

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

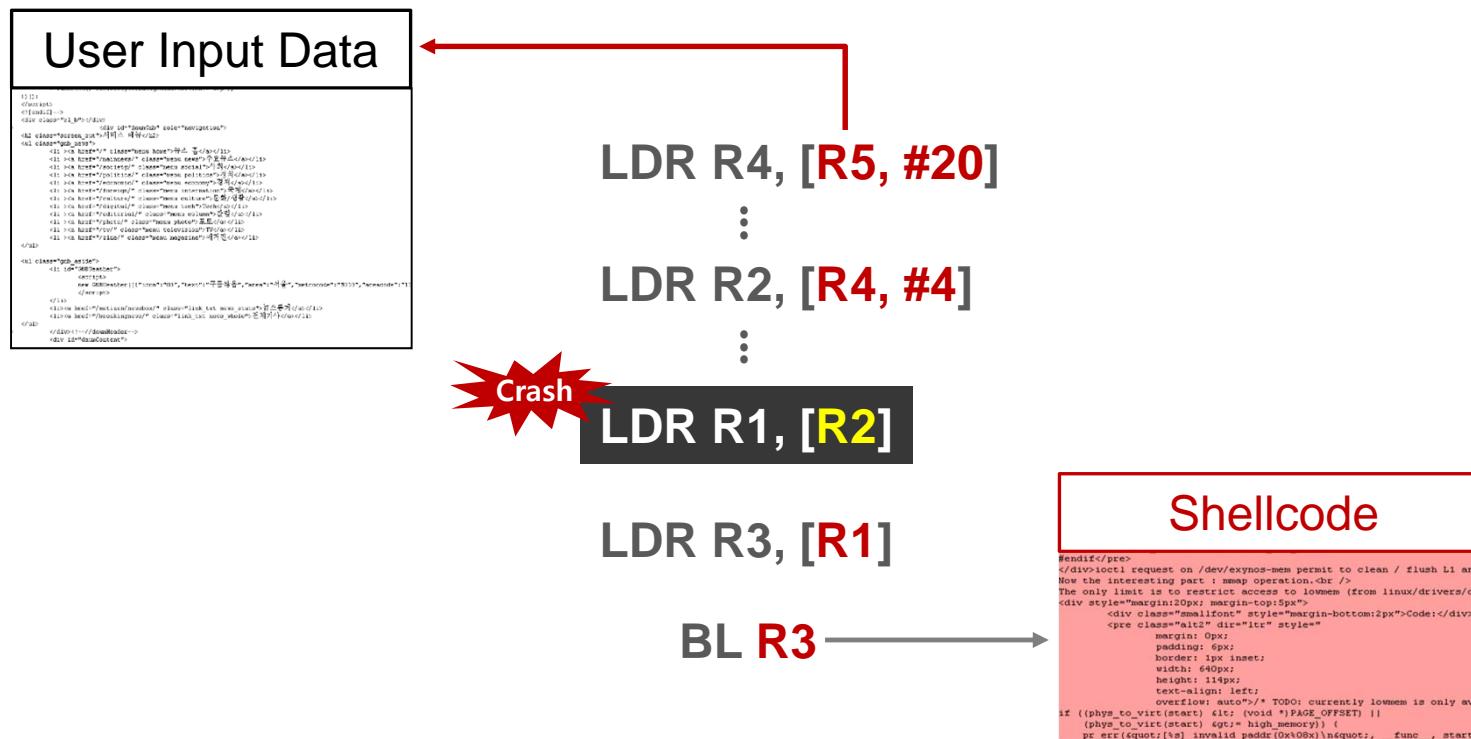
- **Dongwoo Kim** : Hyeon-jeong Lee's Husband
 - Ph.D. Candidate at Chungnam National University in South Korea
 - Majoring in Computer Communications & Security
 - Interested in mobile hacking, digital forensics
- **Sangwho Kim** : Hye-ji Heo's Boyfriend
 - Master's course at the same school
 - Interested 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. ☹

How to determine exploitability

- We have to MANUALLY figure out that the operand at crash point is affected by the user input.



