Effective Memory Diversification in Legacy Systems

Original Scientific Paper

Heesun Yun
Sungshin Women's University
Department of Convergence Security Engineering
02844, Seoul, South Korea
yunheesun718@gmail.com

Daehee Jang
Sungshin Women's University
Department of Convergence Security Engineering
02844, Seoul, South Korea
djang@sungshin.ac.kr

Abstract *– Memory corruption error is one of the critical security attack vectors against a wide range of software. Addressing this problem, modern compilers provide multiple features to fortify the software against such errors. However, applying compiler-based memory defense is problematic in legacy systems we often encounter in industry or military environments because source codes are unavailable. In this study, we propose memory diversification techniques tailored for legacy binaries to which we cannot apply state-of-the-art compiler-based solutions. The basic idea of our approach is to automatically patch the machine code instructions of each legacy system differently (e.g., a drone, or a vehicle firmware) without altering any semantic behavior of the software logic. As a result of our system, attackers must create a specific attack payload for each target by analyzing the particular firmware, thus significantly increasing exploit development time and cost. Our approach is evaluated by applying it to a stack and heap of multiple binaries, including PX4 drone firmware and other Linux utilities.*

Keywords*: Legacy System, Diversification, Memory Layout Randomization, UAV Firmware*

1. INTRODUCTION

Memory corruption vulnerabilities are caused by unexpected changes or reference of memory values, and there are various categories of them [1,2]. Generally, the existence and detailed structures of specific memory regions vary by application, but the stack and heap memory regions of most applications, including legacy binaries, are composed based on a general and common memory layout; thus, its exploitation steps are fairly similar from one to another [3-5].

For instance, memory corruption vulnerabilities in a stack such as buffer overflow vulnerability often target to corrupt a specific/common data so-called return address. Defending such memory corruption-based exploitation is also well established by security researchers: use stack canary and check if the canary value has changed [6]. However, such defense (stack canary) is applied by compilers; thus, it is difficult to apply such defense to legacy binary which cannot be re-compiled.

Various vulnerabilities are also likely to exist in the heap memory area as the heap is dynamically allocated at runtime and it is usually more complicatedly constructed than the stack memory buffers [7]. Similarly, to stack situations, multiple defense systems including address space layout randomization (ASLR) are proposed as a part of the compiler feature or heap-allocator's runtime verification logic [8].

All such state-of-the-art defenses are inapplicable to legacy binaries as they cannot be re-compiled [9]. However, we claim that applying the ASLR effects (even partially) to stack and heap memory regions would apply to some extent solely based on binary analysis and in-place instruction patching [10,11]. This is the part where our idea comes in.

To apply the memory diversification effect to legacy binaries, we use binary analysis techniques and in-place instruction patching to change the runtime memory layout of binary without re-compiling the source code.

The goal is to create different memory layouts for each physical device, without altering the original functionality. More specifically, we patch the binary/firmware to differentiate the size/distance of unused memory pad-

dings in stack and heap. As a result, when our diversification technology is applied to drones, even if the software system is prone to memory corruption, the attacker must adjust the attack payload differently each time when attacking an instance because diversification renders the memory layout of each physical instance differently. As a result, an attacker must create multiple attack payloads by analyzing the instance of each machine, which significantly increases attack time and cost.

Several related works also consider the lack-of-source-code situation. For example, [9] also fortifies the binary to prevent exploitation. The key idea of such work is to patch the ROP-gadget instruction instead of changing the memory layout as our system does. We have surveyed previous studies to this end and summarized more details in section 2.

In short, we proposed binary-patch-based memory layout randomization technique that does not require source code and is thus suitable for legacy systems. The main contributions of this paper are as follows.

1. Our system provides a way to change the memory layout without re-compiling the binary. This is useful to fortify outdated system binaries when their source code is intentionally or unintentionally unavailable.
2. Our system effectively raises the bar of applying memory corruption attacks against multiple physical devices using the same firmware. Because our approach diversifies the memory layout of each target, the attacker must develop different exploit payloads (e.g., ROP payload for buffer overflow vulnerability) for each target device.
3. We implemented and deployed our idea against commercial-off-the-shelf (COTS) software such as Intel PX4 aero drone firmware and popular Binutils binaries in the Linux system.
4. We provide academic analysis and discussion that our approach must consider data alignment issues and code/data interleaving problems in ARM-like CPU architectures. This is a thought-provoking discussion for additional future research to this end.

The rest of the paper is structured as follows. Section 2 covers various related works regarding our research direction. Section 3 proposes a system theory about the legacy system and memory safety. Next, we describe the design of our framework in section 4, and the implementation of algorithms in section 5. Afterward, we present the evaluation results in section 6. Finally, we summarize our paper and conclude in section 7.

2. RELATED WORK

2.1. BINARY PATCH

Because many commercial software applications are developed based on closed-source code, it is difficult to apply security patches to their vulnerable parts. Various researches are addressing this limitation. [12] conducted a study to prevent return-oriented programming (ROP) attacks based on the return instruction without based on debugging symbol information in a Linux 64-bit environment without source code. [13] conducted a patch for a buffer overflow vulnerability by automating the security patch for the Windows x86 binary without the use of source code, debugging information or human intervention. [14] considers a closed-source code patching environment in PowerPC-based binaries. [9] achieves security effectiveness by patching the binary to prevent code reuse attacks. [15] identified vulnerabilities in closed-source code software based on the bug signature. [16] performed a static binary patch to prevent vulnerabilities in ARM-based Internet of Things (IoT) device firmware.

