Memory Corruption Detecting Method Using Static Variables and Dynamic Memory Usage

Jihyun Park
Department of Computer Science & Engineering
Ewha Womans University
Seoul, Korea
pola0527@ewhain.net,

Changsun Park
Department of Computer Science & Engineering
Ewha Womans University
Seoul, Korea
ddangddon@ewhain.net

Byoungju Choi[†]
Department of Computer Science & Engineering
Ewha Womans University
Seoul, Korea
bjchoi@ewha.ac.kr

Gihun Chang
Samsung Research DX
Samsung Electronics
Seoul, Korea
gihun.chang@samsung.com

ABSTRACT

Memory fault detection has been continuously studied and various detection methods exist. However, there are still remains many memory defects that are difficult to debug. Memory corruption is one of those defects that often cause a system crash. However, there are many cases where the location of the crash is different from the actual location causing the actual memory corruption. These defects are difficult to solve by existing methods.

In this paper, we propose a method to detect real time memory defects by using static global variables derived from execution binary file and dynamic memory usage obtained by tracing memory related functions. We implemented the proposed method as a tool and applied it to the application running on the IoTivity platform. Our tool detects defects very accurately with low overhead even for those whose detected location and the location of its cause are different.

CCS CONCEPTS

• Software and its engineering ! Software testing and debugging

KEYWORDS

Memory corruption, Memory fault detection, Runtime fault detection

1 INTRODUCTION

Memory space used in processes can be largely divided into two categories: the static memory area and dynamic memory area. The size of static memory is allocated before a process begins, while dynamic memory is allocated based on the required amount during runtime.

Memory corruption occurs when there is an unintentional change in static or dynamic memory while the process is running [1]. Memory corruption can result when the current process attempts to use or release memory that is not valid, such as memory that was unallocated or uninitialized, or tries to access a memory size that exceeds the allocated amount. Such memory corruption may lead to process malfunctions or even cause system crashes; thus, it is a defect that must be resolved. Further, memory corruption can occur from the use of static memory, such as stack overflow, in addition to dynamic memory; hence, defects detection must go over the entire memory area.

Methods for detecting memory corruption have been the focus of continuous research. These studies have resulted in debugging tools such as Purify, Memcheck, Insure++, and AddressSanitizer [2-5]. However, memory corruption is still one of the more difficult defects that have yet to be resolved [6]. This is because there are often cases where the location of the cause of memory corruption differs from the location of its actual failure, which makes it difficult to correlate the cause and effect. Moreover, symptoms that result from memory corruption such as crashes are non-deterministic [7]; hence, the symptoms often do not reappear.

We propose a method that utilizes information tags in real-time defect detection in static/dynamic memories. These information tags store the usage of static/dynamic memory through the execution binary analyses and memory function hooking. Further, since memory corruption can cause system crashes, we propose a method that can detect memory corruption as soon as a crash occurs through signal hooking

The proposed method shows high accuracy in defect detection since it considers both the dynamic and static memory usage while minimizing the overhead. In addition, even if a system crash occurs, memory corruption can still be detected without data loss, and debugging information is provided that can be used for debugging memory corruption when locations differ between the cause of defect and the actual result.

We implemented the propose method as a Linux-based automation tool. We demonstrate the superior performance of the proposed method through empirical study that compares it to typical memory defect detection tools.

This paper is organized as follows. Section 2 introduces studies related to memory corruption detection. Section 3 present our proposed method and its tool. An empirical study and an actual field study on IoTivity platform are described in

[†]Byoungju Choi is the Corresponding author

AST'18, May 28–29, 2018, Gothenburg, Sweden
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5743-2/18/05...$15.00
https://doi.org/10.1145/3194733.3194741

Section 4 and 5. Section 6 concludes and suggests the direction of future work.

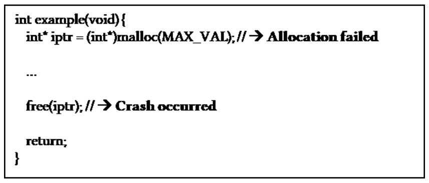
2 BACKGROUND AND RELATED WORK

2.1 Memory corruption

Memory corruption occurs when memory used in a process is changed unintentionally. Once it occurs, the process either malfunctions or the system crashes. Memory corruption is a defect that is difficult to resolve for the following reasons.

