

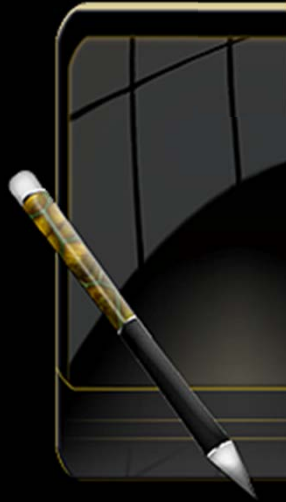
Triaging Crashes with Backward Taint Analysis for ARM Architecture



Dongwoo Kim scotty at home.cnu.ac.kr

Sangwho Kim whoyas2 at home.cnu.ac.kr

Index



Motivation

Related Work

Our Approach

Implementation

Conclusion

Who we are

- **Dongwoo Kim** : Hyeon-jeong Lee's Husband
 - Ph.D. Candidate at Chungnam National University in South Korea
 - Majoring in Computer Communications & Security
 - Interested in mobile hacking, digital forensics

- **Sangwho Kim** : Hye-ji Heo's Boyfriend
 - Master's course at the same school
 - Interested in mobile hacking, vulnerability analysis

Our purpose

- We want to find remote code execution vulnerabilities of real-world Android apps.
- Our targets are apps that consume file data like office file browser.
- We're especially interested in their native libraries that can cause crashes. 😊
- It's not a big deal to make targets get crashed using simple fuzzing.
- The problem is that it's a very time-consuming task to analyze crashes to determine exploitability. 😞

Our goal

1 / 2

- We need something that can let us know whether the operand is affected by the input in an automated manner. (Time is precious!)
- We tried to take advantage of any tools for it.
- However, there is nothing that we can use for our purpose on ARM architecture.
- We have decided to write our own tools using taint analysis based on dynamic binary instrumentation.

Our goal

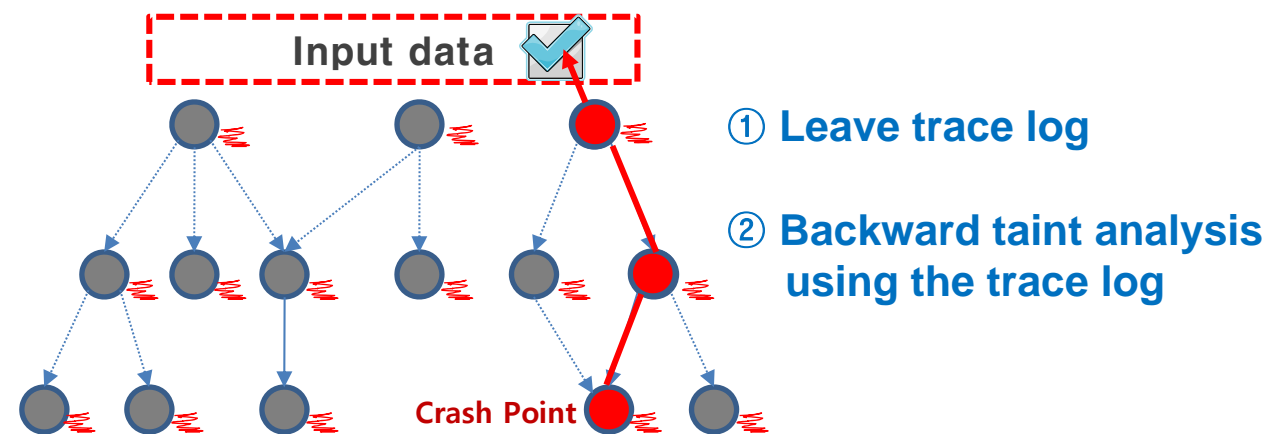
2 / 2

- Our tools should be easy to use on both Android emulator and device for practical use.
- We want our tools to answer the following questions.
 - Q. Operand at crash point is affected by input?
 - A. Yes or No!
 - Q. If yes, where is exactly coming from?
 - A. Offset 0x1004 in the input file

VDT (Visual Data Tracer)

1 / 3

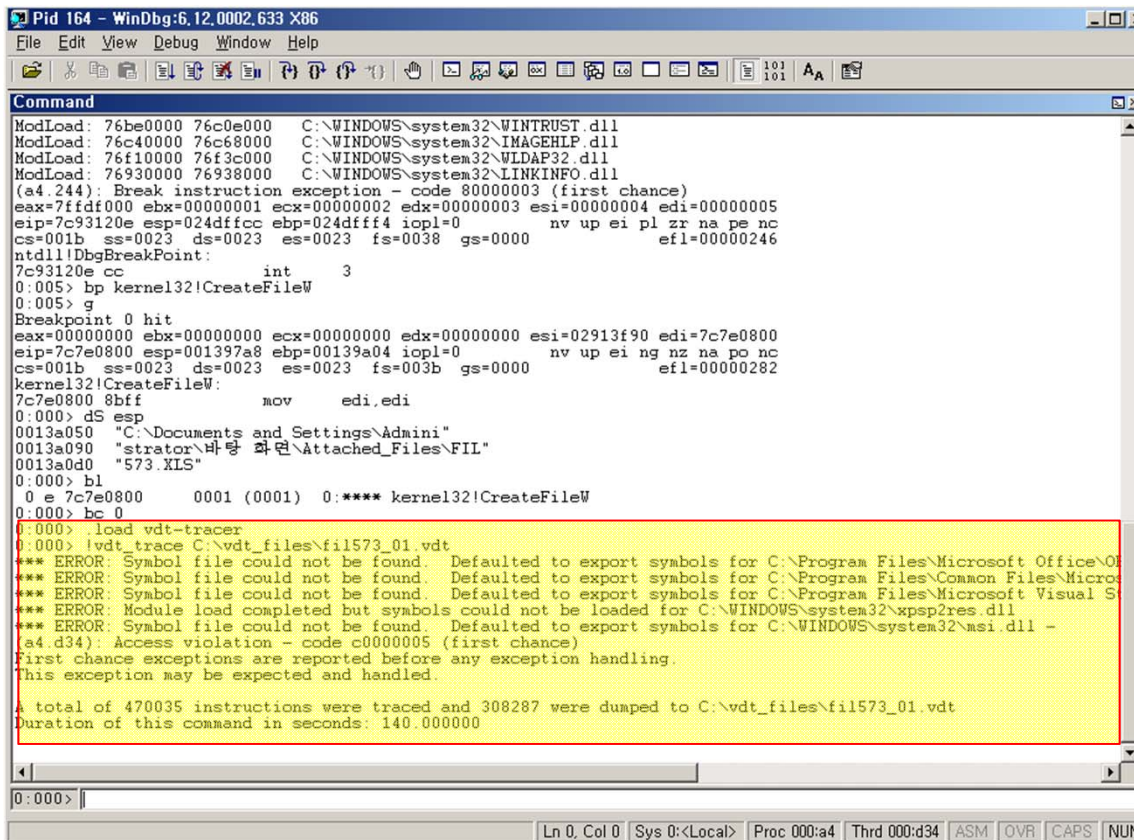
- *Triaging Bugs with Dynamic Dataflow Analysis* presented by Julio Auto at Source 2009 conference
- For crash analysis of user level applications on Windows OS (x86)
- Using taint analysis to determine exploitability



