UAF-GUARD: Defending the Use-After-Free Exploits via Fine-grained Memory Permission Management

Xu, Guangqquan; Lei, Wenqing; Gong, Lixiao; Liu, Jian; Bai, Hongpeng; Chen, Kai; Wang, Ran; Wang, Wei; Liang, Kaitai; Wang, Weizhe

Total number of authors:
12

Published in:
Computers and Security

Link to article, DOI:
10.1016/j.cose.2022.103048

Publication date:
2023

Document Version
Peer reviewed version

Link back to DTU Orbit

Citation (APA):
Xu, G., Lei, W., Gong, L., Liu, J., Bai, H., Chen, K., Wang, R., Wang, W., Liang, K., Wang, W., Meng, W., & Liu, S. (2023). UAF-GUARD: Defending the Use-After-Free Exploits via Fine-grained Memory Permission Management. *Computers and Security*, *125*, Article 103048. https://doi.org/10.1016/j.cose.2022.103048

Journal Pre-proof

UAF-GUARD: Defending the Use-After-Free Exploits via Fine-grained Memory Permission Management

Guangquan Xu, Wenqing Lei, Lixiao Gong, Jian Liu, Hongpeng Bai, Kai Chen, Ran Wang, Wei Wang, Kaitai Liang, Weizhe Wang, Weizhi Meng, Shaoying Liu

PII:	S0167-4048(22)00440-0
DOI:	https://doi.org/10.1016/j.cose.2022.103048
Reference:	COSE 103048

To appear in:	*Computers & Security*

Received date:	20 June 2022
Revised date:	7 October 2022
Accepted date:	1 December 2022

UAF-GUARD: Defending the Use-After-Free Exploits via Fine-grained Memory Permission Management★

Guangquan Xu[a,b], Wenqing Lei[b,*], Lixiao Gong[b], Jian Liu[b,*], Hongpeng Bai[b], Kai Chen[c], Ran Wang[d], Wei Wang[e], Kaitai Liang[f], Weizhe Wang[b], Weizhi Meng[g] and Shaoying Liu[h]

[a]*School of Big Data, Qingdao Huanghai University, Qingdao, P. R. China*
[b]*Tianjin Key Laboratory of Advanced Networking (TANK), College of Intelligence and Computing, Tianjin University, Tianjin, P. R. China*
[c]*Institute of Information Engineering, Chinese Academy of Sciences, P. R. China*
[d]*Security Center, JD.com, P. R. China*
[e]*Beijing Jiaotong University, Beijing, P. R. China*
[f]*Delft university of technology, Netherlands*
[g]*DTU Compute, Techincal University of Denmark, Denmark*
[h]*Graduate School of Advanced Science and Engineering, Hiroshima University, Hiroshima, Japan*

ARTICLE INFO

Keywords:
Use-After-Free vulnerability
fine-grained memory permission management
static instrumentation
Vulnerability Discovery
System Security

ABSTRACT

The defense of Use-After-Free (UAF) exploits generally could be guaranteed via static or dynamic analysis, however, both of which are restricted to intrinsic deficiency. The static analysis has limitations in loop handling, optimization of memory representation and constructing a satisfactory test input to cover all execution paths. While the lack of maintenance of pointer information in dynamic analysis may lead to defects that cannot accurately identify the relationship between pointers and memory.

In order to successfully exploit a UAF vulnerability, attackers need to reference freed memory. However, main existing schemes barely defend all types of UAF exploits because of the incomplete check of pointers. To solve this problem, we propose UAF-GUARD to defend against the UAF exploits via fine-grained memory permission management. Specially, we design two key data structures to enable the fine-grained memory permission management to support efficient relationship search for pointers and memory, which is the key design of our defending scheme against UAF exploits. In addition, UAF-GUARD can precisely locate the position of UAF vulnerabilities, so that malicious programs can be terminated in the place where the abnormality is discovered.

We implement UAF-GUARD on a 64-bit Linux system, and further use UAF-GUARD to transform a program into a suitable version that can defend against UAF vulnerabilities exploits. Compared with main existing schemes UAF-GUARD is able to effectively and efficiently defend against all the three types of UAF exploits with acceptable space overhead (26.4% for small programs and 0.3% for large programs) and time complexity (21.9%).

1. Introduction

UAF (Use-After-Free) vulnerability is a kind of memory corruption flaw, defined by Common Weakness Enumeration (CWE) as "referencing memory after it has been freed can cause a program to crash, use unexpected values, or execute code" (2018) (CWE). UAF Vulnerabilities cannot be exploited individually, which means that a single piece of UAF vulnerability must be exploited along with other heap memory vulnerabilities (e.g., Heap Spray Ratanaworabhan, Livshits and Zorn (2009)). Attackers could utilize UAF vulnerabilities to perform malicious operations such as arbitrary reading, writing back, and code execution. Not only that, once an attacker obtains process information, it will be easier to bypass system security defense tools, for example, Canary UMWiki (2015), PIE Hat (2012), ASLR SearchSecurity (2014). Arbitrary write-back or code execution could cause the attacker to hijack the control flow, furthur to get shell and to obtain all the system permissions.

Memory error detector could be used to capture UAF vulnerabilities at program runtime. By maintaining the state of the allocated memory, we can determine whether the released memory will be used again in the future. But P. Ratanaworabhan et al. proposed Heap Spray Ratanaworabhan et al. (2009), which can force the program to reallocate memory on the freed memory. That means the heap memory allocation process could be easily controlled by attackers.

Control Flow Integrity (CFI) tool may defend UAF vulnerabilities due to an assumption that the ultimate goal of current attacks is to either disclose information or control the target host via hijacking control flow to execute malicious code. The existing tools utilize coarse-grained CFI to avoid

★This work is partially sponsored by National Key R&D Program of China (No. 2019YFB2101700), National Science Foundation of China (No. 62172297, No. 61902276), the Key Research and Development Project of Sichuan Province (No. 2021YFSY0012), and Tianjin Intelligent Manufacturing Special Fund Project (No. 20211097, No. 20201159).

*Wenqing Lei and Jian Liu are the corresponding authors (Email: raidy_518@tju.edu.cn, jianliu@tju.edu.cn).

losin@tju.edu.cn (G. Xu); raidy_518@tju.edu.cn (W. Lei); glx_0826@tju.edu.cn (L. Gong); jianliu@tju.edu.cn (J. Liu); bai931214@tju.edu.cn (H. Bai); chenkai@iie.ac.cn (K. Chen); wangran8088@gmail.com (R. Wang); wangwei1@bjtu.edu.cn (W. Wang); k.liang-3@tudelft.nl (K. Liang); will@tju.edu.cn (W. Wang); weme@dtu.dk (W. Meng); sliu@hiroshima-u.ac.jp (S. Liu)

ORCID(s):

expensive overhead and false positive alarms. However recent works Davi, Sadeghi, Lehmann and Monrose (2014); GÃűktas, Athanasopoulos, Bos and Portokalidis (2014a); GÃűktas, Athanasopoulos, Polychronakis, Bos and Portokalidis (2014b) demonstrate that all the coarse-grained CFI could be easily bypassed.

Our contributions. Facing the fact that all the aforementioned anti-UAF mechanisms suffer from either high false negative rates or being bypassable via certain exploitation techniques, in our previous work, we proposed the UAF-GUARD for UAF defense by maintaining the relationship between pointers and memory. This paper expand the previous workXu, Li, Li and et.al (2020) with addition of instruction types for scanning, completing the data structure of pointers and memory to enhance the overall efficiency, proposing a pointer filtering algorithm to increase the efficiency further and fully analyzing the safety of the method. In addition, an extended utilization of the method is proposed.

By constructing an efficient data structure, our UAF-GUARD is allowed to maintain pointer-to-memory permissions at runtime, so that it could precisely identify the relationship of pointers and memory before target operations for eliminating the vulnerabilities caused by pointer abuse. UAF-GUARD can terminate the runtime program which cannot pass the vulnerability check. Moreover, it can precisely locate the position of the vulnerability which resides at the exact point where the program is terminated. Our new design can effectively prevent the UAF vulnerabilities exploitation and further mitigate the risks in the existing heap management mechanism.

Our contributions are summarized as follows:

1) We improve UAF-GUARD. Using designed key data structure to completely represent the relations of pointer-to-pointer and pointer-to-memory, UAF-GUARD can check the permission of the pointer to memory at runtime for preventing pointer abuse and further defending against all types of vulnerabilities' exploits.
2) We design an approach to optimizing the efficiency of our detection and defense. By leveraging the target instruction filter algorithm, we reduce the overhead to an acceptable level.
3) We illustrate how to develop an automated tool for mining the UAF vulnerabilities by introducing two mature dynamic analysis techniques, namely symbolic execution and fuzzing.

Roadmap. In Section 2, we introduce Use-After-Free exploits, and define three types of UAF vulnerabilities as the basic theory in our designed UAF-GUARD. Section 3 defines the key data structures, and proposes the technical details of UAF-GUARD. In Section 4, we give a theoretical security analysis. Section 5 implements a UAF-GUARD prototype system to evaluate the effectiveness and performance. In Section 6, we discuss the usage scenarios and scalability, and analyze the limitations about overhead. Section 7 gives the related work. Finally, the conclusion and future works are shown in Section 8.

Table 1
The off-the-shelf heap memory management mechanisms.