First, there are often cases where the location of the cause of the memory corruption differs from the location of the actual failure. Figure 1 is an example showing a crash that occurred when pointer iptr was released after an allocation failure. In this example, the crash from memory corruption is caused by the attempt to release the unallocated pointer, but the underlying cause is failure of malloc.

However, the most of memory corruption detection methods only provides the location of the crash, and the reason why iptr cannot be accessed remains unexplained.



```
int example(void){
  int* iptr = (int*)malloc(MAX_VAL); // → Allocation failed

  ...

  free(iptr); // → Crash occurred

  return;
}
```

Figure 1 Example of memory corruption resulting from memory allocation failure

The second reason is because the symptoms of memory corruption are non-deterministic. As shown in Figure 1, memory allocation failures may occur because of wrong allocation size, but they may also occur depending on the system's status. When a buffer overflow occurs, it will malfunction instead of causing a crash if the overflowed memory area is valid. However, if an overflow occurs in an invalid memory area, a crash will occur. As such, since symptoms of memory corruption vary according to the system's status, there is no consistent pattern, making defects difficult to resolve.

Memory corruptions occur while using either dynamic or static memory. A typical example is a stack overflow that occurs when memory is read or written in excess of the array size. Moreover, even if the memory is static, if the address is used as a pointer by adding '&', memory corruptions may still occur in the same way as dynamic memory. Therefore, both static and dynamic memory areas must be analyzed in order to detect and resolve memory corruption.

2.2 Related research

Many tools have been introduced for detecting memory corruption. These tools can be classified into static detection tools and dynamic detection tools depending on whether the target program is executed or not.

Static memory corruption detection analyzes source codes or executable binaries [6]. The array index used in the source code is analyzed to detect defects such as a buffer overflow [8]. However, this method cannot detect memory corruptions in dynamic memory, and because it does not occur in the actual execution environment, defect detection accuracy is not very high.

Dynamic memory corruption detection collects debugging information by adding tracking code into memory-related function during compilation or runtime [9]. Some popular tools include Purify, Memcheck, and AddressSanitizer; these tools operate in the actual execution environment and are highly accurate in the detection of memory corruption.

Purify can detect memory corruptions in dynamic memory. However, the overhead is very high because the code that monitors the memory access is instrumented when object code was linked. Memcheck detects memory corruption while running in a separate execution environment. Process memory is managed through two stages of shadow memory and used for detecting memory corruption. This tool can detect defects such as uninitialized memory access or release, but the execution overhead is also extremely high. AddressSanitzer uses shadow memory for all memory areas to detect memory corruption and has very low overhead. However, there is a limitation that recompilation and change in the executable binary are required.

The shadow memory technique used in Memcheck and AddressSanitizer stores information from all memory areas used by the process in a separate area called the "shadow memory" [10]. Each time an operation is performed in the process, such as memory allocation, release, reading, or writing, the memory operation results are stored in the shadow memory and used to detect runtime memory defects. Typically, shadow memory stores information about validity of access and whether the memory was initialized. This information can be used to detect memory corruption, such as accessing invalid memory spaces in the process or reading uninitialized memory spaces.

Such methods are advantageous because they provide an accurate evaluation of the status of all memory spaces. However, they also require high additional memory usage for the shadow memory and have large performance overhead during the shadow memory search process that corresponds to the location of the memory used in the process. To overcome the limitations of the shadow memory regarding the large overhead, various studies have attempted to minimize the amount of space taken up by the shadow memory and to increase access speed.

3 MEMORY CORRUPTION DETECTION

In this paper, we propose a method of detecting real-time memory corruption in static/dynamic memory, by utilizing their usage. We have classified the categories of memory defects, listed in Table 1, based on common weakness enumeration [11] and existing memory fault classification [12]. The defects in Table 1 are the direct or indirect causes of memory corruption.