Additionally, many studies aim to better identify the causes of security vulnerabilities in binary. [17], [18] enabled finding similar patches or vulnerabilities in different binaries by identifying code portions changed by a binary patch through basic block analysis. [19] detected software vulnerabilities in the binary codes of patched and unpatched programs using the patch diffing technology. Previous studies primarily aimed at preventing specific security vulnerabilities in an environment with binary-only approach without source code. To some extent, our study is similar to these studies in that patch is performed in a binary-only environment, but our framework is mainly focused on diversifying the memory layout and hindering memory corruption exploitation for physical instances (e.g., each drone machine).

2.2. MEMORY RANDOMIZATION

As attacks that exploit memory corruption vulnerabilities have been launched against several commercial software applications, the latest security technologies for memory protection, such as ASLR, have been implemented in most software applications. ASLR is a technology that can defend against attacks using a fixed address by randomly changing the address of the data area whenever a binary is executed. Thus, research to prevent external attacks by randomizing part of the memory is being actively conducted. [20] proposed a technology to randomize the stack layout based on the LLVM compiler to prevent memory corruption vulnerabilities. The position of each object was randomized using source code information so that attackers cannot predict the stack layout. [8] performed randomization for the entire heap memory area by proposing a random memory block allocation algorithm. It is similar to our study in that randomization is performed in the heap memory area, but the detailed process to obtain the randomization effect is different. [21] proposed a solution to prevent just-in-time (JIT) code reuse attacks using memory randomization technology that is subdivided for each process so that code sharing is not disturbed. [22] conducted a study on the kernel defense mechanism by performing device driver randomiza-

tion and proposed a solution to prevent ROP attacks on the Linux kernel by increasing the KASLR entropy. [11] proposed a method for randomizing the memory layout in user-end machines to prevent buffer overflow attacks. As shown in the above examples, memory randomization to reinforce security is being extensively researched. Although some studies do not use source code and symbols, many studies perform memory randomization using the source code information. Our study performed effective memory randomization only for binaries without source code information.

2.3. LEGACY SYSTEM

A legacy system is one that uses software applications from older versions or works in uncommon ways. Currently, the latest software updates and distributions are done quickly based on open sources, but many legacy systems are still being used in military or space exploration systems. Because these legacy systems are mostly based on closed-source code, it is difficult to apply security patches to them. In this situation, research into the reinforcement of the security of legacy systems is indispensable. [23] generates randomized instruction addresses whenever a legacy x86 binary is executed without source code or symbol information. [9] proposed a binary-based rewriting technology for patching legacy binary to prevent code reuse attacks. [24] proposed a solution for detecting control flow integrity (CFI) attacks, such as buffer overflow and ROP attacks. It detected CFI attacks with high accuracy by inserting performance counters and instrumentation hooks through binary editing in the ELF file, which is legacy software.

3. PROPOSED SYSTEM THEORY

3.1. LEGACY SYSTEM

A legacy computer system in this paper refers to a computer device with old software application/firmware that differs from the recent standard. Currently, the development and distribution of software are done quickly based on state-of-the-art compilers, yet many legacy systems are still used as old version binaries in various fields. Because most legacy systems are closed-sourced, and not properly updated, they can be a good target for attackers to abuse system vulnerabilities such as memory errors.

3.2. ASLR

ASLR is a memory layout diversification technology that changes the address of the data whenever a program is executed by randomly placing memory layouts such as stack, heap, and library codes in the unpredicted address space to make memory exploitation attacks difficult. As memory exploitation technologies such as return oriented programming (ROP) are becoming more popular, memory protection techniques such as ASLR are now widely used in most software systems.

3.3. BASIC BLOCK

A basic block in computing is a straight-line code sequence with no branches other than the entry and exit. Because of this characteristic, computer science researchers often analyze binaries based on basic blocks as a unit of algorithm testing/measure.

3.4. MEMORY ALIGNMENT

When data structures or classes are stored in memory in programming languages such as C++, paddings are sometimes inserted in between variables. These paddings are dummy values added by the compiler. The memory is aligned in 4-byte units to optimize the performance when the CPU accesses the memory. Because of the data bus structure between the CPU and memory, all variables that enter the memory must be placed in consideration of memory alignment to improve system performance [25]. In some cases, if data is accessed to an address without considering memory alignment, it can cause an alignment fault rather than causing a performance issue [26]. In our paper, we consider memory alignment problems while applying our patch for memory diversification.

4. DESIGN

4.1. OVERVIEW

The overall framework of this study is shown in Fig. 1. The firmware binary of the legacy system is ①disassembled using the Capstone [27] library and ②functions are extracted based on binary analysis. Afterward, ③ we analyze basic blocks in the function to run our code detection algorithm for proper diversification patch. For the next step, ④ our patch tool performs a diversification patch to the corresponding assembly instructions to adjust the memory layout. After our algorithm is applied, ⑤ all the basic blocks are reassembled into a diversified version of binary. We applied this approach to PX4 drone firmware to confirm its effectiveness and stability. However, in the case of Windows CE binaries, we could not apply the final re-assembly step because the underlying tool only considers Linux-based binary. In such a case, we applied in-place binary patching directly against machine codes with additional heuristics.

4.2. STACK DIVERSIFICATION

The overall methodology for stack memory diversification is to insert a random-sized dummy padding inside each stack frame. Because each binary might have unique compiler options, the details of the stack memory layout may differ from one instance to another. However, the main structure of the stack frame and its uses are mostly similar. To launch a successful attack via buffer overflow vulnerability in the stack; the return address, buffer position, layouts, and offsets among local variable data must be precisely calculated. A general

defense technology against such attacks is to place a stack canary between a local variable and the return address, then check whether the stack canary has been changed before exiting the function. If the canary value has been changed, it indicates that unintended data overwrites occurred during the execution of the function.

