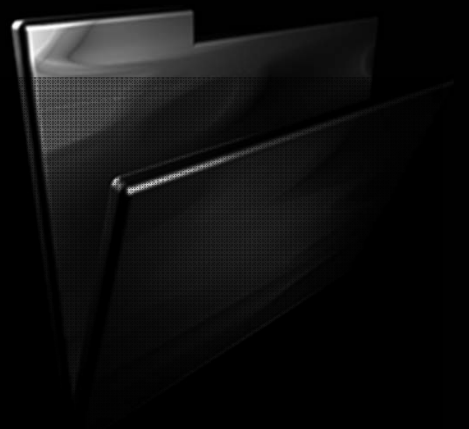
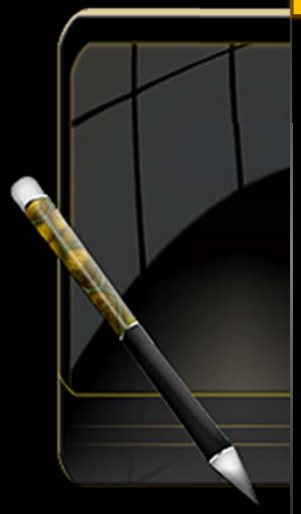


Triaging Crashes with Backward Taint Analysis for ARM Architecture

Dongwoo Kim scotty at home.cnu.ac.kr
Sangwho Kim whoyas2 at home.cnu.ac.kr

ISLAB
INFORMATION SECURITY LABORATORY CNU

Index



Who we are

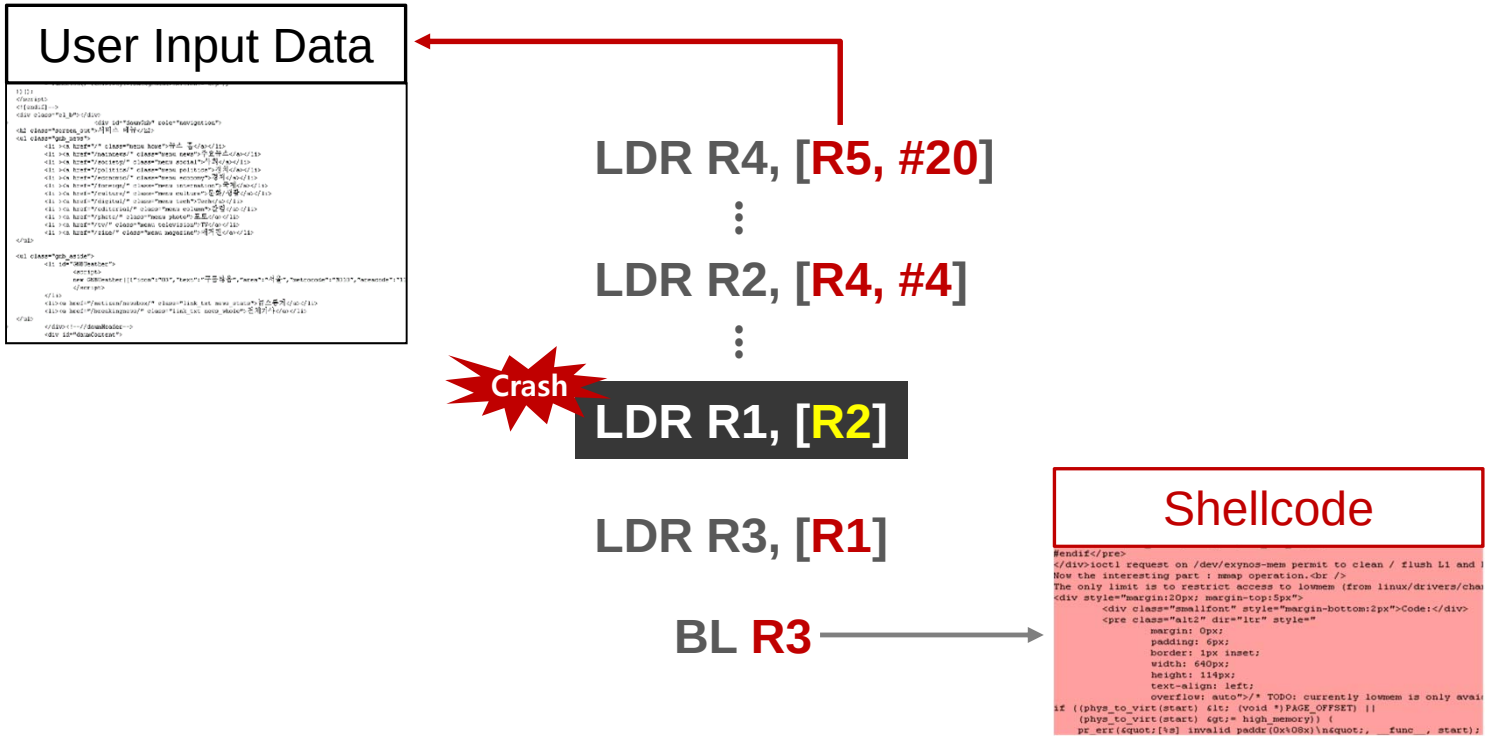
- **Dongwoo Kim** : Hyeon-jeong Lee's Husband
  - Ph.D. Candidate at Chungnam National University in South Korea
  - Majoring in Computer Communications & Security
  - Interested in mobile hacking, digital forensics
- **Sangwho Kim** : Hye-ji Heo's Boyfriend
  - Master's course at the same school
  -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