Platform	Heap Memory Management Mechanism
General purpose allocator	dlmalloc
Glibc	ptmalloc2
FreeBSD & Firefox	jemalloc
Google	tcmalloc
Solaris	libumem

2. Background

With the development of network information, more and more researchers are concentrated in the cyberspace. Recent researches from our laboratory are also worth mentioning, including Xu, Bai, Xing, Luo, Xiong, Cheng, Liu and Zheng (2022a); Xu, Dong, Xing, Lei, Liu, Gong, Feng, Zheng and Liu (2022b). As a security threat to the underlying computer system, uaf vulnerability deserves more attention in this new era of big data network. The principle of uaf vulnerabilities and the basic process of being exploited are introduced in this section. In addition, a definition rule is designed, which divides vulnerabilities into three types.

2.1. Use-After-Free Exploits

The command to allocate dynamic memory for a process in Linux operating system is `brk` or `mmap`, and the memory management mechanism is responsible for allocating memory for processes. The reason is that using memory directly is prone to generate the memory fragmentation. The off-the-shelf heap memory management mechanisms are shown in Table 1. After the allocated memory is freed by the heap memory management, there would be UAF vulnerabilities exploitation if the memory is used in an improper way. A typical process of UAF vulnerabilities exploitation can be described as follows.

step 1: Request for a block of heap memory allocated and managed by the heap memory management mechanism.

step 2: Free the memory block.

step 3: Find a certain pointer, which points to the memory or can access it after calculation.

step 4: Exploit this pointer to read, write or execute operations.

step 5: In some cases, this memory may be re-allocated and reused.

Steps 1-2 are executed by the original program, while steps 3-5 are carried out by attackers.

2.2. Three Types of UAF Vulnerabilities

To clearly demonstrate the superiority of UAF-GUARD, the UAF vulnerabilities is defined into three types according to the source of exploited pointers, which are shown in Table 2

Table 2
Three types of UAF vulnerability.

UAF vulnerability Type	Attack Means
Type I	original pointer
Type II	fulfilled by pointer assignment
Type III	fulfilled by calculating relative offset

```
int main(){
    char *ptr = (char *)malloc(20);
    free(ptr);

    strcpy(ptr, "Use-After-Free");
    return 0;
}
```

Figure 1: An example of Type I UAF vulnerability.

```
 1 int main(){
 2     char *ptr = (char *)malloc(20);
 3     char *ptr2 = ptr;
 4     _int64 val = (_int64)ptr;
 5
 6     free(ptr);
 7     ptr = 0;
 8
 9     strcpy(ptr2, "Use-After-Free");
10     strcpy((char *)(val + 0x8),
11            "Use-After-Free");
12      return 0;
13 }
```

Figure 2: An example of Type II UAF vulnerabilities.

Type I is the most common UAF vulnerability, which can directly use pointers allocated with corresponding memory space. Such vulnerabilities are caused by the failure to cancel the reference of the pointer in time and cause the pointer to be dangling. We provide a simple example in Figure 1.

In Type II, the pointer is obtained through pointer propagation, and the pointer must rely on an existing pointer to the UAF vulnerability. As shown in Figure 2 (at line 9), when the memory space is allocated, the pointer variable `ptr` will be returned, but the value of the pointer variable `ptr2` is the same as `ptr` through assignment. UAF vulnerabilities caused by `ptr2` should belong to type II.

Type III pointers are obtained by directly calculating the offset. Type III vulnerabilities usually need to be combined with other vulnerabilities, such as integer overflow and stack segment overflow. Figure 2 (line 10) shows a simple example of Type III.

3. UAF-GUARD Design

The overview of the UAF-GUARD is presented firstly in this section. In addition, two works have been done to improve the performance of our schema: 1) two key data structures and 2) a target instruction filtering algorithm to optimize overhead. We will introduce our technical details from static instrumentation and runtime defending.

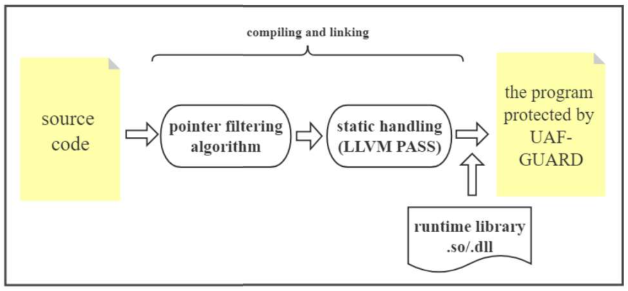

compiling and linking
source code
pointer filtering algorithm
static handling (LLVM PASS)
the program protected by UAF-GUARD
runtime library .so/.dll

Figure 3: The overall framework of UAF-GUARD.

3.1. System Overview

UAF-GUARD checks the memory pointer permission of each pointer operation at runtime, and strictly restricts the use of pointers in user space. This method could effectively detect UAF vulnerabilities theoretically. UAF-GUARD is lightweight, it can reduce the risk of various heap vulnerabilities, prevent the exploitation of heap vulnerabilities, especially for UAF vulnerabilities, and can also pinpoint the location of UAF vulnerabilities. Besides, it cannot be bypassed by off-the-shelf exploitation.

As described in Section 2.1, a successful exploitation over a UAF vulnerability requires: 1) the memory management mechanism frees the allocated memory, 2) attackers read, write or operate the freed memory, and 3) subsequent exploitation, including reallocating the freed memory, information leakage and control flow hijack. Traditional solutions nullify the corresponding pointer to prevent the pointer from reading, writing or operating the memory after *step 1)*. But this is not valid for Type III exploitation. If the exploitation is detected in *step 2)*, it can be defended easily. In essence, *step 2)* is to exploit the pointer to illicitly operate the freed memory. In contrast, we mark each memory block status in UAF-GUARD, which can be represented in meta-data and doubly linked list (see Section 3.2). Different from traditional process memory permissions which are coarse-grained and generally divided by segments, UAF-GUARD manages the memory permission in a fine-grained way to prevent pointer from being abused.

An overview of UAF-GUARD based on the LLVM llvm-admin team (2018) compiler framework/system is shown in Figure 3. UAF-GUARD participates in the compilation and linking process of the source code so that it can: 1) insert the instrumentation code in the compiled target program, 2) link the compiled function library with the binary program during the linking process, and 3) implement the instrumentation function to produce the protected binary program. Besides, we will design and deploy an error-handling mechanism in UAF-GUARD, for the purpose of mitigating the risk of DoS attack yielded by improper pointer handling. Finally, to provide as much as vulnerability information for maintainers, a module to record the vulnerability log is designed.

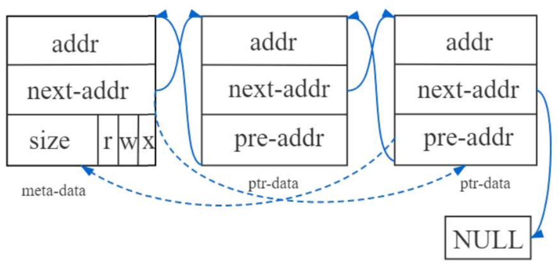


Figure 4: Data structure of meta-data, ptr-data and the relationship between memory and pointer.

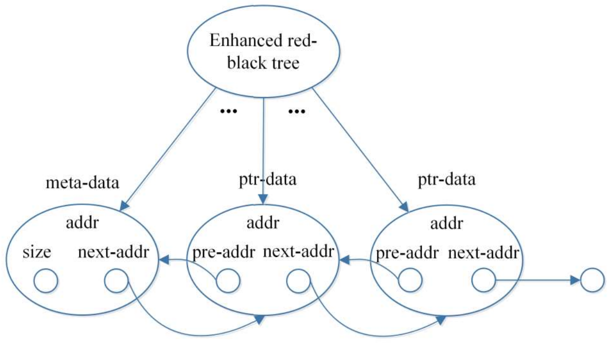


Figure 5: Data structure of the enhanced red-black tree.

3.2. Proposed Key Data Structures

3.2.1. Data Structure for Memory, Pointer and Their Relationship

"Special" data structures is designed to record the information of memory, pointer and their relationship. Metadata is used to describe the basic unit of a storage block. As shown in Figure 4, the metadata includes three fields: `addr`, `next-addr`, and `size|rwx`. `addr` indicates the starting position of the data in the memory block. `next-addr` represents the address of the next data structure in the linked list associated with the memory, and the last field `size|rwx` consists of two parts. `size` is the size of the memory. Since the memory block allocated by `ptmalloc2` is 8 byte aligned in 32-bit operation system - i.e. the size of this field is a multiple of 8, and thus we can utilize the last three bits of this field (`rwx`) to store the permissions (i.e., read, write, execute) of memory.

The ptr_data in Fig 4 depicts the data type ptr-data which describes the pointer information. ptr-data has three fields: `addr`, `next-addr`, and `prev-addr`. `addr` indicates the address of pointer variable. `next-addr` shares the same definition as in meta-data. The last field `pre-addr` indicates the previous data structure address in the linked list associated with the pointer. This field contains two types of addresses: 1) data structure type addresses, and 2) addresses of memory metadata with memory permissions.

Figure 4 shows the data structure of the relationship between the memory and the pointer. UAF-GUARD operates through pointers (including inserting and deleting elements), through a double-linked list, and uses fields containing pointer information for data types of metadata and ptr-data to describe the permissions of pointers to memory.

The data types proposed in this section are used to describe memory and pointer information. The data type of the meta-data is accurate to bit, which greatly decreases the space requirement for UAF-GUARD. The doubly linked list is sufficient to describe the many-to-one relationship between pointers and memory. As for the time overhead, if the size of the doubly linked list is m (where the number of nodes is m), the complexity of traversal operation is bounded by $O(m)$, and the complexity of inserting and deleting operations is $O(c)$ (where c is a constant). In practical applications, m is small, the number of pointers linked to the same memory is relatively small, therefore the complexity of the traversal operation is acceptable.