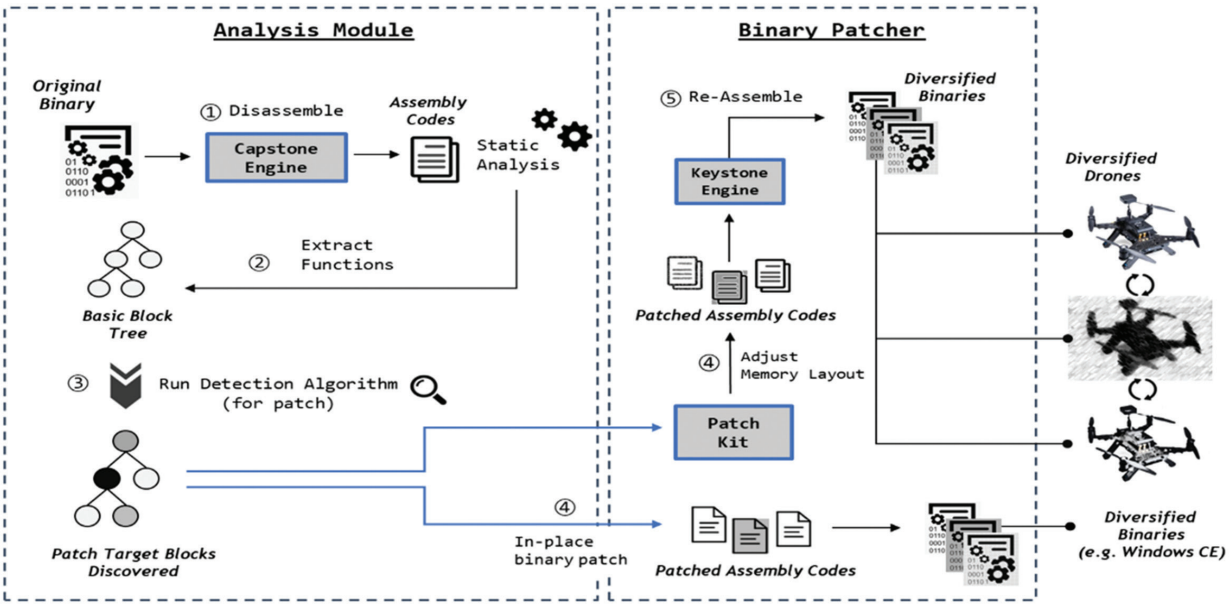

Fig. 1. System Overview. Analysis module spots patch targets based on various pattern detection algorithms. Binary patcher is responsible for modifying and re-assembling the target binary.

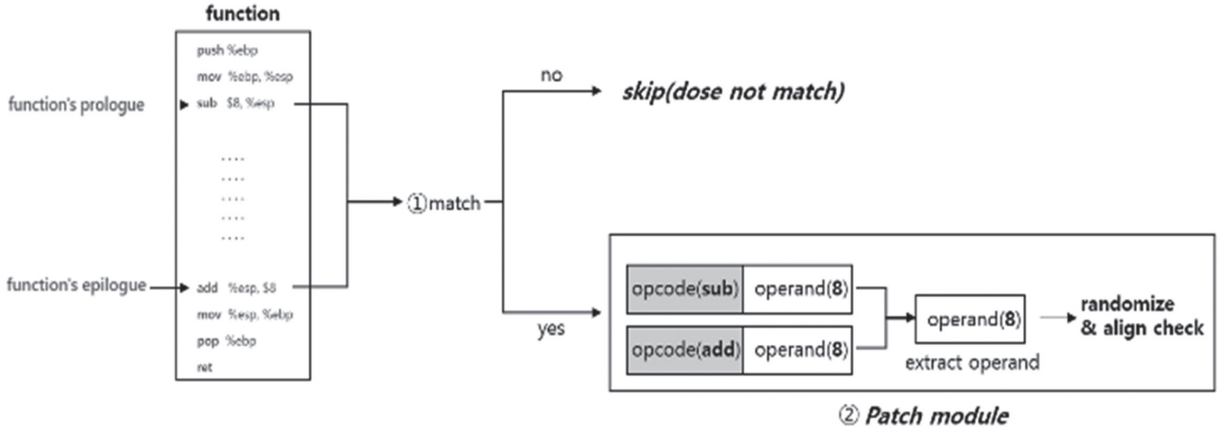

Fig. 2. demonstrates how our patch algorithm checks whether the function's prologue and epilogue match each other. If the condition is satisfied, the patch module performs randomization patch. The patch module extracts the operand from the assembly

However, without re-compiling, the source code, adding stack canary features to legacy systems is difficult. Fig. 3 and 4 show the codes before and after applying the stack canary. Lines 81 to 90 in Fig. 4 represent the stack canary part. The figure shows the same program, but with/without additional assembly codes as the stack canary defense feature. The problem with legacy systems is that it is difficult to *insert* additional assembly code without breaking the other codes. However, it is feasible to in-place *modify* the assembly codes without breaking other codes. Therefore, instead of using a stack canary; in this study, we diversify the distance between the local variable of the stack and the *return* address via inserting a random-sized padding by modifying the stack-offset related variables embedded in the assembly instruction. This can be done relatively simply based on a binary patch, considering how the compiler allocates the stack frame.

The compiler adds the stack allocation and deallocation codes as function prologue and epilogue. Our algorithm detects this code via binary analysis and randomly modifies the size of the stack frame dummy padding differently for each legacy instance. The point of our idea is that we can make this modification solely based on in-place binary patching. To build our system, our patch framework analyzes each binary architecture of the target binary, such as ARM and Intel, and applies

a tailored algorithm to each CPU architecture. For example, ARM uses fixed-length instruction encoding, while Intel uses variable-length instruction encoding.