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

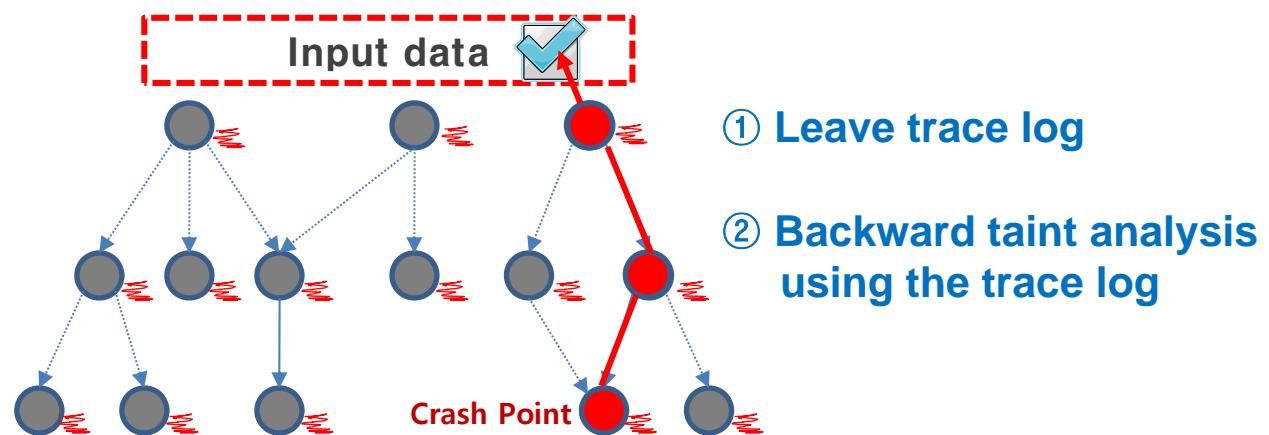
2 / 2

- 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\IMAGEHLP.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 edx=00000002 esi=00000004 edi=00000005
eip=7c93120e esp=024df1cc ebp=024dff14 icpl=0 nv up ei pl nz 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=00139a04 ebp=00139a04 icpl=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> d$ esp
0013a50 "C:\Documents and Settings\Admini"
0013a90 "strator\바탕 화면\Attached_Files\FIL"
0013ad0 "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\f1573_01.vdt
*** ERROR: Symbol file could not be found. Defaulted to export symbols for C:\Program Files\Microsoft Office\Offi...
*** ERROR: Symbol file could not be found. Defaulted to export symbols for C:\Program Files\Common Files\Micro...
*** 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\f1573_01.vdt
Duration of this command in seconds: 140.000000
```



trace.log

VDT (Visual Data Tracer)

3 / 3

- VDT-GUI : Backward taint analysis

The screenshot shows the Visual Data Tracer (VDT) interface. On the left is a window titled "Visual Data Tracer" displaying assembly code. The assembly code is as follows:

```

308215. 30096bc7 8b15ac958530 mov edx,dword ptr [EXCEL!DIIGetLCID+0x8d7e (308595ac)] ds:0
308216. 30096bcd 8b12 mov edx,dword ptr [edx] ds:0023:00f3138=00f4a2c
308217. 30096bcf 83e10f and ecx,0fh
308218. 30096bd2 894dec mov dword ptr [ebp-14h],ecx ss:0023:001372d0=00000004
308219. 30096bd5 8bcb mov ecx,ebx
308220. 30096bd7 c1f904 sar ecx,eax
308221. 30096bd8 034068 add ecx,dword ptr [eax+60h] ds:0023:000402c=00000000
308222. 30096bd9 8b348a mov esi,dword ptr [edx+ecx*4] ds:0023:00f4a2c=0017a900
308223. 30096be8 666b4e2c mov cx,word ptr [esi+2Ch] ds:0023:0017a92c=0080
308224. 30096c00 8bf4e22 movsx ecx,word ptr [esi+22h] ds:0023:0017a922=0360
308225. 30096c04 8b4510 mov eax,dword ptr [ebp+10h] ss:0023:0013724=00000360
308226. 30096c07 83c708 add edi,8
308227. 30096c0c 897db0 mov dword ptr [ebp-50h],edi ss:0023:00137294=00000001
308228. 30096c0f 8945f4 mov dword ptr [ebp-0Ch],eax ss:0023:001372d8=00000001
308229. 30096c18 8b5514 mov edx,dword ptr [ebp+14h] ss:0023:00137218=00000360
308230. 30096c1b 8bc1 mov eax,ecx
308231. 30096c1d 2b45f4 sub eax,dword ptr [ebp-0Ch] ss:0023:001372d8=0000
308232. 30096c20 42 inc edx
308233. 30096c21 8945e4 mov dword ptr [ebp-1Ch],eax ss:0023:001372c8=0000
308234. 30096c24 0fb4f624 movsx eax,word ptr [esi+24h] ds:0023:0017a924=03
308235. 30096c29 8bf7fa mov edi,edx
308236. 30096c32 8bdf1 mov ebx,edi
308237. 30096c34 2b2d8 sub ebx,eax
308238. 30096c36 2bc1 sub eax,ecx
308239. 30096c38 8b2d0 mov edx,eax
308287. 77bf73c4 8b448ef0 mov eax,dword ptr [ebp-10h] ss:0023:001372d8=0000

```

A context menu is open over the assembly code, showing options: "Check Taint Of" and "Scroll To Item".

To the right, a modal dialog box titled "Analysis Results" is displayed. It contains the following text:

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 movzx 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:0023:001379c8=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=00ff5ce4
257969. 77bf72b5 8b750c mov esi,dword_ptr [ebp+0Ch] ss:0023:00137910=00ff5ce4
257977. 77bf73c4 8b448ef0 mov eax,dword_ptr [esi+ecx*4-10h] ds:0023:00ff5ce4=???????

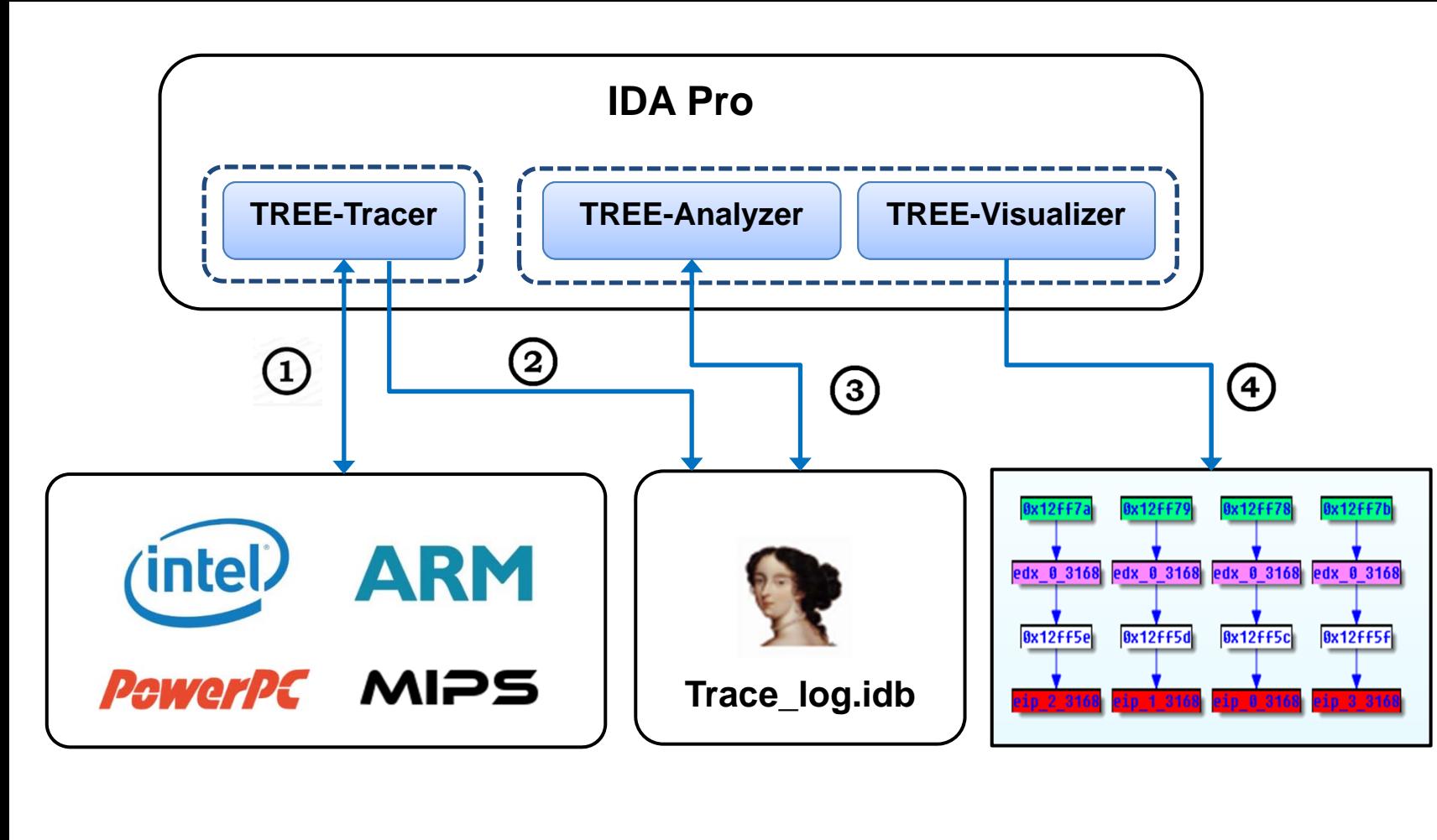
```