3.2.2. Enhanced Red-black Tree For Quick Query

An enhanced red-black tree is utilized to quickly query the meta-data and ptr-data.

Statistically, more than 80% of the operations in the UAF-GUARD query the data of the corresponding data type through the pointer address or the memory address. Therefore, query optimization is necessary. An enhanced red-black tree based on the typical red-black tree algorithm is designed. The data structure is shown in Fig 5.

The enhanced red-black tree is derived as follows:

1) The type of the red-black tree nodes is expanded. The memory or the pointer address and other fields in corresponding data type is used as the key and the node extension contents, respectively. When the key is searched out, the complete data information can be obtained at the same time.
2) The ability of preserving the relationship for the red-black tree is expanded. Through the accurate storage of the meta-data and the ptr-data, the data structure of the doubly linked list could be maintained. Therefore, the associated data can be quickly obtained via traversing the doubly linked list (when the node in the doubly linked list is searched out).
3) The search way of the red-black tree nodes is expanded. The following means is leveraged to decide if a memory address `addr` has been allocated: A memory block in the red-black tree is located firstly, which is closest to the `addr`, whose address is not larger than `addr`. If `addr` is in the range of this memory block, the `addr` is an allocated heap memory address; otherwise, the `addr` is not allocated or it has been freed.

The enhanced red-black tree enjoys the following advantages.

1) It enjoys high query efficiency which is as stable as that of the typical red-black tree. Although the implementation of red-black tree may be sophisticated, it also maintains good efficiency in the worst cases (w.r.t. query). As for the time overhead, when the number of nodes is n, the

complexity of searching, inserting and deleting nodes only requires *O(log n)*.

2) Since UAF-GUARD leverages the address as the unique identity, the address can be used as the query key. Besides, we can easily present the partial order relationship for the integer, thus it is convenient to build a fast and efficient query data structure.
3) Since there is no any overlap between memory blocks, the tree provides a great convenience for searching in UAF-GUARD. There are also plenty of accesses way similar to tree structure in real world situations. For example, the parent structure in the browser contains pointers to multiple substructures. When accessing the content of the substructure through the parent structure pointer, the conversion to IR is an access to a certain address in the heap memory block. In this regard, the tree structure could be appropriate to heap memory blocks.

In conclusion, the tree significantly improves the efficiency of searching the pointers & memory and obtaining the permission of the pointer to memory, and expands the record range of the heap memory so as to precisely locate the location that the pointer points to.

3.3. Technical Details

The design of static detection mechanism and runtime library for defense is introduced.

3.3.1. Static Instrumentation

The UAF-GUARD static tool in the LLVM llvm-admin team (2018) compiler framework is designed and implemented. The goals of UAF-GUARD we designed are as follows:

1) Generate the corresponding meta-data (see Section 3.2) for each memory allocating.
2) In order to protect all heap memory blocks, for each pointer block, propagate associated memory information or update pointer memory permissions.

In the compilation phase, the following handlings is considered.

1) Analyze each instruction in each function through LLVM IR pointer operation, and store related parameters and opcodes through preset variables.
2) Classify according to the instructions parsed in 1), and then insert the correct detection function according to the specific instruction content.

The instrumentation functions and their corresponding instructions are described in Table 3. We will describe in detail the processing details of each type of instruction in LLVM IR below:

1) Memory allocation, reading and writing instructions (at lines 1-4 in Table 3).
   - *Memory allocation/free instructions.* It can be seen that the `alloca` instruction requests to the stack for memory space, thus it is not our target. Obtaining memory space requires calling instructions to send the request to the heap. Taking C++ (Ubuntu Linux x86 system) as an example, we use the `call@malloc(k)` instruction to operate, where *k* is a positive integer and the return value is `(i8*)ptr`. In UAF-GUARD, if the memory block is successfully allocated, the record of the memory block allocated by the detection function `create(ptr, k)` will be inserted into the data structure. When releasing memory, we use `call@free(ptr)` instruction. Before inserting the instruction, the detection function `remove(ptr)` needs to be inserted to delete the record of the memory block. If the record is successfully deleted, the command will be executed normally.
   - *Memory read instruction.* The instruction is read from the memory. In order to prevent the instruction from invalidating the pointer, the check function `check(ptr)` is inserted before the load instruction, usually `%val = load i32, i32* %ptr`. At this time, reading the process memory relies on the pointer, which is equivalent to the pointer used by the read permission. If `check(ptr)` does not raise an exception, the instruction is successfully executed (but not vice versa). If `val` is a pointer type, the detection function `trace(ptr, val)` is inserted before the instruction. Here we introduce "permission propagation": if `ptr` is a resolved effective heap memory address, then `val` will obtain memory permission. Otherwise, the instruction cannot be executed because `ptr` is an invalid address at this time.
   - *Memory write instruction.* This operation uses `store i32 3, i32* %ptr`. The check function `check(ptr)` is inserted before writing the instruction. Because writing to the process memory is operated by pointers, it is equivalent to writing through pointers. In the case that `check(ptr)` does not throw an exception, the instruction will be executed.
2) Arithmetic operation instructions and bit operation instructions (at line 5 in Table 3), including `add/fadd`, `sub/fsub`, `mul/fmul`, `udiv/sdiv/fdiv`, `urem/srem/frem`, `shl`, `lshr`, `ashr`, `an-d`, `or`, `xor`. The instructions associated with the heap memory pointer are UAF-GUARD's only concern. When the pointer `ptr` is included in the arithmetic result, it is necessary to insert a detection function `trace(val, ptr)` before the instruction, and then directly assign the result to the pointer `ptr`. That is, the pointer variable `ptr` will be assigned to the value `val` calculated by the instruction to change the address pointed to by the pointer. If the value is a valid address, `ptr` will get memory permission, and the UAF-GUARD data structure will also update the record of `ptr`'s memory permission.
3) Atomic memory modification instructions (at line 6 in Table 3), including `cmpxchg`, `atomicrmw` and `getelementptr`. Since all of these instructions include memory reading

Table 3
The instrumentation functions and their corresponding instructions.

Instructions Type	Instructions	Instrumentation	Function Description
Memory Request	call@malloc(k)	create(ptr, k)	build the permission of pointer to memory
Memory Free	call@free(ptr)	remove(ptr)	remove the permission of pointer to memory

Memory Read	%val = load i32, i32* %ptr	check(ptr)
		trace(ptr, val)
			check/trace the permission of pointer to memory

Memory Write	store i32 3, i32* %ptr	check(ptr)	check the permission of pointer to memory
Arithmetic operation & bit operation	add/fadd, sub/fsub, mul/fmul udiv/sdiv/fdiv, urem/srem/frem shl, lshr, ashr, and, or, xor	trace(ptr, val)	trace the permission of pointer to memory

Atomic memory modification	cmpxchg, atomicrmw, getelementptr	check(ptr)
		trace(ptr, val)
			check/trace the permission of pointer to memory

and writing, and pointer permission propagation, the instrumentation function `check()`, `trace()` is chosen to be inserted, which is the same as the description in the previous types.

3.3.2. Runtime Defending

The UAF-GUARD runtime library is designed to complete the task of runtime defense, according to the preset detection functions during the compilation of static instructions. There are four main instrumentation functions, namely `create()`, `trace()`, `check()`, and `remove()`. In our design, the information of memory block and pointers is stored in the enhanced red-black tree (please refer to Section 3.2.2), while the relationship of memory and pointer is described by doubly linked list (see Section 3.2.1). In this section, we will elaborate the internal logic of the instrumentation functions.

create(). The establishment of the permission relationship between the pointer and the memory is implemented by the instrument function `create(ptr, k)`. `ptr` is a pointer to the memory address allocated by the heap memory management mechanism, and `k` is the size of the heap memory space requested by the user, in bytes. The technical details of `create()` are as follows:

1) If the value of `ptr` is null, it means that `malloc()` has not successfully allocated memory, and the establishment of permission relationships will be stopped.
2) Otherwise, the establishment of the metadata of the storage block and the ptr data of the pointer is based on the values of `ptr` and `k`, and the relationship between the metadata and the ptr data will also be established in the double-linked list.

The key of meta-data is the memory address, while the key of ptr-data is the pointer address.

check(). The permission to check the pointer to the memory is implemented by the tool function `check(ptr)`. The `ptr` is the pointer to be checked. The process of `check()` is described as follows:

1) Obtain the address and value of the pointer through `ptr`.
2) Search for pointer information in the tree structure. If ptr-data is not found in the pointer, it indicates that the pointer has been abused and the UAF vulnerability will be exploited. The current status information will be recorded including CPU, memory, function-call stack, and the program throw the corresponding exception.
3) Otherwise, if the information of ptr-data is obtained by us, the metadata of the memory associated with the pointer in the doubly linked list can be further tracked, and the range of the memory block can be calculated to determine whether: a) The value of `ptr` is within the range to prevent passing Allocate or calculate the offset to obtain the pointer; b) The pointer has the corresponding authority in the reading, writing and execution of the memory.

