J Supercomput
DOI 10.1007/s11227-016-1777-9

Design and implementation of user-level dynamic binary instrumentation on ARM architecture

Dongwoo Kim[1] · **Sangwho Kim**[1] · **Jaecheol Ryou**[1]

Abstract We have developed a user-level dynamic binary instrumentation (DBI) tool on ARM architecture to enable applying various analysis techniques such as performance evaluation, profiling, and bug detection. Most of existing DBI tools are based on the x86 architecture. As demands for the similar approaches mainly performed in x86 are getting higher in an embedded device domain as well, DBI environment is strongly required for embedded devices especially ARM architecture which has a majority of market share in embedded systems. Only a few tools are currently available to ARM but they are very limited to a specific purpose. Therefore, we designed and implemented a general-purpose DBI tool on ARM to execute an arbitrary code between instructions of a target process. Our tool supports native execution environment which is required for security analysis. We have conducted some experiments with common UNIX utilities and an Android application to validate our tool by tracing and saving context information of every instruction of a desired thread. As a result, we have confirmed that it works as expected without affecting the original context of the target process.

Keywords Dynamic binary instrumentation · Dynamic tracing · Program analysis · Embedded device · ARM

✉ Dongwoo Kim
scotty@home.cnu.ac.kr

Sangwho Kim
whoyas2@home.cnu.ac.kr

Jaecheol Ryou
jcryou@home.cnu.ac.kr

[1] Dept. of Computer Engineering, Chungnam National University, Daejeon, Republic of Korea

Published online: 11 July 2016

1 Introduction

In the recent years, developments of platform technologies have made a number of services ubiquitous. The emergence of ultrafast mobile networks and highly featured embedded devices such as smartphones, tablets, and wearable computing devices has realized the ubiquitous life, but it is still focused on launching new products and services quickly to meet customers' needs without sufficient testing of software on the embedded systems in terms of reliability. As the embedded devices are getting involved in our real lives, various issues about privacy invasion and financial loss have been arising. It is now required to support analysis techniques for embedded software to provide the reliability. However, this kind of research is hindered by lack of dedicated tools like dynamic binary instrumentation (DBI) for the embedded systems.

DBI is a technique to get context information from a process for analysis without source code for re-compilation or binary modification of the target program, which is therefore even applicable to commercial off-the-shelf software [1,2]. It is usually indicated to dynamically inserts an arbitrary code into the target process, which guarantees not affecting the original flow of the target process. Thus, it enables various analysis techniques such as vulnerability analysis and dynamic execution information profiling [3–5].

There are some useful tools for user-level DBI but most of them are only for x86 architecture like Pin and DynamoRio [6,7]. Therefore, we propose a DBI tool for ARM architecture as the basic technology for supporting various analysis techniques. Our final goal is to conduct crash analysis of embedded software on real device to determine exploitability which requires native execution environment rather than simulated environment.

Li and Wang proposed CBASS-TREE [8] which is a similar work related to our final goal. CBASS-TREE is regarded as a multi-platform binary analysis framework based on the intermediate language called REIL (Reverse Engineering Intermediate Language) [9], which supports various features like taint analysis and symbolic execution. However, CBASS-TREE has a limitation in tracing instructions on ARM architecture because the DBI capability depends on a dedicated tool provided by IDA Pro–multi-processor disassembler and debugger that offers a wide range of features for analysis, besides the dedicated tool has problems that GDB also has [10]. The problems are introduced in Sect. 4.

Therefore, as the first step, we have developed a dynamic binary instrumentation tool, ARM-Tracer, for ARM architecture that has majority of market share in embedded systems. Our aim is to extract context information of every user-level instruction from a target process for further analysis. To do this, some challenges like atomic operation and signal handling caused by the multi-threaded environment should be addressed [11,12]. The details are described in the next section.

The paper is organized as follows. Section 2 presents design and implementation of our tool ARM-Tracer, Sect. 3 shows results of the experiment, Sect. 4 briefly deals with related work and finally Sect. 5 concludes the paper.