An arrow points from the text "▼ Instruction chain engaged in data flow" to the "Analysis Results" dialog.

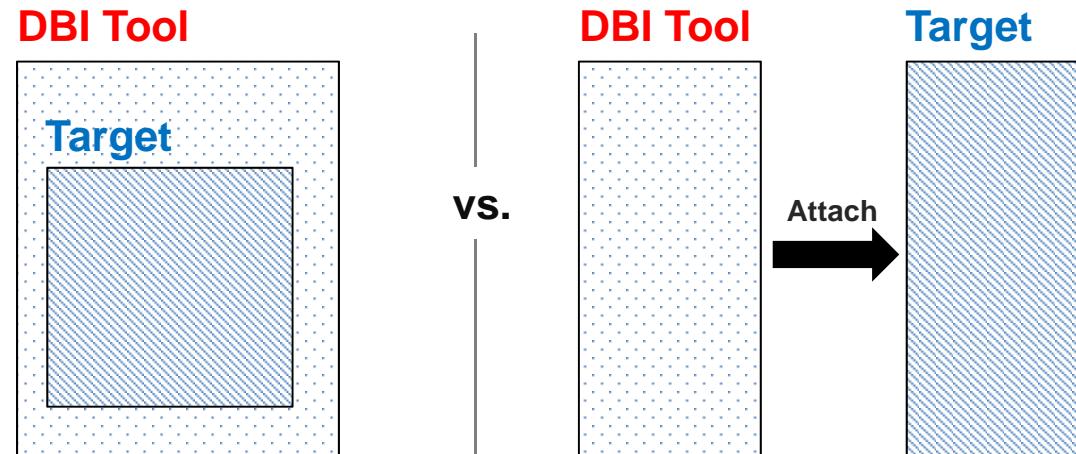
TREE (Tainted-enabled Reverse Engineering Environment) 1 / 2