The `ptr` can pass the test only when the above two conditions are met. Otherwise, there is a unauthorized operation of pointer, which can yield the UAF vulnerabilities. Namely, the current status and the related information of the illegal operation will be recorded and the instruction will be forbidden.

trace(). The transfer of memory permissions between pointers is implemented by the instrument function `trace(ptr1, ptr2)`. The `ptr1` is a heap memory address or a pointer that points to a heap memory block, and the `ptr2` is the pointer to be assigned. Overloaded functions with different parameters are handled by `trace()`. The process of `trace()` is as follows:

1) If `ptr1` is a pointer, its legality is determined by its address and value.
2) If `ptr1` is the heap memory address, whether the address is valid and whether the allocated heap memory address will be the focus of our confirmation.
3) If `ptr1` is illegal, a program exception will be triggered and the status information will be recorded.
4) If `ptr1` is legal, whether the information of `ptr2` has been established will be checked by us. If not, the ptr-data of `ptr2` will be constructed by us and inserted into the tree structure. Finally, after inserting `ptr2` into the two-way linked list of `ptr1`, the permission transfer is complete.

remove(). The authority to delete the pointer to the memory is implemented by the tool function by `remove(ptr)`. `ptr` is the heap memory address or pointer to the heap memory

Algorithm 1 Target Instruction Filtering Algorithm

```
 1: for function in Code do
 2:     for instruction in function do
 3:         lhs = instruction.lhs
 4:         rhs = instruction.rhs
 5:         op = instruction.op
 6:         if isPointer(lhs) and pointToHeap(lhs) then
 7:             goto Processer
 8:         end if
 9:         if isPointer(rhs) and pointToHeap(rhs) then
10:             goto Processer
11:         end if
12:         continue
13:     end for
14: end for
15:
16: Processer(op,lhs,rhs)
```

block. `remove(ptr)` is an overloaded function, the process is as follows.

1) If `ptr` is a heap memory address (in this sense, the `remove()` corresponds to the `free()` in `ptmalloc2`), we check if the address is valid. If no, it indicates that the memory to be freed is the pointer's unprivileged memory, unallocated memory or freed memory. the status information is recorded including current stack frame, registration, and the program throws an exception and will be terminated. Otherwise, the pointer information ptr-data of all associated pointers and the metadata of the corresponding storage block will be searched and the data structure maintained. After that, these elements will be removed from the tree structure and free up space.
2) If `ptr` is a pointer, it dedicate that the pointer life cycle has ended is the removal permission. We need to determine whether `ptr` exists in the tree structure. If they are, they will be deleted from the two-way linked list and tree structure. Otherwise, it indicates that the pointer does not point to the storage block.

3.4. Optimization

Many instructions including the pointer operations are involved in a system. A target instruction filtering algorithm therefore needs to be proposed to filter the unrelated instructions, in which the pointers do not point to the heap memory. In the compilation phase, the algorithm only detects the target pointers which point to the heap memory rather than stack frame. On one hand, there are many protections to increase the difficulty of stack frame utilization, e.g., ASLR SearchSecurity (2014), Canary UMWiki (2015) and CFI Zhang, Wei, Chen, Duan, Szekeres, McCamant, Song and Zou (2013). On the other hand, the pointers in the stack frame have such a short life-cycle that it is difficult to be continuously utilized.

Through the filtering algorithm, the detection of most non-target pointers can be prevented. The details of the algorithm are given in Algorithm 1. At lines 1-2, all the instructions in function are traversed repeatedly. The current instruction information is obtained at lines 3-5. After that, we check if the instruction includes the pointer to heap memory at lines 6-11, if yes, the program will goto `"Processer()"`, and otherwise it jumps out of this loop (at line 12). At the end, we pick out the instructions containing pointers to heap memory.

The completeness of the target instruction filtering algorithm can be analyzed as follows:

1) If the pointer points to heap memory directly, our UAF-GUARD will be able to detect it.
2) If the pointer to the non-heap memory can point to the heap memory through discontinuous access, it will still be detected as the target pointer when subsequent utilization occurs.
3) If the pointer to the non-heap memory realizes the data leakage or overflow through continuously read, write or operating other functions, it will surely go through the no memory mapping regions between the segments and access the memory illegally, which will trigger the "Segmentation fault" to terminate the process.

4. Security Analysis

In this section, we analyze the security of UAF-GUARD in theoretical level. The analysis mainly considers two aspects: one is the detection capability, and the other is the risk of being bypassed.

For Type I, UAF-GUARD establishes the pointer-to-memory relationship right after the heap memory blocks are allocated. This can be used to prevent any memory blocks breaking away from monitoring and therefore, our system is able to reduce the false negative rate. From the view point in the life-cycle of pointer, UAF-GUARD maintains the pointer-to-memory relationship to reduce the false positive rate prior to the operating of pointer. Once the target heap memory is freed, UAF-GUARD removes the permission of pointer to memory immediately. Therefore, when the memory is re-operated by the pointer, the exploitation of the vulnerability can be prevented effectively, and the details of the vulnerability can be reported.

For Type II, UAF-GUARD pays attentions to the problem of pointer permission propagation and further, designs the corresponding instrumentation function for each situation (please refer to Section 3.4). In this way, both the false positive and negative rates can be reduced.

For the more advanced exploitation, Type III, UAF-GUARD could also detect pointers according to their address. This type of UAF vulnerability is to use the pointer obtained from the relative offset to exploit the vulnerability. Regarding this feature of the vulnerability, it doesn't mean that Type III could block or prevent `check()` or any other instrumentation functions. Since the Type III pointers has never been recorded in data structures or has not previously obtained permission to the memory block through the

Table 4
Comparison between UAF-GUARD and DANGNULL. Note the same UAF vulnerabilities are chosen for the comparison. Test I is the implementation of Type I UAF vulnerability in Figure 1, while test II and III are the implementations of Type II, III respectively in Figure 2. By "SIGSEGV", "TRIGGER", "NORMAL" and "ASSERTION" we mean "throw a exception", "trigger the vulnerability", "run in a normal way", and "the security assertion of Chrome", respectively.

Vulnerabilities	Incidence	Position Of Vulnerabilities in Compiled Binary	Type	Detection Result	
				UAF-GUARD	DANGNULL
CVE-2010-2939	OpenSSL 1.0.0a,0.9.8,0.9.7	0x80000000022ba510	II	SIGSEGV	SIGSEGV
CVE-2016-4077	Wireshark 2.0.0-2.0.3	-	II	SIGSEGV	SIGSEGV
CVE-2013-2909	Google Chrome < 30.0.1599.66	0x1bfc9901ece1	II	SIGSEGV	NORMAL
CVE-2013-2909	Google Chrome < 30.0.1599.66	0x7f2f57260968	II	SIGSEGV	NORMAL
CVE-2013-2918	Google Chrome < 30.0.1599.66	0x490341400000	III	SIGSEGV	TRIGGER
CVE-2013-2922	Google Chrome < 30.0.1599.66	0x60b000006da4	II	SIGSEGV	NORMAL
CVE-2013-6625	Google Chrome < 31.0.1650.48	0x897ccce6951	II	SIGSEGV	SIGSEGV
CVE-2012-5137	Google Chrome < 23.0.1271.95	0x612000046c18	II	SIGSEGV	ASSERTION
Our Test I	-	-	I	SIGSEGV	SIGSEGV
Our Test II	-	-	II	SIGSEGV	SIGSEGV
Our Test III	-	-	III	SIGSEGV	TRIGGER

normal process, it cannot pass the security check, and its exploitation will be prevented.

The design of UAF-GUARD is quite similar to that of static analysis, which requires the clear description of the pointer-to-memory relationship. It is noted that UAF-GUARD can only detect the instructions at program runtime. If the undetected part is subsequently exploited, UAF-GUARD can also detect it. That is the reason why UAF-GUARD can be seen as an "upgraded" version of the static analysis.

5. Experimental

To verify the effectiveness and efficiency of UAF-GUARD, the UAF-GUARD prototype is here implemented. Our experiments include the security and performance tests. The security test covers all types of UAF vulnerabilities and some CVE vulnerabilities, while the performance one presents the analysis of the compiling/linking process and actual running costs. We do compare UAF-GUARD with others in the above tests.

From the tests, we show that UAF-GUARD can guarantee the security of the program at runtime but also protect it from UAF vulnerabilities exploitation. At the same time, the overhead of UAF-GUARD can also satisfy the efficiency requirement of the program at runtime.

5.1. Experimental Environment

several mainstream programs are used to evaluate the safety and performance of UAF-GUARD.

1) UAF-GUARD's defense effectiveness is evaluated against all types of UAF vulnerabilities.
2) Further testing for known CVE vulnerabilities in multiple versions of multiple programs including Chrome, Wireshark and OpenSSL.
3) The time cost of UAF-GUARD in the compilation phase is calculated. For the sake of simplicity and fairness in the comparison, the same pending procedures are utilized in related works, such as bzip2 and gcc.
4) Based on the compilation results, the time overhead of UAF-GUARD at runtime is analyzed, and it is proved that UAF-GUARD can ensure the security of the program at runtime.

All the experiments in this article are done on a PC. The configuration is as follows: eight-core Intel Core i7-6700HQ CPU @ 2.60GHz, 16 GB RAM and 500 GB SSD, 64-bit Ubuntu 16.04 (Linux Kernel 4.16).

5.2. Effectiveness of Detection

The test benchmark set used in the experiment is a number of chromium versions that contain more vulnerabilities, which is the same as test benchmark set of DANGNULL. In addition, the open loopholes of OpenSSL and Wireshark are utilized, in order to enhance the reliability of the experiment.