2 Design and implementation

ARM-Tracer is designed to work on 32-bit ARM-based Linux. It extracts context information of every user-level instruction from a target process by executing an arbitrary code whose job is to get the information every time instruction is executed.

There are two different ways to implement user-level DBI to execute the arbitrary code during the target process being run. The one is to execute it in the same process. It translates native instructions of a target program into intermediate language, and locates them with arbitrary code for analysis in code cache [13]. The other one is to do it in a separate process, which is the same manner of attach and control features of a debugger. Both of them have pros and cons but we have chosen the second one although it makes more overhead caused by context switch. It is easier to implement and more stable than the first one [14,15]. It enables stable analysis even on programs that cause crashes; it is, therefore, appropriate for security analysis. It can be implemented by means of system call *ptrace* which allows not only to read and write memory of a debuggee process, but also to get signals from the debuggee process.

When implementing DBI as the separate process model on ARM, four things—single step, atomic operation, kernel segment, and multi-thread signal—should be considered. They are described in detail as follows.

2.1 Single step

To get context information of every instruction from a target process, it should gain control of the target process every time its instruction is executed. It fully depends on the hardware support of the target architecture in terms of implementing the single-step feature. In case of x86, it can be simply implemented by means of trap flag supported by x86. When a process is set to the trap bit, it makes a signal which is delivered to its debugger right after a single instruction is executed and then the debugger can do whatever while the debuggee is stopped. Therefore, it is an easy task to implement the single-step feature in the x86 architecture. On the other hand, ARM does not provide such hardware support for it. For this reason, the single-step feature on ARM should be implemented by software only without hardware supports.

To do this, we can use debug breakpoint mechanism (DBM) for the single step in the environments that have no hardware supports [14]. DBM is to replace an original instruction of the next Program Counter (PC) with a breakpoint instruction so as to make a signal and give a control to a debugger when the debuggee hits the breakpoint. Note that ARM has two types of execution states—ARM and Thumb. They are used together and switched back and forth frequently within a process. Both have the different instruction size—32 bits for ARM and 16 bits for Thumb—which means an opcode size for fetch is determined according to the current execution state. In other words, the ARM architecture has two different breakpoint instructions while x86 has only one breakpoint instruction. Therefore, we should know the execution state of the instruction located at next PC before setting an appropriate breakpoint instruction to gain a control.

Table 1 A list of instructions which directly affects next PC

ARM (32 bit)	Thumb (16 bit)	Thumb2 (16/32 bit)
B Label	B Label	B Label
BL Label	BL Label	BL Label
BX Reg	BX Reg	BX Reg
BLX Label	BLX Label	BLX Label
BLX Reg	BLX Reg	BLX Reg
RFE Reg	CBZ Reg, Label	TBB [RegA, RegB]
ADD PC, Reg, PC	CBNZ Reg, Label	TBH [RegA, RegB, LSL #1]
SUB PC, Reg, PC	ADD PC, SP, PC	RFE Reg
SUBS PC, LR, #imm	MOV PC, Reg	SUBS PC, LR, #imm
MOVS PC, LR	LDR PC, [Reg]	LDR PC, [Reg]
LDR PC, [Reg]	LDM Reg, RegList	LDM Reg, RegList
LDM Reg, RegList		

The single step on ARM is performed in this manner. At first, we should analyze the current opcode to determine the next PC and the execution state of the next instruction. The second is to back up the next instruction, and replace it with the proper breakpoint instruction according to the execution state, then continue the debuggee. It will promptly cause the debuggee to hit the breakpoint. The third is to restore the instruction with the saved one to guarantee the original execution, and repeat it from the first.

The challenge of this procedure is to determine the next PC based on the analysis of the current opcode. In general, the next PC is increased by 4 bytes on the ARM state or 2 bytes on the Thumb state but the Thumb has an extended instruction set called Thumb2. The Thumb2 instruction set consists of 16-bit and 32-bit instructions. Besides, there are some instructions that can directly change the next PC. We figured out the instructions based on ARM compiler user guide [16] and GDB source code auditing. Table 1 shows the instructions classified by instruction set. Thus, it is required to carefully analyze these instructions at opcode level to determine the next PC.

