Ph.D. DISSERTATION

Code Transformation Techniques to Enforce Security Policies for Memory Safety

메모리 보호를 위한 보안 정책을 시행하기 위한 코드 변환 기술

BY

JANGSEOP SHIN

FEBRUARY 2020

DEPARTMENT OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Ph.D. DISSERTATION

Code Transformation Techniques to Enforce Security Policies for Memory Safety

메모리 보호를 위한 보안 정책을 시행하기 위한 코드 변환 기술

BY

JANGSEOP SHIN

FEBRUARY 2020

DEPARTMENT OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Code Transformation Techniques to Enforce Security Policies for Memory Safety

메모리 보호를 위한 보안 정책을 시행하기 위한 코드 변환 기술

지도교수 백 윤 흥

이 논문을 공학박사 학위논문으로 제출함

2020년 2월

서울대학교 대학원

전기 컴퓨터 공학부

신 장 섭

신장섭의 공학박사 학위 논문을 인준함

2020년 2월

위 원 장:	문수묵	(인)
부위원장:	백윤흥	(인)
위 원:	이재욱	(인)
위 원:	이병영	(인)
위 원:	조영필	(인)

Abstract

Computer memory is a critical component in computer systems that needs to be protected to ensure the security of computer systems. It contains security sensitive data that should not be disclosed to adversaries. Also, it contains the important data for operating the system that should not be manipulated by the attackers. Thus, many security solutions focus on protecting memory so that sensitive data cannot be leaked out of the computer system or on preventing illegal access to computer data. In this thesis, I will present various code transformation techniques for enforcing security policies for memory protection. First, I will present a code transformation technique to track implicit data flows so that security sensitive data cannot leak through implicit data flow channels (i.e., conditional branches). Then I will present a compiler technique to instrument C/C++ program to mitigate use-after-free errors, which is a type of vulnerability that allow illegal access to stale memory location. Finally, I will present a code transformation technique for low-end embedded devices to enable execute-only memory, which is a strong security policy to protect secrets and harden the computing device against code reuse attacks.

주요어: Computer Security, Memory protection, Code transformation

학 번: 2013-20813

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Computing devices are ubiquitous in modern human life. We rely on computer systems to handle various jobs for us. As the computer systems become more involved in our everyday life and business, more security sensitive data are processed by the computer systems, which makes it important that these data are protected from unauthorized entity. Also, protecting the integrity of the computer system is important, since attackers can manipulate the operation of computer systems to do harm to us.

In the field of computer security, one of the most critical component is the computer memory. It is the very component that contains security sensitive data. It also contains data for the control of the system—Attackers can control the operation of the system by corrupting computer memory. Therefore, many security solutions focus on the protection of memory. The security solutions differ depending on the given security goal. The security goal can be just to protect secret data from unauthorized users (i.e., to enforce confidentiality), or to prevent data corruption by unauthorized users (i.e., to enforce integrity of the data). There can be different approaches to achieve these goals. Since memory corruption is usually the result of software vulnerabilities, some solutions seek to detect or mitigate the errors resulting from the vulnerabilities. Some solutions manage and enforce access permission for different memory regions so that

there can be some restriction in accessing the memory even if the vulnerabilities are exploited.

In this thesis, I present various code transformation techniques to enforce security policies for memory protection. The advantage of code transformation techniques is that it can be deployed very easily, since it does not require silicon changes and it can be directly applied on existing systems. The code transformation is done by the compiler automatically, so manual intervention is not required. Thus, it is less error-prone and can be efficiently applied.

In Chapter 3, I will present a code transformation technique for efficient implicit information flow tracking. Dynamic information flow tracking (DIFT) is a promising technique for tracking the data inside a computer system. If we mark a security sensitive data, DIFT system will track propagation of the data through registers and memory during program runtime. However, the value of the secret can leak through implicit flows such as conditional branches. If the secret value is used as a condition for branch statements, the attacker can infer the value via the control flow that is taken as the result of the conditional statement. I will show how the proposed technique can help track these kind of implicit flow efficiently.

In Chapter 4, I will present a code transformation technique to efficiently mitigate use-after-free errors. Use-after-free errors are one of the common software vulnerabilities that allow illegal access to already deallocated heap memory. There have been many researches that tackle this problem but all of them suffer from high performance or memory overhead. In this chapter, I present how our technique can mitigate use-after-free error efficiently by automatically tracking reference counts for the heap objects in legacy C/C++ programs.

In Chapter 5, I will present a technique to enforce execute-only memory (XOM) for ARM Cortex-M processors, which is a popular processor for low-end embedded devices. XOM is a promising technique to hide sensitive data or software intellectual

properties. It can also effectively hide randomized code layout from attackers, thus providing protection against code-reuse attacks. Current high-end processors from both Intel and ARM support XOM feature in their CPUs. However, low-end embedded processors currently do not support XOM feature natively. In this chapter, I will show how we transform the code in ARM Cortex-M processor using special instructions to effectively enable XOM without causing too much performance/memory overhead.

Chapter 2

Background

In this chapter, I will introduce typical attack process that starts from memory corruption vulnerabilities. Then I will explain how researches defend against attacks by blocking the certain point of attack process. I will also explain the importance of the role of compiler in defending against these type of attacks.

Figure 2.1 shows how the attacker typically compromise system security starting from the memory corruption vulnerability. The attacker can modify code pointer such as return address or function pointer to hijack control flow of the program. The attacker can alternatively modify data pointer or data variable to manipulate data flow of a program. For example, there can be a variable denoting the authentication status of a user. Attacker can modify the variable by exploiting a memory corruption vulnerability thereby bypassing proper authentication process. Data flow is manipulated in this case, since originally the authentication variable should be toggled inside the authentication function, but the attacker modifies the authentication variable in the abnormal position of the program where the memory corruption occurs. Sometimes the purpose of the attacker can just be using the memory corruption to read illegitimate memory address to leak sensitive data as in the case of Heartbleed vulnerability [1].

Many researches focus on breaking particular point of this chain of attack process.

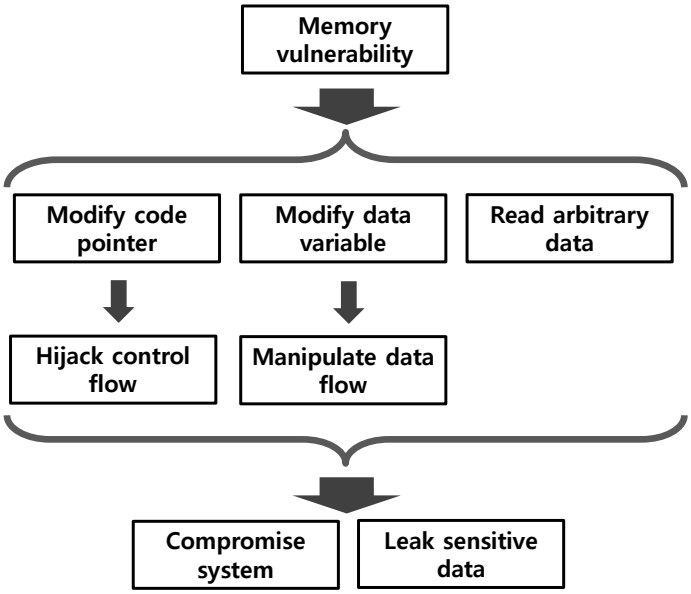


Figure 2.1: Typical attack process starting from a memory vulnerability.

Numerous work have been done to eliminate the source of the memory corruption problem by finding vulnerabilities in the program or developing ways to detect/prevent memory error once triggered by the attacker at runtime. The memory error can be roughly divided as spatial error such as buffer overflow and temporal errors such as use-after-free vulnerabilities. Some work statically analyze the program [39, 112] or dynamically feeding random inputs into the program and trigger crashes [22, 84, 96] to find new vulnerabilities. However, these approach cannot find all the vulnerabilities in the program. Therefore, other researches focus on detecting/preventing errors at runtime. These works typically instrument the program to keep track of data structures and insert checks or invalidate pointers so that illegitimate memory access can be captured [31, 57, 103].

Next line of work focus on isolating the security sensitive data or controlling the access of the attackers to illegitimate memory address so that the attackers cannot modify code pointers or data variables illegally or read memory region which is not

allowed. Code pointer integrity [54] isolates code pointers in a safe region so that attackers cannot temper with code pointers using memory vulnerabilities. WIT [3] assigns a color to the object (by coloring the shadow memory area that corresponds to the object) and assigns the same color to the memory access instructions that can legitimately access the memory object. Before any memory access, the colors are compared to detect possible attack attempt that exploits memory vulnerabilities to access arbitrary data.

Another way to break the attack chain is to tackle the next step in the attack process. When the attacker uses the memory vulnerability to modify illegal data value, she needs to know to which value she wants to convert the data in order to complete her attack. For example, to launch successful code reuse attack, she needs to know the address of the code gadgets that she wants to use for her attack sequence. Various code layout randomization defenses have been developed to prevent the attackers from learning the address of the code gadgets [12, 29, 33]. Data randomization techniques [13, 15, 21] have been proposed to randomize data representation in memory so that attacker cannot know how to change the data value in memory.

Finally, some techniques try to defend against the final goal of the attacker. To compromise a computing system from user space applications, attackers usually have to utilize system calls to do anything meaningful such as launching a shell, reading a file, etc. Therefore, a bunch of work focus on monitoring system calls to detect signs of attack [35, 105]. For another line of work, the sole purpose of the defense is to prevent leakage of sensitive data through the output channels such as standard output and public network/files. Dynamic information flow tracking (DIFT) [97] is a popular approach for solving this problem. It marks the sensitive data and tracks data flow through the program.

Implementation wise, most of the above mentioned solutions require some form of program transformation. Binary level solution is possible and there are plenty of

researches that work on binaries [104, 107, 116]. Binary level solutions have an advantage that the program source code is not required so it can be readily applied to a given binary. However, they are often limited in implementing more advanced security mechanism without involving high performance overhead and they often require extra information such as symbols and relocation [107]. On the other hand, compilers can perform more precise analysis using the given source code, enabling sophisticated security solutions with much lower overhead.

This thesis introduces my work on compiler based security solutions for preventing attacks on computer memory. Chapter 3 introduces a hardware and compiler technique to prevent leakage of sensitive data through implicit information flows. The compiler mainly analyzes program control flow graph and instruments special hardware instructions at the right place for guiding hardware to maintain the program counter tag for tracking data propagation through implicit control flows. Chapter 4 presents a compiler based technique to prevent attacks utilizing use-after-free vulnerabilities. The compiler mainly analyzes the type information in the program intermediate representation and inserts runtime library calls into necessary places to maintain reference counts for controlling memory free operation. Chapter 5 presents a compiler technique to enable execute-only memory on ARM Cortex-M based microprocessors. In this work, the main job of the compiler is to transform certain instructions into special hardware instructions. It also generates verification code and analyzes and inserts check code to sandbox the stack pointer.

Chapter 3

A Hardware-based Technique for Efficient Implicit Information Flow Tracking

3.1 Introduction

In recent years, computer security has been severely threatened by various malicious attacks that intend to leak sensitive information [72]. The general goal of these attacks is to transfer the critical data from sensitive sources (e.g., SIM card, password list) to output channels like network connections so that the attackers can acquire the sensitive information in the system. To achieve the goal, the malicious program or the victim program being exploited by attackers first accesses the critical data and then copies them from the source to the destination at each instruction execution. When they are finally delivered to the output channel, the attacker can leak the sensitive information out of the system.

One of the most widely used solutions against this type of attacks is *Dynamic information flow tracking* (DIFT) [72]. Generally, DIFT sets up rules to taint internal data of interest and keeps track of their taintness throughout the system. At runtime, whenever an instruction is executed, the taintness of sources is propagated to the destinations, to track the information flow associated with the data transfers (data copying

and transformations). An alarm will be triggered as soon as any of the tainted data is involved in potentially illegal activities, such as being included in a data stream on the output channels. In several previous studies [26, 30, 72, 81, 117], it was demonstrated that DIFT is an effective way to detect the attacks which attempt to leak the sensitive information with explicit data transfers.

However, there have been some advanced attacks that can bypass the explicit DIFT approaches by acquiring certain sensitive information only through the execution control flow analysis of a victim program without data transfers. In practice, when a data value affects a conditional branch result, execution flow is altered and it affects other data. Then the affecting value can often be inferred merely by examining the values of the affected ones. In this case, we can see that although there is no *explicit* data copy or transfer, the affecting data value is in effect transferred to other data values via the *implicit flow* along the execution control path. In the previous studies [50, 56], they presented empirical evidence that the attackers can leak the sensitive information by exploiting the implicit flow. Thus, in order to deal with such advanced attacks, a DIFT solution should track the taintness of the sensitive data tags not only through the explicit information flow associated with data copy and transfer operations, but also through the implicit flow associated with conditional branch operations.

For the tracking of implicit flow, several software solutions have been proposed [50, 114]. In these works, they analyze the program code and find the control flow that might be related to the implicit information flow at runtime. Then, they augment the original application with the additional code to keep track of the implicit flow as well as the explicit flow. In spite of their effectiveness, the main drawback of these solutions is that they incur too much runtime overhead, since it takes up to 20 instructions to emulate a single tag propagation operation per instruction.

To reduce the performance overhead for implicit flow tracking, RIFLE [101] resorts to a hardware technique. Although they have shown an impressive improvement

on the overall DIFT computation, their experiment also reveals that they still suffer from the non-negligible performance overhead for implicit flow tracking. This is primarily because their hardware has been designed originally for the information tracking with explicit data transfers. Therefore, to utilize their hardware for implicit flow tracking, they had to convert the implicit flow problem to the equivalent explicit one. For this reason, they instrumented their binary code to transform all implicit information flow operations across conditional branches into explicit data copy operations. According to their experiments, the performance degrades by a factor of two in the worst case, mainly due to the instrumented instructions.

Motivated by previous work, we have developed a dedicated hardware unit to efficiently tackle the implicit flow tracking problem. In this paper, we introduce our hardware engine for implicit flow tracking, called the *implicit flow tracking unit* (IFTU), and the implicit flow tracking scheme designed to work on IFTU. We have built IFTU as an external hardware module attached to the host processor via the system interconnect. To evaluate its effectiveness, we have implemented our solution on an FPGA board. In our experiments, we show that our proposed approach with IFTU successfully tracks the implicit information flow on the system with negligible performance overhead, while the additional logic required for the implicit flow tracking is also small.

3.2 Related Work

There has been much prior work that focuses on explicit information flow tracking [26, 30, 58, 72, 81, 97, 117]. Software approaches in [72, 81] suggest the use of a binary instrumentation technique, which mainly inserts additional instructions to the target code to keep track of the tainted data at runtime. During the program execution, the taintness of data is propagated according to the data dependency, and any misuse of data (e.g. information leak) is detected by their proposed solutions. Other works in-

troduced in [26, 30, 97] suggest the use of specialized hardware logic for DIFT mainly to reduce the performance overhead caused by the DIFT computation. In [26, 97], for instance, they augmented the host processor internals directly, including register files and caches. In [30, 58], they proposed a decoupled DIFT hardware that can be attached to the outside the host. These previous approaches, implemented either in software or hardware, show their effectiveness in the tracking of explicit flows. However, a critical limitation they have is that they do not consider the implicit flows, which can result in the *under-tainting problem* where the values that should be tainted are not tainted [50].

To resolve the under-tainting problem, several software solutions for implicit flow tracking have been proposed. In [36], the authors use dynamic analysis to keep track of the flow of sensitive information processed by the web browser application. To handle the implicit flows, their *taint engine* examines all conditional branch instructions that are encountered during execution. If such an instruction has at least one tainted operand, the taint engine identifies all instructions whose execution is conditionally dependent on the direction of the branch and then it taints the results of those instructions. In [50], another software solution, called DTA++, is proposed. To achieve efficiency in the tracking, instead of examining all conditional branches, DTA++ focuses only on the implicit flows within certain code patterns (i.e., the information-preserving transformations), based on the observation that under-tainting usually occurs at just a few locations. With the proposed tracking strategy, DTA++ can achieve effectiveness and efficiency in implicit flow tracking. Nevertheless, the main drawback of these software approaches is that they still suffer from performance degradation mainly due to the additional code instrumented with the binary translation. For example, although DTA++ only applies the tracking technique to certain cases, the performance overhead is around 1.5X even with the parallel execution of the binary translation.

For this reason, several hardware techniques [100, 101] have been proposed to enhance the tracking performance. RIFLE [101] is a hybrid approach that uses compiler-

assisted binary rewriting to change the program to turn implicit information flows due to condition flags into explicit tag assignments. However, as discussed, the main problem of RIFLE is that it relies on the hardware architecture designed for the explicit flow tracking and thus requires code transformation to convert implicit flows to explicit ones. Since too many additional instructions are added to the original program binary to utilize the hardware, the efficiency is severely degraded and the performance overhead reaches up to 1.5X in the worst case (when the data cache of the system is duplicated to store the tags). On the other hand, in our approach, we propose a hardware engine specialized for implicit flow tracking and thus can overcome the limitation of RIFLE. GLIFT [100] and Leases [99] are interesting hardware solutions that track information flow at the gate level to build a system with strong noninterference properties which can be used to eliminate all forms of information leak, including those from timing and storage channels. While this is a potentially promising approach, all the hardware has to be re-designed from the gates up, requiring unproven new hardware design methodologies and tools. On the other hand, our IFTU can be connected to the commodity processor with an external interface, not requiring the redesign of the off-the-shelf processor architecture, since it is designed as an external module.