Table 1 Memory defect categories

Category	Defects
Allocation	- Memory allocation failure - Zero-size memory allocation

	- Memory leak
Deallocation	- NULL pointer free - Duplicated free - Unallocated memory free - Allocation / deallocation mismatch
Access	- NULL pointer access - Freed memory access - Unallocated memory access - Access defects out of the allocated range that do not conflict with memory space of other variables - Access defect out of the allocated range that conflicts with memory space of other variables

Information tags, which are a type of shadow memory, are used to collect information for detecting the defects in Table 1. Information tags are a data structure that stores and manages the usage on static/dynamic memory. The executable binary file is analyzed to extract the address and size of static global variables used in the process, and then stored as information tags.

Information of dynamic memory such as address, size, function type, and whether to release, allocation/deallocation function are stored as information tags.

While the process is running, the information tags of the memory accessed when calling memory-related functions are used in real-time defect detection. This enables the detection of memory corruption in both static and dynamic memory.

3.1 Information tagging

In order to detect memory defects in real-time, there must first be information that can determine the validity of the memory that will be used. To this end, the shadow memory technique is used. To reduce the memory overhead from shadow memory, instead of managing the entire virtual memory space of process, only the static global variables analyzed from the executable binary and dynamically allocated information on the heap memory area are managed through a data structure called "information tags", as shown in Figure 2.

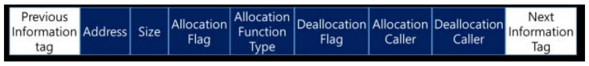


Figure 2 Information tag

To further reduce the performance overhead, the operation formula necessary for mapping information tags corresponding to the memory for defect detection is defined as follows:

Shadow memory address

$$= (\text{No. of static variables}) + (target\ address - heap\ start\ address)$$

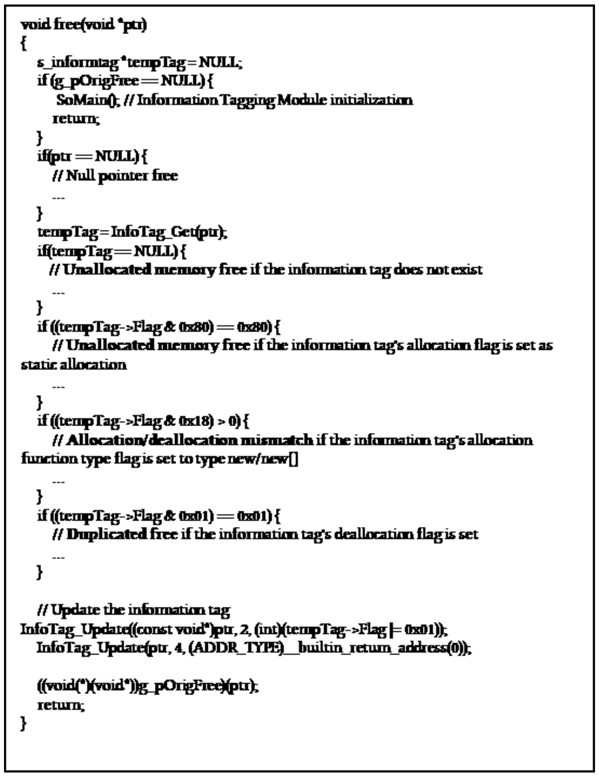
The performance overhead was minimized by allowing the location of information tags of the current memory to be explored through only operations that use the number of static global variables and initial heap memory address.

Memory address, allocation size, static/dynamic allocation status, allocation function type, and release status are saved into information tags in order to detect defects in real-time. Further, information on the function that is called for memory allocation and deallocation are also stored in the information tags so as to overcome the limitations of existing tools that only provide information on the defect's location, even if it differs from the actual location of the defect's cause.

3.2 Real-time memory defect detection

First, to monitor memory usage in the process, the hooking technique is used for memory-related functions. "Hooking" is a technique that intercepts the execution path during runtime and is useful for monitoring memory related-operations [13].