Fig. 3. Typical disassembly result w/o stack canary

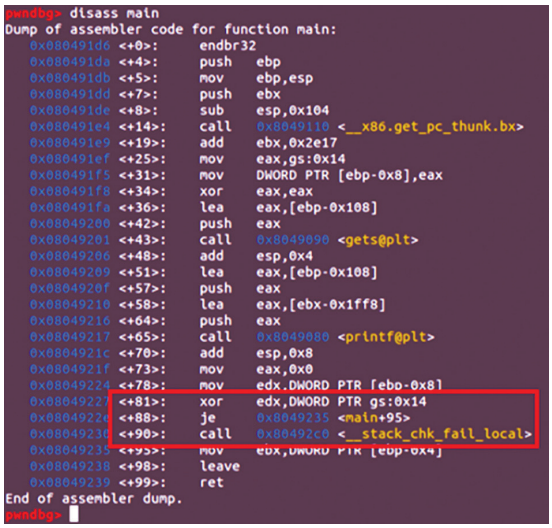

Fig. 4. Typical disassembly result w/o stack canary.

The compiler adds the stack allocation and deallocation codes as function prologue and epilogue. Our algorithm detects this code via binary analysis and randomly modifies the size of the stack frame dummy padding differently for each legacy instance. The point of our idea is that we can make this modification solely based on in-place binary patching. To build our system, our patch framework analyzes each binary architecture of the target binary, such as ARM and Intel, and applies a tailored algorithm to each CPU architecture. For example, ARM uses fixed-length instruction encoding, while Intel uses variable-length instruction encoding.

Fig. 5 shows the general function prologue and epilogue for stack frame allocation and de-allocation. In Fig. 5, the size of the local variable is 0x10. Because 0x10 is part of the 32-bit operand encoding (10 00 00 00) of the instruction, this instruction can be changed in-place without reducing or expanding the code. The essence of our idea is to randomly increase this size to make the distance between the stack buffer and the return address unpredictable. The unnecessarily increased buffer size can be considered as a dummy padding inside memory space which has no harm to program execution. However, if this value is decreased, the program may be damaged as the original buffer cannot hold the given data.

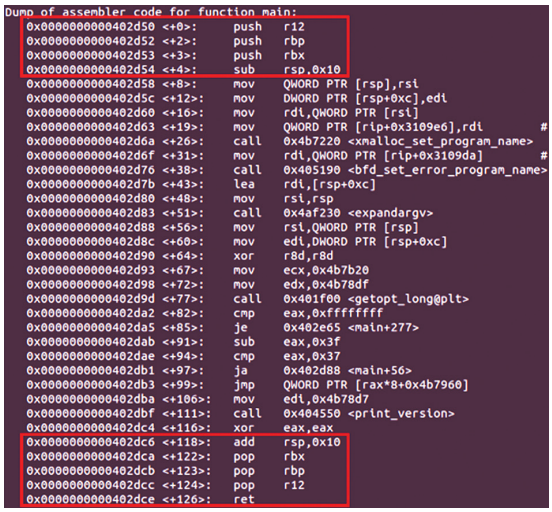

Fig. 5. Typical function's prologue and epilogue code (Binutils cxxfilt). Such code patterns can change depending on compiler options.

The patch appears simple if the stack frame allocation and de-allocation code are symmetric (e.g., add a value and subtract the same value later). However, there are cases where the stack allocation codes are more complicated. For example, when a stack frame of size 100 is allocated by subtracting 100 from to stack pointer and then de-allocated twice with the size of 50, it is difficult to determine which value is the stack frame size because the correspondence of allocation and de-allocation codes do not match based on the same operand size. Therefore, we use additional heuristics to filter out exceptional cases of stack frame allocation.

Furthermore, an additional check is performed to determine whether the changed size is appropriate for the diversification patch. The stack size must be increased with 4-byte granularity considering the CPU word alignment, which improves performance when the CPU accesses the memory and prevents alignment faults [25,28].

4.3. HEAP DIVERSIFICATION

The overall concept of applying diversification to heap memory is similar to the previous stack case (binary analysis and finding the appropriate patch call site), but the detailed method is quite different. Heap memory corruption vulnerabilities are generally related to use-after-free and type confusion. Use-after-free refers to a vulnerability that can occur when a dynamically allocated heap space is freed and reused, whereas type confusion refers to a vulnerability that occurs when the instance of an object confuses the type. The heap is a memory segment with complex data dependency

based on multiple pointer chains. The code pointer in the stack only needs to consider the return address, whereas, in the heap, it is relatively difficult to find a code pointer based on their memory location. Hence, even slight diversification of the heap memory layout can have a significant impact on exploit development.

Our binary analysis algorithm for heap memory diversification differs from the stack analysis. Generally, heap layout can be dynamically changed using standard methods such as IAT hooking in Windows and LD_PRELOAD in Linux. However, this approach is only possible for *dynamically linked binaries*. However most legacy binaries are based on static linking. This raises the question of how the heap allocation call site can be detected based on a generic algorithm. In this paper, we consider the binary has symbols to indicate the heap allocation function. Based on such a premise, our algorithm tracks down heap memory allocation sites in the legacy binary by parsing the symbol table. Then, the call site is backtracked to find the memory allocation size parameter (to modify it). However, a few problems occur while finding this size information.

The first is the data interleaving problem. In Fig. 7, the address 0x807f378 represents a heap allocation call site (malloc). To find the allocation size parameter, the instruction must be backtracked, but there are interleaved data inside the code (0x807f370). This data can be misinterpreted as instruction and hinder the analysis. To solve this problem, our algorithm additionally traces the boundary of each basic-blocks and runs backtrace only within the scope inside the block. However, even within the same basic block, there are exceptional cases.