Table 4 is the comparison results between UAF-GUARD and DANGNULL. For our designed tests for the types of vulnerabilities (Test I, II, and III), we stress that the experimental results of UAF-GUARD and DANGNULL are different through analyzing the exception sinceinformation. The SIGSEGV in UAF-GUARD is thrown by UAF-GUARD itself, UAF-GUARD detects the instruction that the pointer illegally operates the memory, and automatically terminates the process to prevent pointer abusing. However, the SIGSEGV in DANGNULL is caused by its referring the memory of the `0x0` address, which can not be read, written and executed. The type III UAF vulnerabilities cannot be detected by DANGNULL, because the pointer information is insufficient (along with the result) so that the target pointer can not be traced. Note that this is also the untraceable reason for the result of CVE-2013-2918. In contrast, UAF-GUARD is able to detect and terminate the execution of instruction in time. In the case of vulnerability CVE-2013- 2909 and CVE-2013-2922, DANGNULL will not report an exception, because its design principle is to make the dangling pointer

Table 5
Changes in file size during compilation and connection. âĂIJBefore CompilationâĂİ: the size of program before compiling. âĂIJIncreaseâĂİ: the increased size of program. âĂIJInstrumentationâĂİ: the number of instrumentation functions.

Name	Language	File Size					Instrumentation	
		Before	Increase					
			DANGNULL	Percentage	UAF-GUARD	Percentage	DANGNULL	UAF -GUARD
gcc	C	8380KB	768KB	4.7%	584KB	2.3%	9264	6342
soplex	C++	4292KB	453KB	1.9%	422KB	0.7%	264	278
povray	C++	3383KB	513KB	4.2%	458KB	2.0%	941	654
h264ref	C	1225KB	420KB	4.1%	418KB	2.3%	154	167
gobmk	C	5594KB	416KB	0.8%	420KB	0.5%	201	219
chromium	C++	1858MB	10MB	0.5%	7M B	0.3%	140k	87k
sjeng	C	276KB	386KB	5.8%	396KB	2.2%	17	17
namd	C++	1182KB	382KB	1.1%	408KB	1.5%	45	53
hmmer	C	814KB	396KB	3.2%	412KB	2.7%	94	97
sphinx3	C	541KB	389KB	3.5%	416KB	4.8%	170	139
milc	C	351KB	386KB	4.6%	410KB	5.7%	71	82
astar	C++	195KB	378KB	4.1%	408KB	9.2%	54	59
bzip	C	172KB	378KB	4.7%	398KB	4.7%	13	15
mcf	C	53KB	376KB	11.3%	404KB	26.4%	95	106
libquantum	C	106KB	378KB	7.5%	400KB	9.4%	21	21
lbm	C	37KB	374KB	10.8%	393KB	8.1%	9	9

Table 6
The rendering time of accessing web pages by the chromium, which is the mean of the results from one thousand experiments.

Website	Complexity of Page		Rendering Time				
	Requests	DOM Nodes	Original Time(s)	DANGNULL(s)	Increase(%)	UAF-GUARD(s)	Increase(%)
qq.com	92	2604	0.53	0.75	41.5	0.67	30.2
youtube.com	54	2397	3.11	4.13	32.8	3.72	25.4
baidu.com	21	142	0.21	0.25	19.0	0.25	19.0
taobao.com	80	1069	0.31	0.38	22.6	0.38	22.6
google.com	24	360	1.11	1.35	21.6	1.35	22.5
amazon.com	216	1508	2.28	2.66	16.7	2.67	17.1
gmail.com	52	240	1.82	2.23	22.5	2.24	23.1
twitter.com	18	668	3.45	3.81	10.4	3.96	15.1
Average	80	1124	1.73	1.95	21.7	1.94	21.9

empty and allow the program to enter the path of normal execution through other branch structures. As a result, the location of the vulnerabilities cannot be reported as UAF-GUARD does.

Through the above comprehensive analysis based on vulnerabilities, UAF-GUARD is highly competitive with others w.r.t. UAF vulnerabilities at runtime. Although DANGNULL is lightweight, the insufficient pointer information makes it deficient and suffers from the risk of bypassing.

5.3. Performance

Two aspects are considered in the performance test:

1) We tested the program under static compilation and linking, including changes in file size and command functions.
2) We also tested the extra overhead incurred by UAF-GUARD at runtime.

5.3.1. Overhead at Compiling and Linking Process

Program changes are measured by calculating the difference between file sizes before and after compilation and linking, and collected statistical information for detection functions in experiments. The experiment counts related information under the compiling and linking state of 16 programs. Our test of UAF-GUARD is based on the LLVM compiler project, and the experimental results are shown in Table 5

From Table 5, for the smaller programs, the space overhead of UAF-GUARD is quite similar to that of DANGNULL. But, for the larger programs, the space overhead of the program is less than DANGNULL with a decrease from 0.5% to 0.3%.

The fixed runtime library of UAF-GUARD is 390KB, which is a huge increase for small programs like `mcf`. However, the average overhead can reach 5.2% if we minus the size of the fixed runtime library. For the larger programs, the changes of program is less than DANGNULL, since the UAF-GUARD filters out non-target pointers through the target filtering optimization (please refer to Section 3.4). For example, in the `chromium` case, UAF-GUARD just increases 0.3% overhead which is less than that of DANGNULL, 0.5%. Besides, DANGNULL suffers from the redundant operations caused by insufficient pointer information. For the smaller programs, the instrumentation function is required to transmit more information than DANGNULL when handling partial pointer in UAF-GUARD. Accordingly, UAF-GUARD has more variation than DANGNULL. But in terms of numerical values, it should be acceptable in practice. In summary, the performance of UAF-GUARD in the compiling and linking phase is competitive and attractive.

The number of detection functions is different. This is due to the fact that the size of each program is different, and this also leads to the fact that the file size changes are also inconsistent. UAF-GUARD needs the support of the runtime library, which is about 390KB in size. We recommend precise detection and defense against UAF exploits, and this extra overhead is very necessary to record the pointer state.

5.3.2. Runtime Overhead

Due to its distinct features, UAF-GUARD performs better in large scale programs. To simplify the comparisons, we only use one software, namely chromium, in test. The chromium is selected in the experiment because it brings convenience in obtaining the source code and meanwhile, the input of compilation and test can be implemented in a relatively lightweight way.

A chrome plug-in for the UAF-GUARD is designed and implemented, which records the results of accessing the Alexa top 100 websites. Due to space limit, we here only list the results for the top 8 (out of the 100) websites in Table 6. Each set of the results represents the total time taken by the website from the loading to finishing rendering of the webpages, where the time is the mean from the 1000 repeated experiments. From this, we try to precisely reflect the user experience after deploying our UAF-GUARD.

In order to control the variables, the following means need to be taken:

1) All web requests are from the same network environment.
2) Accessing the local caches is not allowed.
3) All processes are terminated that may interfere with data communication.

The complexity of the defense process in our UAF-GUARD is determined by the number of DOM nodes after the webpages are rendered. The results are shown in Table 6. According to the data in Table 6, the time cost of the two methods is similar, and most of them are within 0.5s.

In order to compare the extra overhead of UAF-GUARD with others more comprehensively, select various open source programs are selected as the test set. Different programs contain distinct number of pointer operations and the operations on heap memory. This, we state, may present diverse comparison in defending efficiency. The experimental results are shown in Fig 6

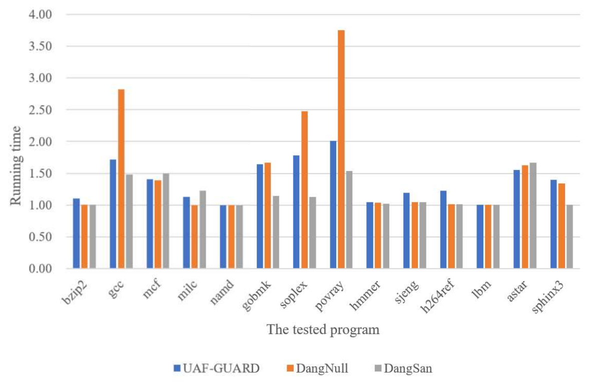


Figure 6: The comparison results of the running time.

UAF-GUARD and DANGNULL have similarities in design, but the main difference is that UAF-GUARD maintains the relationship of pointer-to-memory. Therefore, UAF-GUARD may have an extra constant complexity, which is less favourable when the program size is small or with less heap memory operations. However, the overhead difference (between the two methods) does not exceed 5% and if the program is getting more complicated, the time overhead will be much lower than that of DANGNULL (thanks to the accurate target pointer filtering of UAF-GUARD and the optimization of the data structure). Note DangSan takes the same strategy as DANGNULL and its high efficiency relies on the use of multi-threading. We can conclude that the performance of UAF-GUARD is better than the other two solutions for the case where the programs do not support multi-threading, e.g., astar.

Through the code analysis, if there are many requests for heap memory space in the program, UAF-GUARD will have to generate a huge overhead, e.g., the time overhead will be doubled in the worst case. However, in most cases, the increased overhead is between 0% and 50%. This does outperform DANGNULL whose average increased overhead is 80%. In short, our UAF-GUARD provides an efficient defense solution at program runtime.

In addition, compared with static analysis, UAF-GUARD enjoys the following advantages:

1) It has small overhead, because it does not need the program branch coverage and memory representation. Therefore, UAF-GUARD can protect the program at runtime.
2) UAF-GUARD only detects the current execution path and stores the sufficient and valid pointer and memory information, which helps reduce the false negative rate.

We state that UAF-GUARD is competitive with both the previous static and dynamic analysis for the UAF vulnerabilities.

6. Discussion and Analysis

In this section, some usage scenarios are introduced firstly for UAF-GUARD, and then we discuss how to extend our work for practical cases. We also introduce an overview design of an automated vulnerability mining scheme on top of UAF-GUARD using the symbolic execution and fuzzing (which will be part of our future work). Finally, we have brief discussions over the limitations of UAF-GUARD.