VDT (Visual Data Tracer)

2 / 3

- VDT-Tracer : Leave trace log (Extension of WinDBG)



```
Pid 164 - WinDbg:6.12.0002.633 X86
File Edit View Debug Window Help
Command
ModLoad: 76be0000 76c0e000 C:\WINDOWS\system32\WINTRUST.dll
ModLoad: 76c40000 76c68000 C:\WINDOWS\system32\IMAGEHELP.dll
ModLoad: 76f10000 76f3c000 C:\WINDOWS\system32\WLDAP32.dll
ModLoad: 76930000 76938000 C:\WINDOWS\system32\LINKINFO.dll
(a4.244): Break instruction exception - code 80000003 (first chance)
eax=7ffdf000 ebx=00000001 ecx=00000002 edx=00000003 esi=00000004 edi=00000005
eip=7c93120e esp=024dffcc ebp=024dff4 iopl=0         nv up ei pl zr na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
ntdll!DbgBreakPoint:
7c93120e cc                int     3
0:005> bp kernel32!CreateFileW
0:005> g
Breakpoint 0 hit
eax=00000000 ebx=00000000 ecx=00000000 edx=00000000 esi=02913f90 edi=7c7e0800
eip=7c7e0800 esp=001397a8 ebp=00139a04 iopl=0         nv up ei ng nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000282
kernel32!CreateFileW:
7c7e0800 8bff                mov     edi,edi
0:000> ds esp
0013a050 "C:\Documents and Settings\Admini"
0013a090 "strator\바탕 화면\Attached_Files\FIL"
0013a0d0 "573.XLS"
0:000> bl
0 e 7c7e0800 0001 (0001) 0:**** kernel32!CreateFileW
0:000> bc 0
0:000> .load vdt-tracer
0:000> !vdt_trace C:\vdt_files\fil573_01.vdt
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Program Files\Microsoft Office\O
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\Program Files\Microsoft Visual S
*** ERROR: Module load completed but symbols could not be loaded for C:\WINDOWS\system32\xpsp2res.dll
*** ERROR: Symbol file could not be found.  Defaulted to export symbols for C:\WINDOWS\system32\msi.dll -
(a4.d34): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.

A total of 470035 instructions were traced and 308287 were dumped to C:\vdt_files\fil573_01.vdt
Duration of this command in seconds: 140.000000
0:000>
```



VDT (Visual Data Tracer)

3 / 3

VDT-GUI : Backward taint analysis

```

Visual Data Tracer
File Analysis Help
308215 30086bc7 8b15ac958530 mov     edx,dword ptr [EXCEL!DllGetLCID+0x8d7e (308595ac)] ds:0
308216 30086bcd 8b12    mov     edx,dword ptr [edx] ds:0023:00f3f38=00f4a2c
308217 30086bcd 83e10f  and     ecx,0fh
308218 30086bd2 894dec  mov     dword ptr [ebp-14h],ecx ss:0023:001372d0=00000004
308219 30086bd5 8bcb    mov     ecx,ebx
308220 30086bd7 c19004  sar     ecx,4
308221 30086bda 034868  add     ecx,dword ptr [eax+68h] ds:0023:00f402c=00000000
308222 30086bdd 8b348a  mov     esi,dword ptr [edx+ecx+4] ds:0023:00f4a2c=0017a900
308223 30086be8 66b4e2c mov     cx,word ptr [esi+2Ch] ds:0023:0017a92c=0080
308224 30086c00 0fb4e22 movsx   ecx,word ptr [esi+22h] ds:0023:0017a922=0360
308225 30086c04 8b4510  mov     eax,dword ptr [ebp+10h] ss:0023:001372f4=00000360
308226 30086c07 83c708  add     edi,8
308227 30086c0c 897db0  mov     dword ptr [ebp-50h],edi ss:0023:00137294=00000000
308228 30086c0f 8945f4  mov     dword ptr [ebp-0Ch],eax ss:0023:001372d8=00000001
308229 30086c18 8b5514  mov     edx,dword ptr [ebp+14h] ss:0023:001372f8=00000360
308230 30086c1b 8bc1    mov     eax,ecx
308231 30086c1d 2b45f4  sub     eax,dword ptr [ebp-0Ch] ss:0023:001372d8=00000000
308232 30086c20 42     inc     edx
308233 30086c21 8945e4  mov     dword ptr [ebp-1Ch],eax ss:0023:001372c8=00000001
308234 30086c24 0fb4624 movsx   eax,word ptr [esi+24h] ds:0023:0017a924=0360
308235 30086c2a 8bfa    mov     edi,edx
308236 30086c32 8bdf    mov     ebx,edi
308237 30086c34 2bd8    sub     ebx,eax
308238 30086c36 2bc1    sub     eax,ecx
308239 30086c38 8bd0    mov     edx,eax
308287 77bf73c4 8b448ef0 mov     eax,dword ptr [ebp+10h] ss:0023:001372f4=00000360
Done!