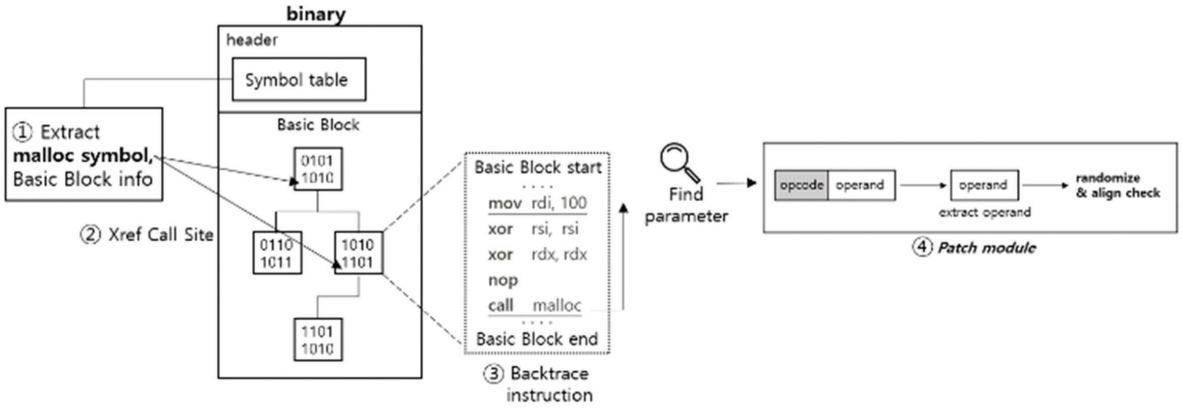

Fig.6. shows the malloc symbol and basic block information extracted from the symbol table in the header of the binary. After extracting the symbol information, the call site information is obtained from the symbol, and the parameter information is obt.

```
.text:0807F368                  BCC     loc_807F30A
.text:0807F36A                  B       loc_807F320
.text:0807F36A ; End of function _ram_flash_wait(sem_s *)
.text:0807F36A
.text:0807F36A ; ---------------------------------------------
.text:0807F36C off_807F36C      DCD _ZL18dm_operations_data
.text:0807F36C                                          ; DATA
.text:0807F36C                                          ; dm_op
.text:0807F370 dword_807F370    DCD 0xF4240             ; DATA
.text:0807F370                                          ; _ram_
.text:0807F374
.text:0807F374 ; =============== S U B R O U T I N E =========
.text:0807F374
.text:0807F374
.text:0807F374 ; _DWORD __fastcall _ram_initialize(unsigned ir
.text:0807F374 _ZL15_ram_initializej                    ; CODE
.text:0807F374                                          ; DATA
.text:0807F374                  PUSH    {R4-R6,LR}
.text:0807F376                  MOV     R5, R0
.text:0807F378                  BL      malloc
.text:0807F37C                  LDR     R6, =_ZL18dm_operations
.text:0807F37E                  MOV     R4, R0
.text:0807F380                  STR     R0, [R6]        ; dm_op
```

Fig. 7. Example case of data interleaving in ARM binary.

```
08019D46 83 68          LDR     R3, [R0,#8]
08019D48 1F 69          LDR     R7, [R3,#0x10]
08019D4A 38 46          MOV     R0, R7
08019D4C 01 F0 9C FA    BL      fat_semtake
08019D50 38 46          MOV     R0, R7
08019D52 01 F0 FB FA    BL      fat_checkmount
08019D56 06 46          MOV     R6, R0
08019D58 30 BB          CBNZ    R0, loc_8019DA8
```

Fig. 8. Exceptional case for tracking function parameter. In the example, it is ambiguous to determine which R0 is the actual parameter for the function fat_checkmount.

Fig. 8 shows two function parameters in the same basic block. Because function parameters are stored in a stack and restored immediately before they are used, it is difficult to determine which parameter is the actual size parameter based on static code analysis.

5. IMPLEMENTATION

To implement our algorithms, we use Patchkit [29], for ELF binary patch, and Capstone [27], Radare2 [30], and Keystone [31] tools for binary analysis and disassembly/reassembly. Overall, implementations are written in 979 lines of Python and 1,495 lines of additional Docker/management codes.

5.1. ARM STACK

Algorithm 1 – The ARM Stack pseudo code. Prefixes/Postfixes such as "BB" denotes the basic block of function, "dis" denotes the dissemble result of the Capstone library, "ins" denote each instruction, and "SP" denotes the stack pointer.

0: **function** PATCH(binary)

1: **for** func in binary_funcs() **do**

```
2:    startBB ← func_startBB_dis
3:    for ins in startBB do
4:        if ins_OpcodeName = "sub" then
5:            src, dst ← ins_operands
6:        else
7:            Skip
8:        end if
9:        if src = SP_reg & dst = stack_size then
10:           prologue ← ins_addr
11:       end if
12:   end for
13:   for ins in reversed(endBB) do
14:       if ins_OpcodeName = "add" then
15:           src, dst ← ins_operands
16:       else
17:           Skip
18:       end if
19:       if src = SP_reg & dst = stack_size then
20:           epilogue ← ins_addr
21:       end if
22:   end for
23:   if prologue = epilogue then
24:       function RANDOMIZE(stack_size)
25:           get (alignment info)
26:           ran_num ← random.randrange(0, 0x10)
27:           new_stack_size ← stack_size +ran_num
28:           Patch()
29:           end function
30:       end if
31: end for
32: end function
```

The overall flow of the stack randomization for ARM 32bit is shown in Algorithm 1. The disassembly result for the first basic block of each function is initially processed, and this result includes instructions such as push and *mov*. To find the address of the subtraction instruction of the stack (e.g., memory allocation), each instruction in the basic block is analyzed, and if the opcode is *sub*, the operand is remembered. If this value is constant (e.g., immediate type value), we conclude this is the stack frame size corresponds to the function prologue. Next, we scan each instruction to find the corresponding add instruction as the match for stack memory deallocation. Afterward, a new stack size is generated using the diversification algorithm, and the stack size is randomly increased respecting the CPU word alignment. Finally, the stack-frame size related instructions are patched in-place.