3.3 Our Approach for Implicit Flow Tracking

We now discuss our approach for efficient implicit flow tracking, inspired by [25] and improved. After briefly explaining the tracking scheme implemented in our work, we will describe our code analysis and transformation technique whose purpose is to enable the scheme to work correctly in real programs.

3.3.1 Implicit Flow Tracking Scheme with Program Counter Tag

The code snippet in Figure 3.1 shows a simple example of implicit information flow in a program. In this example, the value of `x` can be changed to either `0` or `1` according

to the branch result that is affected by the signedness of variable `s`. Clearly, there is a flow of information between the two variables since the value of `s` affects the value of `x`; however, it is not the result of direct data transfers, but rather the result of the branch outcome affected by setting the condition flag through the comparison.

```
x := 2
if s <= 0 then x := 0 else x := 1
```

Figure 3.1: An example code with implicit flow

To handle these implicit flows correctly, language-based static techniques [85] use a tracking scheme that introduces the program counter tag (denoted as t_{PC}), which indicates whether the control flow path is affected by tainted data or not. In this scheme, for every conditional branch, the taintness of data that is used for the condition checking is propagated to t_{PC}. Then, for the instructions after the branch, the value of t_{PC} is propagated to the tags of their destinations to indicate that the values are affected by the branch result. Now, assume that the example described in Figure 3.1 is tracked with this scheme. In this example, the variable `s` is used for the condition checking of the branch. Thus, if the variable `s` contains the sensitive information and its tag is tainted, t_{PC} is also set to `1`, to indicate that the branch result is affected by the sensitive information. Then, by propagating the value of t_{PC} to the tag for variable `x` when it is set, the implicit flow along the branch can be tracked.

In our approach, to handle the implicit flow as well as the explicit ones, we combine the tracking scheme introduced above with the conventional DIFT technique that tracks the data flow [72]. To denote tagging, every location for storing data such as registers and memory is augmented with a tag bit. Then, the tags are propagated during the program execution, based on a set of tag propagation rules that are specified for each basic operation type such as arithmetic, logical, or conditional branch.

Figure 3.2 shows an example code at the assembly level and the associated tag propagation operations. Basically, the tag propagation rules applied in our approach are based on the data dependency, as in the previous works [72, 81]. For example,

when the `ldr` instruction at line 1 is executed, the tags of sources (`%i0` register and the memory location pointed by the register value) are propagated to the tag for register `%g2`. In addition to the basic rules, we add new rules to track the implicit flow along the control path. In principle, a conditional branch has its condition code, such as `equal`, `not equal` or `less than`. When the branch is executed, the processor checks the condition code register (CCR) which generally consists of several condition bits (e.g., N,Z,V,C in SPARC machines), and determines the control path based on the value of CCR. That is, the result of a conditional branch is affected by the value of CCR. In practice, CCR is configured by an arithmetic instruction like `sub`, or a specialized comparison instruction like `cmp`, as in the code at line 3. For this reason, in our solution, when CCR is set by these instructions, the tags of their sources are propagated to t_{CCR}, which is the tag for CCR (see the right column of line 3). Then, when a conditional branch is executed later, the value of t_{CCR} is propagated to t_{PC} (see line 4). (If an unconditional branch is executed, such tag propagation is not performed since the branch is not affected by CCR (see line 6).) Thus, t_{PC} can indicate whether the control path is affected by tainted data or not. Since the value of t_{PC} is propagated to the destination tags (marked in boldface at the right column) at each ordinary instruction execution, we can track the implicit flow along the control dependency.

	Original Code	Tag Propagation
1	ldr [%i0], %g2	tag[%g2] = tag[%i0] or tag[mem[%i0]] **or t_{pc}**
2	sub %g2, %g3, %g1	tag[%g1] = tag[%g2] or tag[%g3] **or t_{pc}**
3	cmp %g1, #0	tag[%ccr] = tag[%g1] **or t_{pc}**
4	be L1 // **branch equal**	tag[%pc] = **t_{ccr} or t_{pc}**
5	mov #2, %g2	tag[%g2] = **t_{pc}**
6	b L2 // **unconditional branch**	none
7	L1: mov '1', %g2	tag[%g2] = **t_{pc}**
8	L2: add %g5, %g2, %g3	tag[%g3] = tag[%g5] or tag[%g2] **or t_{pc}**

Figure 3.2: Example of tag propagation rules

In spite of its effectiveness, there is a challenge in correctly tracking implicit flow

with the propagation rules introduced above. In principle, the taintness of t_{PC} set for a conditional branch should be propagated only to the instructions whose execution is conditionally dependent on the result of the branch according to the definition of implicit flow. Otherwise, the taintness of t_{PC} would be propagated to the tag for the data that is not affected by the branch or the tag that should be tainted would not be tainted. Thus, it is necessary to analyze the code in order to determine the exact scope of every conditional branch, which is a set of instructions that are affected by the branch result. In section 3.3.2, we will discuss a code analysis technique to identify the scopes of conditional branches and the management scheme for correctly clearing t_{PC} based on the analysis.

```
x := 2
if s <= 0 then x := 0
```

Figure 3.3: An example code with implicit flow through the untaken path

Also, from the tag propagation rules in Figure 3.2, once t_{PC} is tainted, all the instructions executed after that will be affected, because their execution is decided by a tagged condition. However, information flow can also exist between the condition of a branch and the instructions that are not executed. For example, in Figure 3.3, if the condition for the `if` statement is true,then x will be tainted according to the propagation rules. However, if the condition is false, x will not be tainted even though the value of x can leak the information about the branch condition. This example shows that only propagating tags according to the executed instructions is not enough, and there is the necessity for tag compensation of the untaken path. In section 3.3.3, we will describe the tag compensation scheme.

3.3.2 t_{PC} Management Technique

In principle, the result of a conditional branch determines the control path which in turn determines the instructions executed by the processor. For example, in the control

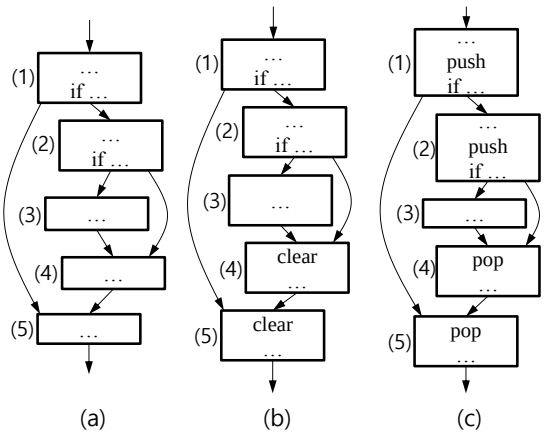


Figure 3.4: t_{PC} setting and clearing example

flow graph (CFG) shown in Figure 3.4(a), the execution of block (2) is determined depending on the result of the conditional branch in block (1). However, at a certain point in a program, the control path is no longer affected by the conditional branch. In general, the influence of a conditional branch ends at the *immediate post-dominator* of the branch. In our example, block (5) is the immediate post-dominator of block (1) because all paths from block (1) to the exit must pass block (5). Thus, the value of t_{PC} set at a conditional branch should be cleared upon the entrance of the immediate post-dominator (5) of the branch in (1) (Figure 3.4(b)).

However, this scheme does not work if the code has multiply-nested branches. For instance, we have a doubly-nested branch in block (2). According to the above scheme, t_{PC} set in block (1) would be cleared in block (4) although it should have remained set until block (5). In order to remedy this, a new t_{PC} *stack* is introduced that is used to save and restore the value of t_{PC} at each nested branch level. Basically, we save the current t_{PC} value by pushing it onto the stack just before a conditional branch, and later when we need to clear t_{PC}, we simply overwrite the current value in t_{PC} with the value popped from the stack (Figure 3.4(c)).

In Algorithm 1, we illustrate our algorithm that finds and marks the places in the

Algorithm 1: Algorithm for inserting push/pop and compensation code

Input : Control flow graph of a function
Output: Control flow graph with push/pop operations and compensation tag set operations inserted for correct implicit flow tracking

```
1  foreach loop l do
2      Insert push at the end of the preheader of l;
3      Insert pop at the start of the (common) immediate post-dominator of
         exiting block(s) of l;
4  end
5  foreach conditional branch block t do
6      Find t's immediate post-dominator block p;
7      if t is inside a loop and p is outside then continue;
8      if t is inside a loop and p does not dominate blocks with loop back edge
         then continue;
9      Insert push before conditional branch in t;
10     Insert pop at the beginning of p;
11     R = set of basic blocks that is reachable from t before reaching p;
12     foreach block b in R ∪ {p} do
13         foreach predecessor block pred of b do
14             if pred is not in R ∪ {t, p} then
15                 Insert push between pred and b;
16             end
17         end
18     end
19     foreach live-in register r of p do
20         D = set of basic blocks in R that defines r;
21         R_D = set of basic blocks in R that are reachable from basic blocks in
             D before reaching p ∪ {p};
22         G = set of basic blocks in R that are guaranteed to pass at least one
             basic block in D before reaching p;
23         foreach edge e connecting a block in R - R_D - G with a block in R_D -
             G do
24             Insert tag compensation for r;
25         end
26     end
27 end
28
```

code where such push/pop operations should be performed. Basically, as explained above, a push operation is inserted before a conditional branch and a pop operation is inserted at the immediate post-dominator of a conditional branch (see lines 9-10). However, real application codes have some exceptional cases that need a more complex algorithm such as ours.

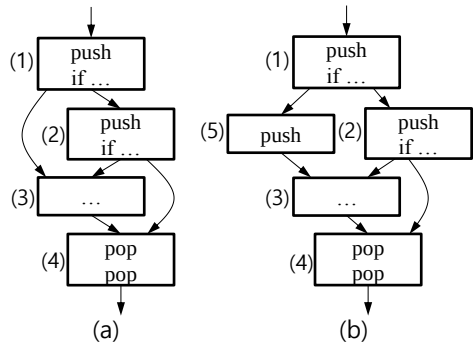


Figure 3.5: Solving push/pop imbalance

In Figure 3.5(a), we can see one of these exceptional cases that should be handled in our algorithm. In the figure, push/pop operations are marked according to the naive algorithm. Now suppose that the control flow takes the path (1)-(3)-(4) at runtime. If so, a push operation will be performed at block (1) and two pop operations will be processed at block (4). This obviously causes an error in the stack management because more entries are popped from the stack than are pushed onto the stack. In general, this problem arises when there is a path that reaches a pop operation inserted for some other conditional branch without passing that conditional branch.

To avoid this problem, for a conditional branch block *t* and its immediate post-dominator *p*, we first define a set *R* consisting of the basic blocks that are reachable from *t* before reaching *p*. Then, among the paths that reach *p*, let *P* be a path (p_1)-...-(p_{r-1})-(p_r)-...-(p) that does not pass *t*, where p_r is the first basic block in the path that is in R. Then, we insert a new basic block that contains a dummy push operation, between p_{r-1} and p_r. (For the example in Figure 3.5(a), such a new block is inserted between blocks (1) and (3) as shown in Figure 3.5(b).) In this way, we can make sure

that the number of push and pop operations are equal along any path. This process corresponds to lines 12-18 of Algorithm 1.

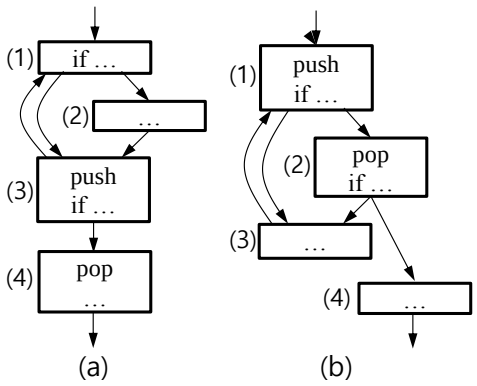


Figure 3.6: Incorrect push/pop insertion for a loop

When the host code includes a loop (e.g., (1), (2), and (3) in Fig 3.6), we must handle a few other exceptional cases. If the immediate post-dominator of a conditional branch block is out of the loop, the push operation marked before the conditional branch is repeatedly executed while the corresponding pop operation is not performed during the loop iteration. In Figure 3.6(a), the immediate post dominator of the conditional branch block (3) is out of the loop, so the push operation may be performed many times while the pop operation will only be executed once in (4). Also, even if the immediate post dominator is in the loop, there can be push/pop imbalances if the block(s) with the backward edge of the loop is not dominated by the immediate post dominator. For example, in Figure 3.6(b), the immediate post dominator of the conditional branch block (1) is block (2). However, there is a path from block (1) that leads to the block with the loop back-edge (block (3)) without crossing block (2). Therefore, the push operation may be performed more than the corresponding pop operation. To handle these exceptional cases, we insert a push operation in the preheader for the loop so that the push is performed only once upon the entrance of the loop, and insert a pop operation in the (common) immediate post dominator of the loop-exiting block(s) of

the loop. We rely on this push-pop pair to handle all the conditional branch blocks that correspond to the above exceptional case. Note that push operations do not affect the t_{PC} value so moving the push operation in front of the loop does not change the correctness of the result. This process is described at lines 1-4 and 7-8 in Algorithm 1.

3.3.3 Compensation for the Untaken Path

To compensate for the untaken path, we analyze the code to find out which register tag needs to be set in which location. There can also be implicit flow through the memory location, but our implementation does not compensate for the memory locations since memory addresses could be determined in the runtime which means that we will have to actually execute the path to determine which memory tag to apply compensation. This could possibly introduce false negatives, but the chances are relatively low since it will also be hard for the attacker to reason about the implicit flow through the memory locations.

Among the registers, we only need to consider the ones that are live-in to the immediate post dominator of the conditional branch block. Registers that are not live at the entry point of the post dominator are not used after the immediate post dominator and cannot be used for the propagation of data. The register tag for the live-in register is set at runtime if there is at least one instruction defining the register on the execution path between the conditional branch and its immediate post dominator. Thus, an implicit flow through an untaken branch will occur if there is an instruction defining the register through some execution path but not all execution paths.

Based on this idea, we find the minimum number of program points where the tag compensation is needed for each register. We first define three sets of basic blocks as described in lines 20-22 in Algorithm 1. In Figure 3.7, we show these sets of basic blocks where block (1) is the conditional branch block currently concerned with. Blocks which define the register are marked with `D`. Lightly shaded blocks are the

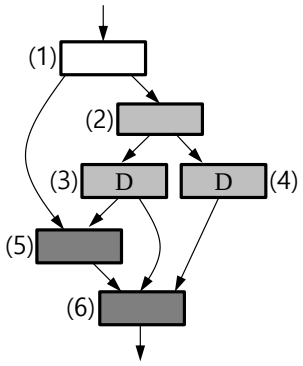


Figure 3.7: Example CFG for tag compensation

blocks which pass at least one basic block which defines the register. Darker blocks are the ones that are reachable from blocks defining the register. After determining these sets of blocks, we start from the conditional branch block (block (1)) and traverse the CFG in depth-first manner. If we encounter a lightly shaded block, we do not need to set the register tag since there will be at least one register definition along that path. If we encounter a dark shaded block, we put the register tag set operation on the edge between the current block (block (1)) and the dark block (block (5)). In this way, we can make sure that the register tag is set for all execution path between the conditional branch block and its immediate post dominator. The entire process corresponds to lines 19-26 in Algorithm 1.

We implemented the code analysis and transformation technique described in Algorithm 1 on the LLVM compiler framework. Our transformation tool inserts the pop and push operations in the host code, which are implemented as special instructions whose encodings are not used by the ISA of the host processor architecture. Thus, at runtime, the host processor regards such marked operations as `nop` operation. Our IFTU processes these operations to manage the t_{PC} stack. For the register tag compensation, we used a dummy `add` instruction that adds 0 to the target register and sets the register to that value. It does not change the semantic meaning of the original program, but it can set the register tag if the PC tag is set at that time.

3.4 Architecture Design of IFTU

In this section, we will discuss the hardware architecture of our solution. After introducing an architectural overview of our solution, we will discuss the detailed structure of IFTU.

3.4.1 Overall System

Figure 3.8 shows the overall system design for our solution, which mainly consists of the host processor and IFTU. In our implementation, as introduced in Section 1, we design our IFTU as an external hardware module and integrate it with the host processor through the system bus, instead of embedding the dedicated hardware logic internally in the host processor [25]. The main advantage achievable from this design strategy is that our proposed solution can be easily adopted by existing commercial platforms such as application processor (AP) SoC platforms for smartphones. Generally in these platforms, the host processors are typically the commodity processors that are quite difficult to modify the internals without tremendous cost and labor from the vendors. Thus, our solution would be adoptable in these platforms as it does not require such modification.

However, such design strategy raises a challenge. As discussed in Section 3, in order to track the information flow of a program, the taintness of tags should be propagated during the program execution according to the propagation rules. Since the rules are dependent on the instruction type and operands, it is necessary for IFTU to know about the instructions executed by the host. For this reason, our IFTU reads the host program code in main memory and extracts the propagation rules as shown in Figure 3.8. Nevertheless, the problem is that, from only the host code, IFTU cannot obtain some essential information required for the correct flow tracking, which is only resolved during code execution. In particular, such information includes (1) an execution path of the original program and (2) memory addresses of load/store instructions.