In addition, condition bits should be also considered. Unlike x86, all the instructions on the ARM state have condition bits which are four most significant bits of the instructions. Their execution depends on Current Program Status Register (CPSR). CPSR holds information about the most recently performed ALU operation. The result of the operation is set to four most significant bits of CPSR as shown in Table 2. Regardless of the condition, every instruction is fetched, and then the condition bits of the instruction are compared to the condition flags of CPSR prior to the execution. If it does not meet the specified condition of CPSR, the instruction is replaced by processor with No Operation (NOP) instruction and then executed. On the other hand, instructions on the Thumb state do not have the condition bits. However, there is an exceptional case on the Thumb state. If–Then (IT) instruction of the Thumb2 instruction set makes the following four instructions conditional.

Table 2 Condition flags of current program status register

Bit	Name	Definition	
[31]	N	Overflow flag:	1 = overflow in last operation
			0 = no overflow
[30]	Z	Zero flag:	1 = result of 0 in last operation
			0 = nonzero result
[29]	C	Carry/borrow flag:	1 = carry or borrow in last operation
			0 = no carry or borrow
[28]	V	Negative/less than flag:	1 = result negative or less than in last operation
			0 = result positive or greater than

When replacing the instruction at the next PC with the breakpoint instruction, we should know the execution state. The state is changed by Branch Exchange (BX) or Branch with Link, and Exchange (BLX) instruction only. The state is dependent on the least significant bit of an operand of BX (BLX). If the bit of the operand is 1, it indicates the state of the next instruction is Thumb. Thus, the breakpoint should be set for the Thumb state. On the other hand, the bit for the ARM state is 0.

2.2 Atomic operation

The ARM architecture does not provide a single atomic instruction whereas Intel x86 does it. Instead, ARM provides instruction sequence for atomic operation. The sequence consists of two instructions—LDREX (Load-Exclusive) and STREX (Store-Exclusive). The sequence is used to atomically update memory. For instance, Table 3 shows the mutex mechanism using the sequence. The sequence conducts a memory check repeatedly until it is unlocked.

When single stepping the sequence with DBM, it can cause unnecessary tracing which leads sometimes infinite loop of the sequence due to the characteristic of the instructions. The important thing is that the sequence does not have to be traced. Therefore, it should be addressed by skipping the sequence rather than tracing every

Table 3 Instruction sequence of atomic operation

```
        MOV r1, #0x1                    ; load the 'lock taken' value
try
        LDREX r0, [LockAddr]            ; load the lock value
        CMP r0, #0                      ; is the lock free?
        STREXEQ r0, r1, [LockAddr]      ; try and claim the lock
        CMPEQ r0, #0                    ; did this succeed?
        BNE try                         ; no try again
        ...                             ; yes we have the lock
```

instruction of the sequence to reduce the tracing time and guarantee not affecting the original flow.

We made a handler for dealing with the sequence. When STREX instruction is found, the handler is called. Then, the handler seeks branch instruction that makes the sequence is repeated. Finally, the handler sets breakpoint at the next of the branch instruction to escape the loop.

2.3 Kernel segment

There exists a special purpose kernel segment which is accessible to user-level code. EVT (Exception Vector Table) resides in the segment. EVT contains a jump table to exception handlers. When breakpoint instruction that is set by DBM is executed by processor, it causes an exception which makes processor to change PC to EVT to eventually invoke an appropriate handler. After the exception handler finishes, PC is automatically changed to which the exception occurs. The exception handling is not directly performed by user-level code. It is handled by processor, thus there is no need to handle exception-related instructions.

However, there are some functions to help user space in the shared segment other than EVT. The functions are used to provide user space with some operations that need kernel support due to unimplemented native features like memory barrier. Therefore, the functions can be directly invoked by user-level code but a problem is that user-level code is not allowed to modify instructions in the kernel segment. Since it is required to replace an existing instruction with breakpoint instruction to stably trace instructions, the functions in the segment should be carefully handled. To do this, we simulate the functions in the segment. If the next PC turns out to point to a specific function in the segment, ARM-Tracer simulates the function and sets breakpoint instruction at the return address in user space. It can guarantee seamless tracing.