5.2. X86 STACK

The x86 stack randomization evaluation is performed using the Windows CE binary. Because the Windows CE binary cannot use the Patchkit library [29], our framework implemented a custom code detection for our algorithm. For patching, our tool traces instructions starting from the entry point and applies the same algorithm as ARM stack. In the Windows binary, it was more difficult to identify the function location, size, and internal structure compared to ARM. However, it was easy to apply the patch if the function frame used *leave* instruction for function epilogue as the instruction automatically calculates the required stack size for deallocation.

5.3. ARM HEAP

Algorithm 2 – The ARM Heap pseudo code. Prefixes/Postfixes such as "BB" denotes the basic block of function, "dis" denotes the dissemble result of the Capstone library, "ins" denote each instruction.

```
0: function PATCH(binary)
1: for target_addr in malloc_BB do
2:     bb ← malloc_BB[target_addr]
3:     startpos ← target_addr – len(bb) + 4
4:     for ins in dis(bb, startpos) do
5:     if ins_mnemonic is "movs" then
6:         if ins_operands is register then
7:             Reg ← ins_reg_name
8:         end if
9:         if Reg = "r0" then
10:            MRU ← (ins_addr, ins_str, bb[offset|opcode])
11:        end if
12:    end if
13:    end for
14:    if MRU ≠ None then
15:        get(alignment info)
16:        pad ← random.randrange(0xf0 - imm)
17:        Patch(MRU, pad)
18:    end if
19: end for
20: end function
```

The overall flow of the heap randomization for ARM 32bit is shown in Algorithm 2. Before performing randomization, the malloc position is first found by parsing the symbol table of the target binary. Then, all the basic blocks that contain the malloc address are loaded and saved. At this point, analysis is performed considering the ARM and THUMB modes.

The core of the algorithm is to find a pattern corresponding to the heap size allocation of malloc. When movs instruction is found, we check if the target register corresponds to *r0* because *r0* is the register used as the first function parameter. We determine that the value for *r0* is the malloc size based on heuristics and save all the size data. All values that contain *r0* inside the basic block are collected, and if there are multiple values, the closest value to the malloc call site is used as the malloc size parameter. The randomization patch is then performed. In the heap, memory alignment must follow 16-byte (128-bit) granularity because of the ARM NEON instructions.

5.4. X86 & X64 HEAP

To perform the heap randomization on x86 and x64, we use the malloc address from the symbol table and *basic block border* information of the entire functions based on binary analysis. We use basic block border information to apply our algorithm solely to a single block. Similarly to the ARM case, if the *mov* instruction operand is the immediate type and is closest to the call site, we consider the information as a size parameter. Finally, we also respect the memory alignment of 16-byte (128-bit) granularity considering the use of SIMD instructions using 128-bit MMX registers.

6. EVALUATION

In this study, experiments were performed by porting all related codes into a Docker container image for efficiency. The server environment for our evaluation is composed of 16GB RAM, and 1TB SSD, based on Ubuntu 16.04 64bit server. The Intel PX4 Autopilot Drone firmware (ARM 32-bit) was mainly used for stack diversification testing. The soundness of patched drone firmware was checked with basic system functionalities after the booting process.

6.1. STACK

1) Buffer overflow Toy example

We evaluated our system using a simple x86-based toy binary that has memory vulnerability. In the experiment, we used a general stack buffer overflow vulnerability exploitation as a test case.

When the example code is executed, buffer overflow is triggered, and a segmentation fault error occurs because the return address is broken. However, when a buffer overflow vulnerability was triggered for a binary with diversified stack sizes using our patch tool, the result of the attack was unpredictable.

```
void test(char *input, int len) {
    char buf[4];
    memcpy(buf, input, len);
}

int main(int argc, char* argv[]) {
    if(argc!=2){
    printf("usage: ./test [bof size]");
    return 0;
}
```

Fig. 9. A toy program for testing memory diversification against stack.

```
running buffer-overflow to unpatched binary...
./run.sh: line 15:    60 Segmentation fault

==========================================

now running buffer-overflow to patched varients
bof test
no bof
```

Fig. 10. Screen capture of running toy example case.

Fig. 11. Result after applying stack memory diversification against PX4-based drone. The screen on the left side indicates internal command/communication operates without problem after patching. Right side of the picture shows LED blinking of the UAV that indicates its normal operation.

We evaluated our system using PX4 Autopilot Drone firmware (ARM 32-bit) binary as a test case of ARM binary. When the stack memory layout in the PX4 firmware was randomized using our patch tool and then executed in the same manner as the firmware before the patch, all functions worked normally despite thousands of codes being modified for diversification. Fig. 11 shows *nsh* program, the shell interpreter of the drone. All the commands in *nsh* shell operated correctly before/after our patch.

3) Windows CE binary

We evaluated our system using a collection of 42 binaries, including *pmkdir*, *pmemmap*, and *pdebug*, which are tools that can be used in Windows CE provided by [32]. Wine [33] was used for compatibility to run Windows-exclusive programs on the Linux operating system. All functions worked normally when the diver-

sification-patched binaries were executed. Randomization patch was performed in several tens of places in the case of the Windows CE binary.