6.1. Usage Scenarios

Being designed to detect and defend UAF vulnerabilities, our analysis shows that UAF-GUARD outperforms the other existing works in terms of the overall performance with acceptable overload. In a word, it has the following usage scenarios:

1) UAF-GUARD is able to defend UAF vulnerabilities at runtime. As demonstrated in Section 5, UAF-GUARD has an acceptable overhead, and hence it is suitable for defending UAF vulnerabilities at runtime.
2) UAF-GUARD can be used for back-end UAF detection. Many zero-day attacks are based on UAF vulnerabilities Swamy, Cristian and Elia (2013); Liu, Zhang and Wang (2018). UAF-GUARD can detect these attacks.
3) Considering the fact that our UAF-GUARD performances better in terms of overhead in complicated programs, it is more suitable for such large scale software like Chromium.
4) UAF-GUARD can not only defend all the three types of UAF vulnerabilities completely, but also be expediently transplanted to further defend other types of heap vulnerabilities, such as Heap Overflow, and Double Free.

6.2. Scalability

As mentioned previously, UAF-GUARD is able to precisely identify the location of vulnerabilities, based on what it can achieve the automated vulnerability mining through introducing automatic analysis techniques, e.g., symbolic execution and fuzzing Stephens, Grosen, Salls, Dutcher, Wang, Corbetta, Shoshitaishvili, Kruegel and Vigna (2016); Xu, Kashyap, Min and Kim (2017). In the following paragraphs, we will show how to achieve the automated vulnerability mining using symbolic execution, and provide an overview idea for automated auditing on program vulnerabilities. Note that Angr Yi, Yang, Guo, Wang, Liu and Zhao (2018), which is one of the classic symbolic execution techniques, will be merged with our UAF-GUARD.

6.2.1. Basic Scheme of Automated Vulnerability Mining

Angr can help us analyze programs automatically. Specifically, Angr is able to define any content in a program as a symbol, traverse almost all execution paths automatically and further retain the logical expression containing the symbol according to such structures as program branch. If the symbol of the logical expression is solvable, we thus can obtain the corresponding symbol value which covers the code execution path. The overview steps of the scheme are as follows:

1) All variables are defined that users can control as symbols.
2) According to the design of UAF-GUARD, the path with the UAF vulnerability is selected to be the code execution path.
3) Using Angr to analyze and handle the program automatically, we can get a large number of logical expressions which contain the input symbols.
4) SMT Solver (Satisfiability Modulo Theories Solver, such as Z3 Research (2017)) is leveraged here to get the value of the logical expressions.
5) To further verify the vulnerability and give a PoC to prove the vulnerability, we artificially reproduce the vulnerability based on the value of the symbol.

The UAF vulnerability mining scheme based on the combination between UAF-GUARD and symbolic execution is presentable, because the PoC report is sufficient enough to prove the existence of a UAF vulnerability. Meantime, the scheme will enjoy efficiency and low overhead, since it focuses on covering the targeted path while "ignoring" other paths (which saves time in detection), where by the targeted path we mean those with vulnerabilities but verified by our UAF-GUARD.

6.2.2. Improved Scheme via Fuzzing

Although being able to locate the logical expression of the corresponding execution path, the basic scheme suffers from some difficulties in the expression evaluation. One of the difficulties is that the symbolic values may fall into a large domain (for example, a value may be represented as a string, or a file type) so that the SMT Solver has to take a long time to solve it. Luckily, such a problem can be optimized empirically. Another difficulty comes from the operators in logical expression (e.g., modulo and power operations), which will also need to spend a long time to solve it.

We stress that fuzzing (which is a common way of software testing nowadays due to its simple design pattern) can complement our overview scheme in such a way that we can use a certain data set for logical expression verification, and solve part of the symbols in order to accelerate the solving process.

In terms of algorithm implementation, we can easily improve the basic vulnerability mining scheme via fuzzing, so that the resulting scheme is applicable to the systems with considerable computing resources. In addition, we may still need to select an appropriate fuzzing data set to provide good code coverage.

6.3. Limitations

Although UAF-GUARD could defend the UAF vulnerabilities efficiently at runtime, its space and time overhead should be further saved, especially for the programs with restricted performance resources. In future works, we will try to reduce or merge the usage of instrumentation function

Table 7
Comparison of UAF-GUARD with other solutions. Note that *Runtime Method* indicates if the detection algorithm is for runtime, *Explicit Check* shows if the detection provides explicitly instrument check for UAF vulnerabilities except with pointer propagation, *False positive rate* indicates if the algorithm yields high/low number of false alarm on benign input, *Bypassing* shows how the detection can be bypassed with the exploitation techniques, and N/A indicates that the detection cannot be bypassed by far.

Detection Algorithm	Method	Runtime Method	Explicit Check	Low False Positive Rate	Limitation/Bypassing
DANGNULL Lee, Song, Jand, Wang, Kim, Lu and Lee (2015)	Dangling pointer nullifacation	✓	×	✓	Can't detect Type III UAF
Undangle Caballero, Grieco, Marron and Nappa (2012)	Dangling pointer taint tracing	×	✓	✓	N/A
CETS Nagarakatte, Zhao, Martin and Zdancewic (2010)	Associating pointer with memory and taint tracing	×	✓	×	N/A
AddressSanitizer Serebryany, Bruening, Potapenko and Vyukov (2012)	Memory error detector	×	✓	✓	Fabricate memory chunk or metadata in heap
Memcheck Nethercote and Seward (2007)	Memory error detector	×	✓	×	N/A
SafeDispatch Jang, Tatlock and Lerner (2014)	Control flow integrity	×	✓	✓	Modify non-control data
Cling Akritidis (2010)	Safe heap memory management mechanism	✓	×	✓	Fabricate heap memory chunk/controlling
Dieharder Novark and D. Berger (2010)	Safe heap memory management mechanism	✓	×	✓	Fabricate heap memory chunk/controlling
UAF-GUARD	Check permission of pointer to memory	✓	✓	✓	N/A

"check()", optimize the strengthen red-black tree structure and double linked list, and consider the memory reuse in the implementation.

7. Related Work

7.1. Static Detection and Defense

Static analysis uses small instruction sets, for example, REIL IL Dullien and Porst (2009), Bincoa Bardin, Herrmann, Leroux, Ly, Tabary and Vincent (2011) and BAP Brumley, Jager, Avgerinos and J. Schwartz (2011), to convert binary programs into intermediate representations, then use abstract interpretation Dolan-Gavitt, Hulin, Kirda, Leek, Mambretti, Robertson, Ulrich and Whelan (2016) and symbolic execution techniques Ye, Zhang and Han (2014) to analyze the intermediate codes, build control flow graphs, and extract vulnerabilities from the analysis results of the reverse software Guilfanov (2017). There are two long-lasting difficulties in static analysis: a) when a detector encounters a loop, it has to calculate the possible values of all the variables in the loop and to know when the loop terminates. This is known as the Turing Halting Problem and can't be fully addressed. b) the value of each memory address must be changed dynamically with corresponding execution paths, which may not scale well in practice.

7.2. Dynamic Detection and Defense

Fortunately, the above challenges do not exist in dynamic analysis. Below, we will have some discussions on the categories of dynamic analysis.

7.2.1. *Direct Detection*

Direct Detection usually defend UAF exploits by detecting the pointer directly.

Dangling pointer nullification. DANGNULL Lee et al. (2015) and DangSan Kouwe, Nigade and Giuffrida (2017) are the two classic direct detection systems which need to nullify dangling pointer in time. When carrying out the pointer propagation, DANGNULL and DangSan leverage data structure to nullify all pointers which point to the same freed memory. However, they yield the false negative w.r.t. the Type III UAF vulnerabilities and meanwhile, they fail to locate the position of the vulnerabilities.

Dangling pointer taint tracing. Undangle Caballero et al. (2012) could modify memory allocation function in heap memory management mechanism. It returns a unique label and the memory block address while allocating the heap memory and uses the dynamic taint analysis technique to track the propagation of these labels. However, this technique has to maintain the information of pointer propagation that yields an "extra" of detection overhead. In addition, it cannot be used to detect Type III UAF.

Associating pointer with memory and taint tracing. This technique, e.g., CETS Nagarakatte et al. (2010), generates a unique metadata when allocating the heap memory space to record the associated pointer. When taint propagation occurs during pointer allocation and computation, only the metadata is updated to maintain the relationship between pointers and memory. However, this approach does not consider the case where multiple pointers may point to the same memory after pointer propagation, which is therefore inevitable to yield the false negative.

7.2.2. *Indirect Detection*

Unlike direct detection, the indirect technique makes use of memory status detection, control flow integrity and secure heap memory management mechanism.

Memory error detection. Most of this type, such as AddressSanitizer Serebryany et al. (2012), and Memcheck Nethercote and Seward (2007), can detect the UAF vulnerabilities by maintaining the allocated memory status. Memcheck is a prevalent solution that is built on top of Valgrind to facilitate program debugging. However, high memory and CPU overhead required for error detection comes at a price. AddressSanitizer Serebryany et al. (2012) optimizes the representation of memory blocks, and its memory status acquisition methods are based on existing advanced error detection. However, this technique could be bypassed if the heap memory allocation process is compromised. For example, a possible strategy is to use Heap Spray Ratanaworabhan et al. (2009) to force the memory block to be reallocated on the released memory block, that construct a condition, in which the methodâĂŹs premise is invalid, and the exploitation of the UAF vulnerability could continue. On the contrary, UAF-GUARD could record the information of pointers and memory usage, while slidecode in Heap Spray is working. The detailed information records by two data structure enables attacks to be detected by `check()`.