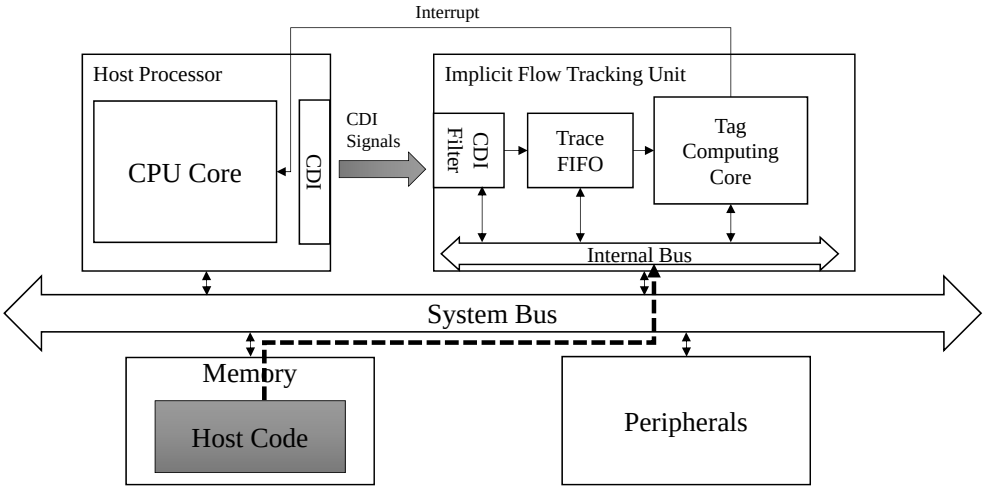


Figure 3.8: Overall system design

Without this information, our IFTU cannot perform the tag operations correctly, while following the execution of the host program.

To resolve the problem, in our solution, we utilize the core debug interface (CDI) in the host processor, as was done in the hardware-based solution introduced in [58]. CDI is an interface placed in recent commodity processors, whose main role is to provide the external debug modules with the processor's internal status information required for debug/trace, without affecting the performance of the host. Based on the specification of CDI in commercial processors and the prior works that utilize CDI [7, 58, 59, 109], in our prototype, we assume that CDI provides a set of signals as follows: instruction address, current context ID (or process ID), data address/value of memory access instructions, branch type/source address/target address, exception and privilege mode information. Since the set of signals includes the necessary runtime information for flow tracking, our IFTU can follow the execution of the host and perform the tag operations correctly.

As shown in Figure 3.8, IFTU consists of three components: the *CDI filter*, the *trace FIFO* and the *tag computing core* (TCC). Although the CDI in the host processor pro-

vides plenty of signals, our IFTU needs only a subset of those signals. The role of the CDI filter is to filter out unnecessary signals and leave the ones that are necessary for the tracking: the current process ID (PID), the address of memory data accessed by a load/store and the target address of a branch. (The current PID is necessary to recognize the active process running on the host. If the monitored program goes into sleep mode, the main controller informs the trace FIFO to ignore the traces from the CDI filter.) IFTU consumes the traces containing such information to obtain necessary information and store them in the trace FIFO in order at runtime.

3.4.2 Tag Computing Core

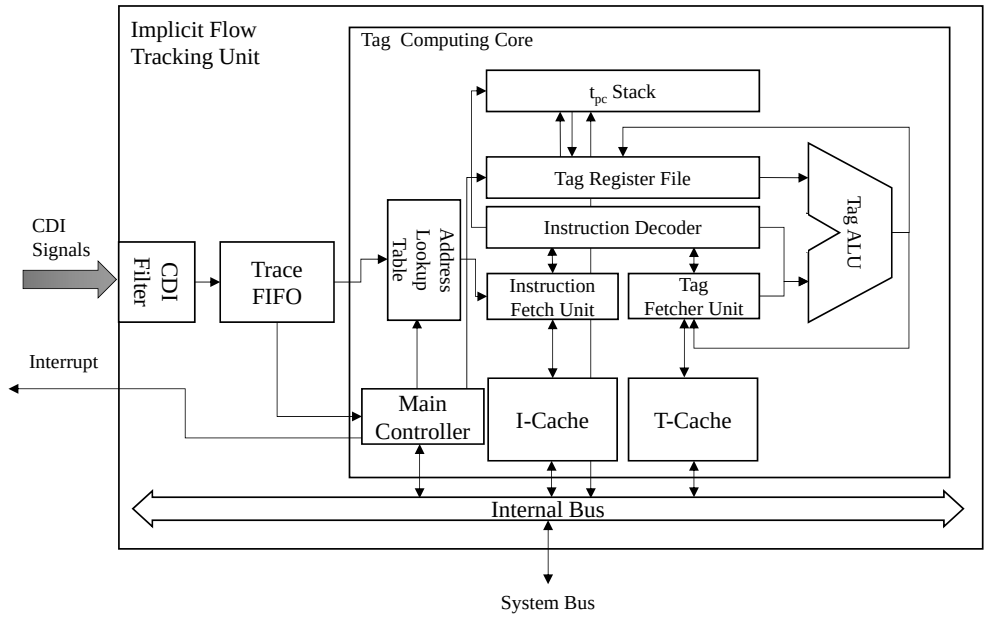


Figure 3.9: Tag computing core architecture design

Figure 3.9 shows the microarchitecture of TCC, whose main role is to manage all tags and perform the tag operations. The overall operation of IFTU is controlled by the *main controller* in TCC. It contains several configuration registers and the host processor can control the functions of IFTU by setting the registers, such as the tag

initialization that marks the location of tainted data. To track the flow of information, we augment the tags to the processor registers and memory locations, as in other previous approaches [72, 97]. The tags for registers are stored to a special register file in TCC called the *tag register file* (TRF). Each entry of TRF represents the 1-bit tag for the corresponding processor register. We also add two register tags t_{PC} and t_{CCR} to the basic structure of TRF, which are used only for the implicit flow tracking. For the memory tags, we reserve a special region called the *tag space* in the main memory. Each bit of the tag space represents the tag for a memory word (32-bit). The *T-cache* is employed in our TCC design to reduce the access latency of tag fetching.

The branch target addresses transferred from CDI are consumed by TCC in order to follow the execution path of the host program. However, since the addresses stored in the trace FIFO are virtual addresses, they cannot be used directly for fetching the host code from main memory. To resolve this problem, TCC includes the *address lookup table* (ALT) where an entry of ALT is comprised of the process ID and the virtual-to-physical address mapping information [58]. At runtime, the host OS kernel updates the entries whenever a code page is allocated on the host, and by using the information the *instruction fetcher unit* reads the host code from the memory with the translated physical addresses. To reduce the access latency required for the instruction fetch, we employ the *I-cache* in TCC as done in previous work [58].

After the host code is fetched, it is delivered to the *instruction decoder* which extracts the propagation rules from the opcode and operands of the instruction. TCC accesses TRF to fetch the tag values if the rule requires register tags. If the operand is the memory address for a load/store, TCC firstly accesses the trace FIFO to acquire the exact address. (Since all load/store instructions generate the CDI signals for the access addresses and the trace containing such information is stored in the trace FIFO in order, it is guaranteed that TCC can obtain the address for the memory instruction.) Then, TCC loads the memory tag corresponding to the address from the T-cache. If

a miss occurs, the *tag fetcher unit* accesses the tag space to handle the miss. Finally, once all the tags are prepared, the *tag ALU* performs the tag propagation with the tags and the resulting values are written to TRF or the T-cache.

To support the management scheme for t_{PC} introduced in Section 3, TCC includes the hardware for the t_{PC} stack as shown in Figure 3.9. As discussed, in the instrumented host code, the push/pop operations for the t_{PC} stack are included. When the instruction decoder encounters such operations, TCC takes the corresponding actions: for push operations, TCC reads the value of t_{PC} from TRF and pushes it to the stack. For pop operations, the top entry of the stack is popped and overwrites the t_{PC}. As our current hardware stack implementation has 32 entries, the stack will overflow if the nested level of branches exceeds 32. To cope with this case, we reserve a memory region for the entries to be stored to if the stack is full. Then the tag value for PC is saved or restored from the memory region.

3.5 Performance and Area Analysis

To evaluate our approach, we have built a full-system prototype on an FPGA board. In this prototype, we used SPARC V8 processor as the host processor which has separate 4KB instruction/data caches. The AMBA2 AHB compliant bus is used to interconnect the host processor with our IFTU and Linux 2.6 is used as the host OS kernel. IFTU is implemented as described in Section 4 and it includes 4KB I-cache/512B T-cache. Based on the parameters, we synthesized our prototype on to a Xilinx Virtex5 FPGA board. Table 3.1 presents the design statistics of our implemented hardware. Our experiment shows that our IFTU incurs a hardware resource overhead of 28.18% for LUTs, and the memory requirement is increased by about 4.5 KB (mainly due to the caches) when compared to the baseline system that includes the host core. It is noteworthy that the amount of logic required to perform the implicit flow tracking is very small. In our approach, the two tag registers (i.e., t_{PC}, t_{CCR}) and the t_{PC} stack are the

components installed for the implicit flow tracking. In our experiment, the hardware resources for these components are estimated to be about only 5.7% of the overall IFTU. This clearly shows that our approach can be implemented with a small amount of additional logic on top of the DIFT hardware for the explicit flow tracking.

Category	Component	LUTs
Baseline System	Host Processor Core	4876
	Bus components and Memory Controller	844
	Peripherals (TIMER, UART, Interrupt Controller and etc.)	963
	Total Baseline System	**6683**
IFTU	Components for CDI (CDI Filter, Trace FIFO, Address Lookup Table)	826
	Main Controller and Bus Interface	330
	Instruction Cache	293
	Tag Cache	180
	Instruction/Tag Fetcher Unit	97
	Instruction Decoder	35
	Tag ALU	109
	t_{pc} Stack	13
	Total IFTU	**1883**
	% IFTU over Baseline System	**28.18%**

Table 3.1: Synthesis Result

To measure the performance overhead of our approach, we chose eight applications from the *mibench* benchmark suite and executed them on three systems each with different configurations. The first configuration is *Native*, which stands for a system that performs the original application without information flow tracking. In the *Explicit* configuration, our IFTU performs only the explicit flow tracking and therefore the host code is not instrumented. Finally, in *Explicit+Implicit*, IFTU performs the tracking scheme proposed in this paper.

Figure 3.10 shows the execution times for the three configurations normalized to that of *Native*. The results show that the *Explicit* incurs about 1.6% performance overhead although the host code is not instrumented in this configuration. The performance loss is mainly due to the resource competition between the host processor and our IFTU. Since both modules are connected to the same system bus and share

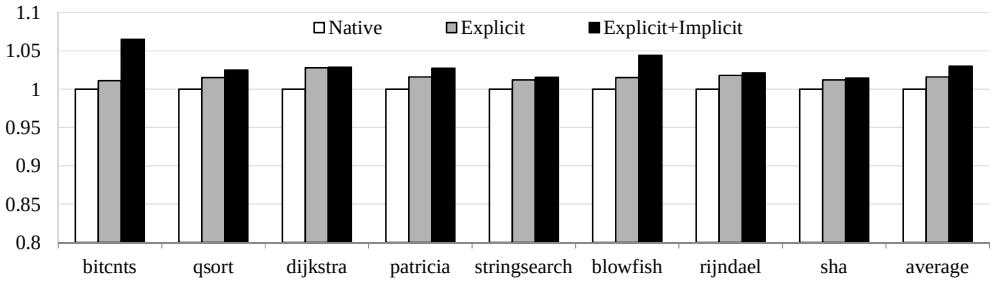


Figure 3.10: Performance Comparison

the same main memory, the bus transactions of IFTU for accessing the main memory slightly degrades the host performance. *Explicit+Implicit*, which stands for our proposed approach, shows an average performance overhead of about 3%. This shows that the overhead caused by our code instrumentation is negligible. Overall, the performance of our approach is much greater than that of the previous hardware approach, RIFLE [101].

We also measured the code size increase due to the code instrumentation. For the given benchmarks, the code size is increased by 0.3% on average. This shows that the code size overhead of our approach is also negligible.

3.6 Security Analysis

To evaluate the accuracy of our taint propagation methods, we used a program that has explicit and implicit information flow. We chose a π computing program that calculates the digits of π and uses the `sprintf` library function to put the ASCII representation of π in the specified buffer. While not strictly a security-related program, we found it adequate for evaluating our implicit flow tracking methods. The program is shown in Listing 1. The program calculates the 1002 digits of π, refining the value at each iteration. The final value will be transformed into the ASCII form by the `sprintf` call. To transform the `sprintf` library function to track implicit flow, we have copied the corresponding functions from *dietlibc*. We have slightly modified the core of the

```
long a[337],p,q,k=4000,t=1000;
char buffer[5000];
int j,n=0;
for(;a[j=q=0]+=2,--k;)
  for(p=1+2*k; j<337; q=a[j]*k+q%p*t,a[j++]=q/p)
    k!=j>2?:(n+=sprintf(&buffer[n],
             "%.3d",a[j-2]%t+q/p/t));
```

Listing 1: π computing program

`sprintf` function so that it involves implicit flow when translating decimal digits to the ASCII form.

In the program, if the memory location for the array `a` is tagged at the start, the correct explicit and implicit information flow tracking scheme should tag the part of the `buffer` array where the ASCII characters are written. We ran our information tracking hardware after tagging array `a`, and examined the tagged memory locations after the program is finished. A total of 593 words were tagged, of which 337 were for array `a` and 250 for `buffer`. 6 other locations were additionally tagged. We have analyzed the execution trace to find out why those 6 locations were tagged and why there is one missing tag for the `buffer` array. Since there are 1002 digits of π, 251 words should have been tagged in the `buffer` array. We found out that all 6 additionally tagged locations are for temporary data in the stack frame that is destroyed when the `sprintf` function is returned. Those temporary data contained the π digit, its ASCII representation, or the length of the character written for the π digit. Thus, we do not need to regard them as false positives. For the `buffer` array, we found that the tag corresponding to the last word of the character string has been reset at the end because the `sprintf` function has put a *null* value at the end of the character string. Since there is a 1-bit tag for each 4-byte word, the tag corresponding to the last word was reset even though there were two tainted bytes.

The analysis of the results shows that our implicit information flow tracking scheme effectively catches the implicit information flow without significant false positive rates.

Although the `sprintf` function is quite complicated, our t_{PC} stack maintenance technique clears the t_{PC} at the right time so that the tainted tags do not spread throughout memory locations. Although there can also be false positives and false negatives introduced by the granularity of the memory tag, we can expect its impact to be small since character data is usually grouped together.

3.7 Summary

This paper presented IFTU, our external hardware engine for implicit (and explicit) flow tracking. To keep track of the implicit flows in a program, we employed a tracking scheme which utilizes a t_{PC} register and stack together with the code analysis and instrumentation technique that help us correctly manage the value of t_{PC}. To perform this task efficiently, we have installed within IFTU hardware logic specialized for the task, such as the t_{PC} stack. We have connected IFTU with the host processor via CDI to acquire the runtime information necessary for tracking, while minimizing the host performance degradation. Our experiments on an FPGA prototype showed that our IFTU can perform both the explicit and implicit flow tracking with only about 3% performance loss. In addition, the synthesis result revealed that the hardware resources required for efficient implicit flow tracking are only about 5.7% of the overall resources for IFTU.

Chapter 4

CRCount: Pointer Invalidation with Reference Counting to Mitigate Use-after-free in Legacy C/C++

4.1 Introduction

Use-after-free (UAF) errors refer to unlawful dereferences of *dangling pointers*, which are the pointers that still point to a freed and thus stale object. UAF errors constitute a serious threat to software security because they are considered significantly difficult to identify by compilers and manual analyses. This difficulty is attributed to the fact that the temporal distances between arbitrary pointer operations, such as setting a pointer to the address of the object, freeing the object, and dereferencing the pointer, can be very long and hence very difficult to analyze accurately in reality. This difficulty has led attackers to leverage UAF errors as a primary source for exploitation in their attempts [65,110,115] to access or corrupt arbitrary memory locations in a victim process.

In the past decade, mitigation against UAF errors has been approached by many researchers from various directions. In one group of studies [66, 67, 69, 87, 89], researchers attempted to detect the UAF error when a pointer is dereferenced to access its referred object (or *referent*). Their goal is to validate the access to the object by

carrying out a sequence of operations to check whether the referent is stale. To support this *access validation* mechanism, each time an object is allocated, they label the object with a unique *attribute* that identifies the allocation. Later, when a pointer is dereferenced, they examine the attribute of its referent to check whether or not the access is made by a dangling pointer whose referent no longer holds the original valid allocation in memory.

Although mitigation techniques based on access validation are claimed to be extensive and complete, they tend to incur an excessively high performance overhead. This high overhead is attributed to the fact that the attribute checks must be executed exhaustively for every memory access, thereby considerably increasing the total execution time. More recently in a different group of studies, as a new direction of UAF defense research to reduce this performance overhead, some researchers have proposed an approach based on *pointer invalidation* [57, 103, 115]. Their mitigation approach against UAF errors is to deter the violations preemptively by getting rid of the dangling pointers at the outset. As a pointer becomes dangling when its referents get freed, this approach in principle can succeed by invalidating all the related pointers when an object is freed such that an exception is triggered when one of the invalidated pointers is dereferenced afterwards. However in practice, for this approach to be successful, we need to address the problem of accurately tracking down every change, such as the creation, copy, or destruction of pointers, and hence, of identifying pointers and their referents located anywhere on the execution path. Unfortunately, this *pointer tracking* problem in general is prohibitively difficult and expensive to solve with high accuracy because the pointers may be copied into a number of different data structures during program execution.