6.2. HEAP

1) Toy example

We use a simple toy example to evaluate our heap diversification patch. Fig. 13 shows the before/after result of the diversification patch to the example program. The randomization result showed that the heap allocation address was all changed, however all *printf()* functions in the example code were executed normally. The top portion of the allocation address was changed because of the ASLR effect (not because of our diversification), however, and the bottom offset portion (marked with a red box) was also changed by our patch tool (this part should not change in the normal case). We used gcc and clang with multiple optimization levels ranging from zero to three, and Fig. 13 is based on the optimization level zero.

```
void f1(){
    char* ptr = malloc(100);
    ptr[5] = 1;
    printf("heap buffer at %p\n", ptr);
}
void f2(){
    char* ptr = malloc(40);
    ptr[5] = 1;
    printf("heap buffer at %p\n", ptr);
}
void f3(){
    char* ptr = malloc(256);
    ptr[5] = 1;
    printf("heap buffer at %p\n", ptr);
}
void f4(){
    unsigned int size = getpid()*1000;
    size = size%1000;
    char* ptr = malloc(size);
    memset(ptr, 0xff, size);
    printf("heap buffer? at %p\n", ptr);
}

int main(){
    f1();
    f2();
    f2();
    f2();
    f3();
    f4();
    printf("end\n");
    return 0;
}
```

Fig. 12. A toy program for testing memory diversification against heap.

Fig. 13. Before/after applying heap randomization patch. The red box indicates the internal heap offset is changed across the patch (the upper part of the address is changed due to ASLR).

2) Binutils

We applied our system to the GNU Binutils binaries for performance evaluation. The Binutils version in our evaluation is based on 2.31.1 and the architecture is 64-bit Intel. After the randomization patch, all binaries are executed as originally intended. To measure the execution time delay due to the unnecessarily expanded memory layout, we repeatedly tested the execution time of various Binutils programs before/after we applied our system. The program execution time is measured based on a simple python script.

Table 1 summarizes the result (program execution time) of the comparison between binaries before and after patching. We noticed that there was tiny degradation in the execution time before and after the patch. However, the overall amount of performance degradation is negligible.

Table 1. This table is organized by comparing the binary execution time before and after randomizing to the Binutils binary using our tool. Table 1 is the result of 1000 runs each, and the following options were used to test.

Binary	Before	After	Option
addr2line	0.809s	0.732s	Check the file related to the execution file address and the line information
ar	0.877s	1.014s	Generate an archive
nm-new	0.916s	0.921s	Check the symbol information of the file
objcopy	0.751s	0.749s	Generate a new file after extracting the instructions and data of the file
objdump	1.113s	0.867s	Check all contents of the object file
ranlib	0.826s	0.825s	Register the library so that it can be used
readelf	3.516s	3.682s	Output the dependency of 'ld' including type
size	0.805s	0.848s	Output the file size
strings	1.874s	1.904s	Output the offset of the execution file
strip-new	0.879s	0.938s	Remove the symbol

3) PX4 drone

We used the PX4 Autopilot Drone firmware binary for heap diversification evaluation as well. Similarly to the stack experiment, all functions worked normally when the heap memory layout in the PX4 firmware was randomized with our patch tool and the binary was executed in the same manner as the firmware before the patch.

7. CONCLUSION

With the emergence of memory corruption vulnerabilities, defense technologies that apply diversification to software systems, such as stack canary and isolated heap, are being extensively researched. However, because the

source codes are unavailable, such defense technologies cannot be applied to very old legacy systems. To remedy such a problem, this study presents an alternative approach to apply memory layout adjustment solely based on binary analysis and patching. We mainly apply diversification against two memory components: stack and heap. As an experimental approach, we chose stack and heap as diversification targets because they are the most common memory section of any computer system. By applying our idea, it is expected that the attack time and cost will increase significantly as the attacker must adjust the attack payload differently each time when attacking an instance. We conducted experiments using real-world programs such as Intel PX4 Autopilot Drone firmware (ARM 32bit) binary, and Binutils in the Linux system. The results demonstrated our diversification did not break any original semantics of the program yet successfully changed the memory layout for the attacker.

8. ACKNOWLEDGMENT

This work was supported by the Sungshin Women's University Research Grant of 2022.

9. REFERENCES

[1] K. S. Lhee, S. J. Chapin, "Buffer overflow and format string overflow vulnerabilities", Software: practice and experience, Vol. 33, No. 5, 2003, pp. 423-460.

[2] J. C. Foster et al. "Buffer overflow attacks", Syngress, Rockland, CA, USA, 2005.

[3] S. Govindavajhala, A. W. Appel, "Using memory errors to attack a virtual machine," Proceedings of the Symposium on Security and Privacy, Berkeley, IL, USA, May 2003, pp. 154-165.

[4] A. Francillon, D. Perito, C. Castelluccia, "Defending embedded systems against control flow attacks", Proceedings of the 1st ACM workshop on Secure execution of untrusted code, Chicago, USA, November 2009, pp. 19-26.

[5] A. Gupta, S. Kerr, MS. Kirkpatrick, E. Bertino, "Marlin: A fine grained randomization approach to defend against ROP attacks", Proceedings of the International Conference on Network and System Security, 3-4 June 2013, pp. 293-306.

[6] B. A. Kuperman, C. E. Brodley, H. Ozdoganoglu, T. N. Vijaykumar, A. Jalote, "Detection and prevention of stack buffer overflow attacks", Communications of the ACM, Vol. 48, No. 11, 2005, pp. 50-56.