After hooking memory-related functions, the hooked function is executed each time a memory related-function is called. At this point, information tags are used and defects are detected in real-time, as shown in Figure 3.



```
void free(void *ptr)
{
   s_informtag *tempTag=NULL;
   if (g_pOrigFree==NULL){
      SoMain(); // Information Tagging Module initialization
      return;
   }
   if(ptr ==NULL){
      // Null pointer free
      ---
   }
   tempTag=InfoTag_Get(ptr);
   if(tempTag==NULL){
     // Unallocated memory free if the information tag does not exist
      ---
   }
   if ((tempTag->Flag & 0x80) == 0x80){
      // Unallocated memory free if the information tag's allocation flag is set as
static allocation
      ---
   }
   if ((tempTag->Flag & 0x18) > 0){
      // Allocation/deallocation mismatch if the information tag's allocation
function type flag is set to type new/new[]
      ---
   }
   if ((tempTag->Flag & 0x01) == 0x01){
      // Duplicated free if the information tag's deallocation flag is set
      ---
   }

   // Update the information tag
InfoTag_Update((const void*)ptr, 2, (int)(tempTag->Flag |= 0x01));
   InfoTag_Update(ptr, 4, (ADDR_TYPE)__builtin_return_address(0));

   ((void(*)(void*))g_pOrigFree)(ptr);
   return;
}
```

Figure 3 Example of real-time memory defect detection

After first storing the address and size of static global variables derived from the execution binary analysis as information tags, the address, size, dynamic allocation status, allocation function type, and the address of the function that calls for memory allocation in the information tag are stored each time memory is dynamically allocated.

When memory is accessed or released, the validity of the information tag that corresponds to the target memory address is verified. The address and dynamic allocation status of the information tag are used to determine whether the address space that permits the access or release of the current memory space must be accessed or released. If memory is accessed in the middle, such as "addr+n", then the address and size of the nearest information tag is used to verify its validity. Even if the address can be accessed, if access exceeds the size that was restricted by a size value, this is regarded as an access defect that deviates from the approved range.

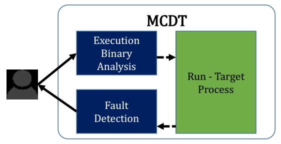
3.3 Signal hooking

Memory defects, such as access to unallocated memory or release of memory that was already freed, may cause system crashes. In such cases, defect information which consists of defect type, defect location, function parameter, return value and call stack to be collected at the point of the crash may be lost. In order to detect defects without losing information when a system crashes, a signal handler that is raised during a crash is hooked. When a system crash occurs, most operating systems create a signal that is linked to the cause, and the operation from when the signal goes off is executed by the signal handler in charge. Therefore, when the signal handler is hooked, and defect detection information is collected from the hooked signal handler, there is no loss of information. In the Linux system, signals related to memory defects include SIGABRT, SIGBUS, and SIGSEGV.

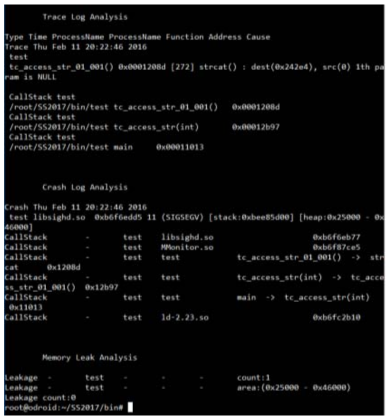
3.4 Automation Tool

We implemented the proposed method as an automated tool that runs on an ARM-based Linux system. Currently, the kernel version of Linux that implemented is 4.9.27 and is designed to work on ARM based systems.

As shown in Figure 4(a), when the user inputs an execution file, the execution file is automatically analyzed, and the execution file is run with the tool. While the tool is running, it detects memory defects in real-time and collects information. When the operation ends normally or abnormally, the tool analyzes and reports the defects it detected, as shown in Figure 4(b).