```

▼ Instruction chain engaged in data flow

Analysis Results

OK

Possible source of taint found!
 Printing (possibly a part of) the tainting instruction: 300ce493 f3a5 rep movs dword ptr es:[edi],dword ptr [esi]
 Destination operand: +00138e00
 Source operand: +3085d40e

Printing dataflow path:

255383	300ce493 f3a5	rep movs dword ptr es:[edi],dword ptr [esi]
255532	300c6caa 0fb74e02	movsx ecx,word ptr [esi+2] ds:0023:00138e02=0360
255534	300c6cb6 51	push ecx
255539	300df7e2 8b542408	mov edx,dword ptr [esp+8] ss:0023:001379dc=00000360
255542	30120db1 52	push edx
255560	30086e1b ff7510	push dword ptr [ebp+10h] ss:0023:001379c8=00000360
257915	30086c04 8b4510	mov eax,dword ptr [ebp+10h] ss:0023:0013799c=00000360
257918	30086c0f 8945f4	mov dword ptr [ebp-0Ch],eax ss:0023:00137980=00000001
257956	30086cbc 8b4df4	mov ecx,dword ptr [ebp-0Ch] ss:0023:00137980=00000360
257958	30086cc0 8d04c8	lea eax,[eax+ecx*8]
257959	30086cc3 50	push eax
257962	30009acd ff742408	push dword ptr [esp+8] ss:0023:0013791c=00f5ce4
257969	77bf72b5 8b750c	mov esi,dword ptr [ebp+0Ch] ss:0023:00137910=00f5ce4
257977	77bf73c4 8b448ef0	mov eax,dword ptr [esi+ecx+4-10h] ds:0023:00f5ce4=????????