- *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

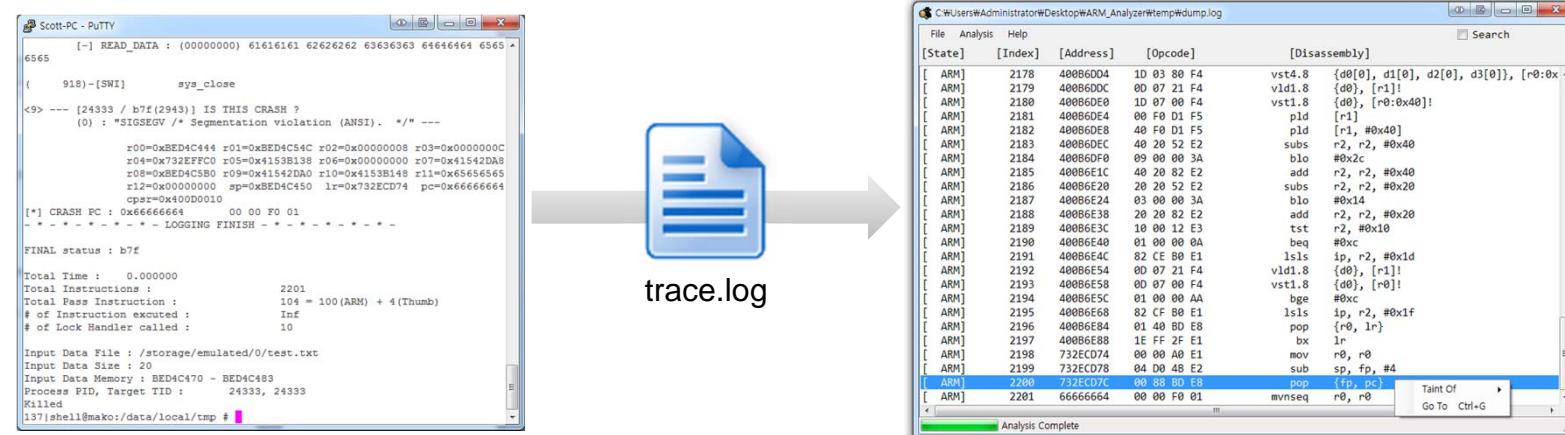


Type of DBI (Dynamic Binary Instrumentation)



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



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

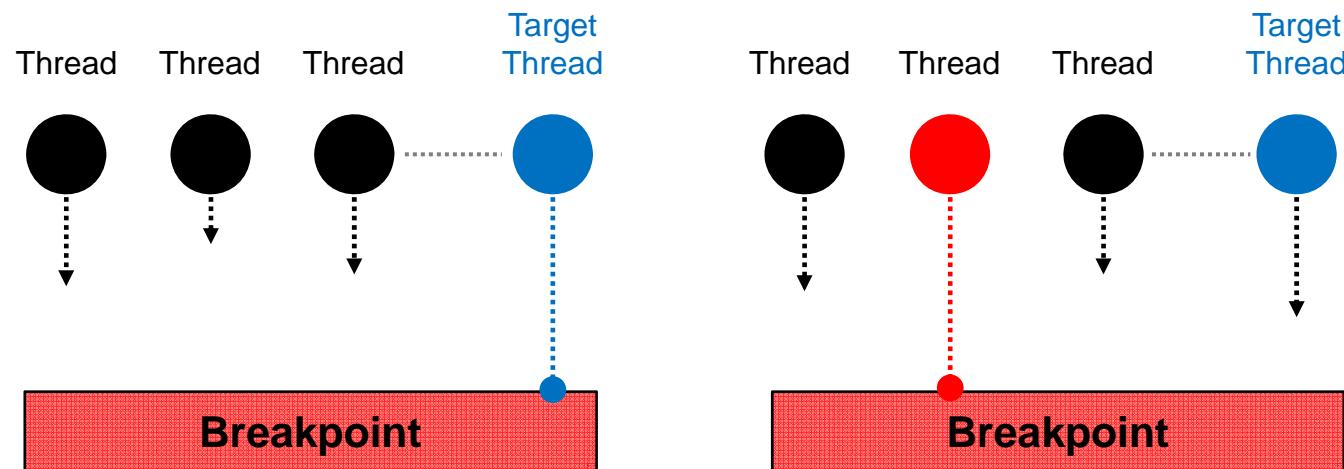
- Considerations on calculating Next PC
 - We have to identify opcode of instructions according to instruction state. (based on GDB)