For precise pointer tracking, DANGNULL [57] uses dynamic range analysis to monitor the pointer arithmetic operations that alter the reference relationships between the pointers and the memory objects. Unfortunately, DANGNULL suffers from a high

performance overhead. A majority of this overhead is attributed to the design element that requires the system to immediately update the metadata for the objects when there is a change in the reference relationships. To alleviate this performance overhead, DangSan [103] takes a different approach wherein the total cost for updating the reference relationships is reduced by discarding the transitional changes intermittently produced in a sequence of pointer arithmetic operations. More specifically, in this approach, when any of the existing reference relationships is changed by pointer arithmetic, this change is not reflected immediately in the relationships (thus saving CPU cycles); instead, the change is merely stored in a history table as a record for future reference. The actual reference relationships are checked later when the object is freed. Experiments on DangSan have proven the effectiveness of this approach by showing that it achieved a considerably lower performance overhead than DANGNULL. However, the experiments also show that the history table can become unbearably large when benchmark programs use pointers intensively. For example, the memory overhead of `omnetpp` benchmark was more than a hundred times the original memory consumption. As UAF errors are more likely to be prevalent in programs with a heavy use of pointers, such an immense memory overhead might be a significant obstacle for a broad application of this approach.

From the observations on previous work, we found that such a high overhead, either performance-wise or space-wise, of the existing pointer invalidation techniques is basically caused by the approach that when an object is freed, these techniques promptly locate and explicitly invalidate all the pointers referring to the object. This *explicit pointer invalidation* approach seems to be intuitive as it mitigates UAF errors by eradicating the root cause (i.e. dangling pointers), but it is usually very costly as it demands expensive algorithms or a large amount of space to maintain the up-to-date list of pointer locations linking to each object at all times during program execution. DANGNULL spends many CPU cycles to manage binary trees as the data structures

to store pointer locations. Every time there is a change in one of the locations, the trees are traversed and modified accordingly, consuming a considerable amount of the execution time. Even worse, the total performance overhead increases in proportion to the numbers of pointers and referents, which can increase considerably for programs, such as `omnetpp`, that perform frequent arithmetic operations on a myriad of pointers.

Our findings motivated us to take an alternative approach, which we have named *implicit pointer invalidation* to contrast with the existing explicit approach. The goal of our approach is to prevent dangling pointers by enforcing a basic principle that permits an object to be freed only if there is no pointer currently referring to it. Of course in C/C++, users may deallocate an object at their disposal by invoking the `free` (or `delete`) function irrespective of the existence of pointers linking with the object. Therefore, to enforce the principle in the legacy C/C++ code, we augment each memory object with a single integer, known as the *reference counter*, that records the number of pointers referring to the designated object. When the user intends to `free` an object in the original code, we ignore the function by doing nothing explicit if the corresponding reference counter has a non-zero value. The object is disposed of without explicit effort for invalidation once the counter comes to zero. Indeed, in most real code, reference counters eventually decrease to zero in a sequence of repeated pointer operations, such as assignment, nullification, and deallocation of pointer variables. This implies that even without explicit invalidation time and effort, the proposed scheme can prevent dangling pointers by holding an object remain undeleted until all the links between the object and its referring pointers vanish by themselves, which is tantamount to the implicit invalidation of the referring pointers.

This implicit invalidation scheme sounds plain and naive at the first glance, but its practical application to the existing C/C++ code is very challenging from several aspects. The first aspect of concern is the increase in the memory overhead. In C/C++, `free`/`delete` is purposed to instantly release the memory space occupied by objects

and reclaim the space for reuse. However, such reclamation of memory will be hindered by our implicit scheme that delays the release of a "to-be-free" object, which thus remains *undeleted* until its reference count reduces to zero. Consequently, our scheme could suffer from a memory overhead due to undeleted (and thus unreclaimed) objects, particularly if their number becomes large. Luckily, as will be empirically demonstrated, the overhead was manageably small for most real cases as far as we could accurately compute the reference counts and timely delete the undeleted objects. In fact, this very problem of reference count computing is another important aspect to be considered for the practical application of our scheme to legacy code because the notorious C/C++ programming practices heedlessly violating type safety tend to extremely complicate this problem. For example, common practices, such as unrestricted pointer arithmetic and abusive uses of type casts or unions in C/C++, make it difficult to pinpoint exactly when pointers are generated and deleted at runtime, which in turn results in imprecise and incorrect reference counting.

Because the legacy C/C++ code is such full of type unsafe operations, previous attempts based on reference counting could not effectively tackle the UAF problem in the legacy code [4, 38]. In this paper, we propose *CRCount*, an effective and efficient solution developed to mitigate UAF errors on the basis of implicit invalidation. As reasoned above, the key to the success of our solution depends on the accuracy of reference counting. To compute reference counts with high precision, CRCount adopts a technique called *pointer footprinting*, which tracks down the memory locations of live heap pointers along the execution flow. Our pointer footprinting technique is centered around a special data structure, called the *pointer bitmap*, that represents the up-to-date memory locations where the heap pointers are stored. The bitmap is updated by means of program instrumentation coupled with the runtime library. The empirical results show that, assisted by the footprinting technique, CRCount could track C/C++ pointers with a relatively low overhead and compute the reference counts with high ac-

curacy. CRCount is implemented as a compiler pass in LLVM. Therefore, any C/C++ program can be fortified against attacks exploiting UAF errors merely by compiling its source code with CRCount enabled.

4.2 Related Work

In this section, we will continue the discussion on CRCount by relating it to previous solutions that also aimed to thwart UAF errors in C/C++ code.

Explicit Pointer Invalidation. We divide the explicit pointer invalidation techniques into two folds depending on the manner in which updates on the reference relationships between pointers and objects are reflected. We deem that DANGNULL [57] and FreeSentry [115] update the reference relationships in an *eager* manner because they always update their metadata for pointers and objects right after the pointer arithmetic operations affecting the relationships. In contrast, we deem that DangSan [103] opt for the *lazy* manner in updating these relationships. This enables DangSan to achieve much better performance, but DangSan's memory overhead is often too large, as the size of the history table grows extremely large for programs with heavy uses of pointers. In principle, CRCount embraces the same eager update strategy as DANGNULL in such a way that when an object is linked/delinked with a pointer by pointer arithmetic, the reference relationships are updated instantly by modifying the object's reference count accordingly. However, CRCount does not suffer from the performance issue as it manages much lighter metadata. Moreover, our implicit pointer invalidation scheme does not suffer from the performance overhead that was mandated by DANGNULL to explicitly invalidate all the pointers referring to an object when the object is freed.

Implicit Pointer Invalidation. Thus far, several studies have come close to CRCount in the sense that they benefit from the implicit pointer invalidation even if this fact is not expressed clearly in the literature [14, 75, 88, 111]. To be more specific, their solutions are exempt from additional force required to explicitly invalidate dangling

pointers by delaying the reuse of the recently freed objects in the hope that the number of pointers referring to freed objects would gradually decrease to zero during program execution. However, these approaches differ from CRCount in one important aspect. They do not have notions, such as reference counting, to measure the number of pointer references at runtime. Therefore, they cannot determine exactly how long they should hold the freed objects back from being reused by the memory allocator, and their common schemes are to release the objects simply when specific conditions are met, such as after a random amount of time or when the total size of objects being held reaches a certain limit. Unfortunately, such naive schemes can be easily circumvented by calculated attacks such as heap spraying [32,57] or heap fengshui [94]. In contrast, CRCount, by maintaining precise reference counters for every object, can guarantee the safe release of freed objects for reuse with no presence of dangling pointers.

Object Access Validation. Many security solutions [69,89] have attempted to prevent UAF errors by exhaustively validating every object access via pointers. To this end, they use a lock-and-key mechanism that can check the validity by (1) assigning a unique lock to each object at the creation time, and (2) monitoring whether the object accesses are made by the pointers having the correct key matching the target object's lock. This mechanism realizes a thorough defense against UAF errors. However, they are at a disadvantage as compared to CRCount it terms of accuracy and performance: they generate a number of false positive alarms because of their strictness that goes beyond the common programming practices, and incur a huge performance overhead necessary to intervene in every object access.

Secure Layout of Object. Some systems prevent the exploitation of UAF errors by using prudent layouts of objects. Cling [2] forces new objects to be created only in a memory block that has either never been allocated or has been allocated to objects of the same type. In brief, Cling mitigates UAF errors by ensuring the type safety of the allocated objects. Although efficient, it still allows UAF errors between objects of the

same type. Oscar [31] defeats UAF errors through a careful arrangement of objects. For this, Oscar never reuses the (virtual) memory, such that all the objects are created in a unique memory space, thereby completely blocking the UAF bugs. Oscar facilitates an effective measure against UAF errors. The downside, however, is that it suffers from a higher performance and memory overhead than CRCount because it abandons the efficiency that could otherwise be gained through the maximal reuse of the memory space.

Garbage Collection. Garbage collection makes a program robust against UAF errors through an automatic mechanism that frees an object after confirming that there is no reference to the object. Unlike the case of JAVA and C# in which garbage collection is built into, no hint to distinguish pointers from ordinary objects is provided by compilers in C/C++. Therefore, Boehm-Demers-Weiser garbage collector (BDW GC) [17], a representative garbage collector for C/C++, uses a conservative approach that regards any pointer-size word as a potential pointer value. Such a conservative approach may result in memory leaks in the case of an erroneous recognition of pointer values, although it has been reported that the problem rarely occurs in 64-bit architectures [46]. Garbage collection is also known to cause a non-negligible memory overhead as it trades space for performance [45]. BDW GC works based on dedicated APIs. Although it provides a way to automatically redirect traditional C memory allocation routines to the corresponding APIs, some porting efforts may be required for large real-world programs, especially for C++ programs.

Smart Pointer. To enforce safe and automatic memory management in C++, an extended data type is provided, called the *smart pointer* [4], which encapsulates a raw pointer with a reference counter. Conceptually, a smart pointer owns one raw pointer, meaning that it is responsible for deleting the object referred to by its raw pointer. During program execution, it keeps track of the reference counter through the language's scoping rules and deletes the referred object from the heap when the reference

counter becomes zero. The smart pointer is similar to CRCount in that it is based on the reference counting mechanism. However, there is a critical downside of using smart pointers to enhance memory safety: programmers must take full responsibility of smart pointers. In order for a legacy C++ program to be free from UAF issues, all the raw pointers in the program must be converted manually to smart pointers. Unfortunately, such a complete conversion of every raw pointer is a very time-consuming task to achieve. In fact, this is almost impossible for legacy code unless the entire code is completely re-written by hand from scratch. Furthermore, a smart pointer is basically an extended data type consisting of a raw pointer and the inline metadata, i.e., a reference counter. Unlike other work using extended data types with disjoint metadata [68], a UAF defense solution based on smart pointers cannot maintain the data structure layout compatibility with the existing legacy code.

Taint Tracking. Undangle [20] utilizes taint tracking [73] to detect dangling pointers. It assigns labels to the heap pointers created from memory allocation routines and keeps track of how the pointer is copied through the registers and memory by taint tracking. Later, at memory deallocation time, it checks the labels for the pointers in the program and determine whether the pointer is unsafe based on how much time has passed since the pointer is created. Since it is based on dynamic taint tracking, it can be more precise in determining pointer locations, compared to CRCount, which relies on static type information. However, dynamic taint tracking causes significant performance overhead. It also determines unsafe dangling pointers based on the ad-hoc definition of a lifetime, which can result in an undetected UAF vulnerability.

Hardware-based Approaches. There have been several attempts to extend hardware architectures to handle UAFs efficiently. Watchdog [66] keeps disjoint metadata associated with every pointer, propagates them through the pointer operations, and checks the validity of the pointer upon every access. WatchdogLite [67] provides a fixed set of additional instructions coupled with compiler support to catch UAFs without signifi-

cant hardware modifications. CHERI architecture [62] models pointers as capabilities that include information such as base and bound of the accessible memory region and distinguishes them at the hardware level so that there is no need to separately track pointers in memory as CRCount does by the means of pointer footprinting. CHERI itself does not have native support for preventing UAFs, but it does provide a foundation for accurate garbage collection.

4.3 Threat Model

We assume that the target C/C++ programs have UAF errors. The attacker can trigger a UAF exploit by letting a dangling pointer read/write a value from/to an object that is allocated into the same region that the previous object referred to by the dangling pointer was once allocated to. We do not consider other types of memory errors such as buffer overflow and type confusion. We assume that the integrity of the data structure and algorithm of CRCount is enforced through the security techniques that are orthogonal to CRCount [53, 55]. This assumption is consistent with previous UAF defenses relying on additional metadata [57, 69, 103, 115].

4.4 Implicit Pointer Invalidation

As stated in §4.1, the implicit pointer invalidation scheme enables a safe, efficient defense against UAF errors, but the complications involved in reference counting hinder a wide adoption of this scheme in legacy C/C++. In this section, first, we will provide an overview of how the implicit scheme works with reference counting and then present the challenging problems to be addressed for a successful application of the scheme to real C/C++ code.

4.4.1 Invalidation with Reference Counting

```
1 struct node { struct node *next; int data; };
2 struct node *ptrA, *ptrB;
3
4 ptrA = malloc(sizeof(struct node));  // objA
5 ptrB = malloc(sizeof(struct node));  // objB
6
7 ptrB->next = ptrA;
8
9 /* code execution */
10
11 free(ptrA);
12
13 /* code execution */
14
15 ptrA = malloc(sizeof(struct node));
16 free(ptrB);
```

Listing 2: Code example showing the defense against UAF errors via reference counting

In Listing 2, we present an example code to explain how UAF errors are tackled by the implicit pointer invalidation scheme coupled with the reference counters. Here, RC_{obj} denotes the reference count of a memory object *obj*. In lines 4 and 5, two heap objects, `objA` and `objB`, are created and pointed to by two pointer variables, `ptrA` and `ptrB`, respectively. At this moment, the reference count of each heap object is set to one. Next, `ptrA` is assigned to `ptrB->next`, and RC_{objA} is increased from one to two. Then, in line 11, the `free` function is invoked to deallocate `objA`. Now, note that $RC_{objA} > 0$ as it is still referred to by `ptrA` and `ptrB->next`. In the explicit invalidation scheme [57, 103, 115], both the pointers are delinked with `objA` by explicitly invalidating them right after the object is deleted. However, in the implicit invalidation scheme, the further actions inside the `free` function are interrupted to leave `objA` undeleted, and the pointers remain intact, linking with the object. In line 15, `ptrA` is reassigned to point to a newly allocated object. In this case, without any explicit effort, `ptrA` is in effect invalidated with respect to `objA` because of the delinking of their reference relationship. To reflect this change, RC_{objA} is decreased from two to one. Finally, in line 16 where `objB` is freed, `ptrB->next` can also be considered

to be implicitly invalidated because it is no longer legitimately accessible,[1] thus being effectively delinked with `objA`. Now, $RC_{objA} = 0$, and thus, the object is released and can be reused safely by the memory allocator.

4.4.2 Reference Counting in C/C++

In the above example, we demonstrated how the implicit invalidation scheme with reference counting can preemptively prevent UAF errors by delaying object deletions until the reference counts are decreased to zero. Clearly, the prerequisite for this scheme is flawless reference counting, for which we developed a special mechanism to keep an accurate track of the reference relationships between the pointers and the objects along the execution flow. The reference relationship relevant to an object is expressed by the object's reference count which is dynamically increased or decreased as a pointer is linked or delinked with the object, respectively. Therefore, to accurately monitor such incessant changes in the reference count of an object, we need to pinpoint the moments at runtime when the object is linked or delinked with the pointers. We say that the referring pointers are *generated* or *killed* if the pointers are linked or delinked with their referred objects, respectively. In the code, a referring pointer is generated when its value is stored in the memory, and the pointer is killed when another value overwrites the pointer (see line 15 of Listing 2) or the pointer goes out of scope (see line 16 of Listing 2). In reality, however, perfect reference counting in C/C++ is quite problematic mainly because these languages have weak typing that places no restrictions on the type conversion of objects. For instance, with weak typing, a subfield of an object can be interpreted as either a pointer or a non-pointer alternatively at the time of execution, which makes it extremely challenging to accurately capture all the generations and kills of the pointers, and accordingly update the reference counter of every corresponding referred object.