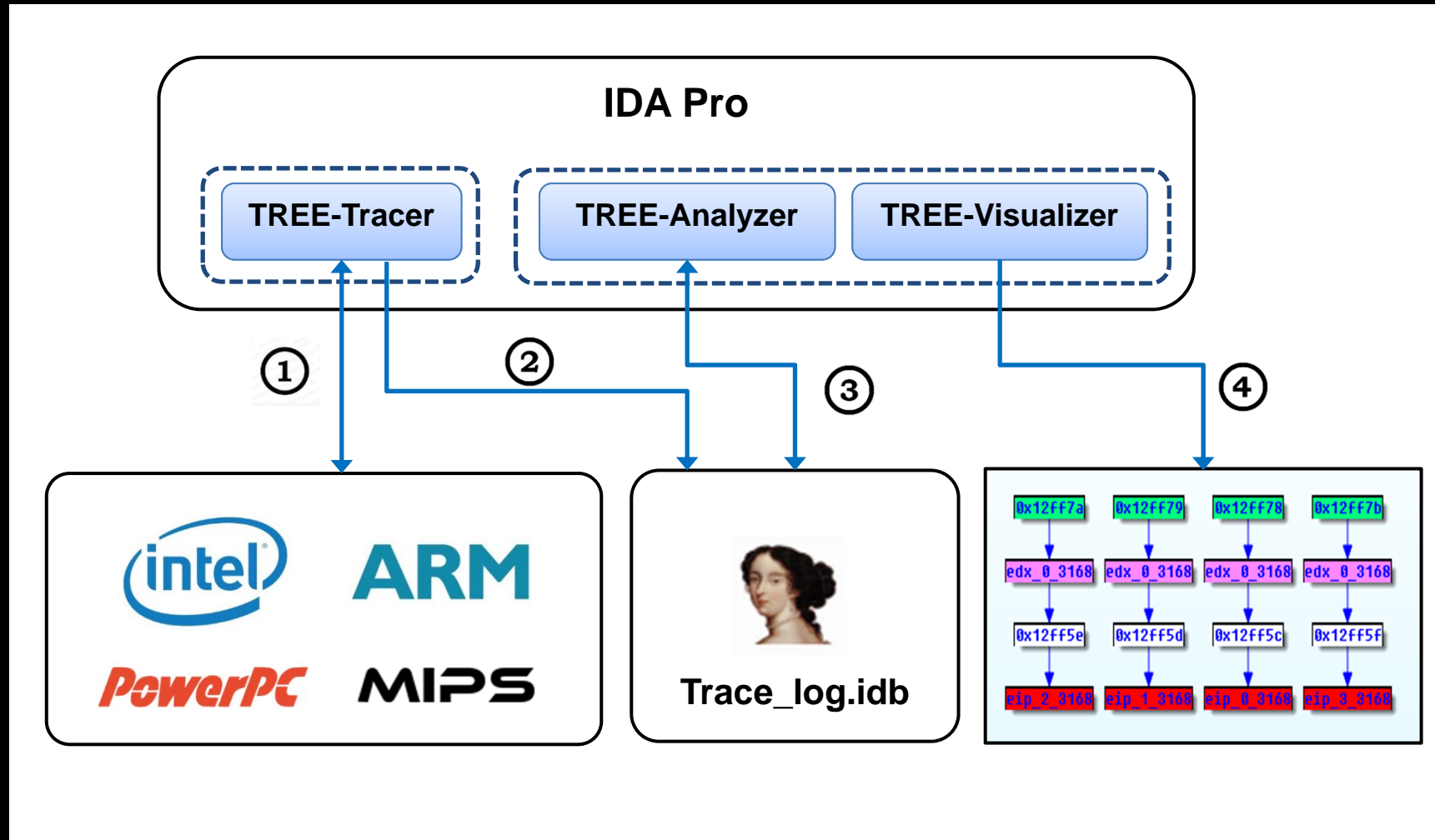
Check Taint Of ▶

Scroll To Item

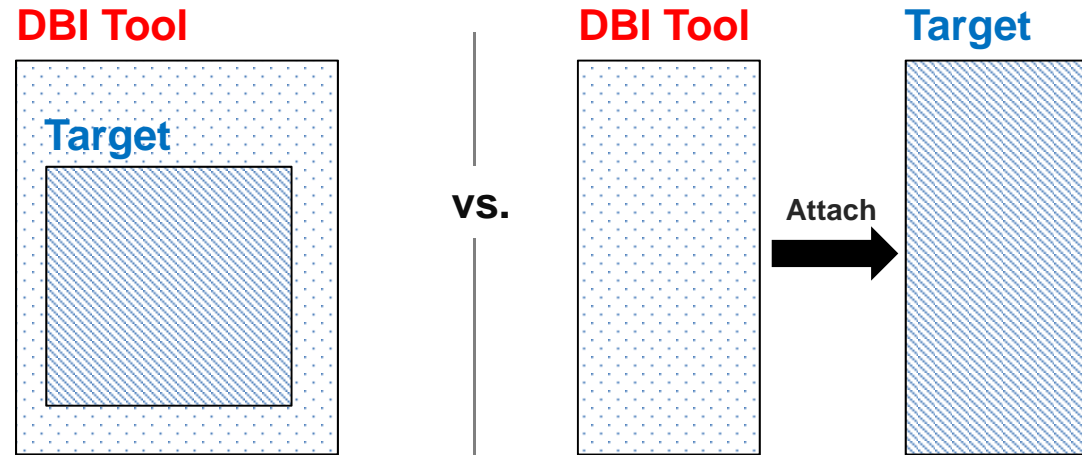
TREE (Tainted-enabled Reverse Engineering Environment) 1 / 2

- *Dynamic Analysis and Debugging of Binary Code for Security Applications* by Lixin Li and Chao Wang in 2013
- For crash analysis of user level applications on various architectures based on debugging feature of IDA Pro
- Using taint analysis to determine exploitability.

TREE (Tainted-enabled Reverse Engineering Environment) 2 / 2



Type of DBI (Dynamic Binary Instrumentation)



vs.

Type	- Same process	- Separate process ← Our choice!
Pros	- OS support NOT required (Low overhead)	- Appropriate for crash analysis
Cons	- NOT appropriate for crash analysis	- OS support required (High overhead)

Overview of our tools

```

[ ] READ_DATA : (00000000) 61616161 62626262 63636363 64646464 6565
6565
( 918)-[SWI] sys_close
<9> --- [24333 / b7f(2943)] IS THIS CRASH ?
(0) : *SIGSEGV /* Segmentation violation (ANSI). */ ---

r0=0xRED4C444 r01=0xRED4C54C r02=0x00000008 r03=0x0000000C
r04=0x732EFC0 r05=0x4153B138 r06=0x00000000 r07=0x41542DAB
r08=0xRED4C5B0 r09=0x41542DA0 r10=0x4153B148 r11=0x65656565
r12=0x00000000 sp=0xRED4C450 lr=0x732ECD74 pc=0x66666664
cpsr=0x40000010
[*] CRASH PC : 0x66666664 00 00 F0 01
- * - * - * - * - LOGGING FINISH - * - * - * - * -
FINAL status : b7f
Total Time : 0.000000
Total Instructions : 2201
Total Pans Instruction : 104 = 100(ARM) + 4(Thumb)
# of Instruction excutod : Inf
# of Lock Handler called : 10
Input Data File : /storage/emulated/0/test.txt
Input Data Size : 20
Input Data Memory : BED4C470 - BED4C483
Process PID, Target TID : 24333, 24333
Killed
137|shell@make:/data/local/tmp #
    
```



[State]	[Index]	[Address]	[Opcode]	[Disassembly]
[ARM]	2178	40086004	1D 03 80 F4	vst4.8 {d0[0], d1[0], d2[0], d3[0]}, [r0:0x...
[ARM]	2179	4008600C	00 07 21 F4	vld1.8 {d0}, [r1]!
[ARM]	2180	400860E9	1D 07 00 F4	vst1.8 {d0}, [r0:0x40]!
[ARM]	2181	400860E4	00 F0 01 F5	pld [r1]
[ARM]	2182	400860E8	40 F0 01 F5	pld [r1, #0x40]
[ARM]	2183	400860EC	40 20 52 E2	subs r2, r2, #0x40
[ARM]	2184	400860F9	09 00 00 3A	blo #0x2c
[ARM]	2185	40086E1C	40 20 82 E2	add r2, r2, #0x40
[ARM]	2186	40086E20	20 20 52 E2	subs r2, r2, #0x20
[ARM]	2187	40086E24	03 00 00 3A	blo #0x14
[ARM]	2188	40086E38	20 20 02 E2	add r2, r2, #0x20
[ARM]	2189	40086E3C	10 00 12 E3	tst r2, #0x10
[ARM]	2190	40086E40	01 00 00 0A	beq #0xc
[ARM]	2191	40086E4C	82 CE B0 E1	lsls ip, r2, #0x1d
[ARM]	2192	40086E54	00 07 21 F4	vld1.8 {d0}, [r1]!
[ARM]	2193	40086E58	00 07 00 F4	vst1.8 {d0}, [r0]!
[ARM]	2194	40086E5C	01 00 00 AA	bge #0xc
[ARM]	2195	40086E68	82 CF B0 E1	lsls ip, r2, #0x1f
[ARM]	2196	40086E84	01 40 80 E8	pop {r0, lr}
[ARM]	2197	40086E88	1E FF 2F E1	bx lr
[ARM]	2198	732ECD74	00 00 A0 E1	mov r0, r0
[ARM]	2199	732ECD78	04 00 48 E2	sub sp, fp, #4
[ARM]	2200	732ECD7C	00 88 80 E8	pop {fp, pc}
[ARM]	2201	66666664	00 00 F0 01	mvnseq r0, r0