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

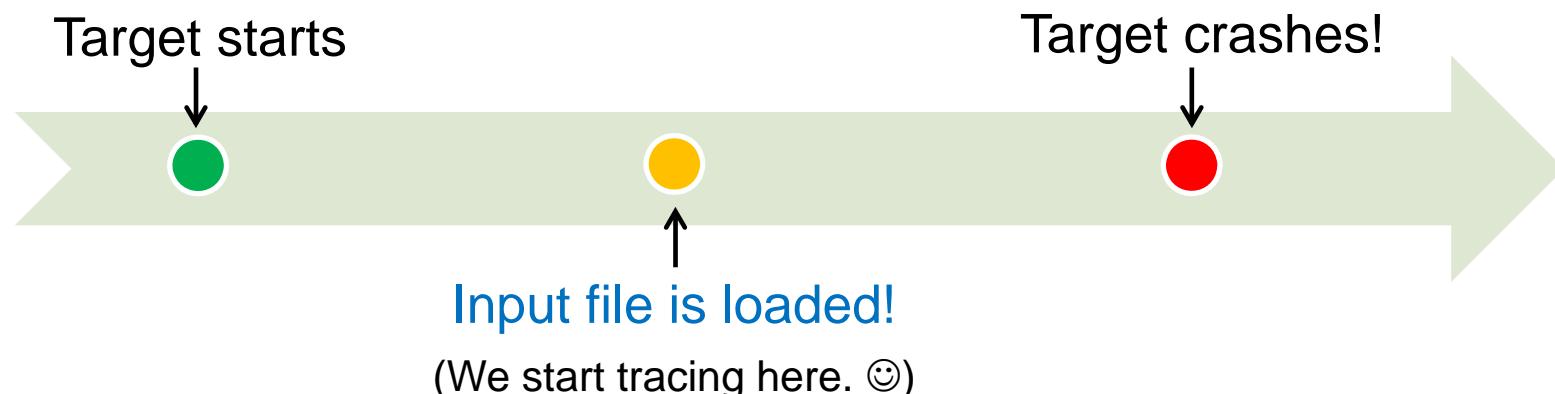
- Handling instruction sequence for atomic operation
 - ARM does not provide atomic instruction.
 - Instead, it provides sequence for it. (LDREX/STREX)
 - We should not intervene the sequence otherwise, it may cause infinite loop. ☺

```
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] ← infinite loop
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
```

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.

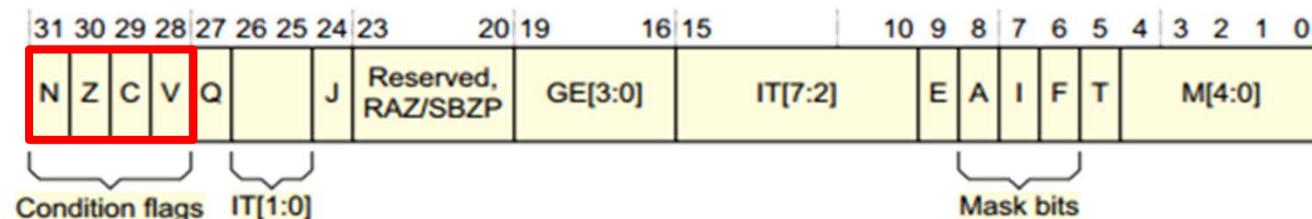


ARM-Tracer

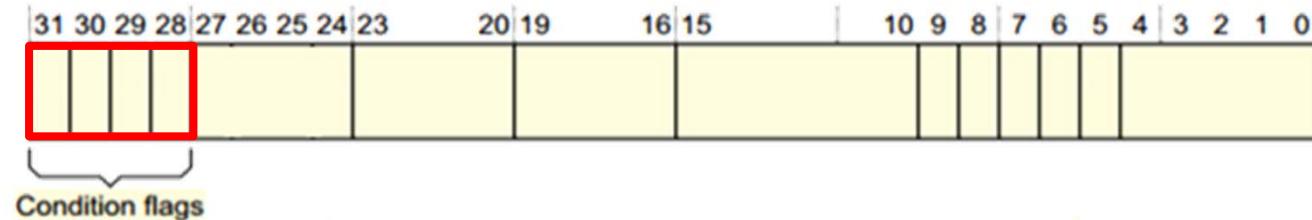
7 / 8

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

[CPSR]



[ARM Instruction]