2.4 Multi-thread signal

There is an issue in the multi-threaded environment. It is about signal handling. Since code section is shared by all the threads in a process, every thread can hit a breakpoint even if the breakpoint is set for tracing a specific thread. This is inevitable in environments using DBM. Therefore, it should be finely designed for handling signals such as SIGSEGV, SIGSTOP, SIGTRAP, SIGILL, and EXIT caused by threads. It should be able to distinguish where signals come from, between a target thread and other threads, and then perform different jobs in accordance with the thread type.

Figure 1 shows the entire flowchart of ARM-Tracer. It is designed to trace a specific thread chosen by user. At first, it stops all the threads. It chooses the target thread, and gets register context of the target thread. Then, it determines the next PC by analyzing opcode of the current PC with the register context. When the sequence of the atomic operation is found, the handler deals with it. The next step is to set breakpoint at the next PC to inform the ARM-Tracer that a thread hits the breakpoint. To catch the signal, *wait* system call is used. The *wait* system call gets signals from the signal queue one by one. The important thing in this part is that there can be signals made

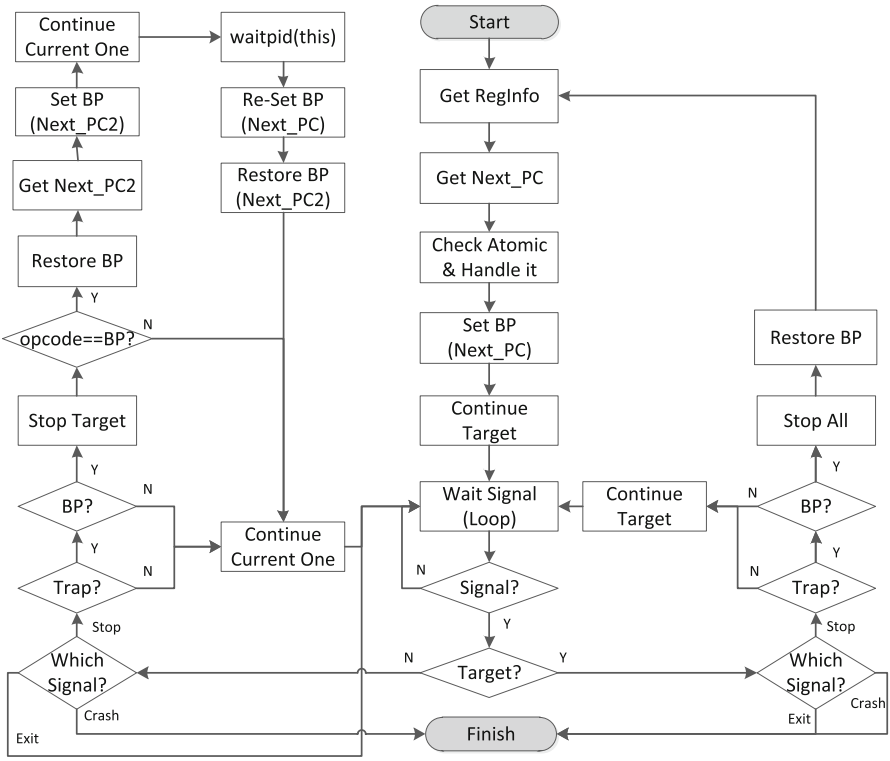

Fig. 1 Entire flowchart of ARM-Tracer

by other threads as well as the target thread. Therefore, it is required to handle signals appropriately in accordance with the target or other threads as follows.

Target thread When the signal is sent by the target thread, it is necessary to see the type of the signal first. In case of SIGTRAP, it checks whether the signal is made by the breakpoint. If not, it indicates that the signal is made by thread creation or process fork. However, if the signal turns out to be for the breakpoint, it stops all the threads to avoid a side effect caused by other threads before restoring the breakpoint with the original one. After the restoration, the phase for determining the next PC to set the breakpoint is repeated until the target thread finishes or gets crashed.