① ARM-Tracer (Online)

- CLI Interface
- Working on 32bit ARM-based Linux (Android emulator and real device)
- Extracting context of every instruction until the target gets crashed

② ARM-Analyzer (Offline)

- GUI Interface
- Working on Desktop for efficiency
- Parsing trace.log and show the list of executed instructions
- Allowing a user to choose an object for backward taint analysis

Challenges in ARM-Tracer

- No hardware support for single-stepping whereas Intel x86 provides it known for trap flag.
 - We can implement it with DBM (Debug Breakpoint Mechanism).
- It requires various considerations which are not necessary in x86.
 - Such as calculating Next PC, handling signals in multi-threaded environment, handling atomic instruction sequence.

Challenges in ARM-Analyzer

- Not a simple task to identify semantic of ARM instructions in terms of data propagation, and distinguish their syntax.
- SIMD (Single Instruction Multiple Data) instruction set is very annoying!
- SIMD is for multimedia like SSE (Streaming SIMD Extensions) in x86 which has its own register bank that size is 256 bytes in total.

ARM-Tracer

1 / 8

- Instruction tracing with DBM
 - single-stepping using *ptrace* system call
 - Breakpoint instruction differentiate according to the instruction state

[Step 1] Determine Next PC

```

→ 0x1004 01 10 C0 24
   0x1008 01 00 BD E8
   0x100C 1E FF 2F E1
  
```

1. Analyze current instr.
0x1004 01 10 C0 24
2. Determine Next PC
Next PC = 0x1008

[Step 2] Set BP

```

→ 0x1004 01 10 C0 24
   0x1008 "Breakpoint"
   0x100C 1E FF 2F E1
  
```

3. Backup instr. at Next PC
0x1008 01 00 BD E8
4. Set BP at Next PC
0x1008 "Breakpoint"

[Step 3] Restore Instr.

```

   0x1004 01 10 C0 24
   → 0x1008 01 00 BD E8
   0x100C 1E FF 2F E1
  
```

5. Execute
0x1004 01 10 C0 24
6. Restore Instr.
0x1008 01 00 BD E8

ARM-Tracer

2 / 8

- Instruction state

State	Instruction	Size
ARM state	ARM instruction	32
Thumb state	Thumb instruction	16
	Thumb2 instruction	16/32

- Instruction state change (interworking) can happen by BX/BLX instructions.

ARM-Tracer

3 / 8

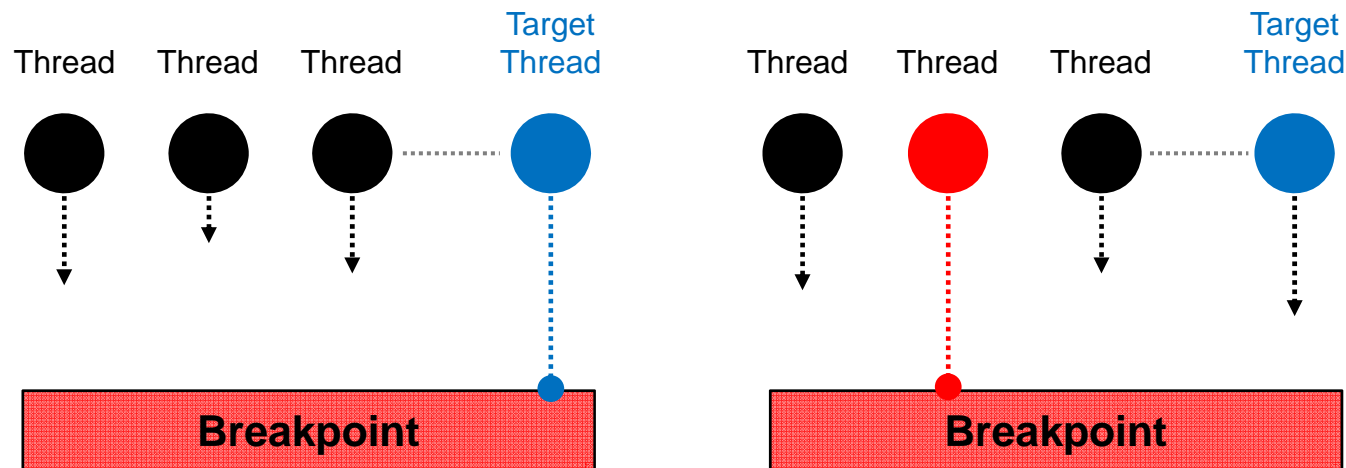
- Considerations on calculating Next PC
 - We have to identify opcode of instructions according to instruction state. (based on GDB)

ARM (32bit)	Thumb (16bit)	Thumb2 (16/32bit)
BLX #Offset	POP {(RegList,) PC}	B #Offset
BLX <Reg>	B #Offset	BL #Offset
BX <Reg>	BX <Reg>	BLX #Offset
LDR PC, [<Reg>]	BLX <Reg>	SUBS PC, LR, #Offset
LDM <Reg>, {(RegList,) PC}	MOV PC, <Reg>	LDMIA <Reg>, {(RegList)}
B #Offset	CBZ <Reg>, #Offset	LDMDB <Reg>, {(RegList)}
BL #Offset	CBNZ <Reg>, #Offset	RFEIA <Reg>
		RFEDB <Reg>
		MOV PC, <Reg>
		LDR PC, [<Reg>]
		TBB [<RegA>, <RegB>]
		TBH [<RegA>, <RegB>]

ARM-Tracer

4 / 8

- Addressing interference by other threads
 - Caused by code sharing



- We have to guarantee all the threads run properly.