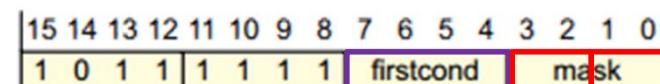
ARM-Tracer

8 / 8

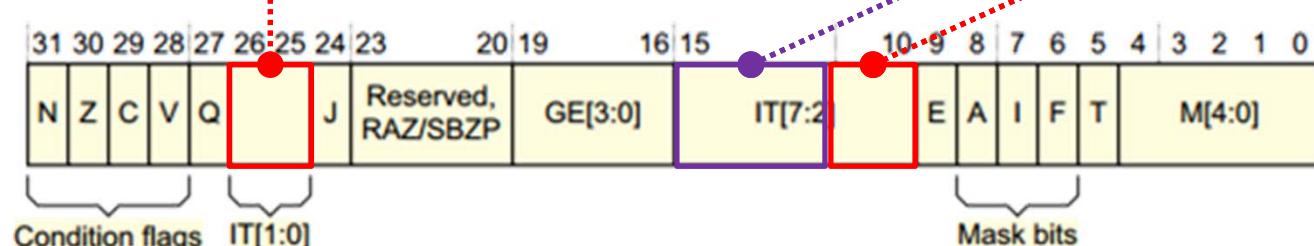
- Before logging, filter out instructions not executed (Thumb2)

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

[Thumb2 – IT Instruction]



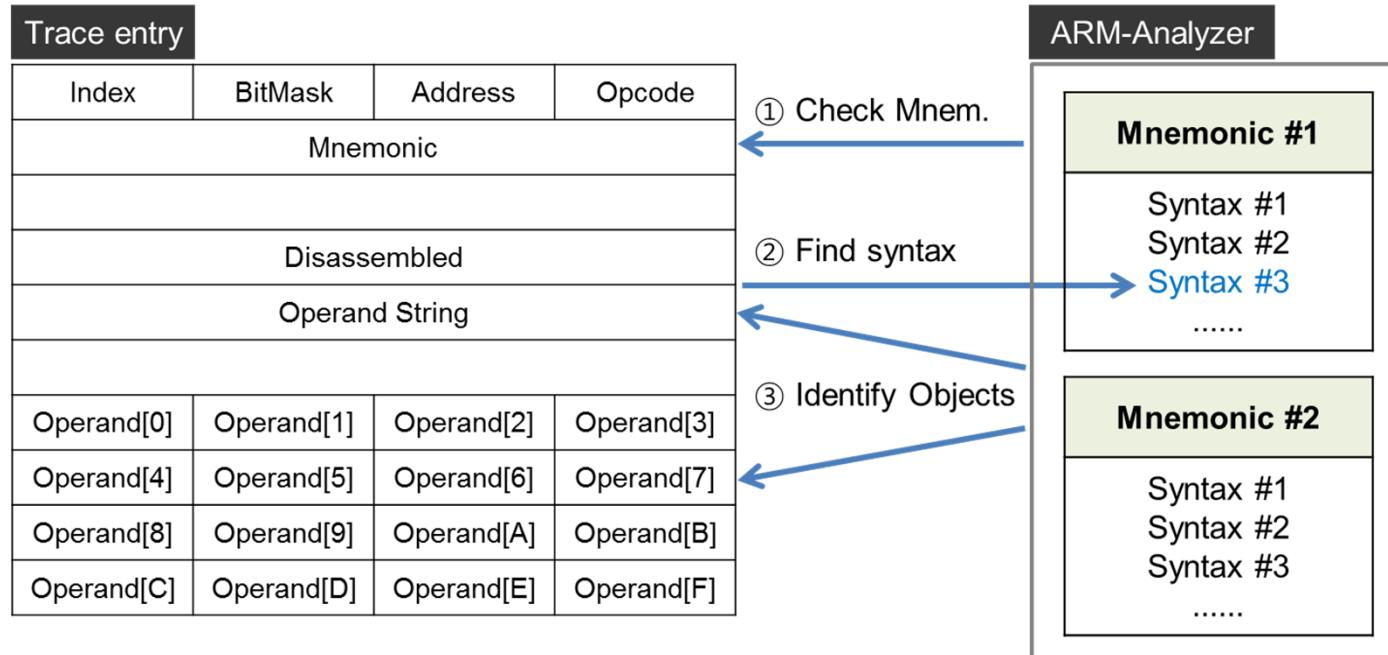
[CPSR]