Other threads When the signal is sent by other threads, only SIGTRAP caused by the breakpoint is regarded as important. Other signals are handled just by continuing the current thread right away. If the breakpoint is recognized, it stops the target thread first to prevent the target thread from passing the current point where the breakpoint is set. Then, it checks the opcode at the current PC to see if it is the breakpoint instruction again. This is because the opcode can be restored with the original one in advance by other threads. In this case, the current thread is just continued. When the opcode is still identified as the breakpoint, ARM-Tracer restores it with the original one. The important thing here is that the original instruction that has been just restored

should be replaced with the breakpoint again after the current thread passes, which guarantees tracing the target thread without missing a single instruction. To do this, it is required to determine the next PC again to set the breakpoint as the second point, and continue the current thread. Meanwhile, it waits for the current one to reach the breakpoint at the second point. As soon as the current one reaches the breakpoint that has been set temporarily, it resets the breakpoint at the first point for capturing the target thread, and then restores the current breakpoint at the second point with the original one. Finally, the current thread is continued. All these steps can guarantee that the target thread is traced without missing a single instruction in harmony with other threads.

3 Experiments and results

We have conducted two experiments for validation of ARM-Tracer on a real device which is the smartphone Nexus4–ARMv7, Quad-core 1.5GHz, 2GB DDR2, Android 4.3 and Linux kernel 3.4.0. First, we have tested the DBI feature in the single thread environment by tracing various UNIX utilities of BusyBox. Second, we have applied our tool to an Android application that has a number of threads [17]. Our tool is supposed to save a trace log to a file containing disassembly and register context of every instruction. When disassembling, it takes an advantage of *Capstone* that is a lightweight multi-platform, multi-architecture disassembly framework [18].

3.1 Experiment 1

We carried out some tests with utilities provided by BusyBox to validate the DBI feature of our tool. BusyBox combines small versions of a wide range of common UNIX utilities into a single executable file for providing a fairly complete environment for embedded systems [19]. Table 4 shows the result of tracing some utilities of BusyBox 1.23.1. We tested many utilities but it represents only some of them for want of space.

Each of them is executed in a single thread. We can see that about 30 mnemonics are found in each utility. We also conducted tracing them step by step using GDB for comparison. As a result, we have confirmed that the result is the same with the log created by our tool, which means that the DBI feature of our tool using single stepping with the debug breakpoint mechanism is reliable for use.

3.2 Experiment 2

We applied our tool to an Android application—Polaris Office 6.0.1—which is a widely used application to read, edit, and share various documents. In this case, we performed tests with input files that make the target application get crashed. The crashes generated by fuzzing the target application will be analyzed to determine the exploitability as our future work. An Android application basically creates a number of threads. In addition, the target application creates additional threads. One of them is in charge of

Table 4 Results of tracing some UNIX utilities of BusyBox

	Cal		Date		Id		Pwd	
1	CMP	12,585	CMP	2378	CMP	2455	LDR	911
2	LDR	11,960	LDR	1866	LDR	2215	CMP	883
3	MOV	10,178	ADD	1486	B	1667	ADD	778
4	ADD	7761	B	1407	ADD	1437	B	658
5	B	7339	MOV	1369	MOV	1286	MOV	592
6	BL	3182	STR	701	STR	795	STR	406
7	SUB	2933	SUB	477	SUB	327	SUB	230
8	STR	2748	BL	387	BL	277	STM	171
9	BX	2408	SUBS	288	STM	247	BL	166
10	SUBS	2141	TST	281	PUSH	181	TST	113
11	TST	1089	BX	276	ORR	176	PUSH	102
12	POP	819	STM	217	POP	174	POP	96
13	ORR	803	ORR	207	SUBS	169	SUBS	91
14	STM	791	AND	141	TST	162	BX	85
15	PUSH	692	POP	126	AND	157	AND	77
16	AND	646	PUSH	123	LSL	116	ORR	57
17	MVN	433	EOR	96	BX	106	MVN	57
18	RSB	397	LSL	85	RSB	83	EOR	48
19	LSL	387	CLZ	75	MVN	66	RSB	40
20	CLZ	313	MVN	73	LSR	64	BLX	24
21	CMN	310	RSB	65	EOR	37	LSL	22
22	EOR	215	LSR	63	BIC	33	LSR	20
23	LSR	209	SRS	62	CLZ	30	CMN	19
24	SRS	174	MUL	38	SRS	29	MLA	18
25	MUL	170	CMN	26	CMN	25	LDM	16
26	BIC	165	BIC	20	BLX	20	CLZ	15
27	SVC	150	LDM	20	LDM	19	BIC	12
28	TEQ	122	MLA	19	SVC	17	SVC	11
29	ADC	114	BLX	19	MLA	14	ASR	4
30	MLA	98	SVC	18	MUL	5	MUL	2
31	LDM	96	ASR	12	ASR	4	TEQ	1
32	UMULL	57	TEQ	9	TEQ	3	SRS	1
33	ASR	42	ADC	6				
34	BLX	30	UMULL	3				
35	SMLAL	19	SMLAL	1				
Total instr	71,576		12,440		12,396		5726	
Taken time	25 s		4 s		4 s		2 s	