[1] The pointers enclosed inside a freed object can still be accessed through a UAF vulnerability. For full security protection, these pointers must be nullified upon freeing their enclosing object.

```
1  struct node { struct node *next; int data; };
2  union unode { struct node *next; int data; };
3
4  char *chunk = malloc(CHUNK_SIZE);
5  struct node *ptrA=malloc(sizeof(struct node)); //objA
6  struct node *ptrB=
7   (struct node *)&chunk[n*sizeof(struct node)]; //objB
8  union unode *ptrC=malloc(sizeof(union unode)); //objC
9
10 ptrB->next = ptrC->next = ptrA;
11
12 /* code execution */
13
14 free(ptrA);
15 ptrA = NULL;
16
17 /* code execution */
18
19 free(chunk);
20 ptrC->data = 1;
```

Listing 3: Code example showing the challenges in reference counting in a legacy C/C++ program

Listing 3 shows the practical hurdles in precise reference counting. For simplicity, we only consider RC_{objA} in the code. There are several heap objects created in the code: `objA` and `objC` are newly allocated by `malloc` while `objB` is created by a type conversion of a subregion in the existing array `chunk`. In line 5, by linking the pointer `ptrA` with `objA`, RC_{objA} is set to one. The pointers `ptrB` and `ptrC` are initialized to refer to `objB` and `objC`, respectively. In line 10, `ptrA` is assigned to `ptrB->next` and `ptrC->next`, which results in $RC_{objA} = 3$. In line 14, the programmer wants to delete `objA` when $RC_{objA} > 0$, but as mentioned earlier, this deletion will be denied. In the next line, where `ptrA` is assigned NULL, RC_{objA} is decremented by one. The last two lines of the code exhibit two challenging problems pertaining to reference counting. Firstly, when the array `chunk` is deleted, RC_{objA} has to be decreased as `objA` is referred to by a pointer, `ptrB->next`, which is inside the deleted object. Unfortunately, as `chunk` is declared as an ordinary array, the compiler cannot provide any information with regard to the existence of a pointer inside at

runtime. Therefore, for correct reference counting, we need some mechanism to separately track the location of the pointers inside `chunk`. The code in line 20 presents another practical problem. Here, when `ptrC->data` is set to 1, the previously stored pointer, `ptrC->next`, is simultaneously overwritten by the same value. Therefore, according to the implicit invalidation scheme, RC_{objA} should be reduced as the pointer referring to the object has been technically killed. Here, for correct reference counting, we need to analyze the code and mark the point so that we can decrease the reference count at runtime, and we also need to track whether the pointer is currently stored at the location of `ptrC->data` at that moment.

From all these examples, we can conclude that without a detailed tracking down of all the operations affecting the generations and the kills of the referring pointers, the accuracy of reference counting would be severely limited. This would in turn damage the overall feasibility of the implicit invalidation scheme for mitigation against UAF errors. To summarize, as hinted by the examples, the identification of all the pointer generations and kills in the legacy code is prohibitively complex. The failure to find some pointer generations will result in underestimated reference counts, inducing loopholes in the mitigation of UAF errors. The opposite (i.e., failure to find the kills) will lead to overestimated counts, which, in turn, will result in memory leaks. In the subsequent sections, we will show how CRCount addresses this challenge.

4.5 Design

In this section, we elaborate on the design of CRCount, our UAF error prevention system based on implicit pointer invalidation. First, we present a brief overview of the entire system, and then we provide a more detailed explanation of each component.

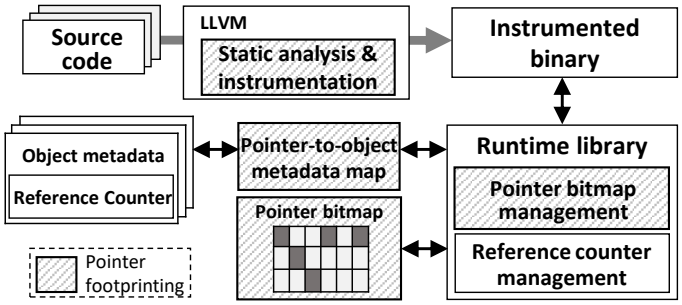


Figure 4.1: Overview of CRCount

4.5.1 Overview

Figure 4.1 shows an overview of CRCount. At the core, CRCount uses a technique called *pointer footprinting* (§4.5.2) to overcome the challenge described in §4.4. The pointer footprinting technique tracks exactly when and where in memory the pointers to heap objects are generated and killed. This technique is centered around a special data structure, the *pointer bitmap*, that represents the exact locations of the heap pointers scattered throughout the program memory. The bitmap is managed by the *runtime library*, which keeps track of the generations and the kills of the pointers at runtime, and reflects the changes into the bitmap by setting or clearing the corresponding bits, respectively. Invocations to the runtime library are instrumented into the target program by CRCount's *LLVM plugin*, which utilizes a static analysis to minimize the number of instrumentation points while preserving the precision in tracking pointers. The idea of using the bitmap to indicate pointer locations has been proposed in previous work on garbage collection [74, 91]. However, unlike previous work, with the help of the compiler instrumentation and the runtime library, we automatically and accurately track down the heap pointers in the entire memory space to enable successful mitigation of UAF errors.

CRCount depends heavily on the pointer footprinting technique for precise reference counting. It associates each heap object with per-object metadata (§4.5.3) that

include the reference counter for the object. Every generation or kill of a pointer is detected and handled by the runtime library. CRCount takes the stored/killed pointer value and consults with the pointer-to-object metadata map to find and increase/decrease the reference count of the object referred to by the pointers (§4.5.3). When the `free` function is called to deallocate an object, CRCount first checks the object's reference counter. If the count is zero, then CRCount lets the function deallocate the object. Otherwise, it halts the function and leaves the object intact. Later, when there is a change (either increment or decrement) in the reference count, CRCount kicks in and checks whether the count is zero. Finally, when the count decreases to zero, implying that the pointers having referred to the object are all implicitly invalidated, CRCount hands the object over to the memory allocator, which will free and reuse the object.

4.5.2 Pointer Footprinting

To enable the precise tracking of heap pointers, CRCount uses the pointer footprinting technique, which is centered around the pointer bitmap data structure that enables us to efficiently locate all the pointers in the memory that refers to the heap objects. The pointer bitmap is basically a shadow memory for the entire virtual memory space, which marks the locations where the heap pointers are stored. We assume that pointers are aligned to an 8-byte boundary, which would be true in most cases as pointer-type variables are typically arranged in an 8-byte alignment by the compiler in a 64-bit system.[2] Note here that the current prototype of CRCount only supports a 64-bit system. Based on this assumption, each bit of the pointer bitmap corresponds one-to-one to all the 8-byte-aligned addresses in the virtual memory space; thus, we can identify the exact pointer locations through the pointer bitmap. Owing to the structural simplicity and compactness of our bitmap, the runtime library can efficiently manipulate it with a combination of simple bit operations such as shifting and masking. The bitmap oc-

[2]We have encountered a few cases where this assumption does not hold true. We will explain these cases in §4.9.

Runtime library function	Invoked at	Description
`crc_alloc`	Heap allocation	Add a mapping for the new heap object to the pointer-to-object metadata map
`crc_store`	Candidate store Instruction	Handle a pointer generation and/or kill due to memory store
`crc_memset`	Memset	Handle pointer kills due to memset'ing a region with identical bytes
`crc_memcpy`	Memcpy	Handle pointer generations and/or kills due to copying of a memory region
`crc_free`	Heap deallocation	Handle pointer kills by heap object deallocation
`crc_return`	Function return	Handle pointer kills by stack frame deallocation

Table 4.1: The list of runtime library functions of CRCount

cupies 1/64-th of the virtual memory space and is reserved at the start of the process through the `mmap` system call. This might seem like a large amount of memory, but fortunately, because of the demand paging mechanism of OSs that delays the allocation of a physical memory block (i.e., frame) until there is an actual access, the bitmap does not occupy much physical memory at runtime. Furthermore, as the access to the pointer bitmap follows the original locality of the memory accesses, in practice, the physical memory overhead for the bitmap is negligible.

The pointer bitmap is managed by the runtime library. Table 4.1 shows a list of the runtime library functions, along with the program points where they are invoked and their tasks at these points. The function `crc_alloc` does not update the pointer bitmap, but when a new heap object is allocated, it adds a new mapping for the object to the pointer-to-object metadata map to be used in the reference count management (refer to §4.5.3 for details). Moreover, as we are only interested in the pointers to the heap objects, the runtime library functions look up the pointer-to-object metadata map before setting the bits in the pointer bitmap. The function `crc_store` sets or clears, respectively, the corresponding bit in the bitmap when a new heap pointer is

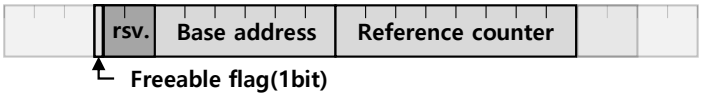


Figure 4.2: Layout of per-object metadata. `rsv.` field is reserved for C++ support (§4.6) and garbage collection (§4.7).

stored (generated) or the previously stored pointer is overwritten (killed) by a store instruction. The functions `crc_memset` and `crc_memcpy` set and clear the bits in the pointer bitmap corresponding to the pointers that are killed and/or duplicated by `memset` or `memcpy`. The functions `crc_free` and `crc_return` clear the bits in the pointer bitmap corresponding to the pointers invalidated by the heap object deallocation and the stack frame deallocation, respectively.

At the time of compilation, the calls that invoke the runtime library functions are instrumented into the program so that the runtime library can reflect the generations and the kills of the pointers into the pointer bitmap. The instrumentation is done by the CRCount's LLVM plugin that provides an additional pass over the intermediate representation (IR) during the compilation phase. All the runtime library calls except `crc_store` are instrumented in a straightforward manner at every corresponding program point. In the case of `crc_store`, instrumenting all the store instructions will cause the excessive performance overhead. It will be overkill if we consider that only a part of these instructions are actually related to pointer generations and kills. However, as the store of a non-pointer-type value can kill a pointer, as discussed in §4.4.2, a simple examination of the type of stored value in LLVM IR is not sufficient to identify all the instructions that need to be instrumented. To solve this, our LLVM plugin performs a static analysis of the program code to identify the minimum set of instrumentation points required to enable an efficient yet precise tracking of pointers.

Listing 4 shows the pseudo-code of CRCount's LLVM plugin for instrumenting memory store instructions. In the LLVM IR, store instructions assign a source value `val` to a destination address `dest`. We should definitely insert a `crc_store` call

```
for storeInst in program:
  dest = storeInst.dest
  val = storeInst.val

  if !isPointerType(val) && !isCastFromPtr(val):
    if !shouldInstrument(storeInst.dest):
      continue

  if isLoadStoreSame(dest, val):
    continue

  callInst = createCallInst(crc_store, dest, val)
  storeInst.insertBefore(callInst)
```

Listing 4: Pseudo code for instrumenting the store instructions.

when a *pointer* value is written; therefore, the plugin first examines `val` to check whether it is a pointer. It is obviously a pointer if it has a pointer type (`isPointerType`), but sometimes, it can be a pointer even if it does not have a pointer type. For example, the programmer could have cast a pointer into an integer type. In this case, in the IR code, there will be a `bitcast` instruction that casts the type of `val` somewhere before the store instruction. In this context, our LLVM plugin conducts a backward data flow analysis to check whether `val` has been cast from a pointer type prior to the store instruction (`isCastFromPtr`). If it has, then the store instruction is instrumented.

Even if `val` is not a pointer type value, store instructions might implicitly invalidate an existing pointer by overwriting it with a non-pointer value. Thus, the LLVM plugin performs a backward data flow analysis on `dest` to check whether the store instruction can potentially kill a pointer and thus should be instrumented with a call to `crc_store` (`shouldInstrument`). There are two main cases where the instrumentation is necessary. First, the plugin instruments the store instruction if `dest` has been cast from a double pointer type, because in this case, the memory pointed to by `dest` can hold a pointer value. Another case that the plugin mainly looks for in the data flow analysis is a case wherein `dest` is a field of the union type that can hold both a pointer value and a non-pointer value (as shown in Listing 3). However, the determi-

nation of whether or not a specific field of the union can be a pointer type in LLVM IR is non-trivial because the IR code generation phase collapses the type information for the union type, and thus, union types in LLVM only has the type information for a single member field whose in-memory representation is the largest in size among all the fields in the union. For example, if a union type has a pointer type member and a struct type member with the size bigger than that of a pointer, only the struct type is shown as the member of the union type in IR. Nevertheless, with the backward data flow analysis, we can at least determine whether the field pointed to by `dest` is a part of a union type. Consequently, we conservatively instrument the store instruction if the underlying type of the memory object is a union type even if it does not have a pointer type member field at the specific offset.

The LLVM plugin also performs a similar optimization done in DangSan that skips the instrumentation if it can be statically determined that `val` points to the same object that the pointer stored in `dest` points to (`isLoadStoreSame`). In this case, `crc_store` will increment and decrement the same reference counter, so there is no need for the runtime library call to be instrumented. This mainly deals with the case where a pointer is simply incremented or decremented and thus the reference counter of the target object does not change.

4.5.3 Delayed Object Free

To achieve its objective, CRCount enforces the *delayed object free* policy that delays the freeing action as briefed in §4.5.1. CRCount manages the reference counters of objects by using the pointer footprinting technique. When a programmer invokes the function `free`/`delete` to free an object, CRCount checks the object's reference count and stops the function from freeing the object if this count is non-zero. To implement this, we modified the `free` function so that the function cannot automatically free objects. In CRCount, the decision on when to free an object is exclusively made

by our runtime library. Therefore, any manual attempt of a programmer to delete an object is intercepted by the library which will eventually permit the memory allocator to free the object for reuse when the object's count becomes zero.

Per-object Metadata

To realize the delayed object free policy, we must maintain a reference counter for each heap object. To do this, CRCount uses METAlloc [43] to augment the heap objects with the per-object metadata. METAlloc internally maintains a trie-based pointer-to-object metadata map [71]. Given a pointer value, METAlloc retrieves the map and returns a pointer to the object metadata allocated separately when the heap object is allocated. The per-object metadata (Figure 4.2) include not only the reference counter but also two additional pieces of information: the *base address* and a 1-bit *freeable flag*. The base address is required for the memory allocator to free the object when the reference count becomes zero. Note that the `free` function needs the base address of the target object as its unique argument. However, when the last pointer that points to the object is killed, and the reference count is set to zero, there is no guarantee that this pointer will hold the base address of the object. Therefore, CRCount separately keeps the base address of each object to invoke the `free` function correctly. The freeable flag is required for CRCount to mark some objects as *freeable*. When the `free` function is called for an object and its reference count is non-zero, CRCount just halts the function and sets the freeable flag of the object. Thereafter, when the reference counts of objects become zero, CRCount allows only the freeable objects for which the `free` function has been called, to be actually freed by the memory allocator. This is important for CRCount because there are some exceptional cases (discussed in detail in §4.9) that may hinder the correct maintenance of the reference counter. These exceptional cases would decrease even the reference counters of non-freeable objects to zero, and if CRCount mistakenly decides to free these non-freeable objects, the program may crash.

Even though such cases are known to be unusual in the normal programming practices of C/C++ [83], we adopt this freeable flag-based approach for maximum compatibility with the legacy C/C++ applications.

Reference Counter Management

The runtime library includes the code for reference counter management that can update the reference counter according to the pointer generations and kills. When a heap object is allocated, the associated per-object metadata are also allocated. Here, the reference count is initialized to zero, and the base pointer is set to the base address of the allocated memory region. Every time a pointer is stored by a store instruction, CRCount reads the corresponding per-object metadata by using the pointer-to-object metadata map and increases the reference count. For `memcpy`, CRCount first examines the pointer bitmap mapped to the source memory region to find the pointers that are to be duplicated and increases the reference counts corresponding to the objects referred to by these pointers. Every time a pointer is invalidated, either by a store instruction or by any of the `memset`/`memcpy`/`free`/`return` function/instruction, CRCount checks the pointer bitmap to identify the pointers from the destination memory region and decreases the reference counts of the objects referred to by these pointers. For `free` and `return`, CRCount also nullifies all the pointers inside an object or a stack frame to completely block wrongful uses of them. Finally, when CRCount finds that the reference count for an object has become zero and the object's freeable flag is set, it gives the object to the memory allocator that will free the object.

It is noteworthy that CRCount handles `memset`/`memcpy` as well. Not only are they very commonly used in C/C++ programs, but they are also often introduced by compiler optimizations when a contiguous range of memory is set or copied. The previous work on pointer invalidation, such as FreeSentry or DangSan, does not handle these functions for performance reasons, leaving the system exposed to UAF errors.

Note that CRCount is immune to the so-called *reference cycle* problem [108]. Automatic memory management systems (i.e., garbage collector) relying on reference counting suffer from the problem wherein the reference counters of a group of objects are never decreased to zero when the objects are cross-referenced. Since the purpose of automatic memory management systems is to deallocate memory objects automatically without relying on explicit free requests, the reference counts of objects pointing each other will never decrease to zero. To avoid this problem, many systems introduce the notion of *weak* references, which the programmers must wisely use to prevent reference cycles [60, 77]. CRCount does not suffer from reference cycles as it operates based on the `free` functions that already exist in the legacy code. When the `free` function is called for one of the objects involved in the reference cycle, CRCount forcibly kills the pointers enclosed in the freed object and decrements the reference counter of the other object, thereby breaking any reference cycles.

4.6 Implementation

We have implemented the CRCount LLVM plugin as an LTO (Link Time Optimization) module based on LLVM 3.8. The runtime library is written in C and is statically linked into the program binary. The LLVM plugin and the runtime library each consists of approximately 1k lines of code.

Allocation of the per-object metadata. METAlloc provides an efficient mapping between a given pointer and the associated per-object metadata, but it does not provide any way to allocate the metadata itself. We sought for a way to avoid the additional overhead that comes from metadata allocations, since whenever heap object is allocated, the corresponding per-object metadata also needs to be allocated. If we use `malloc` for this purpose, an overhead incurred by `malloc` would be doubled, which could be non-negligible as more memory objects are allocated [40]. Fortunately, each of our per-object metadata mapped to the objects has a fixed size. Thus we can miti-

gate the metadata allocation overhead by using the concept of an object pool. We first reserve an object pool using `mmap` and provided a custom allocator for the per-object metadata, eliminating the costs involved with `malloc`. The current implementation of CRCount performs a linear search over this memory pool to find an empty slot for the allocation of the metadata.