Control-Flow Integrity (CFI). CFI Jang et al. (2014); Carlini, Barresi, Payer, Wagner and Gross (2015); Zhang et al. (2013) enforces the indirect function call to be legitimate, so as to prevent such control-flow hijacking (e.g., ROP attack Carlini and Wagner (2014)) effectively. In practice, most CFI methods leverage coarse-grained CFI to avoid heavy overhead and false positive. Nevertheless, recent work Davi et al. (2014); GÃűktas et al. (2014a,b); Chen, Xu, Sezer, Gauriar and Iyer (2005) has found that all current coarse-grained CFI methods can be bypassed.

Secure heap memory management mechanism. In Akritidis (2010), only the objects with the same type can reuse the memory. Nonetheless, the mechanism still can be bypassed through heap manipulation, such as heap memory forgery and the header modification of memory block. Oscar H.Y. Dang, Maniatis and Wagner (2017) is a page-permissions based realizations of lock-and-key protection scheme. It implements UAF defense by changing the shadow memory where the memory block is located. However, recycling of virtual pages need to be considered.

8. Conclusion

In this work, previous schema called UAF-GUARD is improved, based on the design of fine-grained memory permission management, aiming to identify and locate UAF vulnerabilities. We design a target instruction filtering algorithm to improve the efficiency of UAF-GUARD by reducing the detection overhead. We also expand the types of instructions scanned by UAF and improve the data structure stored between pointers and memory. Finally, we complete experiments on a 64-bit linux system, and the results show that UAF-GUARD is superior to other methods in accuracy and cost defending against three types of uaf vulnerabilities. In the future, we plan to improve the design and implementation of UAF-GUARD by symbolic execution and design an automated UAF vulnerability discovery tool.

References

Akritidis, P., 2010. Cling: A memory allocator to mitigate dangling pointers, in: Proceedings of the 19th USENIX conference on security (USENIX Security '10), USENIX Association, Berkeley, CA, USA. pp. 12–12.

Bardin, S., Herrmann, P., Leroux, J., Ly, O., Tabary, R., Vincent, A., 2011. The bincoa framework for binary code analysis, in: Proceedings of the 23rd international conference on Computer aided verification (CAV '11), Springer-Verlag, Berlin. pp. 165–170.

Brumley, D., Jager, I., Avgerinos, T., J. Schwartz, E., 2011. Bap: A binary analysis platform, in: CAV 2011: International Conference on Computer Aided Verification, Springer, Berlin. pp. 463–469.

Caballero, J., Grieco, G., Marron, M., Nappa, A., 2012. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities, in: Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA 2012), ACM, New York, NY, USA. pp. 133–143. URL: http://dx.doi.org/10.1145/2338965.2336769.

Carlini, N., Barresi, A., Payer, M., Wagner, D.A., Gross, T.R., 2015. Control-flow bending: On the effectiveness of control-flow integrity, in: 24th USENIX security symposium, USENIX Association, Washington. pp. 161–176.

Carlini, N., Wagner, D.A., 2014. Rop is still dangerous: breaking modern defenses, in: Proceedings of the 23rd USENIX conference on security symposium (SEC'14), USENIX Association, Berkeley, CA, USA. pp. 385–399.

Chen, S., Xu, J., Sezer, E.C., Gauriar, P., Iyer, R.K., 2005. Non-control-data attacks are realistic threats, in: Proceedings of the 14th conference on USENIX security symposium, Berkeley, CA, USA. pp. 12–12.

(CWE), C.W.E., 2018. Cwe-416: Use after free. URL: https://cwe.mitre.org/data/definitions/416.html. accessed October 11, 2018.

Davi, L., Sadeghi, A.R., Lehmann, D., Monrose, F., 2014. Stitching the gadgets: on the ineffectiveness of coarse-grained control-flow integrity protection, in: Proceedings of the 23rd USENIX conference on security symposium (SEC'14), USENIX Association, Berkeley, CA, USA. pp. 401–416.

Dolan-Gavitt, B., Hulin, P., Kirda, E., Leek, T., Mambretti, A., Robertson, W.K., Ulrich, F., Whelan, R., 2016. Lava: Large-scale automated vulnerability addition, in: 2016 IEEE Symposium on Security and Privacy (SP), San Jose, CA. pp. 110–121. URL: https://doi.org/10.1109/SP.2016.15.

Dullien, T., Porst, S., 2009. Reil: A platform-independent intermediate representation of disassembled code for static code analysis, in: Proceedings of Cansecwest, Vancouver.

Guilfanov, I., 2017. Hex-rays. URL: https://www.hex-rays.com/. accessed October 11, 2018.

GÃűktas, E., Athanasopoulos, E., Bos, H., Portokalidis, G., 2014a. Out of control: Overcoming control-flow integrity, in: Proceedings of the 2014 IEEE Symposium on Security and Privacy (SP '14), IEEE Computer Society, Washington. pp. 575–589. URL: https://doi.org/10.1109/SP.2014.43.

GÃűktas, E., Athanasopoulos, E., Polychronakis, M., Bos, H., Portokalidis, G., 2014b. Size does matter: why using gadget-chain length to prevent code-reuse attacks is hard, in: Proceedings of the 23rd USENIX conference on security symposium (SEC'14), USENIX Association, Berkeley, CA, USA. pp. 417–432.

Hat, R., 2012. Position independent executables (pie). URL: https://access.redhat.com/blogs/766093/posts/1975793. accessed October 11, 2018.

H.Y. Dang, T., Maniatis, P., Wagner, D., 2017. Oscar: A practical page-permissions-based scheme for thwarting dangling, in: Proceedings of the 26th USENIX Security Symposium, Vancouver, BC, Canada. URL: https://dx.doi.org/10.14722/ndss.2019.23541.

Jang, D., Tatlock, Z., Lerner, S., 2014. Safedispatch: Securing c++ virtual calls from memory corruption attacks, in: Symposium on Network and Distributed System Security (NDSS '14), San Diego, CA, USA. pp. 23–26. URL: http://dx.doi.org/10.14722/ndss.2014.23287.

Kouwe, E.v.d., Nigade, V., Giuffrida, C., 2017. Dangsan: Scalable use-after-free detection, in: Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17), ACM, New York, NY, USA. pp. 405–419. URL: https://doi.org/10.1145/3064176.3064211.

Lee, B., Song, C., Jand, Y., Wang, T., Kim, T., Lu, L., Lee, W., 2015. Preventing use-after-free with dangling pointers nullification, in: Symposium on Network and Distributed System Security (NDSS), ACM, San Diego, CA, USA. pp. 8–11. URL: http://dx.doi.org/10.14722/ndss.2015.23238.

Liu, D., Zhang, M., Wang, H., 2018. A robust and efficient defense against use-after-free exploits via concurrent pointer sweeping, in: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18), ACM, New York, NY, USA. pp. 1635–1648. URL: https://doi.org/10.1145/3243734.3243826.

Nagarakatte, S., Zhao, J., Martin, M.M.K., Zdancewic, S., 2010. Cets: compiler enforced temporal safety for c, in: Proceedings of the 2010 international symposium on Memory management, ACM, New York, NY, USA. pp. 31–40. URL: https://doi.org/10.1145/1837855.1806657.

Nethercote, N., Seward, J., 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation, in: Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07), ACM, New York, NY, USA. pp. 89–100. URL: https://doi.org/10.1145/1250734.1250746.

Novark, G., D. Berger, E., 2010. Dieharder: securing the heap, in: Proceedings of the 17th ACM conference on Computer and communications security (CCS '10), ACM, New York, NY, USA. pp. 573–584. URL: https://doi.org/10.1145/1866307.1866371.

Ratanaworabhan, P., Livshits, V.B., Zorn, B.G., 2009. Nozzle: a defense against heap-spraying code injection attacks, in: Proceedings of the 18th conference on USENIX security symposium (SSYM'09), USENIX Association, Berkeley, CA, USA. pp. 169–186.

Research, M., 2017. The z3 theorem prover. URL: https://github.com/Z3Prover/z3/. accessed October 11, 2018.

SearchSecurity, 2014. Address space layout randomization (aslr). URL: https://searchsecurity.techtarget.com/definition/address-space-layout-randomization-ASLR. accessed October 11, 2018.

Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D., 2012. Addresssanitizer: a fast address sanity checker, in: Proceedings of the 2012 USENIX conference on Annual Technical Conference (USENIX ATC '12), USENIX Association, Berkeley, CA, USA. pp. 28–28.

Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., Vigna, G., 2016. Driller: Augmenting fuzzing through selective symbolic execution, in: Symposium on Network and Distributed System Security (NDSS), San Diego, CA, USA. URL: http://dx.doi.org/10.14722/ndss.2016.23368.

Swamy, S.N., Cristian, C., Elia, F., 2013. Software vulnerability exploitation trends. URL: https://zh.scribd.com/document/262748190/Software-Vulnerability-Exploitation-Trends. accessed October 11, 2018.

llvm-admin team, 2018. The llvm compiler infrastructure. URL: https://llvm.org/. accessed October 11, 2018.

UMWiki, 2015. Canary (buffer overflow). URL: http://www.cbi.umn.edu/securitywiki/CBI_ComputerSecurity/MechanismCanary.html. accessed October 11, 2018.

