<td>get current context</td>
<td>interpret Java object</td>
<td></td>
</tr>
<tr>
<td>task list</td>
<td>get/set DVM state</td>
<td></td>
</tr>
<tr>
<td>flag set/check</td>
<td>taint set/check objects</td>
<td></td>
</tr>
<tr>
<td></td>
<td>disable JIT</td>
<td></td>
</tr>
</tbody>
</table>
Plugins

- API Tracer
  - System calls
    - `open, close, read, write`, includes parameters and return values
  - Native library calls
  - Java API calls
    - Java Strings converted to C Strings
- Native and Dalvik Instruction Tracers
- Taint Tracker
  - Taints ARM instructions
  - One bit per byte
  - Data movement & Arithmetic instructions including barrel shifter
  - Does not support control flow tainting

Roadmap

- External instrumentation
  - Linux context
  - Dalvik context
- Extensible: plugin-support / event-based interface
- Evaluation
  - Performance
  - Usage
Implementation

• Configuration
  – QEMU 0.10.50 – part of Gingerbread SDK
  – Gingerbread
    • “user-eng”
    • No changes to source
  – Linux 2.6.29, QEMU kernel branch

Performance Evaluation

• Seven free benchmark Apps
  – AnTuTu Benchmark
  – (ABenchMark) by AnTuTu
  – CaffeineMark by Ravi Reddy
  – CF-Bench by Chainfire
  – Mobile processor benchmark (Multicore) by Andrei Karpushonak
  – Benchmark by Softweg
  – Linpack by GreeneComputing

• Six tests repeated five times each
  – Baseline
  – NO-JIT Baseline – uses a build with JIT disabled at runtime
  – Context Only
  – API Tracer
  – Dalvik Instruction Trace
  – Taint Tracker
Select Performance Results

Usage Evaluation

- Use DroidScope to analyze real world malware
  - API Tracer
  - Dalvik Instruction Tracer + dexdump
  - 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

- Three encrypted payloads
  - ratc (Rage Against The Cage)
  - killall (ratc wrapper)
  - gisvro (udev exploit)
- Three execution methods
  - piped commands to a shell (default execution path)
  - Runtime.exec() Java API (instrumented path)
  - JNI to native library terminal emulator (instrumented path)
  - Instrumented return values for isVersion221 and getPermission methods

Droid Kung Fu: TaintTracker

getDeviceId()

String @ 0x40524d80
"1234567890123456"

UrlEncodedFormEntity mime()

String @ 0x4056a448
"mime=1234567890123456&extype="

AbstractHttpClient.execute()

byte[] @ 0x405967d0 / void @ 0x405967d0
"POST search/key.php HTTP/1.1"

sys_write(34, 0x405967d0, 397)
DroidDream

- Same payloads as DroidKungFu
- Two processes
  - Normal `droiddream` process clears logcat
  - `droiddream:remote` is malicious
- xor-encrypts private information before leaking
- Instrumented `sys_connect` and `sys_write`
DroidDream: crypt trace

[DroidDream code]

Summary

- DroidScope
  - Dynamic binary instrumentation for Android
  - Built on Android Emulator in SDK
  - External Introspection & Instrumentation support
  - Four plugins
    - API Tracer
    - Native Instruction Tracer
    - Dalvik Instruction Tracers
    - TaintTracker
  - Partial JIT support
Related Works

- Static Analysis
  - ded, Dexpler, soot
  - Woodpecker, DroidMoss
- Dynamic Analysis
  - TaintDroid
  - DroidRanger
  - PIN, Valgrind, DynamoRIO
  - Anubis, TEMU, Ether, PinOS
- Introspection
  - Virtuoso
  - VMWatcher

Challenges

- JIT
  - Full JIT support
  - Flushing JIT cache
- Emulation detection
  - Real Sensors: GPS, Microphone, etc.
  - Bouncer
- Timing assumptions, timeouts, events
- Closed source systems, e.g. iOS
Questions?

Q0. Where can I get DroidScope?

ratc

• Vulnerability
  – setuid() fails when RLIMIT_NPROC reached
  – adbd fails to verify setuid() success

• Three generation (stage) exploit
  – Locate adbd in /proc and spawns child
  – Child fork() processes until -11 (-EAGAIN) is returned then spawns child – continues fork()
  – Grandchild kill() adbd and waits for process to respawn
ratc: exploit diagnosis

Symbol Information

- Native library symbols - Static
  - From `objdump` of libraries
- Java symbols - Dynamic
  - Dalvik data structures -> address of string
      - Given address, load from
          - Memory
          - File mapped into memory
      - `dexdump` as backup