Handling realloc. `realloc` can migrate an object from its original memory region to another memory region. Such behavior of `realloc` necessitates an exceptional handling by CRCount. First, when the contents of the target object are copied to another region, the pointers belonging to the object are copied as well. Therefore, to keep track of the copied pointers correctly, the corresponding bits of the pointer bitmap also have to be copied. Next, after the migration, `realloc` frees the original memory region. In CRCount, however, the free action only has to be allowed when the reference count becomes zero. To enforce this rule, we modified `realloc` to let the runtime library decide when to free the original region, as was done in the `free` function. The runtime library (1) allows the memory allocator to perform the free action if the reference counter is zero or (2) just sets the freeable flag otherwise.

Multithreading support. Multithreading support can be enabled in CRCount by defining `ENABLE_MULTITHREAD` macro variable when building the runtime library. Two major data structures—the reference counters and the pointer bitmap—have to be updated atomically to support multithreading. The reference counters need atomic operations because multiple threads can store or kill the pointers to the same heap object at the same time. As a reference counter is just a single word, we simply used the atomic operations defined in the c11 standard library. We assume that the threads in the target program do not write to the same pointer concurrently without a proper synchronization. We believe that this is a reasonable assumption as it indicates a race condition in the original program. The pointer bitmap also must be maintained in an atomic fashion. Even if we have the above assumption, multiple threads could write

pointers to the nearby memory locations which could end up in the same word in the pointer bitmap. Thus, we also use the atomic operations whenever the bitmap is updated. Besides the reference counters and the pointer bitmap, we made a small change in the per-object metadata allocation/deallocation routine to ensure thread safety.

Note that all of the data structure updates in CRCount only require touching just one word which makes multithreading support very simple and also very efficient in most cases. However, we have encountered a worst case in one of the benchmarks that we tested, where only a small number of objects are allocated and their reference counters are frequently updated by multiple threads. In this case, there will be many lock contentions for the reference counters, which results in a considerable performance overhead. We will give more detail in §4.7.

Double free and invalid free. We can simply implement the prevention capability for double frees on CRCount. As a freeable flag of per-object metadata indicates that `free` has been called for an object, we can easily detect if `free` is called multiple times for the object. CRCount can also be extended to prevent invalid frees. If `free` is called for an invalid pointer, CRCount can easily detect it because there either will be no valid mapping for the pointer in the pointer-to-object map, or the base address of the object metadata will not match the pointer value.

C++ support. For the most part, CRCount can naturally support C++ because CRCount instrumentation operates on LLVM IR, which is language independent and thus does not distinguish between C and C++. C++ concepts like templates, dynamic binding, etc. are lowered to basic functions and LLVM instructions and do not require separate handling by our LLVM plugin. However, C++ `new` and `delete` require some special care. Recall that CRCount delays freeing of the object until its reference count becomes zero. For C++, CRCount must invoke the correct deallocation function according to the function that was used to allocate the object. `malloc`, `new`, and `new []` are three possible choices for the allocation of the object, and the corresponding

deallocation function must be used to deallocate the object. To achieve this, CRCount uses the additional bits next to the freeable flag in the per-object metadata to record and call the right function for the deallocation of the object.

4.7 Evaluation

In this section, we evaluate CRCount by measuring the performance overhead and the memory overhead imposed by CRCount in well-known benchmarks and web server applications. All the experiments have been conducted on Intel Xeon(R) CPU E5-2630 v4 platform with 10 cores at 2.20 GHz and 64 GB of memory, running Ubuntu 64-bit 16.04 version. We applied minor patches to a few of the benchmarks to assist our reference counter management. In §4.9, we will explain these cases in detail.

4.7.1 Statistics

The performance and memory overhead of CRCount can vary depending on the characteristics of the target program. In particular, the number of pointer store operations and the memory usage of CRCount can be a crucial indicator for analyzing the experimental results. Thus, we gathered some of the statistics for the SPEC benchmarks [44] which we will refer to when analyzing the performance and memory overheads in this section. Table 4.2 shows the results for the SPEC CPU2006 benchmarks.

Here, we first compare the number of pointer stores tracked down by CRCount with that by DangSan. We will explain other metrics later in this section. As shown in the `# ptr stores by inst.` column and the `# ptrs` column, in many benchmarks, we can see that the number of pointer stores by the store instruction measured in CRCount is larger than the one in DangSan. The differences are mainly due to a small patch we applied to LLVM in order to ease our static analysis. Specifically, we disabled a part of the bitcast folding optimization, which complicates our backward data flow analysis in tracing the casting operations. We expect the numbers to be de-

benchmark	CRCount							DangSan
	# tot alloc.	# ptr stores by inst.	# ptr stores by memcpy	max mem.	max undeleted	max undel. / max mem.	leaks	# ptrs
400.perlbench	350m	44507m	242m	1103 MB	5838 KB	0.005	1680 B	40490m
401.bzip2	264	2200k	0	3362 MB	0	0	0	2200k
403.gcc	28m	9328m	13m	4075 MB	7491 MB	1.838	288 KB	7170m
429.mcf	21	10086m	574k	1676 MB	0	0	0	7658m
433.milc	6531	2663m	0	679 MB	21 MB	0.032	0	2585m
444.namd	1340	2998k	1198	46 MB	0	0	0	2970k
445.gobmk	622k	609m	10	117 MB	34 KB	0	0	607m
447.dealII	151m	30m	13m	791 MB	2048 KB	0.003	0	117m
450.soplex	236k	876m	1731k	877 MB	27 MB	0.032	0	836m
453.povray	2427k	5784m	2409m	2 MB	18 KB	0.007	0	4679m
456.hmmer	2394k	4458k	0	41 MB	12 MB	0.291	0	3829k
458.sjeng	21	4	0	172 MB	0	0	0	4
462.libquantum	165	130	72	96 MB	32 B	0	0	130
464.h264ref	178k	11m	845m	111 MB	1609 KB	0.014	0	11m
470.lbm	20	6004	0	409 MB	0	0	0	6004
471.omnetpp	267m	13099m	14m	154 MB	1301 KB	0.008	481 KB	13099m
473.astar	4800k	1235m	7667k	471 MB	91 MB	0.195	0	1235m
482.sphinx3	14m	326m	0	44 MB	11 KB	0	0	302m
483.xalancbmk	135m	1711m	1633	385 MB	1018 KB	0.003	576 B	2387m

Table 4.2: Statistics for the SPEC CPU2006 benchmarks. `# tot alloc.` denotes the total number of object allocations. `# ptr stores by inst.` denotes the number of tracked pointer stores by the store instructions, while `# ptr stores by memcpy` denotes the number of pointer stores by `memcpy`. `max mem.` shows the maximum amount of memory occupied by the objects that are allocated but not freed. `max undeleted` shows the maximum amount of memory occupied by the undeleted objects. `max undel. / max mem.` shows the ratio between `max mem` and `max undeleted`. `leaks` shows the memory leak caused by an error in the pointer footprinting. The last column shows the number of pointers tracked down by DangSan.

creased if we elaborate on our static analysis to support the optimization, which would also give a small performance improvement for CRCount.

In the case of `dealII` and `xalancbmk`, CRCount kept track of a fewer number of pointer stores than that of DangSan. This is due to a minor hack in our LLVM plugin that is applied to avoid the problem of incorrect reference counter management in the programs that use the C++ templates from the C++ standard library. Specifically, the problem occurred because only the part of the library code for the template functions defined in the header file was instrumented by our plugin while the rest was not instrumented. In order to solve this problem, we compiled the program with the `-g` option to include the debug symbols and excluded the instructions originating from the library during the instrumentation. Another way to solve this problem would be to compile and instrument the entire standard library with CRCount.

4.7.2 Performance Overhead

To measure the performance overhead of CRCount, we ran and recorded the execution times for several benchmarks and server programs. We compare the performance overhead of CRCount with DangSan and Oscar, which are the latest work in this field. We used the open-sourced version of DangSan for our evaluation while using the numbers reported in the paper for Oscar. We also report the performance overheads for BDW GC. To use BDW GC, programs must use special APIs for memory allocation routines (e.g., GC_malloc instead of malloc) to let the GC track and automatically release the object when there are no references to it. BDW GC provides an option to automatically redirect all of the C memory management functions to use the APIs. We used this option and another option that makes GC to ignore the *free* function. As we later specify, we were not able to compile or correctly run some C application, which indicates that some porting efforts are required to use BDW GC for UAF mitigation. Also, simple API redirection does not work for C++ applications. Instead, all the class

needs to inherit from special `gc` class which provides a new definition of `operator new`. Classes that already have a custom `operator new` function will have to be changed. Because of these reasons, we only show the results for the subset of the C benchmarks which we were able to compile and run correctly.

First, to measure the performance impact on the single-threaded applications, we ran the SPEC CPU2006 benchmark suite [3]. Figure 4.3 shows the results. For CRCount, the geometric mean of all benchmarks is 22.0%, which is approximately the half that for DangSan and Oscar, which respectively are 44.4% and 41%. The performance efficiency of CRCount is even more evident in the pointer intensive benchmarks (`omnetpp`, `perlbench`, and `xalancbmk`). For these benchmarks, CRCount only incurs an average overhead of 92.0%, while both DangSan and Oscar show over 300%. For `povray`, CRCount incurs a higher performance overhead than Oscar and DangSan. For the case of Oscar, note that Oscar does not instrument any memory access. This gives Oscar performance advantages for some benchmarks like `povray` which have relatively a large number of pointer stores (see Table 4.2). For the case of DangSan, let us note that DangSan does not track down any pointers copied through `memcpy`. In contrast, CRCount does track down such pointers for higher accuracy (thus also security), which explains the larger performance overhead of CRCount. For `dealII` and `xalancbmk`, we should consider the advantage that CRCount might obtain by not instrumenting the template-based standard library functions. However, considering the difference between the number of tracked pointers described in Table 4.2, we still expect that the performance overhead of CRCount would be lower than those of DangSan and Oscar. For BDW GC, we could not run `gcc` benchmark. The geometric mean of the performance overhead for the rest of the C benchmark is 0.7% for BDW GC and 13.9% for CRCount, which shows that the current highly optimized and multi-threaded BDW GC can be very efficient for single-threaded work-

[3] linked with the single-threaded version of CRCount runtime library

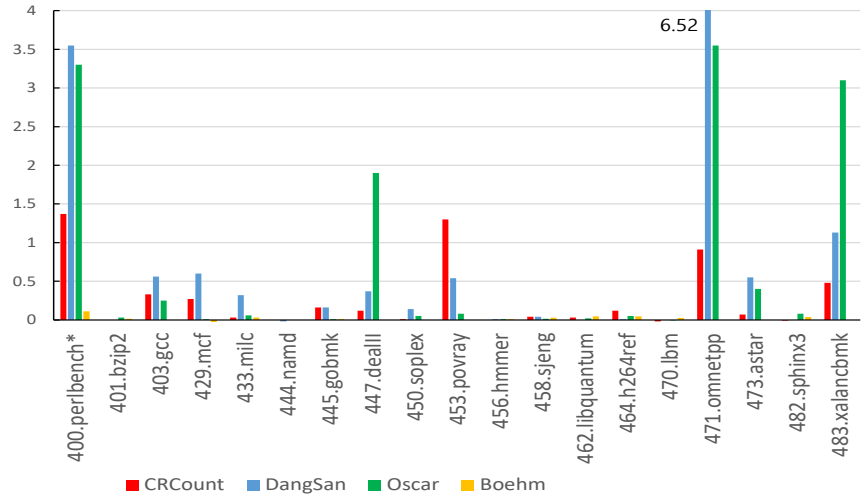


Figure 4.3: Performance overhead on SPEC CPU2006. We use the reported numbers in the original papers for `perlbench` of DangSan, which fails to run, and all the benchmarks of Oscar. For Boehm GC, we were able to run only C benchmarks excluding `gcc`.

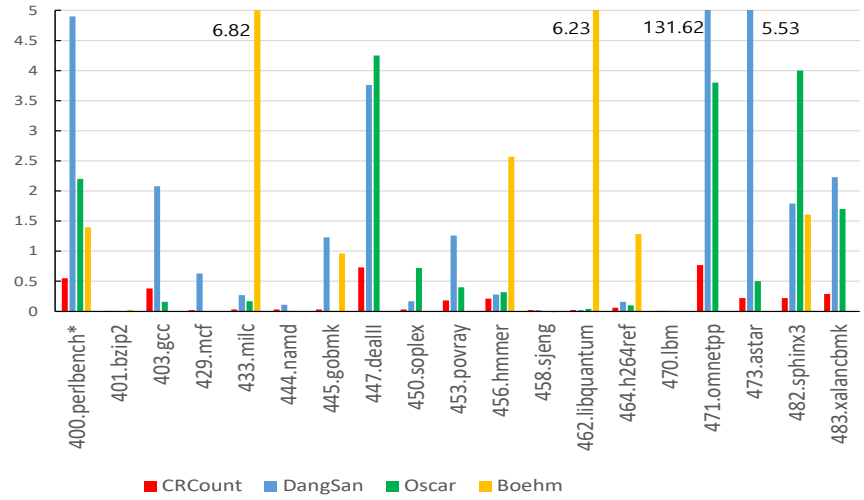


Figure 4.4: Memory overhead on SPEC CPU2006. Some numbers are those that have been reported in the original paper as in Figure 4.3.

loads compared to CRCount which suffers from instrumentation overheads.

We also conducted a set of experiments with the PARSEC [16] benchmarks to evaluate the scalability of CRCount in multithreaded programs. Figure 4.5 shows the results in comparison to the baseline and DangSan. The geometric mean of the

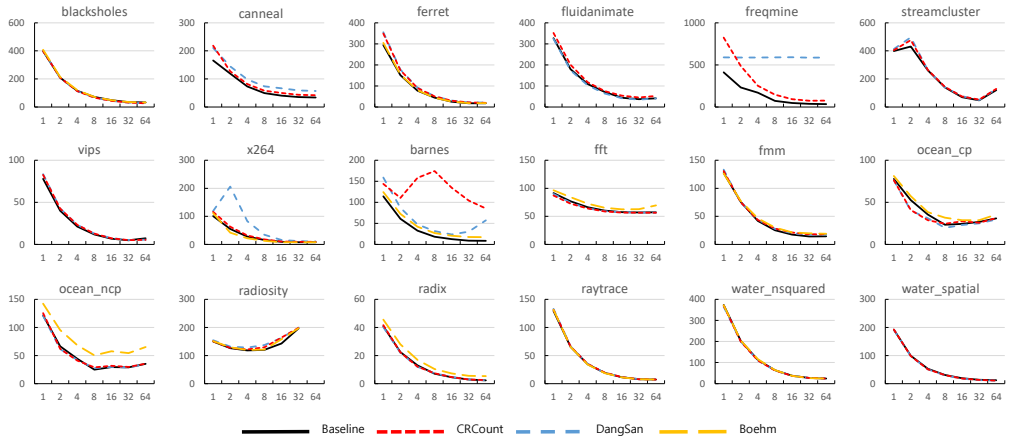


Figure 4.5: Comparison of the execution time on PARSEC. We could not get the correct result for `freqmine` for DangSan because we could not enable OpenMP with DangSan, which is required to run `freqmine` in the multithreaded mode. The results for Boehm GC is only included for the subset of the C benchmarks that we could run.

overheads (excluding `freqmine`) ranges from 6.1% to 22.4% in CRCount and from 6.3% to 17.0% in DangSan, as more threads run concurrently. Overall, CRCount and DangSan show comparable performance overhead in most of the benchmarks. Even though CRCount uses atomic operations to maintain its data structures, it does not introduce critical sections because only a single word needs to be updated at a time. Also, simultaneous accesses to the same reference counter or the same word in the pointer bitmap are rare. Thus, CRCount can be scaled to multiple threads in most cases. `barnes` shows an interesting behavior as it is run with more threads. In `barnes`, only a few large objects are allocated with around 6 billion pointer stores. As the total number of objects is so small, we expect that frequent lock contentions occur when updating the reference counts, which explains such an irregular result. For the subset of the benchmarks we could test with BDW GC, the geometric mean of the overheads ranges from 5.3% to 28.9% in BDW GC and 4.9% to 28.6% in CRCount. CRCount performs comparable to BDW GC for multithreaded workloads.

We conducted additional experiments for evaluating the performance of CRCount on web server applications, including Apache 2.4.33 (with `worker MPM`), Nginx

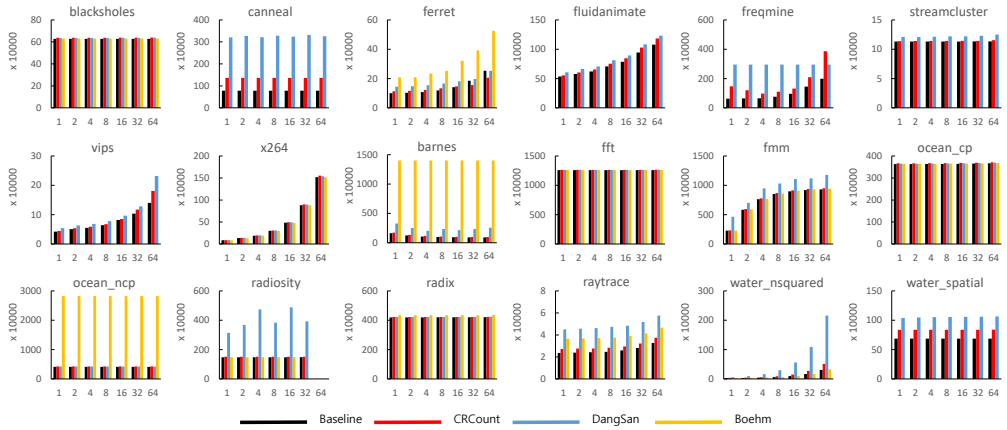


Figure 4.6: Memory overhead on PARSEC.