(a) Fault Detection Process



```
        Trace Log Analysis

Type Time ProcessName ProcessName Function Address Cause
Trace Thu Feb 11 20:22:46 2016
 test
 tc_access_str_01_001() 0x0001208d [272] strcat() : dest(0x242e4), src(0) 1th pa
ram is NULL

 CallStack test
 /root/SS2017/bin/test tc_access_str_01_001()   0x0001208d
 CallStack test
 /root/SS2017/bin/test tc_access_str(int)       0x00012b97
 CallStack test
 /root/SS2017/bin/test main     0x00011013



        Crash Log Analysis

Crash Thu Feb 11 20:22:46 2016
 test libsighd.so  0xb6f6edd5 11 (SIGSEGV) [stack:0xbee85d00] [heap:0x25000 - 0x
46000]
CallStack       -       test    libsighd.so                             0xb6f6eb77
CallStack       -       test    MMonitor.so                             0xb6f87ce5
CallStack       -       test    test            tc_access_str_01_001()  ->  str
cat      0x1208d
CallStack       -       test    test            tc_access_str(int)  ->  tc_acce
ss_str_01_001()  0x12b97
CallStack       -       test    test            main  ->  tc_access_str(int)
 0x11013
CallStack       -       test    ld-2.23.so                              0xb6fc2b10



        Memory Leak Analysis

Leakage  -      test    -       -       -       count:1
Leakage  -      test    -       -       -       area:(0x25000 - 0x46000)
Leakage count:0
root@odroid:~/SS2017/bin# 
```

(b) Tool Screen

Figure 4 Scenario of using the automation tool

The automation tool consists of four modules that are loaded into the Linux system, as shown in Figure 5. The role of each module is as follows.

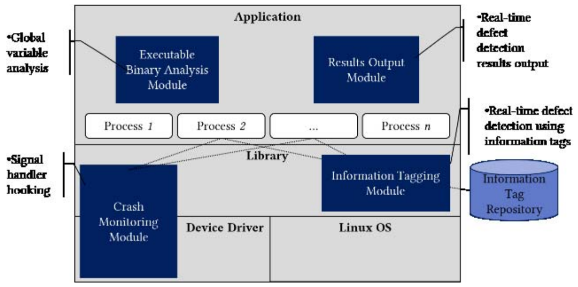

Application
•Global variable analysis
Executable Binary Analysis Module
Results Output Module
•Real-time defect detection results output
Process 1
Process 2
...
Process n
•Real-time defect detection using information tags
•Signal handler hooking
Library
Information Tagging Module
Information Tag Repository
Crash Monitoring Module
Device Driver
Linux OS

Figure 5 Automation tool composition

- Executable binary analysis module: Extracts static global variables to store in information tags from the binary file as mentioned in Section 3.1.
- Defect detection module using information tagging: Detects defects using information tags each time a memory-related function is called by hooking memory-related functions as mentioned in Section 3.2.
- Crash monitoring module: Uses the signal hooking technique in Section 3.3 to hook signal handlers and collect defect detection information without losing data, even when a system crashes.
- Result output module: Generates the results on defects that were detected in real-time after completing the target defect detection process.

4 EMPIRICAL STUDY

In order to analyze the effects of the proposed method, benchmarking was used for defect detection and an analysis was performed to check if defects were truly detected.

4.1 Test Subjects

The target of the empirical study was the ITC benchmark [14]. The ITC benchmark is a program that was developed for defects occurring in actual vehicle software, and contains a mix of various defects, such as deadlock and typecasting, in addition to memory defects. Of these, 311 cases of 19 different memory defects from Table 2 were applied to the tool.

Table 2 Target

Bench mark	Fault		TC Count	Crash occurrence
ITC Bench mark	TC_02	buffer_overrun_dynamic	32	1
	TC_03	buffer_underrun_dynamic	39	3
	TC_04	cmp_funcadr	2	0
	TC_11	deletion_of_data_structure_sentinel	3	0
	TC_12	double_free	12	12
	TC_16	free_nondynamic_allocated_memory	16	16
	TC_17	free_null_pointer	14	9
	TC_24	invalid_memory_access	17	1
	TC_25	littlemem_st	11	4