handling the input files. To extract a trace log for analysis from a thread that causes crash while parsing and processing the input file, we need to identify a specific thread of all the threads which is trying to open the input files. Once the moment when the

Table 5 Results of tracing crash samples of Polaris Office Viewer; top 25 instructions

	Crash #1		Crash #2		Crash #3		Crash #4	
1	LDR	167,211	LDR	226,551	LDR	1,181,296	LDR	1,720,569
2	B	97,006	MOV	137,710	B	919,524	B	1,306,681
3	MOV	93,594	B	131,991	MOV	810,170	MOV	1,152,527
4	CMP	78,411	STR	116,576	CMP	679,755	CMP	996,965
5	ADD	76,122	CMP	105,656	STR	607,233	STR	864,062
6	STR	61,001	ADD	101,129	ADD	576,831	ADD	796,701
7	BL	33,635	BL	46,437	BL	206,183	BL	293,650
8	ORR	20,999	POP	26,863	POP	163,935	POP	245,785
9	BX	19,566	BX	25,767	PUSH	145,101	PUSH	216,025
10	POP	18,917	PUSH	25,439	BX	113,769	BX	158,419
11	PUSH	17,724	ORR	23,696	SUB	99,135	SUB	144,901
12	SUBS	13,057	SUBS	16,346	ORR	97,837	LSL	124,347
13	PLD	12,127	SUB	15,985	LSL	86,641	ORR	121,853
14	SUB	11,940	PLD	13,354	SUBS	72,901	SUBS	104,935
15	LSL	11,136	MOVT	13,277	PLD	71,866	PLD	103,131
16	MUL	10,169	LSL	12,569	AND	69,218	AND	100,374
17	STM	9785	STM	11,015	VMOV	69,176	RSB	81,407
18	LDM	7300	MUL	10,181	CBZ	59,162	MOVT	77,406
19	MOVT	6654	LDM	8503	RSB	58,712	VMOV	72,203
20	RSB	5812	RSB	8015	MOVT	55,333	LSR	72,054
21	AND	4982	AND	7527	LSR	49,573	TST	65,325
22	LSR	3386	BIC	3523	TST	45,297	CBZ	59,765
23	TST	2917	TST	3359	STM	41,020	STM	55,251
24	BIC	2623	LSR	3223	VLDR	40,206	BIC	54,626
25	SRS	1922	UXT	2153	BIC	35,048	BLX	53,776

input file is loaded is captured, ARM-Tracer starts instruction tracing until the target gets crashed. The important thing here is that it should guarantee other threads to be run as expected without any error, otherwise it results in a side effect.

When tracing a specific thread with GDB–version 7.10.1 that is the latest version as of December 2015 [20], it often misses some instructions of the target thread when the thread switching happens occasionally. For this reason, GDB supports a special option called scheduler-locking. The option allows tracing a specific thread consistently by avoiding the thread switching but it does not guarantee other threads to be run, which can cause a side effect in multi-threaded environment.