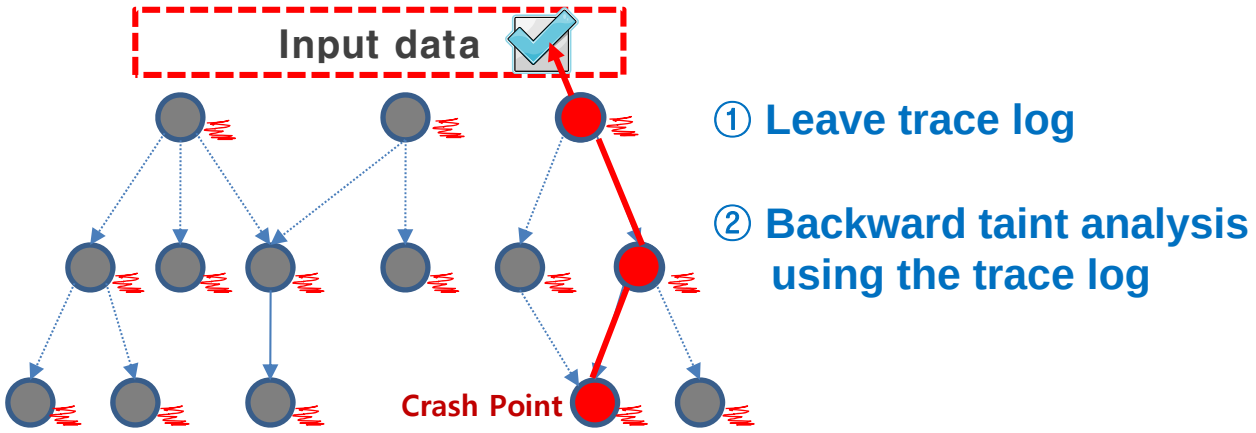
- We need something that can let us know whether the operand is affected by the input in an automated manner. (Time is precious!)
- We tried to take advantage of any tools for it.
- However, there is nothing that we can use for our purpose on ARM architecture.
- We have decided to write our own tools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