[7] P. O. Sullivan, "Preventing Buffer Overflows with Binary Rewriting", University of Maryland, Electrical Engineering, College Park, Md, PhD Thesis, 2010.

[8] Z. Jin, Y. Chen, T. Liu, K. Li, Z. Wang, J. Zheng, "A novel and fine-grained heap randomization allocation strategy for effectively alleviating heap buffer overflow Vulnerabilities", Proceedings of the 4th International Conference on Mathematics and Artificial Intelligence, Chegndu, China, April 2019, pp. 115-122.

[9] P. Wang, J. Zhang, S. Wang, D. Wu, "Quantitative Assessment on the Limitations of Code Randomization for Legacy Binaries", Proceedings of the IEEE European Symposium on Security and Privacy, Genoa, Italy, September 2020, pp. 1-16.

[10] M. Prasad, T. Chiueh, "A Binary Rewriting Defense Against Stack based Buffer Overflow Attacks", Proceedings of the USENIX Annual Technical Conference, San Antonio, TX, USA, 9-14 June 2003, pp. 211-224.

[11] V. Iyer, A. Kanitkar, P. Dasgupta, R. Srinivasan, "Preventing overflow attacks by memory randomization", Proceedings of the IEEE 21st International Symposium on Software Reliability Engineering, San Jose, CA, USA, November 2010, pp. 339-347.

[12] S. Xu, P. Xie, Y. Wang, "AT-ROP: Using static analysis and binary patch technology to defend against ROP attacks based on return instruction", Proceedings of the International Symposium on Theoretical Aspects of Software Engineering, Hangzhou, China, December 2020, pp. 209-216.

[13] K. Chen, Y. Lian, Y. Zhang, "Automatically generating patch in binary programs using attribute-based taint analysis", Proceedings of the International Conference on Information and Communications Security, December 2010, pp. 367-382.

[14] U. Müller, E. Hauck, T. Welz, J. Classen, M. Hollick, "Dinosaur Resurrection: PowerPC Binary Patching for Base Station Analysis", Proceedings of the Network and Distributed System Security Symposium 2021, February 2021, pp. 21.

[15] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, T. Holz, "Cross-architecture bug search in binary executables", Proceedings of the IEEE Symposium on Security and Privacy, May 2015, pp.709-724.

[16] M. Huang, C. Song, “ARMPatch: A Binary Patching Framework for ARM-based IoT Devices”, Journal of Web Engineering, Vol. 20, No. 6, 2021, pp. 1829-1852.

[17] P. Sun, Q. Yan, H. Zhou, J. Li, "Osprey: A fast and accurate patch presence test framework for binaries", Computer Communications, Vol. 173, No. 9, 2021, pp. 95-106.

[18] Z. Xu, B. Chen, M. Chandramohan, Y. Liu, F. Song, "Spain: security patch analysis for binaries towards understanding the pain and pills", Proceedings of the 39th International Conference on Software Engineering, Buenos Aires, Argentina, May 2017, pp. 462-472.

[19] L. Zhao, Y. Zhu, J. Ming, Y. Zhang, H. Zhang, H. Yin, "Patchscope: Memory object centric patch diffing", Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, USA, November 2020, pp. 149-165.

[20] S. Lee, H. Kang, J. Jang, BB. Kang, “Savior: Thwarting stack-based memory safety violations by randomizing stack layout”, IEEE Transactions on Dependable and Secure Computing, Vol. 19, No. 4, 2021, pp. 2259-2575.

[21] M. Backes, S. Nürnberger, "Oxymoron: Making {Fine-Grained} Memory Randomization Practical by Allowing Code Sharing", Proceedings of the 23rd USENIX security symposium, Sandiego, CA, USA, August 2014, pp. 433-447.

[22] R. Nikolaev, H. Nadeem, C. Stone, B. Ravindran, “Adelie: Continuous Address Space Layout Rerandomization for Linux Drivers“, Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating System, Lausanne, Switzerland, February 2022, pp. 483-498.

[23] R. Wartell, V. Mohan, K. W. Hamlen, Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code", Proceedings of the ACM conference on Computer and communications security, Raleigh, NC, USA, October 2012, pp. 157-168.

[24] A. Biswas, Z. Li, A. Tyagi, "Performance Counters and DWT Enabled Control Flow Integrity", SN Computer Science 3.1, Vol. 3, No. 48, 2022, pp. 1-19.

[25] M. Singh, “Data Structure Alignment: How data is arranged and accessed in Computer Memory?”, https://www.geeksforgeeks.org/data-structure-alignment-how-data-is-arranged-and-accessed-in-computer-memory (accessed: 2021)

[26] ARM Developer, “Cortex-R4 and Cortex-R4F Technical Reference Manual r1p3”, https://developer.arm.com/documentation/ddi0363/e/memory-protection-unit/mpu-faults/alignment-fault (accessed: 2021)

[27] Github, “Capstone Engine”, https://github.com/capstone-engine/capstone (accessed: 2021)

[28] Microsoft, “Align (C++)”, https://docs.microsoft.com/en-us/cpp/cpp/align-cpp?view=msvc-170 (accessed: 2021)

[29] Github, “patchkit”, https://github.com/lunix-bochs/patchkit (accessed: 2021)

[30] Github, Radare2: Libre Reversing Framework for Unix Geeks, https://github.com/radareorg/radare2 (accessed: 2021)

[31] Github, “Keystone Engine”, https://github.com/keystone-engine/keystone (accessed: 2021)

[32] W.J. Hengeveld, “Rapi tools”, https://itsme.home.xs4all.nl/projects/xda/tools.html (accessed: 2022)

[33] WINEHQ, “What is Wine?”, https://www.winehq.org/ (accessed: 2022)