ARM-Tracer

5 / 8

- Handling instruction sequence for atomic operation
 - ARM does not provide atomic instruction.
 - Instead, it provides sequence for it. (LDREX/STREX)
 - We should not intervene the sequence otherwise, it may cause infinite loop. ☹️

```

0x40918960 <dvmLockObject+56>: 8a b9  cbnz    r2, 0x40918986 <dvmLockObject+94>
0x40918962 <dvmLockObject+58>: 43 ea 08 02  orr.w   r2, r3, r8
0x40918966 <dvmLockObject+62>: 54 e8 00 cf  ldrex  r12, [r4]
0x4091896a <dvmLockObject+66>: 4f f0 00 00  mov.w  r0, #0
0x4091896e <dvmLockObject+70>: 9c ea 03 0f  teq    r12, r3
0x40918972 <dvmLockObject+74>: 08 bf  it     eq
0x40918974 <dvmLockObject+76>: 44 e8 00 20  strexeq r0, r2, [r4]
0x40918978 <dvmLockObject+80>: 00 28  cmp   r0, #0
0x4091897a <dvmLockObject+82>: f4 d1  bne.n 0x40918966 <dvmLockObject+62>
0x4091897c <dvmLockObject+84>: bf f3 5f 8f  dmb    sy

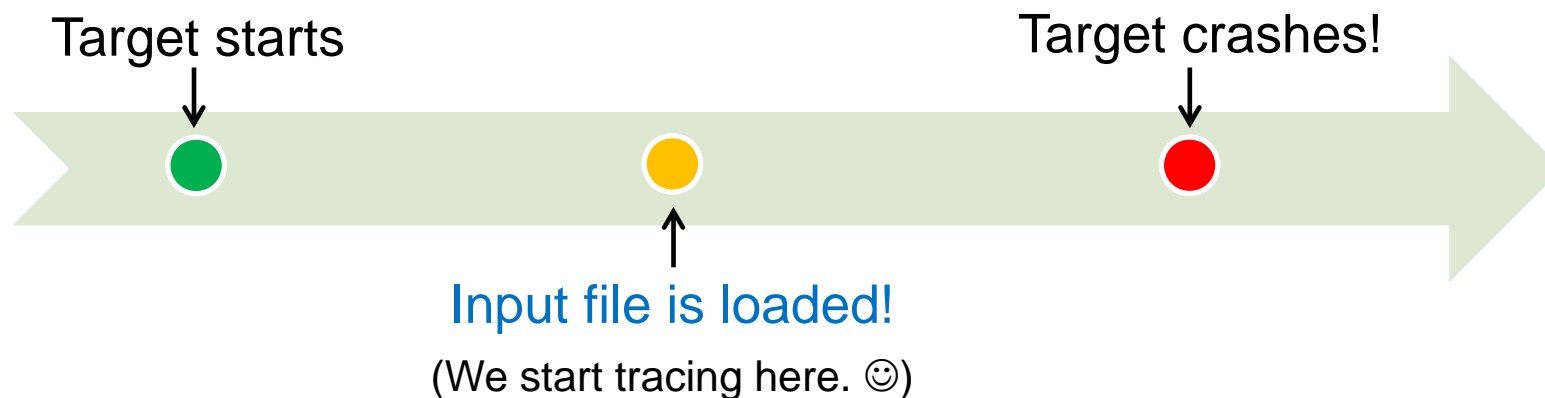
```

infinite loop

ARM-Tracer

6 / 8

- The “good” starting point
 - We designate a specific thread as the target thread which opens the input file.
 - We can know memory address where the input file is loaded by checking open and read functions.

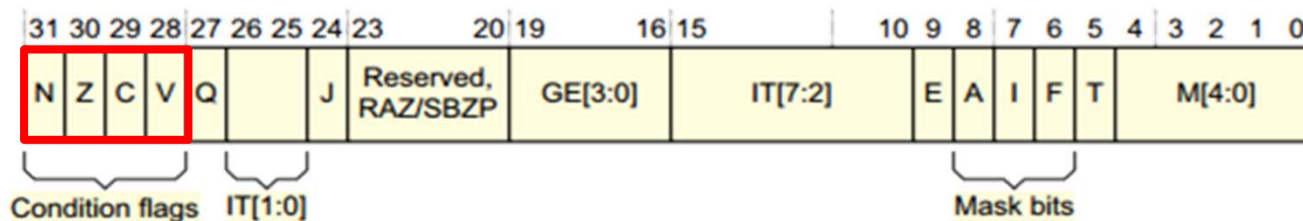


ARM-Tracer

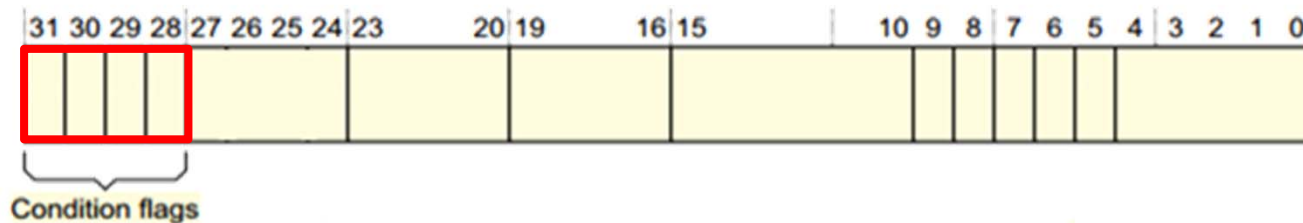
7 / 8

- Before logging, filter out instructions not executed (ARM)