Our goal

- 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

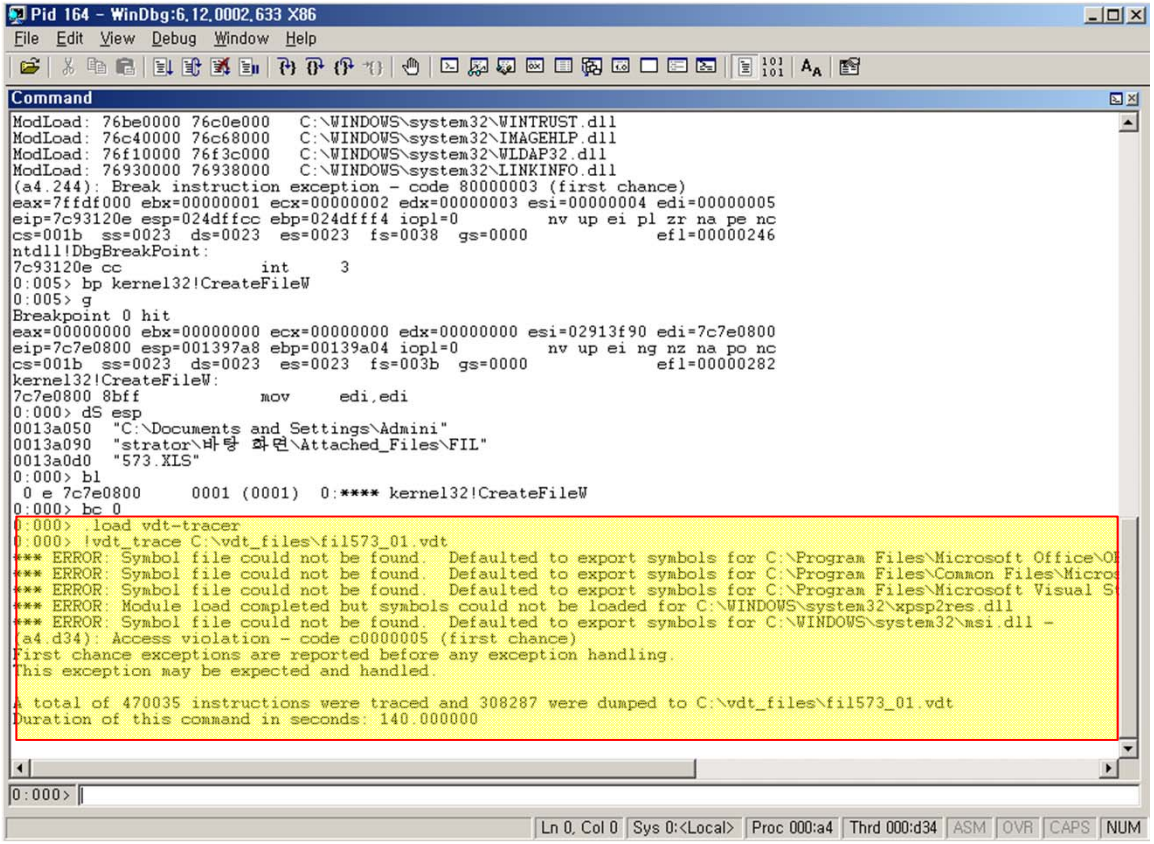
VDT (Visual Data Tracer) 1 / 3

- *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



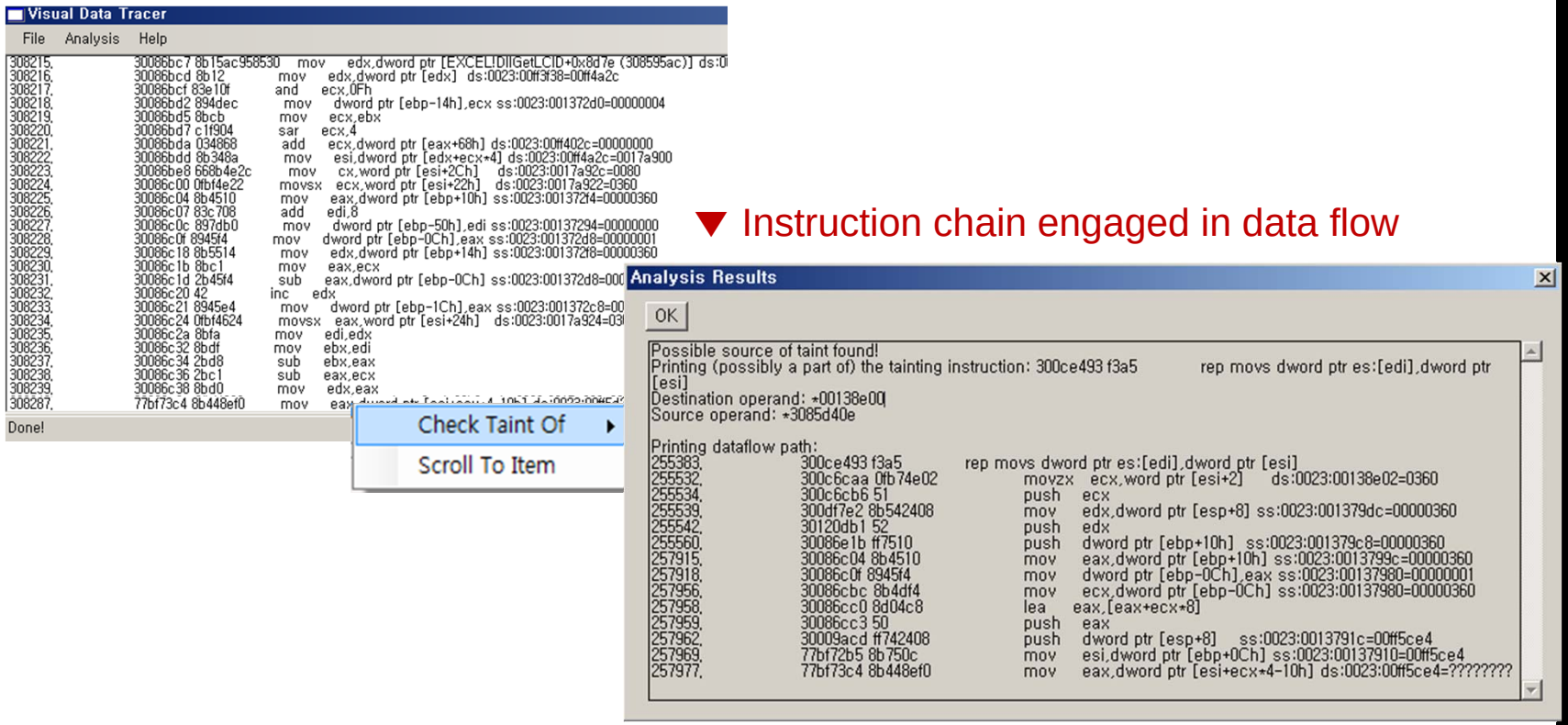
VDT (Visual Data Tracer)