Table 5 shows the results of tracing crash samples of the Android application. Unlike the test results of BusyBox, over 100 unique instructions are identified. Thus, we represent only 25 instructions that are most frequently appeared. We also classified them by groups (see Table 6). Instructions in data process and memory access groups take account for over 70 % in each case. We can see that our tool is able to trace over

Table 6 Classification of total instructions by groups

	Crash #1	Crash #2	Crash #3	Crash #4
Data processing	334,837 (41.89 %)	456,225 (41.00 %)	2,774,614 (41.46 %)	3,959,702 (41.84%)
Memory access	296,033 (37.04 %)	430,293 (38.66 %)	2,250,840 (33.63 %)	3,258,189 (34.42 %)
Branch	151,610 (18.97 %)	206,506 (18.55 %)	1,334,726 (19.94 %)	1,878,030 (19.84 %)
Multiply	10,350 (1.29 %)	10,383 (0.93 %)	12,745 (0.19 %)	16,085 (0.17 %)
Packing	3429 (0.42 %)	4686 (0.42 %)	20,636 (0.30 %)	36,071 (0.38 %)
Coprocessor	1112 (0.14 %)	1975 (0.18 %)	9070 (0.14 %)	12,824 (0.14 %)
Control	53 (0.01 %)	50 (0.00 %)	22,224 (0.33 %)	24,431 (0.26 %)
Miscellaneous	24 (0.00 %)	26 (0.00 %)	478 (0.00 %)	985 (0.01 %)
NEON/VFP	1728 (0.21 %)	2613 (0.23 %)	266,788 (3.98 %)	277,204 (2.93 %)
Total instr traced	799,176	1,112,757	6,692,121	9,463,521
Taken time	215 s	292 s	1454 s	2,365 s
# of atomic handling	46	54	2798	3982

4000 instructions per second. The bottom line of the table indicates the number of atomic handler called to skip the sequence related to atomic operation.

4 Related work

There have been many related works in x86 architecture but there are only few works in ARM architecture as follows.

Pin, a well-known dynamic binary rewriting system developed at Intel, was ported to ARM in 2006 but it is not available at the moment [21].

DynamoRIO is a dynamic code manipulation framework that enables transforming any part of code in a program at runtime. It did not support ARM architecture but the community that manages DynamoRIO recently started working to port it for ARM.

Valgrind is a widely used framework that provides some tools based on DBI to automatically detect memory-related bugs like memory leak [22]. Valgrind uses simulation techniques to instrument user-level instructions, which makes difference in buggy program. For example, if a program gets crashed because of accessing memory that is unaddressable, it could be possible that the memory would not be unaddressable when run under Valgrind. This is the nature of the way Valgrind works. Therefore, it is not appropriate for crash analysis with Valgrind [23].

ADBI, Android dynamic binary instrumentation, is developed to hook a function of ARM binary to monitor and modify parameters, which is not for instruction tracing [24].

GDB seems to be very similar to our tool in terms of debugger and debuggee model but GDB has heavy overhead due to its complex data structures, besides the current GDB, decisively, cannot single step a specific thread completely. It misses some instructions when thread switching occurs. GDB also supports an option for tracing

a specific thread but it has a side effect that it does not guarantee the harmonious execution of other threads as mentioned in Sect. 3. Hence, it can cause unexpected results in the multi-threaded environment.

5 Conclusion and future work

We have developed a fine and lightweight DBI tool working on ARM architecture, which can extract desired context information of every instruction from a target process and even a specific thread. We also considered the multi-threaded environment. We conducted some experiments for validation by applying our tool to common UNIX utilities and an Android application. As a result of the evaluation, we found that it works correctly in harmony with other threads without affecting the original context of the target thread as we designed. We expect that it will be used for further researches like security analysis on embedded software which is our final goal. We plan to perform crash analysis for determining exploitability by means of taint analysis in the future. For the taint analysis, a capability to trace every instruction related to data propagation is essential. Therefore, we will perform the taint analysis based on our DBI tool.