[CPSR]



[ARM Instruction]



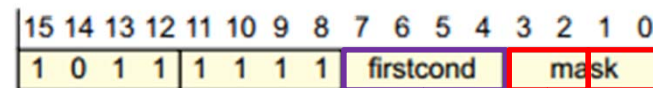
ARM-Tracer

8 / 8

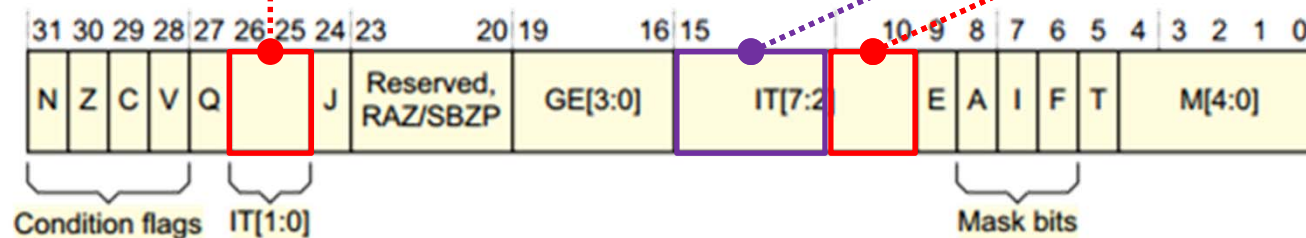
- Before logging, filter out instructions not executed (Thumb2)

AB BF	ITETE GE
23 6D	LDRGE
A3 89	LDRLTH
1B 18	ADDGE
23 F4 80 53	BICLT.W

[Thumb2 – IT Instruction]



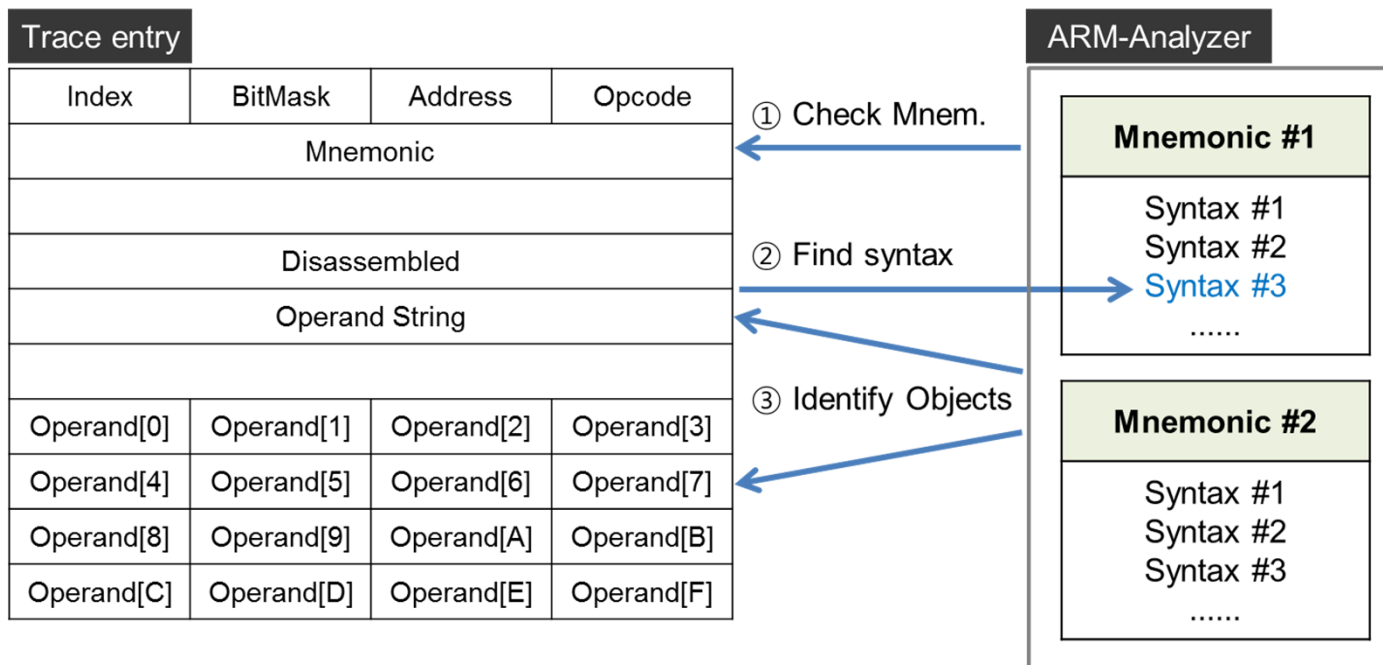
[CPSR]



ARM-Analyzer

1 / 3

- Parsing each entry from the trace log file
 - Identify instruction syntax based on disassembly
 - Identify object : register, memory address (byte level)



ARM-Analyzer

2 / 3

- Classification of instructions

- ARM Architecture Reference Manual ARMv7-A Edition

Group	Mnemonic	Target	Syntax	Impl.
Memory access	16	8	39	54
General data processing	32	27	37	70
Multiply	25	22	22	28
Saturating	6	6	6	10
Parallel	4	4	4	5
Packing and unpacking	10	10	10	28
Branch and control	10	0	0	0
Coprocessor	14	0	0	0
Total	117	77	118	195


- We have also considered some SIMD instructions (vld, vst).