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



VDT (Visual Data Tracer) 3 / 3

- VDT-GUI : Backward taint analysis

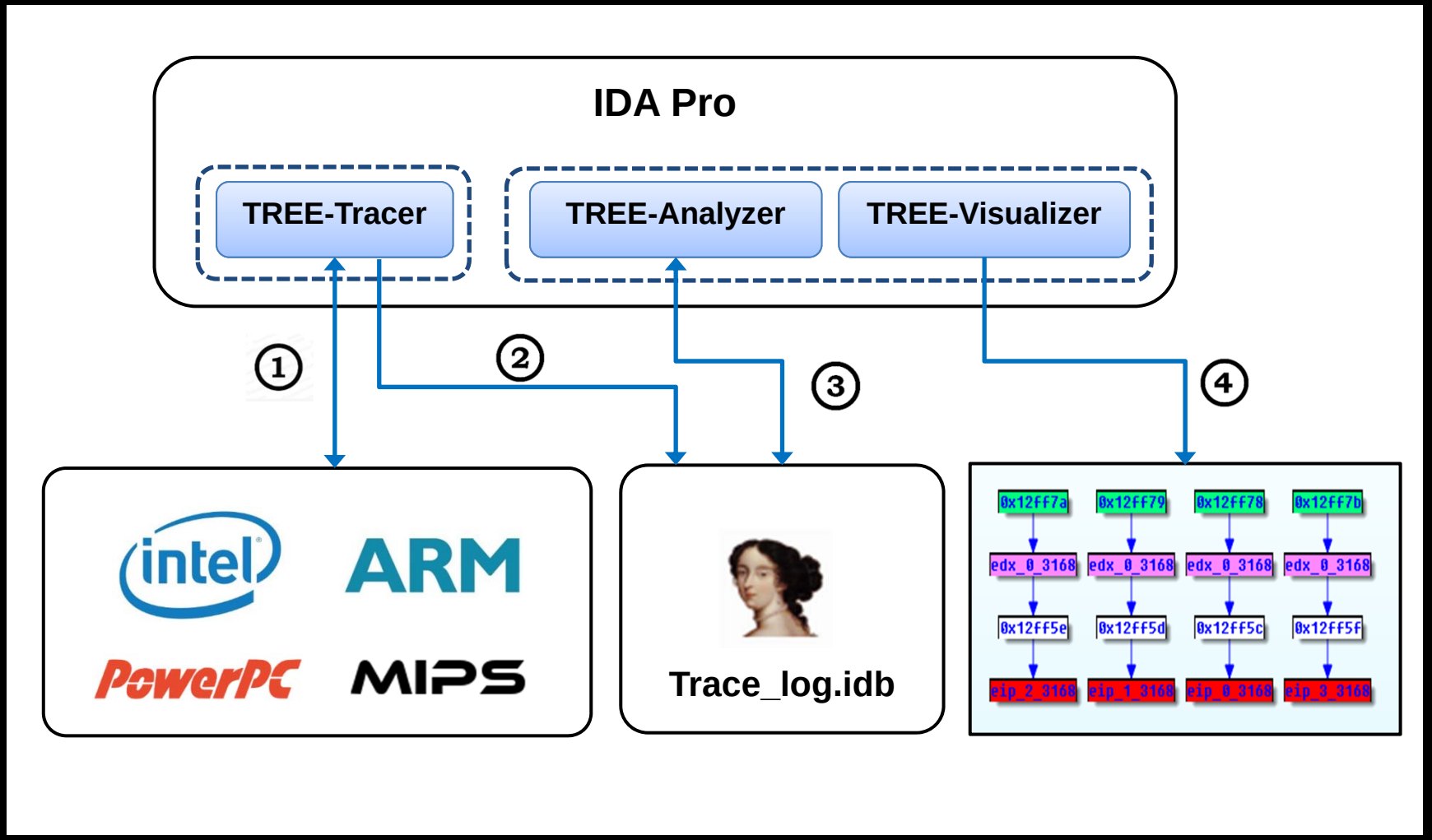


▼ Instruction chain engaged in data flow

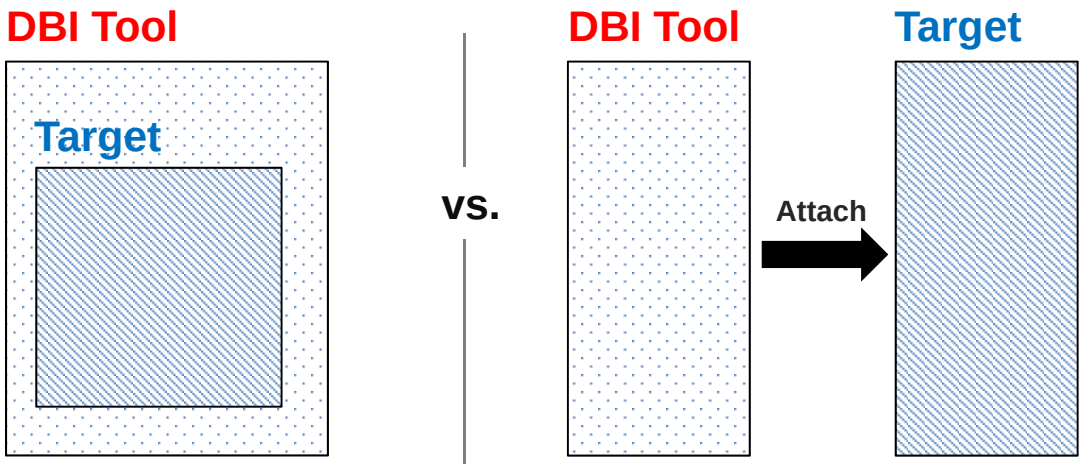
TREE (Tainted-enabled Reverse Engineering Environment) 1 / 2

- *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