1.14.0, and Cherokee. We tested each web server with the default configuration files through Apachebench (with 128 concurrent connections and 1,000,000 requests), and measured the throughput in terms of requests per second (RPS). For Apache, the throughput of the baseline is 24024 RPS, while it is decreased to 23051 RPS (slowdown of 4.1%) in CRCount and 22774 RPS (slowdown of 5.2%) in DangSan. The results for other web servers are similar. For Nginx, the throughput of the baseline was 29514 RPS, but it is 20553 RPS (slowdown of 30.4%) in CRCount and 20144 RPS (slowdown of 31.7%) in DangSan. Lastly, for Cherokee, the baseline throughput of 25993 RPS is decreased to 25615 RPS (slowdown of 1.5%) and 24756 RPS (slowdown of 4.8%) in CRCount and DangSan, respectively.

4.7.3 Memory Overhead

In CRCount, let alone its data structures, undeleted objects may be one major factor that potentially consumes substantial memory. To evaluate the overall memory overhead of CRCount, we have recorded the maximum resident set size (RSS) while running the same benchmarks as in §4.7.2.

Figure 4.4 shows the memory overhead of our CRCount, DangSan, Oscar, and

BDW GC for SPEC CPU 2006 benchmarks. Our geometric mean of all benchmarks is 18.0%, which is significantly lower than 126.4% of DangSan and 61.5% of Oscar. BDW GC shows a memory overhead of 125.7% for the tested benchmarks while that of CRCount is 9.7%. Figure 4.6 shows the maximum RSS values for PARSEC benchmarks for baseline, CRCount, DangSan, and BDW GC. The geometric mean (without `freqmine`) of the overhead is from 9.2% to 11.6% in CRCount and from 45.0% to 52.7% in DangSan as the number of threads increases from 1 to 64. The geometric mean of the memory overhead for the benchmarks tested with BDW GC ranges from 56.6% to 70.9% for BDW GC and 5.4% to 6.0% for CRCount.

Finally, we measured the memory overhead for three web server applications used in §4.7.2. The maximum RSS of Apache is 7.8MB in the baseline, 9.9MB in CRCount (26% overhead), and 106.8MB in DangSan (1263% overhead). For Nginx, the maximum RSS is 6.0MB in the baseline, 6.5MB in CRCount (8.2% overhead) and 10.4MB in DangSan (73.3% overhead). For Cherokee, the recorded maximum RSS is 32.1MB, 41.2MB (28.5% overhead) and 62.9MB (95.9% overhead), in the baseline, CRCount and DangSan, respectively.

All those experimental results, we believe, consistently testify the efficiency of CRCount in terms of memory usage. Such memory efficiency of CRCount would be attributed to its compact data structures, but more importantly, to the relatively low memory usage by undeleted objects that remains persistently small in practice. To investigate the relative overhead of undeleted objects further, see Table 4.2 where the `max mem.` and `max undeleted` columns respectively show the maximum total memory for the heap-allocated objects and the undeleted objects. In the `max undel./max mem.` column, we compute the relative overhead of undeleted objects in memory, which is clearly shown to be very small for most benchmarks. On top of that, we have discovered that the majority of these undeleted objects tend to be eventually deleted and handed over by CRCount to the allocator for safe reuse during program execution.

We credit such favorable outcomes mainly to the capability of CRCount that is able to correctly decrease the reference counts whenever generated pointers are killed.

There are still the cases where CRCount fails to accurately keep up the reference counts, thereby being unable to delete undeleted objects even when no more pointers refer to them (see §4.9). The `leaks` column in Table 4.2 denotes the total amount of memory occupied by such undeleted objects. To calculate the numbers in the column, right after program termination, we scanned the entire pointer bitmap to decrease the reference counters corresponding to the pointers still residing in the global variables or the heap objects for which the `free` function has not been called during the execution. The existence of the undeleted objects that still have a non-zero reference count after this process signifies that some pointer kills were not tracked properly, failing to decrease the reference count of these objects. Note that once CRCount fails to track a pointer kill, it is no longer able to delete the corresponding object as the reference count of the object will never decrease to zero. Obviously, these objects are the source of the memory leak induced by CRCount. Luckily, we can see that the numbers on the `leaks` column are negligibly small (or even zero) for almost all benchmarks, indicating that CRCount in fact quite accurately perform reference counting in legacy C/C++ code.

The numbers in Table 4.2 only inform us of the maximum memory space that has once been occupied by heap and undeleted objects during program execution, but it does not give us any clue how much space has been dynamically consumed by these objects at runtime. To obtain this, we have regularly measured the changes in the amount of the memory taken up by undeleted objects and memory leaks over the entire period of each benchmark executions. As can be expected from 4.2, in most benchmarks, the total memory overhead due to the undeleted objects steadily remains low throughout the execution. However in some benchmarks like `gcc` with `200.i` input file (see Figure 4.7), the overhead can sometimes become noticeably high at

Application	CVE	Vulnerability	Original	CRCount	CRCount-det
openlitespeed-1.3.7	2015-3890	UAF	No effect	No effect	Detected UAF
wireshark-2.0.1	2016-4077	UAF	No effect	No effect	Detected UAF
PHP-5.5.9	2016-3141	UAF	Crash (double free)	Detected double free	Detected UAF
PHP-5.5.9	2016-6290	UAF	No effect	Detected double free	Detected UAF
PHP-5.5.9	2016-5772	Double free	Crash (double free)	Detected double free	Detected double free
ed-1.14.1	2017-5357	Invalid free	Crash (invalid free)	Detected invalid free	Detected invalid free

Table 4.3: Real world vulnerabilities tested with CRCount. The `Original` column shows the behavior of the original program when run with the exploit input. We disabled the `Zend` allocator in `PHP` to test the exploits.

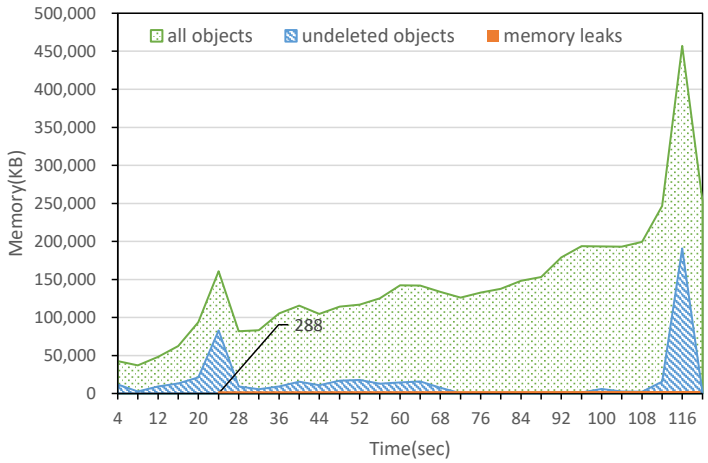


Figure 4.7: Changes in memory usage during the execution of `gcc` with `200.i` input file. `all objects` denotes the total amount of memory allocated for heap-allocated objects and the undeleted objects. `undeleted objects` and `memory leaks` indicate the amount of the memory occupied by undeleted objects and memory leaks, respectively.

some point during program execution although it remains low for most of the execution times. Figure 4.7 displays two peaks in the memory consumption when a large amount of memory is consumed by undeleted objects, but most of it is soon freed as the result of program's normal execution. Figure 4.7 also displays the amount of leaked memory. Note that once a memory leak occurs at some point in the execution, it will never disappear afterward. For instance in Figure 4.7, we have a memory leak of 288 KB in the middle of execution which exists until the end of execution. Fortunately, the amount of wasted memory due to memory leaks is negligible in comparison with the total program memory space, for all the benchmarks we tested.

Although memory leaks are not the major cause of memory overhead in our experiments, they may be a serious problem with long-running programs like server applications where leaks can stack up indefinitely over a long period of program execution. One promising way to cope with the problem is to integrate to CRCount a garbage collection mechanism for reclaiming the leaked memory. Whenever the amount of memory occupied by the undeleted objects exceeds certain limit, we can scan the en-

tire memory of a program and mark all the objects that are referred to by pointers. At this time, all the undeleted objects that have not been marked while scanning the memory obviously correspond to the memory leaks. Now we can reclaim the memory occupied by the identified memory leaks by releasing forcibly. Since CRCount already has a bitmap that pinpoints the pointers from the vast program memory, the garbage collection can be performed more efficiently and accurately than conservative garbage collectors. We have implemented a simple garbage collector to measure how much performance overhead it incurs. The garbage collection starts from the pointers in the stack, the global variables, and the registers, and follows the pointers recursively to scan the pointers in the heap region. All the objects referred to by the pointers are marked (using the reserved field in the per-object metadata) and all the memory leaks are released at the end. We ran `gcc` with the garbage collector enabled because it shows the largest amount of memory occupied by the undeleted objects and thus is expected to give us the worst case performance overhead among the benchmarks. We used three different threshold values (64MB, 128MB, and 256MB) and let the garbage collector run whenever the amount of memory occupied by the undeleted objects exceeded these values. Compared to the version without the garbage collector, it showed an overhead of 2.3%, 1.1%, 0.4%, respectively. We believe that this overhead is acceptable to be integrated into CRCount.

4.8 Security Analysis

In this section, we perform the security evaluation by running CRCount-enabled programs with real vulnerability exploits. We also discuss some of the security considerations for CRCount.

4.8.1 Attack Prevention

To evaluate the effectiveness of CRCount in mitigating UAF errors, we ran several applications with publicly available vulnerability exploits. Table 4.3 shows the list of vulnerabilities tested with CRCount. CRCount successfully detected the double free and invalid free vulnerabilities. We explain the test results with the UAF exploits below.

All the UAF exploits we used accessed the freed region only before it is reallocated. Thus, the UAF accesses in the exploits did not affect the original build of the target program. Note that CRCount is purposed to prevent the attackers from reallocating an object in the memory region still pointed to by the dangling pointers; thus, it did not affect the tested exploits. However, in order for these exploits to eventually be developed into serious attacks, the freed region should be reallocated so that the UAF access can read from/write to the allocated victim object. If CRCount correctly keeps track of the reference counts in the tested programs, it will properly mitigate these advanced exploits. We will show that it is indeed the case in a moment.

For CVE 2016-6290, CRCount detected a double free vulnerability while the original build of the program did not. We found that the double free was triggered by a pointer that still referred to a freed object. The original build of the program did not detect it because another object was allocated at the same address before the free function is called with the dangling pointer. This shows that CRCount successfully delayed the freeing of the object with pointers still referring to it.

To verify that CRCount properly delays the reuse of problematic memory region in the exploits, we have also implemented an extended version of CRCount with a UAF detection capability, called *CRCount-det*. CRCount-det performs checks on every memory access to see if the accessed heap object is marked as freeable. While extra checks on memory accesses cause non-trivial performance overhead, we would immediately know if a pointer is used to access an undeleted object. In our experi-

ments, CRCount-det could detect all the UAF attempts we tested, which also implies that CRCount would properly delay freeing of the object to prevent malicious attempts utilizing the tested vulnerabilities.

4.8.2 Security considerations

One of the concerns about the security guarantee of CRCount is how effective a delayed-memory-reuse based mitigation is against UAF exploits. Recall that one key condition in exploiting an UAF exploit is to locate an attacker-controlled object into the freed memory region pointed to by dangling pointers in order to arbitrarily control the results caused by dangling pointer dereferences. However, in a victim process that CRCount is applied, when an object is freed, no objects are allocated until the reference count becomes zero. At this point, the objects can be accessed only through the existing links (pointers), maintaining their original semantics. Namely, the attacker can no more implant any controllable object into the freed memory region where dangling pointers still point to. As a result, the attackers' capabilities are limited to performing the actions that are originally allowed for the object in the program, unless the attackers use another kind of vulnerability. This makes it impossible, or makes it significantly complicated at least, for the attackers to achieve their goal. It is noteworthy that CRCount nullifies any heap pointers inside the object when the object is freed, so the attackers are further restricted from reusing the heap pointer inside the object.

4.9 Limitations

Custom Memory Allocator. While applying CRCount to the benchmark programs, we encountered some cases (i.e., `gcc` in SPEC CPU2006 and `freqmine` in PARSEC) where the program had to be patched in order for our technique to work correctly. Specifically, the problem occurred mainly due to the use of a custom memory allocator that internally allocates objects from a reserved chunk of memory without

going through the expensive heap management functions. If different types of objects are allocated to the same memory region, the pointers that were stored in the previous object can be overwritten by a non-pointer-type value in the newly allocated object. Had CRCount been able to identify the custom deallocator paired with the custom allocator, it would insert a runtime library call to handle the pointers enclosed in a freed region. Since it was not, we needed to manually identify these custom memory deallocators and explicitly insert the CRCount's runtime library calls to update the pointer bitmap and the reference counts. Specifically, we added 2 lines to `gcc` and 1 line to freqmine to call `crc_free` upon custom memory deallocation.

Unaligned Pointer. Another problem we met in the experiments is that some of the programs stored pointers in 4-byte aligned addresses, which is finer than the assumed alignment (i.e. 8-byte) in the pointer bitmap. Specifically, PARSEC's `freqmine` benchmark used a custom allocator that aligns objects at a 4-byte boundary. We addressed this by modifying the custom memory allocator to align objects at a 8-byte boundary. Also, `Apache` web server used `epoll_event` struct defined with `__attribute__((packed))`, which made the pointer inside the struct to be located at a 4-byte boundary. We addressed this by wrapping the struct so that the pointer is located with an 8-byte alignment. Note that CRCount could just ignore the unaligned pointer store by not increasing the reference count for the stored pointer. We chose to patch the code for more complete protection. 12 lines were modified in `freqmine` and 10 lines in `apache` to ensure pointers are stored at aligned addresses.

Vectorization Support. Our prototype CRCount implementation currently does not support vectorization in LLVM IR. DangSan also does not support vectorization—it simply ignores the stores of vector types. Even though vector operations rarely have to do with pointer values, as ignoring the vector types could adversely affect reference counter management, we instead turned off vectorization in all the experiments. It is our future work to correctly deal with the vector types in our analysis and instrumen-

tations.

Limitations of Pointer Footprinting. There are cases where our static analysis fails to determine whether a particular store instruction should be instrumented or not. We perform only intra-procedural backward data flow analysis. Thus, if a pointer is cast before being passed to a function, we cannot analyze how the pointer is cast, and thus we may fail to correctly decide whether to instrument the store instruction or not. However, since we used LLVM link-time optimization (LTO), many functions are inlined to their caller, which enabled us to get much information from the backward data flow analysis. Another problem regarding static analysis is that we cannot track type unsafe pointer propagation through memory. For example, a pointer could be cast to an integer, stored in some integer field of a struct type variable, and passed around the program through memory as an integer. The pointers stored as an integer data in this process will not increase the reference counts of their corresponding objects. This is a common limitation faced by every approach based on pointer tracking [57, 68, 69, 103, 115]. Like all the approaches based on source code, we cannot instrument the libraries distributed as a binary file. This can cause errors in reference counter management if a pointer stored in the instrumented program is killed in such uninstrumented binary libraries.

4.10 Summary

CRCount is our novel solution to cope with UAF errors in legacy C/C++. For efficiency, CRCount employs the implicit pointer invalidation scheme that avoids the runtime overhead for explicit invalidation of dangling pointers by delaying the freeing of an object until its reference count naturally reduces to zero during program execution. The accuracy of reference counting greatly influences the effectiveness of CRCount. Therefore in our work, we have developed the pointer footprinting technique that helps CRCount to precisely track down the location of every heap pointer along the exe-

cution paths in the legacy C/C++ code with abusive uses of type unsafe operations. CRCount is effective and efficient in handling UAF errors in legacy C/C++. It incurs 22% performance overhead and 18% memory overhead on SPEC CPU2006 while attaining virtually the same security guarantee as other pointer invalidation solutions. In particular, CRCount has been more effective for programs heavily using pointers than other solutions. We claim that this is an important merit because UAF vulnerabilities are more likely prevalent in those programs.

Chapter 5

uXOM: Efficient eXecute-Only Memory on ARM Cortex-M

5.1 Introduction

When it comes to the security of a computing system, the protection of the code running on the system should be of top priority because the code defines security critical behaviors of the system. For instance, if attackers are able to modify existing code or inject new code, they may place the victim system under their control. Fortunately, code injection attacks nowadays can be mitigated by simply enforcing the well-known security policy, W$\oplus$X. Since virtually all processors today are equipped with at least five basic memory permissions: read-write-execute (RWX), read-write (RW), read-execute (RX), read-only (RO) and no-access (NA), W$\oplus$X can be efficiently enforced in hardware for a memory region solely by disabling RWX.

However, even if attackers are not able to modify the system's code, the system can still be threatened by *disclosure attacks* that attempt to read part of or possibly the entire code. Because code often contains intellectual properties (IPs) including core algorithms and sensitive data like cryptographic keys, disclosure attacks severely damage the security of victim systems by exposing critical information to unauthorized users.