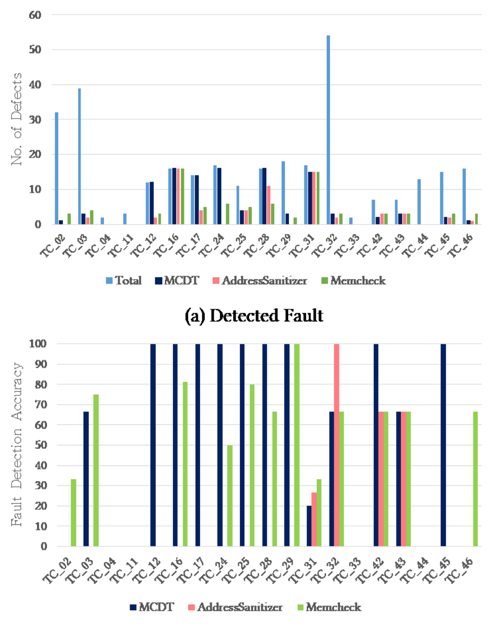
	TC_28	memory_allocation_failure	16	7
	TC_29	memory_leak	18	0
	TC_31	null_pointer	17	15
	TC_32	overrun_st	54	3
	TC_33	ow_memory	2	0
	TC_42	st_overflow	7	7
	TC_43	st_underflow	7	3
	TC_44	underrun_st	13	0
	TC_45	uninit_memory_access	15	2
	TC_46	uninit_pointer	16	1
Total			311	84

In order to perform an objective comparative analysis on the effects of the proposed method, the empirical study was also performed on AddressSanitizer and Memcheck. Since both defect detection tools use shadow memory. In particular, AddressSanitizer uses shadow memory to manage stack and heap spaces, they are appropriate for comparing the effects of the proposed method.

4.2 Results

(1) Accuracy of Defect Detection.

Upon applying 19 different defect types to the ITC benchmark, AddressSanitizer detected 65 out of 311 defects, Memcheck detected 80 defects, and the proposed method was able to detect 111 defects, as shown in Figure 6(a). AddressSanitizer and Memcheck had difficulty detecting defects related to 'free non-dynamic allocated memory' (TC_16) and 'null pointer access' (TC_31). On the other hand, the proposed method used information tags to determine whether or not the memory was initialized and also managed the freed memory through information tags, thereby displaying outstanding performance in defect detection.



(a) Detected Fault

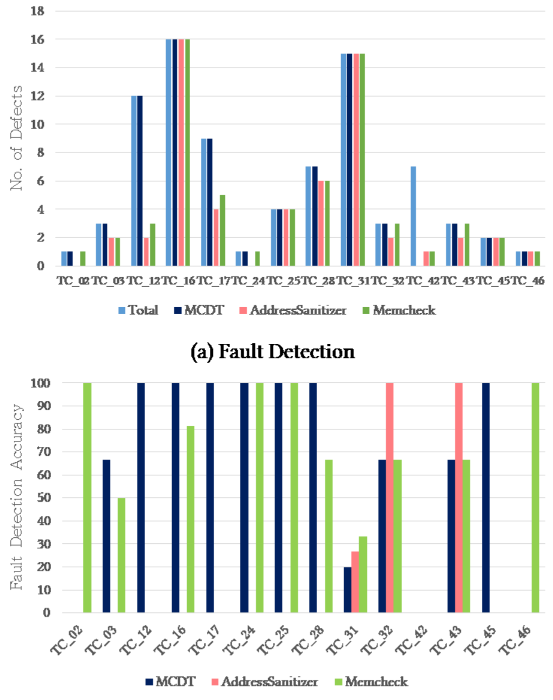
(b) Fault Detection Accuracy

Figure 6 Defect detection results

Further, the cause of defects was analyzed based on the information provided when defects were detected in the tool. The proposed method accurately analyzed the cause of 94 defects (84.69%). AddressSanitizer was only able to accurately analyze the cause of 10 defects (15.39%), while Memcheck only analyzed 43 defects (53.75%). The proposed method collected information from the moment when the defect was detected as well as information on the potential cause of the defect during defect-related memory use. As a result, the accuracy of the proposed method was high because information provided upon defect detection was used while analyzing the cause of the defect. In contrast, since AddressSanitizer and Memcheck only provided information on the moment when a defect was detected, these tools had difficult to identify the cause of defects when the location and cause of the detected defect differed.

(2) Crash occurrence.