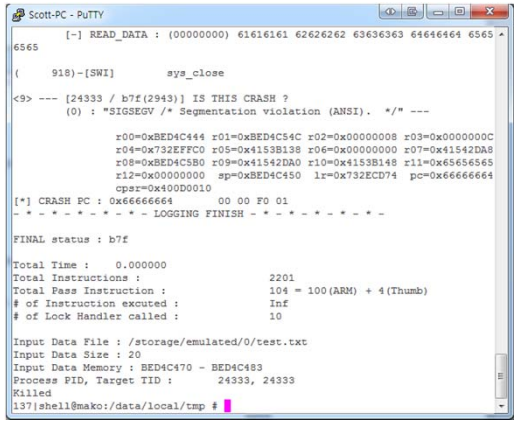


Type of DBI (Dynamic Binary Instrumentation)

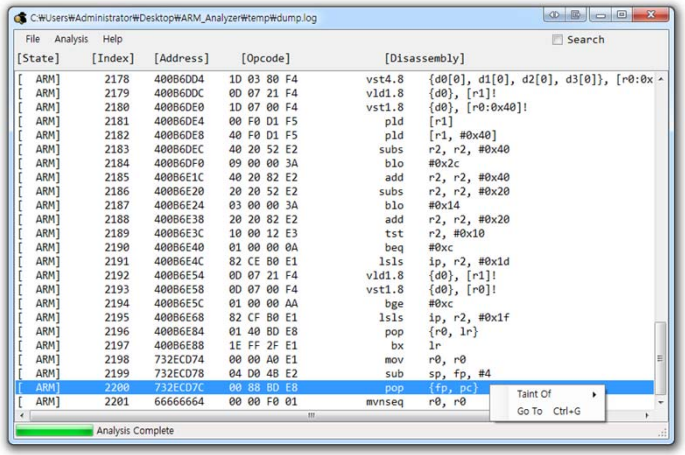


Type	- Same process	- Separate process ← Our choice!
Pros	- OS support NOT required (Low overhead)	- Appropriate for crash analysis
Cons	- NOT appropriate for crash analysis	- OS support required (High overhead)