Xu, G., Bai, H., Xing, J., Luo, T., Xiong, N.N., Cheng, X., Liu, S., Zheng, X., 2022a. Sg-pbft: A secure and highly efficient distributed blockchain pbft consensus algorithm for intelligent internet of vehicles. Journal of Parallel and Distributed Computing 194, 1–1–1. URL: https://doi.org/10.1016/j.jpdc.2022.01.029.

Xu, G., Dong, W., Xing, J., Lei, W., Liu, J., Gong, L., Feng, M., Zheng, X., Liu, S., 2022b. Delay-cj: A novel cryptojacking covert attack method based on delayed strategy and its detection. Digital Communications and Networks URL: https://doi.org/10.1016/j.dcan.2022.04.030.. available online 13 May 2022.

Xu, G., Li, M., Li, X., et.al, 2020. Defending use-after-free via relationship between memory and pointer, in: Proceedings of the 2020 EAI CollaborateCom 2021 - 17th EAI International Conference on Collaborative Computing: Networking, Applications and Worksharing.

Xu, W., Kashyap, S., Min, C., Kim, T., 2017. Designing new operating primitives to improve fuzzing performance, in: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17), ACM, New York, NY, USA. pp. 2313–2328. URL: https://doi.org/10.1145/3133956.3134046.

Ye, J., Zhang, C., Han, X., 2014. Poster: Uafchecker: Scalable static detection of use-after-free vulnerabilities, in: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14), ACM, New York, NY, USA. pp. 1529–1531. URL: https://doi.org/10.1145/2660267.2662394.

Yi, Q., Yang, Z., Guo, S., Wang, C., Liu, J., Zhao, C., 2018. Eliminating path redundancy via postconditioned symbolic execution. IEEE Transactions on Software Engineering , 25–43URL: https://doi.org/10.1109/TSE.2017.2659751.

Zhang, C., Wei, T., Chen, Z., Duan, L., Szekeres, L., McCamant, S., Song, D., Zou, W., 2013. Practical control flow integrity and randomization for binary executables, in: Proceedings of the 2013 IEEE Symposium on Security and Privacy (SP '13), IEEE, Washington. pp. 559–573. URL: http://dx.doi.org/10.1109/SP.2013.44.

Biography

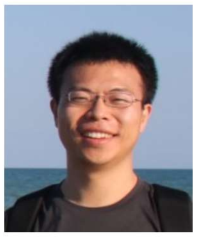
Guangquan Xu is a Ph.D. and full professor at the Tianjin Key Laboratory of Advanced Networking (TANK), College of Intelligence and Computing, Tianjin University, China. He received his Ph.D. degree from Tianjin University in March 2008. He is a IET Fellow, members of the CCF and IEEE. His research interests include cyber security and trust management. He is the director of Network Security Joint Lab and the Network Attack & Defense Joint Lab. He has published 100+ papers in reputable international journals and conferences, including IEEE IoT J, FGCS, IEEE access, PUC, JPDC, IEEE multimedia, and so on. He served as a TPC member for IEEE UIC 2018, SPNCE2019, IEEE UIC2015, IEEE ICECCS 2014 and reviewers for journals such as IEEE access, ACM TIST, JPDC, IEEE TITS, soft computing, FGCS, and Computational Intelligence, and so on.

Wenqing Lei is a master's student at the College of Intelligence and Computing, Tianjin University, China. He received his B.S. degree from the School of Mechenical Engineering, Tianjin University, China in 2016. His current research interests include blockchain, Internet security and cryptography.

Lixiao Gong is working toward the masterâĂŹs degree in the College of Intelligence and Computing, Tianjin University, China.She received the bachelor degree in computer and science from Institute of Technology of Tianjin University of Finance and Economics in 2021.Her research interests include IoT Security and Cryptographic Algorithms.

Jian Liu received the B.S. and Ph.D. degrees from the School of Mathematical Sciences at Nankai University, Tianjin, China, in 2009, and 2015, respectively. She was a visiting Ph.D. student at the Department of Mathematics, University of Paris VIII, Paris, France. She is currently an associate professor with the School of Cybersecurity, College of Intelligence and Computing, Tianjin University, Tianjin, China. Her research interests include cryptography and coding theory.

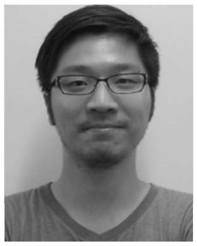
Hongpeng Bai is a Eng.D. student at the College of Intelligence and Computing, Tianjin University, China. He received his master degree from Changchun University of Science and Technology, China, in 2021. His current research interests include Android Malware detection and Zero Trust. Email: bai931214@tju.edu.cn



Kai Chen received his Ph.D. degree in the University of Chinese Academy of Science in 2010; then he joined the Chinese Academy of Science in January 2010. He became the Associate Professor in September 2012 and became the full Professor in October 2015. His research interests include software analysis and testing; smartphones and privacy.

Ran Wang received the Master degree from the School of Computer science and technology, Tianjin University, in 2019. He is currently with the security center, JD.com, China. His main research field is the network attack and defense.

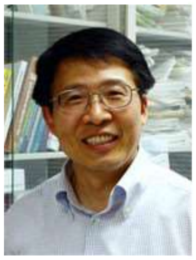
Wei Wang received the Ph.D. degree in control science and engineering from XiâĂŹan Jiao- tong University, in 2006. He was a Postdoc- toral Researcher with the University of Trento, Italy, from 2005 to 2006. He was a Postdoctoral Researcher with TELECOM Bretagne, and also with INRIA, France, from 2007 to 2008. He is currently a Full Professor with the School of Com- puter and Information Technology, Beijing Jiao- tong University, China. He is an Editorial Board member of Computers & Security and a Young AE of the Frontiers of Computer Science. He has authored or coauthored over 80 peer-reviewed papers in various journals and international conferences. His main research interests include mobile, computer, and network security. He was a European ERCIM Fellow of the Norwegian University of Science and Technology (NTNU), Norway, and in Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, from 2009 to 2011.



Kaitai Liang received the Ph.D. degree from the Department of Computer Science, City University of Hong Kong, in 2014. He is currently an Assistant Professor with Delft university of technology, Netherlands. His research interests are applied cryptography and information security in particular, encryption, blockchain, post-quantum crypto, privacy enhancing technology, and security in cloud computing.

Weizhe Wang received his bachelor degree from Changchun University of Science and Technology in 2018. He is currently pursuing his master degree at both College of Intelligence and Computing, Tianjin University and School of Information Science, Japan Advanced Institute of Science and Technology. His research interests include cyber security, privacy protection.

Weizhi Meng is currently an assistant professor in the Cyber Security Section, Department of Applied Mathematics and Computer Science, Technical University of Denmark (DTU), Denmark. He obtained his Ph.D. degree in Computer Science from the City University of Hong Kong (CityU), Hong Kong. Prior to joining DTU, he worked as a research scientist in Info-comm Security (ICS) Department, Institute for Infocomm Research, A*Star, Singapore, and as a senior research associate in CS Department, CityU. He won the Outstanding Academic Performance Award during his doctoral study, and is a recipi-ent of the Hong Kong Institution of Engineers (HKIE) Outstanding Paper Award for Young Engineers/Researchers in both 2014 and 2017. He is also a recipient of Best Paper Award from ISPEC 2018, and Best Student Paper Award from NSS 2016 and Inscrypt 2019. His primary research interests are cyber security and intelligent technology in security, including intrusion de-tection, smartphone security, biometric authentication, HCI security, trust computing, blockchain in security, and malware analysis. He served as program committee members for 50+ international conferences. He is a senior member of IEEE.



Shaoying Liu holds a B.Sc and a M.Sc degree in Computer Science from Xi'an Jiaotong University, China, and the Ph.D in Computer Science from the University of Manchester, U.K. He worked as Assistant Lecturer and then Lecturer at Xi'an Jiaotong University, Research Associate at the University of York, and Research Assistant in the Royal Holloway and Bedford New College at the University of London, respectively, in the period of 1982 - 1994. He joined the Department of Computer Science at Hiroshima City University as Associate Professor in April 1994, and the Department of Computer Science in the Faculty of Computer and Information Sciences at Hosei University in April 2000. In April 2001 he was promoted to a Professor. From 1st April 2020, he has been working at Hiroshima University as a Professor. He was invited as a Visiting Research Fellow to The Queen's University of Belfast from December 1994 to February 1995, a Visiting Professor to the Computing Laboratory at the University of Oxford from December 1998 to February 1999, and a Visiting Professor to the Department of Computer Science at the University of York from April 2005 to March 2006. From 2003 he is also invited as an Adjunct Professor to Shanghai Jiaotong University, Xi'an Jiaotong University, Xidian University, and a Visiting Professor to Shanghai University, Xi'an Polytechnic University, Bejing Jiaotong University, and Beijing University in China, respectively. He is IEEE Fellow, British Computer Society (BCS) Fellow, and member of Japan Society for Software Science and Technology.

Declaration of interests

☒ The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

☐ The authors declare the following financial interests/personal relationships which may be considered as potential competing interests:

CRediT Author Statement

Guangquan Xu: Supervision, Conceptualization, Funding acquisition **Wenqing Lei**: Methodology, Software, Writing- Review & Editing **Lixiao Gong**: Validation, Formal analysis. **Jian Liu**: Investigation **Hongpeng Bai**: Resources **Kai Chen**: Visualization **Ran Wang**: Writing – Original Draft **Wei Wang**: Funding acquisition **Kaitai Liang**: Project administration **Weizhe Wang**: Project administration **Weizhi Meng**: Writing – Review & Editing **Shaoying Liu**: Writing – Review&Editing