Since memory corruption is highly likely to cause a system crash [15], information on defect detection cannot be lost because of a crash. Hence, signal hooking was used for this purpose. In the test subjects in Table 2, a crash occurred when 84 TCs were performed. Figure 7 shows the result. The proposed method detected 77 defects (91.67%) that led to the crash, while AddressSanitizer and Memcheck detected 57 defects (67.86%) and 63 defects (75%), respectively.



(a) Fault Detection

(b) Fault Detection Accuracy

Figure 7 Defect detection results upon crashing

Upon analyzing the cause of defects that were detected at the moment of the crash, the proposed method accurately analyzed the cause of 58 defects (65.71%). AddressSanitizer was only able to accurately analyze the cause of 8 defects (16.9%), while Memcheck only analyzed 34 defects (54.61%). The accuracy of the two tools in determining the cause of defects was low because they only collected information on the moment of defect detection at the crash and presented this information as the cause of the defect, as analyzed in (1). While the proposed method may have displayed lower accuracy in analyzing the cause of all defects in (1), it still provided relatively accurate information on the cause of defects compared to the other tools.

(3) Performance.

The proposed method had a 6.67% performance overhead when monitoring defect detection. Memcheck is a heavyweight method with an extremely large performance overhead because it only operates from a separate virtual machine called Valgrind. AddressSanitizer showed a 26.7% performance overhead. While AddressSanitizer operates by changing the execution file to detect defects, the proposed method may be considered a very lightweight method because it only adds a defect detection agent without changing the execution file.

5 CASE STUDY

The automation tool was applied to detect defects of an application that operates on an actual IoTivity platform [16]. As shown in Table 3, aside from 4 cases that did not reappear out of the 9 cases, defects were detected for all 5 cases. For the 3 closed defects, defects that were detected by the tool were identical to the resulting causes.

Table 3 IoTivity application targets

Bug ID	Title	Status	Reappearance	Detection
IOT-2951	translateEndpointsPayload leaks memory	open	O	O
IOT-2413	[AddressSanitizer] Heap-use-after-free in InProcServerWrapper.cpp	closed	O	O
IOT-2411	[AddressSanitizer] Attemtping double-free in otmunittest.cpp	closed	O	O
IOT-2200	Valgrind reports invalid memory read causing intermittent CI build failures	closed	X	-
IOT-2373	Intermittent CI build failure due to invalid memory read/write in notification consumer tests	closed	X	-
IOT-2246	checkSslOperation continues using peer object already freed by DTLSHandshakeCB	closed	O	O
IOT-2503	[GRL][BUG] [UBUNTU] Segmentation Fault error for running DirectPairingClient sample app	open	O	O
IOT-2441	OCRepPayloadClone with OCByteString array may crash	resolved	X	-
IOT-2647	CT1.7.8.3: printCRL read buffer overrun	resolved	X	-

IOT-2411 is a defect with a "segmentation fault" during execution. Upon using the tool, when the SIGSEGV signal was raised, a duplicated free defect was detected, and the defect's cause was analyzed by providing the initial freed location stored in the information tag from when the defect was detected.

IOT-2951 and IOT-2503 are unresolved defects, but when the tool was used, memory corruptions were detected as a result of respective memory leakage and access to invalid memories.

6 CONCLUSION

Memory corruption is a remaining problem to be resolved. To resolve this issue, defects must be detected across all memories used in a process, and it is important to accurately identify the location of the cause of the defect instead of just the moment when the defect occurred.

In this paper, we proposed a method that detects memory corruption by analyzing static variables and dynamic memory use. This method can be used to detect defects by managing static variables from static memory used in the process, and dynamic memory usage through information tag. Further, signal hooking was applied to collect information on defects without data loss, even if a crash occurs. The proposed method was implemented as an automated tool, and its performance was compared with other tools.

In the future, we will conduct further experimental study to measure the performance of our tool by fault injection technique. Moreover, we plan to expand the static memory used for defect detection to the entire static memory area, in addition to static variables

ACKNOWLEDGMENTS

This work was supported by the Samsung Electronics. At present an industrial field study is in progress for applying the IoTivity.