Overview of our tools



trace.log



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

- 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

- 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

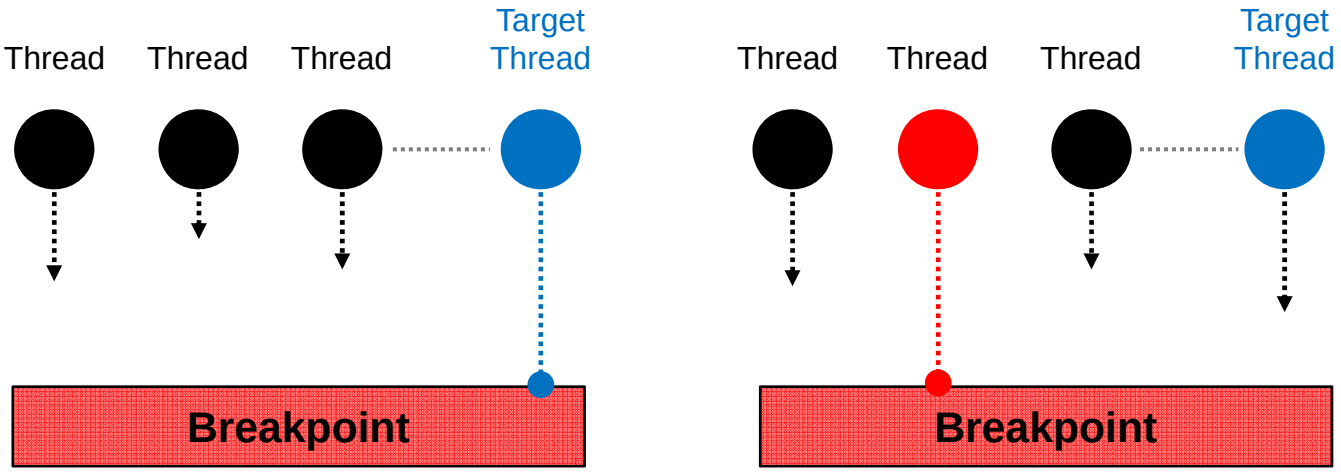
- Considerations on calculating Next PC
  - We have to identify opcode of instructions according to instruction state. (based on GDB)

ARM (32bit)	Thumb (16bit)	Thumb2 (16/32bit)

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

ARM-Tracer

- Addressing interference by other threads
  - Caused by code sharing



  - We have to guarantee all the threads run properly.