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

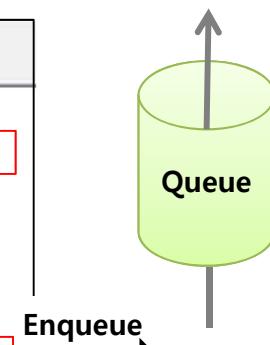
- How it works – Backward taint analysis

▼ 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

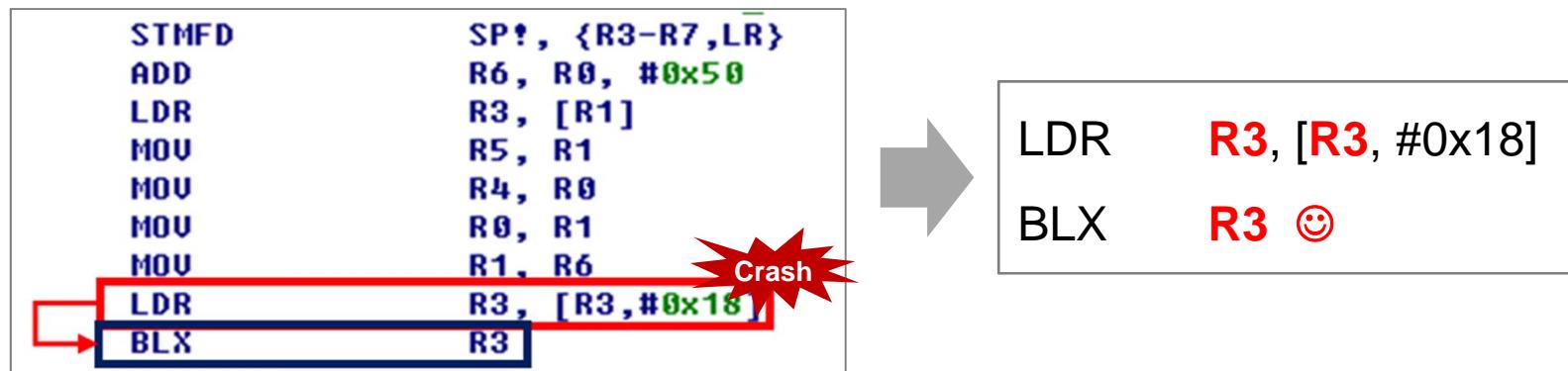
[Index]	[Address]	[Opcode]	[Disassembly]
6926429	78CB8788	50 60 80 E2	add Dst : r6 / Src : r0
② 6926430	78CB878C	00 30 91 E5	ldr Dst : r3 / Src : r1, *0x2224
6926431	78CB8790	01 50 A0 E1	mov Dst : r5 / Src : r1
6926432	78CB8794	00 40 A0 E1	mov Dst : r4 / Src : r0
6926433	78CB8798	01 00 A0 E1	mov Dst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

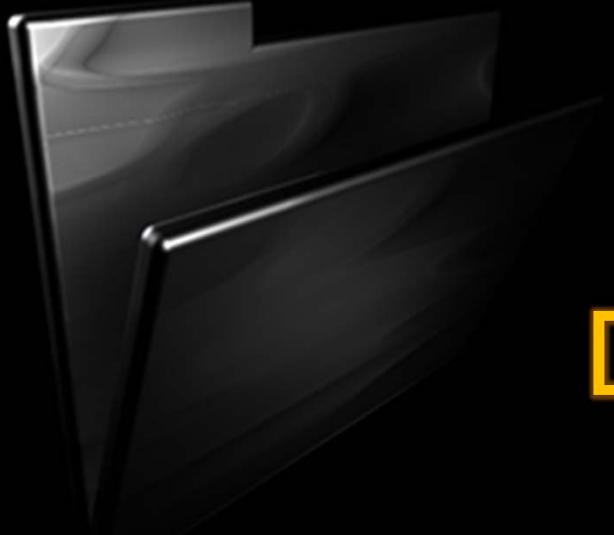
Experiment

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] → *(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 ☺