REFERENCES

[1] Jun Xu, Dongliang Mu, Ping Chen, Xinyu Xing, Pei Wang, and Peng Liu. 2016. Credal: Towards locating a memory corruption vulnerability with your core dump. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. ACM, New York, NY, USA, 529-540. DOI: http://dx.doi.org/10.1145/2976749.2978340

[2] Reed Hastings, and Bob Joyce. 1991. Purify: Fast detection of memory leaks and access errors. In Proceedings of the Winter 1992 USENIX Conference. Usenix Association, 125-138.

[3] Nicholas Nethercote and Julian Seward. 2007. How to shadow every byte of memory used by a program. In Proceedings of the 3rd international conference on Virtual execution environments (VEE '07). ACM, New York, NY, USA, 65-74. DOI=http://dx.doi.org/10.1145/1254810.1254820

[4] Parasoft Insure++. 2017. White paper: Runtime and Memory Error Detection and Visualization with Parasoft Insure++. Retrieved from http://www.parasoft.com/products/insure

[5] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. USENIX Annual Technical Conference. 309-318

[6] Feng Qin, Shan Lu, and Yuanyuan Zhou. 2005. SafeMem: Exploiting ECC-memory for detecting memory leaks and memory corruption during production runs. High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on. IEEE. 291-302

[7] Yichen Xie, Andy Chou, and Dawson Engler. 2003. ARCHER: using symbolic, path-sensitive analysis to detect memory access errors. In Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT

international symposium on Foundations of software engineering (ESEC/FSE-11). ACM, New York, NY, USA, 327-336. DOI=http://dx.doi.org/10.1145/940071.940115

[8] Christopher Erb, Mike Collins, and Joseph L. Greathouse. 2017. Dynamic buffer overflow detection for GPGPUs. In Proceedings of the 2017 International Symposium on Code Generation and Optimization (CGO '17). IEEE Press, Piscataway, NJ, USA, 61-73

[9] Qin Zhao, Rodric Rabbah, Saman Amarasinghe, Larry Rudolph, Weng-Fai Wong. 2008. How to do a million watchpoints: Efficient debugging using dynamic instrumentation. International Conference on Compiler Construction. Springer, Berlin, Heidelberg, 147-162, DOI= https://doi.org/10.1007/978-3-540-78791-4_10

[10] Kostyantyn Vorobyov, Julien Signoles, and Nikolai Kosmatov. 2017. Shadow state encoding for efficient monitoring of block-level properties. SIGPLAN Not. 52, 9 (June 2017), 47-58. DOI: https://doi.org/10.1145/3156685.3092269

[11] CWE. 2018. A community-developed list of common software security weaknesses. Retrieved from https://cwe.mitre.org/index.html

[12] Glenn R. Luecke, James Coyle, Jim Hoekstra, Marina Kraeva, Ying Li, Olga Taborskaia, Yanmei Wang. 2006. A survey of systems for detecting serial run-time errors. Concurrency and Computation: Practice and Experience. 18, 5 (April 2006), 1885-1907. DOI= https://doi.org/10.1002/cpe.1036

[13] Jooyoung Seo, Byoungju Choi, and Suengwan Yang. 2011. A profiling method by PCB hooking and its application for memory fault detection in embedded system operational test. Inf. Softw. Technol. 53, 1 (January 2011), 106-119. DOI=10.1016/j.infsof.2010.09.003 http://dx.doi.org/10.1016/j.infsof.2010.09.003

[14] ITC-benchmarks. 2018. Static analysis benchmarks from Toyota ITC. Retrieved from https://github.com/regehr/itc-benchmarks

[15] Jun Xu, Peng Ning, Chongkyung Kil, Yan Zhai, and Chris Bookholt. 2005. Automatic diagnosis and response to memory corruption vulnerabilities. In Proceedings of the 12th ACM conference on Computer and communications security (CCS '05). ACM, New York, NY, USA, 223-234. DOI=http://dx.doi.org/10.1145/1102120.1102151

[16] IoTivity. 2018. An open source software framework enabling seamless device-to-device connectivity to address the emerging needs of the Internet of Things. Retrieved from https://git.iotivity.org/iotivity