ARM-Tracer

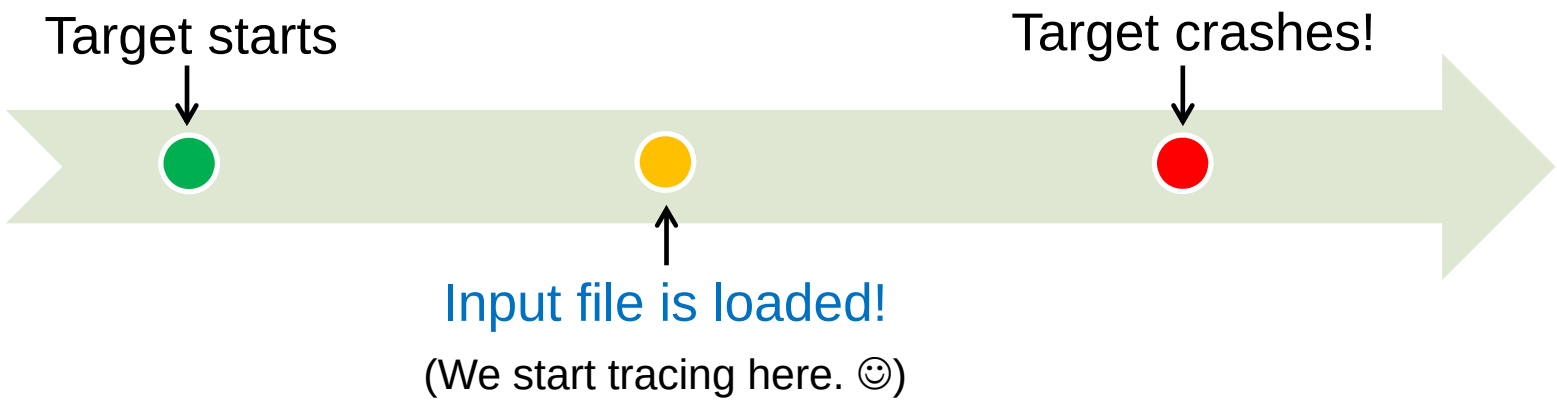
- Handling instruction sequence for atomic operation
  - ARM does not provide atomic instruction.
  - Instead, it provides sequence for it. (LDREX/STREX)
  - We should not intervene the sequence otherwise, it may cause infinite loop. ☹

```
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

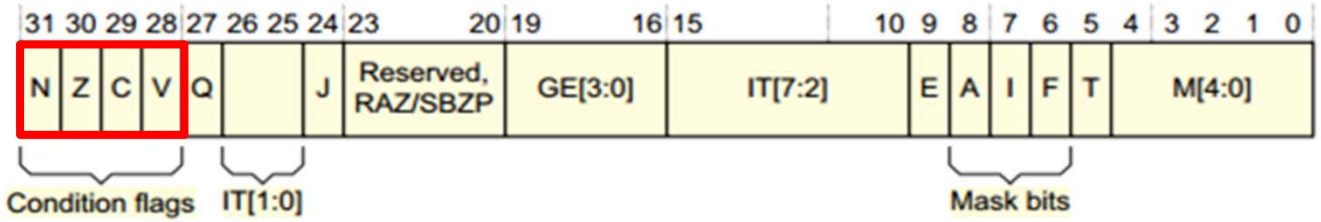
- The “good” starting point
  - We designate a specific thread as the target thread which opens the input file.
  - We can know memory address where the input file is loaded by checking `open` and `read` functions.



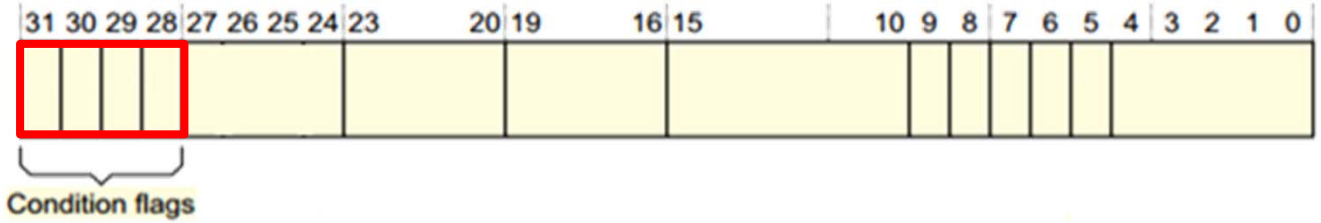
ARM-Tracer

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

[CPSR]



[ARM Instruction]

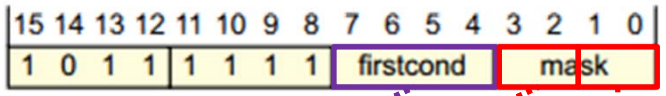


ARM-Tracer

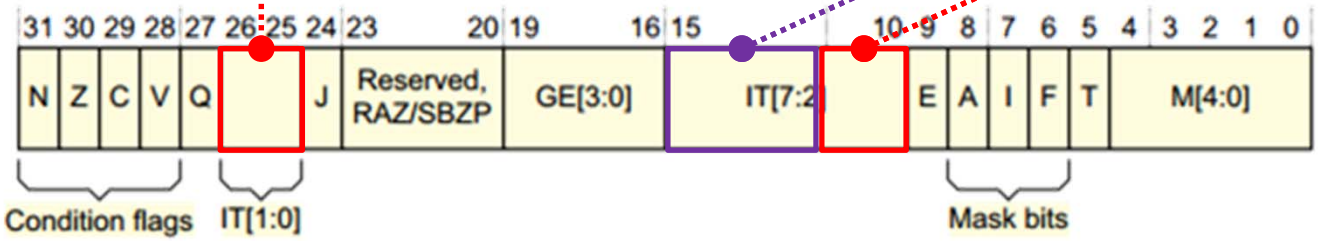
- Before logging, filter out instructions not executed (Thumb2)