Acknowledgments This research was supported by Next-Generation Information Computing Development Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (No. NRF-2014M3C4A7030648)

References

1. Lee D, Kim I, Kim J, Jun HK, Kim WT, Lee S, Eom YI (2013) Light-weight kernel instrumentation framework using dynamic binary translation. J Supercomput 66(3):1613–1628
2. Henderson A, Prakash A, Yan L, Hu X, Wang X, Zhou R, Yin H (2014) Make it work, make it right, make it fast: building a platform-neutral whole-system dynamic binary analysis platform. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis, pp 248–258
3. Nicholas N (2004) Dynamic binary analysis and instrumentation. Dissertation, University of Cambridge
4. Zaddach J, Bruno L, Francillon A, Balzarotti D (2014) Avatar: a framework to support dynamic security analysis of embedded systems firmwares. In: Proceedings of the 21st Symposium on Network and Distributed System Security
5. Enck W, Gilbert P, Han S, Tendulkar V, Chun B, Cox L, Jung J, Mcdaniel P, Sheth A (2014) TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones. ACM Trans Comput Syst 32(2):5
6. Luk CK, Cohn R, Muth R, Patil H, Klauser A, Lowney G, Wallace S, Reddi VJ, Hazelwood K (2005) Pin: building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, vol 40, pp 190–200
7. Uh GR, Cohn R, Yadavalli B, Peri R, Ayyagari R (2006) Analyzing dynamic binary instrumentation overhead. In: WIBA Workshop at ASPLOS
8. Li L, Wang C (2013) Dynamic analysis and debugging of binary code for security applications. In: Proceedings of the 4th International Conference on Runtime Verification, Springer, Berlin
9. DullienT, Porst S (2009) REIL: a platform-independent intermediate representation of disassembled code for static code analysis. In: CanSecWest
10. Mihajlovi B, Zilic Z, Gross WJ (2014) Dynamically instrumenting the QEMU emulator for linux process trace generation with the GDB debugger. ACM Trans Embed Comput Syst 13(5s), article no. 167

11. Liu C, Chen J, Yang W, Hsu W (2014) Dynamically translating binary code for multi-threaded programs using shared code cache. J Electron Sci Technol 04:434–438
12. Hong D, Wu J, Yew P, Hsu W, Hsu C, Liu P, Wang C, Chung Y (2014) Efficient and retargetable dynamic binary translation on multicores. IEEE Trans Parallel Distrib Syst 25(3):622–632
13. Payer M, Kravina E, Gross T (2013) Lightweight memory tracing. In: USENIX annual technical conference, pp 115–126
14. Paxson V (1990) A survey of support for implementing debuggers
15. Rodriguez R, Artal J, Merseguer J (2014) Performance evaluation of dynamic binary instrumentation frameworks. IEEE Latin Am Trans 12(8):1572–1580
16. ARM (2015) ARM compiler armasm user guide version 6.3. http://infocenter.arm.com/help/topic/com.arm.doc.dui0801d/DUI0801D_armasm_user_guide. Accessed 8 Dec 2015
17. Sun H, Zheng Y, Bulej L, Villazn A, Qi Z, Tma P, Binder W (2015) A programming model and framework for comprehensive dynamic analysis on Android. In: Proceedings of the 14th International Conference on Modularity, ACM, pp 133–145
18. Capstone: lightweight multi-platform, multi-architecture disassembly framework. http://www.capstone-engine.org. Accessed 8 Dec 2015
19. van der Kouwe E, Giuffrida C, Tanenbaum AS (2014) Evaluating distortion in fault injection experiments. In: 2014 IEEE 15th International Symposium on High-Assurance Systems Engineering (HASE), pp 25–32
20. GDB 7.10.1. http://www.gnu.org/software/gdb/download. Accessed 8 Dec 2015
21. Kim H, Klauser A (2006) A dynamic binary instrumentation engine for the ARM architecture. In: Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems
22. Nicholas N, Seward J (2007) Valgrind: a framework for heavyweight dynamic binary instrumentation. ACM Sigplan Not 42(6):89–100
23. Valgrind FAQ (2015) On crash. http://valgrind.org/docs/manual/faq.html#faq.crashes. Accessed 8 Dec 2015
24. ADBI: Android dynamic binary instrumentation. https://github.com/crmulliner/adbi. Accessed 8 Dec 2015