ARM-Analyzer

3 / 3

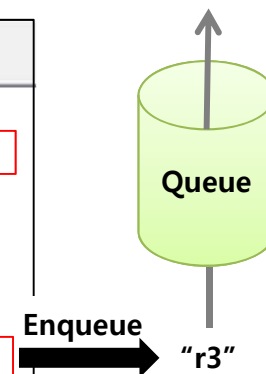
How it works – Backward taint analysis

View for user

[Index]	[Address]	[Opcode]	[Disassembly]
6926429	78CB8788	50 60 80 E2	add r6, r0, #0x50
6926430	78CB878C	00 30 91 E5	ldr r3, [r1]
6926431	78CB8790	01 50 A0 E1	mov r5, r1
6926432	78CB8794	00 40 A0 E1	mov r4, r0
6926433	78CB8798	01 00 A0 E1	mov r0, r1
6926434	78CB879C	06 10 A0 E1	mov r1, r6
6926435	78CB87A0	18 30 93 E5	ldr r3, [r3, #0x18]  Crash

Inside of ARM-Analyzer

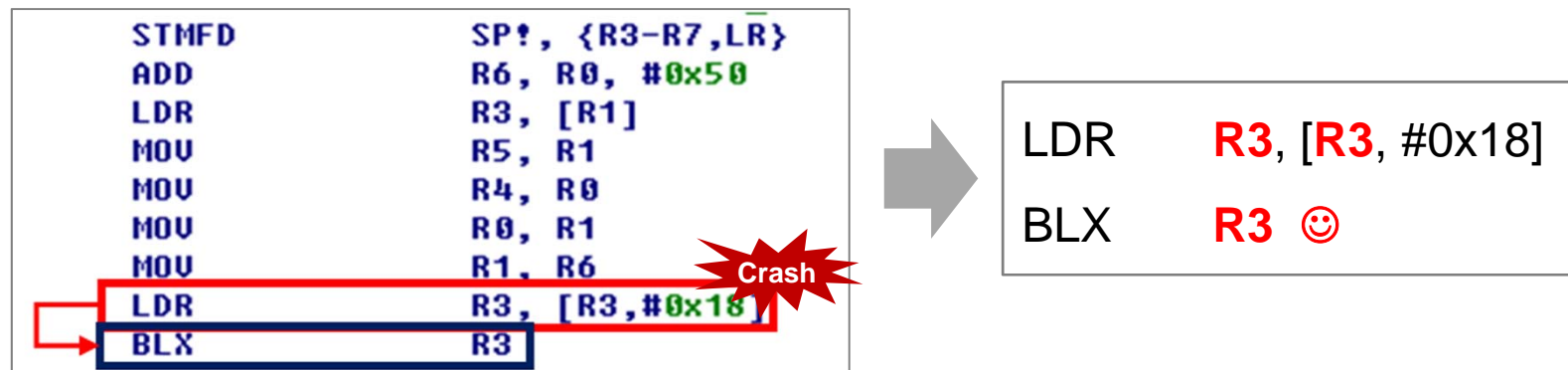
[Index]	[Address]	[Opcode]	[Disassembly]
6926429	78CB8788	50 60 80 E2	add Dst : r6 / Src : r0
② 6926430	78CB878C	00 30 91 E5	ldr Dst : r3 / Src : r1, *0x2224
6926431	78CB8790	01 50 A0 E1	mov Dst : r5 / Src : r1
6926432	78CB8794	00 40 A0 E1	mov Dst : r4 / Src : r0
6926433	78CB8798	01 00 A0 E1	mov Dst : r0 / Src : r1
6926434	78CB879C	06 10 A0 E1	mov Dst : r1 / Src : r6
① 6926435	78CB87A0	18 30 93 E5	ldr Dst : r3 / Src : r3, *0x1018



Experiment

1 / 3

- We generated crashes against Polaris Office 6.0.1.
- Among them, we chose 7 crashes that look cool!
- Such as..



- Let's try to put them into our tools!

Experiment

2 / 3

- Tested on GalaxyS4
 - 2.3 GHz Quad-core, 2GB RAM, Android 4.4.2, Kernel 3.4.0

ARM-Tracer	Crash 1	Crash 2	Crash 3	Crash 4	Crash 5	Crash 6	Crash 7
# of instructions executed	6,804,072	6,830,983	7,008,764	7,048,261	10,000,000+	10,000,000+	10,000,000+
# of instructions filtered out	585,093	584,841	601,177	607,208	900,000+	900,000+	900,000+
# of atomic handler	2,600	2,600	2,662	2,630	3,800+	3,800+	3,800+
Taken time (sec)	1,563	1,562	1,616	1,673	2,300+	2,300+	2,300+
Dump file size (MB)	1,038	1,042	1,069	1,075	1,500+	1,500+	1,500+

Experiment

3 / 3

- Tested on Desktop
 - 3.3 GHz Quad-core, 16GB RAM, Windows 7

ARM-Analyzer		Crash 1	Crash 2	Crash 3	Crash 4
Probably Exploitable		X	O	X	O
# of instructions executed		6,804,072	6,830,983	7,008,764	7,048,261
Taken time to full scan	Fast Mode	10 ~ 15 sec			
	Normal Mode	A couple of days..... ☹			

- Fast Mode enqueues only **effective address** of source into the search queue.
 - ex) `LDR R1, [R2, R3]` \rightarrow `*(R2+R3) // 0x1004`
 0x1000 0x4



DEMO

ARM-Tracer + ARM-Analyzer → Exploitable Crash 😊

- We have developed tools for crash analysis of user-level applications on ARM architecture.
 - It can avoid non-deterministic behavior.
 - We can efficiently analyze crashes in a limited time.
- We have tested it with real-world app on Android device.
 - As a result, we got two exploitable crashes after short testing our tools with crash samples that we have already generated.
- Before long, we're going to release our tools with source code after some revisions for those who are interested in them.
 - Please participate in improving our tools.

Q&A

Thank you 😊