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

[Thumb2 – IT Instruction]



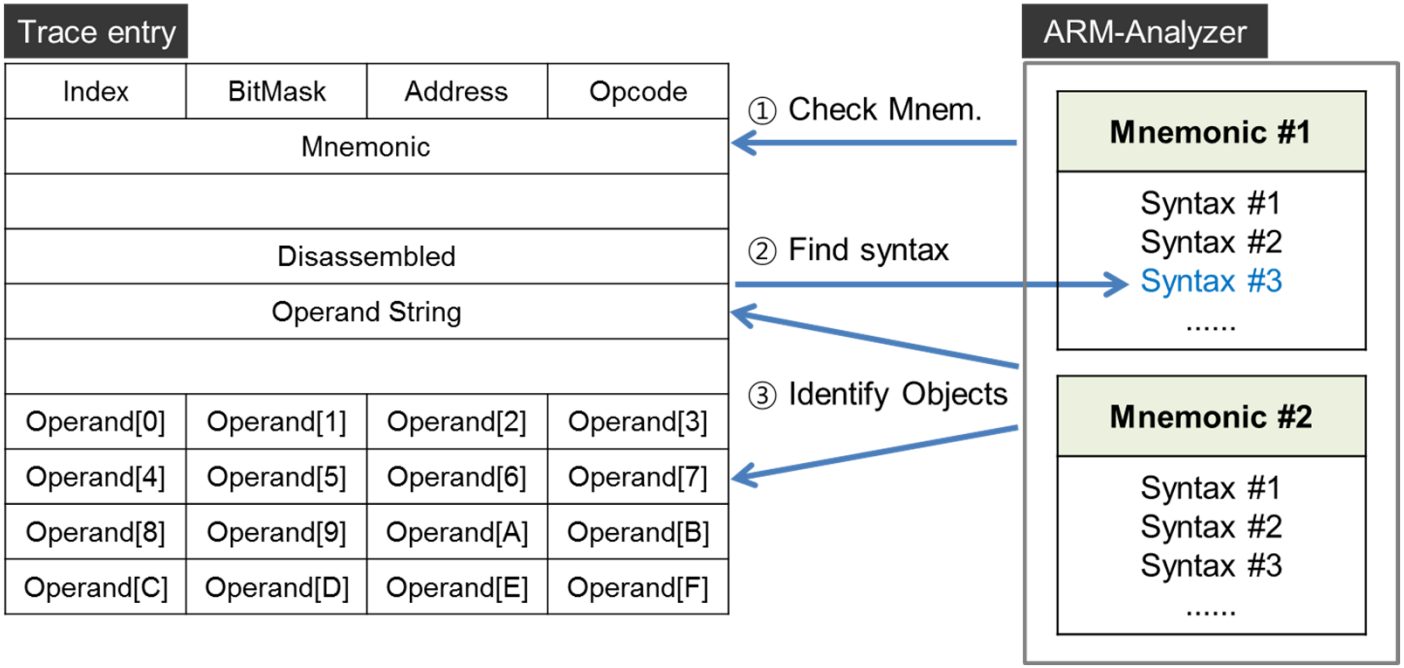
[CPSR]



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

- 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

- How it works – Backward taint analysis

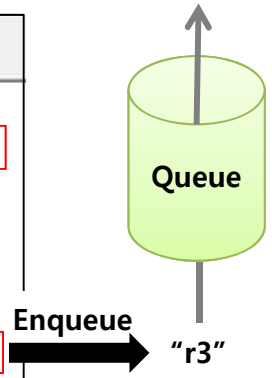
▼ View for user

[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

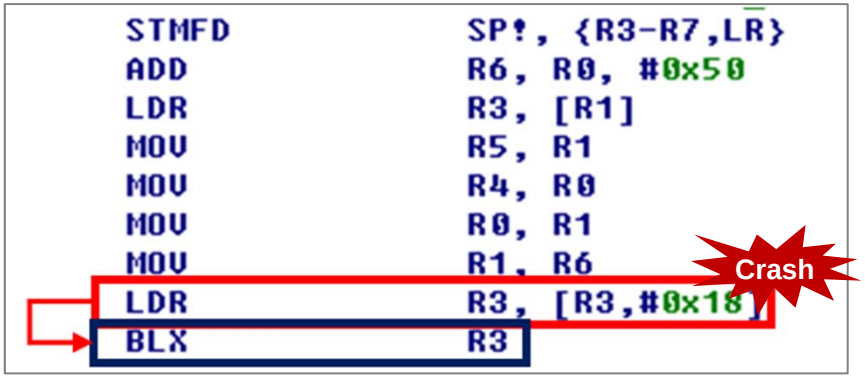
▼ Inside of ARM-Analyzer

	[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

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




```
LDR   R3, [R3, #0x18]
BLX   R3  ☺
```

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

Experiment

- Tested on GalasxyS4
  - 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

- 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) L D R R 1 , [R 2 , R 3] ➔ * (R 2 + R 3) // 0 x 1 0 0 4
      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 ☺