Even worse, disclosure attacks can be abused by attackers to launch *code reuse attacks* (CRAs), which allow the attacker to perform adversarial behaviors without modifying its code contents. It has been shown that attackers who can see the instructions in the code may launch a CRA wherein they craft a malicious code sequence by chaining the existing code snippets scattered around the program binary [90].

In order to prevent disclosure attacks, *eXecute-Only-Memory* (XOM) has been a core security mechanism of various countermeasure techniques [11, 18, 19, 29, 41, 42, 80, 95]. XOM is a strong memory protection mechanism that defines a special memory region where only instruction executions are allowed, and any attempts for instruction reads or writes are prohibited. Thus, as long as sensitive information such as IPs and the code contents are stored inside the region protected by XOM, developers are in principle able to prevent direct exposure of the code content as well as the code layout. This simple but tangible security benefit of XOM has led several researchers to propose hardware-assisted XOM on various architectures. For example, some have proposed an architecture that implements XOM by encrypting executable memory and decrypting instructions only when they are loaded [61]. However, since their approach mostly imposes significant changes and overhead on the underlying hardware, it cannot be adopted readily by the processor vendors for their existing products. Instead, many vendors opt for a less drastic approach that simply augments the basic memory permissions with the new execute-only (XO) permission [19, 24].

As of today, many high-end processors provide XOM capabilities by supporting augmented memory permissions. Consequently, by taking benefits from the hardware support for XOM, low-cost security solutions have been built to mitigate real attacks [19, 24, 29, 41]. However, these security benefits are confined to computing systems for general applications since the XO permission is only available in relatively high-end processors targeting general-purpose machines such as servers, desktops and smartphones. More specifically, applications running on tiny embedded devices cannot

enjoy such benefits because only the basic memory permissions (not XOM) are supported in their target processors, which are primarily intended for use in low-cost, low-power computations. As one example of such processors that hardware-level XOM is not built into, we have the ARM *Cortex-M* series, which are prominent processors adopted by numerous low-cost computing devices today [98].

Fortunately, researchers have demonstrated that software fault isolation (SFI) techniques can be used to thwart these prevalent attacks without hardware-level XOM [18, 80]. They are purely software techniques, and thus are able to cope with any types of processors regardless of the underlying architectures. However, the drawback we observed is that SFI-based XOM techniques perform less optimally on certain types of processors, including Cortex-M in particular. More importantly, such techniques can even be circumvented, leading to critical security issues (refer to §5.6.4). Motivated by this observation, this paper proposes a novel technique, called *u*XOM, to realize XOM in a way that is secure and highly optimized to work on Cortex-M processors. Since performance is a pivotal concern of tiny embedded devices such as Cortex-M, efficiency must be the most important objective of any technique targeting these low-end processors. To achieve this objective, *u*XOM leverages a special type of instructions, called *unprivileged loads/stores*, provided by the instruction set architecture for ARM Cortex-M. In an ARM-based system, memory can be divided into two classes of regions according to privilege levels: *non-privileged* and *privileged* memory regions. Unprivileged loads/stores can only access non-privileged memory regions, irrespective to the processor's current privilege level (either in a privileged or non-privileged). On the contrary, ordinary loads/stores are permitted to access privileged regions as long as they are executed under the privileged level. This striking difference between unprivileged and ordinary load/store instructions is the key enabler of our technique.

By capitalizing on this difference, we also need to exploit a unique style of running embedded software on the processors to achieve this ultimate goal of *u*XOM.

In computing systems, software entities are typically assigned certain privileges during execution. For instance, user applications run as unprivileged, and the OS kernel as privileged. In practice, however, applications and the kernel in tiny embedded devices are designed to operate with the same privilege level [28, 51]. This is because these embedded systems are typically given real-time constraints, and the privilege mode switching involved in user-kernel privilege isolation is considered very expensive [51]. For the goal of *u*XOM stated above, we utilize these unique architectural characteristics of Cortex-M processors. More specifically, *u*XOM converts all memory instructions into unprivileged ones and sets the code region as privileged. As a result, converted instructions cannot access code regions, thereby effectively enforcing the XO permission onto the code regions. Since the processor is running with privileged level, code execution is still allowed without any permission error.

However, in order to actually realize *u*XOM, we need to tackle the problem that some memory instructions cannot be changed into unprivileged memory instructions. For example, memory instructions accessing critical system resources, such as an interrupt controller, a system timer and a Memory Protection Unit (MPU), should not be converted. Accesses to these resources always require privilege, so the program will crash if instructions accessing these resources are converted to unprivileged ones. In addition, load/store exclusive instructions, which are the special memory instructions for exclusive memory access, do not have unprivileged counterparts. For these instructions, there is no way to implement the intended functionality with unprivileged memory instructions. Therefore, we should analyze the code thoroughly to find these instructions and leave them as the original instructions.

Unfortunately, these unconverted memory instructions can be exploited by attackers to subvert *u*XOM. For example, if the attackers manage to execute these instructions by altering the control flow, they may bypass *u*XOM by (1) turning off the MPU protection or (2) reading the code directly. To prevent such attacks, the unconverted

memory instructions need to be instrumented with verification routines to ensure that each memory access using these instructions does not break *u*XOM 's protection. However, the attackers can still bypass the verification routines and directly execute the problematic memory instructions. To handle this challenge, we have devised the *atomic verification* technique that virtually enables memory instructions to be executed atomically with the verification routine, thereby preventing potential attackers from executing the memory instructions without passing the verification.

Another important problem *u*XOM needs to handle is that the attackers can alter control flow to execute *unintended* instructions, which may result from unaligned execution of 32-bit Thumb instructions or execution of the data embedded inside the code region [9]. Among the unintended instructions, attackers may find useful instructions for bypassing *u*XOM, such as ordinary memory instructions. To mitigate this attack vector, *u*XOM analyzes the code to find all potentially harmful unintended instructions and replaces them with alternative instruction sequences that have an equivalent function but do not contain any exploitable unintended instructions.

Built upon LLVM compiler and Radare2 binary analysis framework [82], *u*XOM automatically transforms every software component (i.e., real-time operating systems (RTOSs), the C standard library, and the user application) into a *u*XOM-enabled binary. Currently, *u*XOM supports processors based on ARMv7-M architecture, including Cortex-M3/4/7 processors. To evaluate *u*XOM, we experimented on an Arduino Due board, which ships with a Cortex-M3 processor. Our experiment confirms that *u*XOM works efficiently, empowered with the optimized use of the underlying hardware features. In particular, *u*XOM incurs only 15.7%, 7.3% and 7.5% overhead for code size, execution time and energy, while SFI-based XOM incurs overhead of 50.8%, 22.7%, and 22.3%, respectively. To demonstrate the compatibility of *u*XOM with other XOM-based security solutions, we discuss two use cases of *u*XOM: secret key protection and CRA defense. We implemented and evaluated the second use case,

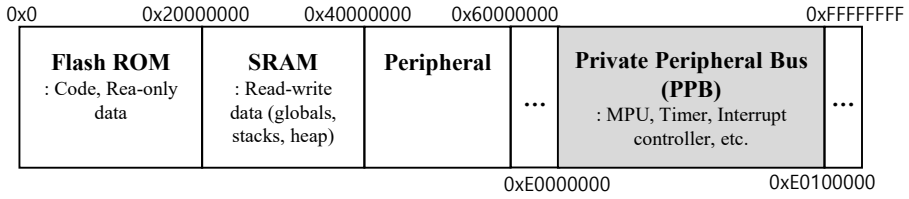


Figure 5.1: System address map for ARMv7-M [47]

the CRA defense. Even when the CRA defense is applied on top of *u*XOM, it shows only moderate performance overhead, which is 19.3%, 8.6% and 9.7% for code size, execution time and energy, respectively.

The remainder of this paper is organized as follows. §5.2 provides the background information. §5.3 explains the threat model and assumptions. §5.4 and §5.5 describe the approach and design of *u*XOM, respectively. §5.6 provides experimental results for *u*XOM and its use cases. §5.7 presents several discussions regarding *u*XOM, and §5.8 explains related works. §5.9 concludes the paper.

5.2 Background

Cortex-M(3/4/7) processors targeted in this paper implement the ARMv7-M architecture, the microcontroller ('M') profile of the ARMv7 architecture, which features low-latency and highly deterministic operation for embedded systems. In this section, we give background information on the key architectural features of ARMv7-M that are required to understand the design and implementation of *u*XOM.

5.2.1 ARMv7-M Address Map and the Private Peripheral Bus (PPB)

ARMv7-M does not support memory virtualization and the regions for code, data, and other resources are fixed at specific address ranges. Figure 5.1 shows the system address map for ARMv7-M architecture. The first 0.5 GB (0x0-0x20000000) is the region where the flash ROM is typically mapped. Code and read-only data are placed

here. The memory range 0x20000000-0x40000000 is the SRAM region where read-write data (globals, stack, and heap) are placed. Devices only use a small subset of each region; our test platform (SAM3X8E) has 512KB of flash and 96KB of SRAM. The memory range 0x40000000-0x60000000 is where device peripherals, such as GPIO and UART, are mapped. The 1 MB memory region ranging from 0xE0000000 to 0xE00FFFFF is the PPB region. Various system registers for controlling system configuration and monitoring system status, such as the system timer, the interrupt controller and the MPU, are mapped in this region. The PPB differs from the other memory regions of the system in that only privileged memory instructions are allowed to read from or write to the region. Generally, access permissions for memory regions can be configured through the MPU which we describe in detail below. However, the access permission for the PPB is fixed and even the MPU cannot override the default configuration.

5.2.2 Memory Protection Unit (MPU)

The MPU provides a memory access control functionality for Cortex-M processors. The biggest difference between the MPU and the Memory Management Unit (MMU) equipped in high-end processors is that the MPU does not provide memory virtualization and thus the access control rules are applied on the physical address space. Depending on the setting of the MPU's memory-mapped registers between 0xE000ED90 and 0xE000EDEC, a limited number (typically 8 or 16) of possibly overlapping regions can be set up, each of which is defined by the base address and the region size. Each region defines separate access permissions for privileged and non-privileged access through the combination of eXecute-Never (XN)-bit and Access Permission (AP)-bits. The available permission settings are RWX, RW, RX, RO, and NA, but in any case, unprivileged access is granted the same or more restrictive permission than privileged accesses. For example, when RO permission is given to a privileged access,

unprivileged access can only have NA or RO permissions. If two or more regions have overlapping ranges, the access permission for the higher-numbered region takes effect. For access to memory ranges not covered by any region, it can be configured to always generate a fault or to follow the default access permission, which depends on the specific processor implementation. It is important to note that the read permission should be included in order for the memory region to be executable. This is the reason that XOM cannot be implemented simply by configuring the MPU in Cortex-M processors.

5.2.3 Unprivileged Loads/Stores

The ARMv7-M architecture only supports a thumb instruction set, which is a variable-length instruction set including a mix of traditional 16-bit thumb instructions and 32-bit instructions introduced in Thumb-2 technology. The unprivileged loads/stores are special types of memory access instructions provided in the instruction set architecture [47]. The main distinction of these instructions is that they always perform memory accesses as if they are executed as unprivileged regardless of the current privilege mode. Thus, memory accesses using these instructions are regulated by the MPU's permission setting for unprivileged accesses. Unprivileged loads/stores are only available in 32-bit encoding and only have immediate-offset addressing mode. They do not support exclusive memory access. They are distinguished by the common suffix '`T`' (e.g., `LDRT` and `STRT`).

5.2.4 Exception Entry and Return

An exception is a special event indicating that the system has encountered a specific condition that requires attention. It typically results in a forced transfer of control to a special software routine called an exception handler. On ARMv7-M, the location of the exception handlers corresponding to each exception are specified in the vector table pointed to by the Vector Table Offset Register (`VTOR`). Note that unlike the

other ARMv7 profiles, the ARMv7-M has introduced a hardware mechanism that automatically stores and restores core context data (in particular, Program Status Register (xPSR), return address[1], lr, r12, r3, r2, r1 and r0) on the stack upon exception entry and return. The ARMv7-M profile also exhibits an interesting feature where an exception return occurs when a unique value of EXC_RETURN (e.g., 0xFFFFFFF1) is loaded into the pc via memory load instructions, such as POP, LDM and LDR, or indirect branch instructions, such as BX. Another thing to note about the exception handling in ARMv7-M is that different stack pointer (sp) can be used before and after the exception. ARMv7-M provides two types of sp, called main sp and process sp. The exception handler can only use main sp but the non-handler code can choose which of the two sps to use. The type of stack pointer being currently used is internally managed through CONTROL register, so that stack pointers are always represented as sp in the binary regardless of its actual type.

5.3 Threat Model and Assumptions

Several conditions must be met to realize *u*XOM. First, the target processor must support the MPU and the unprivileged load/store instructions. We also assume that the target devices run standard bare-metal software in which all included software components, such as applications, libraries, and an OS, share a single address space. Notably, we assume that the entire software executes at a privileged level as mentioned in §5.1.

Next, we define the capabilities of an attacker. We assume that attackers are only capable of launching software attacks at runtime. We do not consider offline attacks on firmware images, such as disassembling, manipulating, or replacing the firmware, because we believe that these attacks can be thwarted by orthogonal techniques such as code encryption or signing. We also leave hardware attacks, such as bus probing [23]

[1] the value of the program counter (pc) at the moment of the exception

and memory tampering [52] out of consideration. However, we believe that our attackers are still strong enough to jeopardize the security of the target devices. The bare-metal software installed in the device is considered benign but internally holds software vulnerabilities, so that the attackers may exploit the vulnerabilities and ultimately have arbitrary memory read and write capability. With such a strong memory access capability, attackers can access any memory region including code, stack, heap and even the PPB region for system controls. They can also subvert control flow by manipulating function pointers or return addresses. We do not trust any software components, including the exception handlers. Event-driven nature of tiny embedded systems signifies that exception handlers can take a large portion of embedded software components [34], so we cannot just assume the security of these handlers. Thus, we assume that attackers can trigger a vulnerability inside the exception handler and manipulate any data including the cpu context saved on exception entry.

5.4 Approach and Challenges

*u*XOM aims to provide XO permission, which enables effective protection against disclosure attacks for code contents, for commodity bare-metal embedded systems based on the Cortex-M processor. *u*XOM tries to minimize the performance penalty by utilizing hardware features, such as unprivileged memory instructions and the MPU provided by Cortex-M processors. Ideally, *u*XOM converts *all* memory instructions into unprivileged ones. It then configures the MPU upon system boot to set code regions to RX for privileged access and NA for unprivileged access. It also sets the other memory regions (i.e., data regions) to non-executable for both privileged and unprivileged accesses. After the configuration, *u*XOM executes code as privileged. All the converted memory instructions (i.e., unprivileged memory instructions) are allowed to access the data regions in the same way as before. However, these instructions are prohibited from accessing the code region and the PPB region in which the MPU and `VTOR` are located

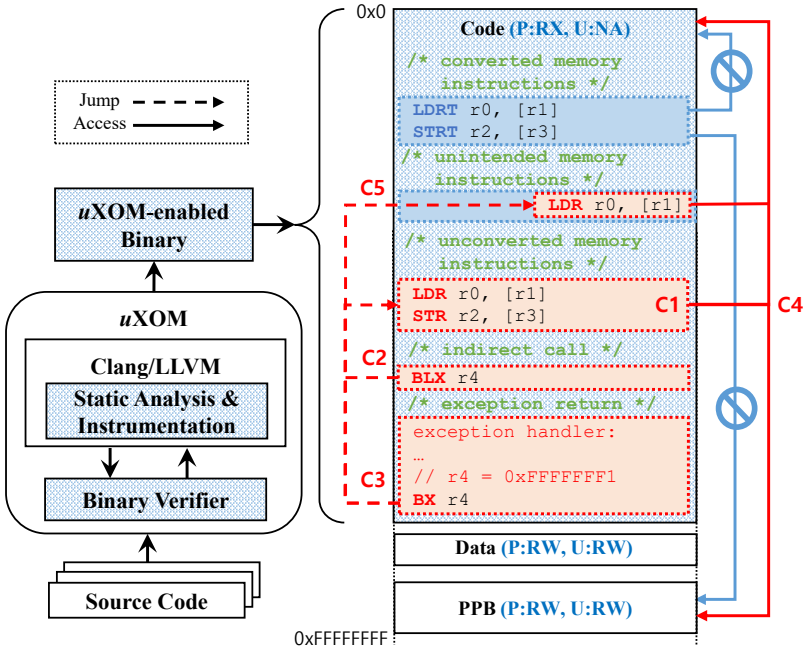


Figure 5.2: *u*XOM approach

that are essential for the security of *u*XOM (see the blue arrows in Figure 5.2). This is because these regions are set to the NA memory permission for unprivileged accesses. As all of the memory instructions have been converted to unprivileged ones, code disclosure attacks are effectively thwarted. In addition, *u*XOM by default enforces W⊕X policy that prevents code execution from writable regions. Therefore, any attempt to inject ordinary memory instructions for code disclosure is blocked as well.

Challenges. The basic principle of *u*XOM is simple and intuitive as described above. To realize *u*XOM in practice, however, we have to overcome some challenges to build a system that works for real programs and cannot be bypassed by any means. We summarize the challenges of realizing *u*XOM as follows.

- **C1. Unconvertible memory instructions:** To implement *u*XOM, we initially tried to convert all memory instructions into unprivileged ones. However, this naïve attempt will be unsuccessful because unprivileged memory instructions do not support

the exclusive memory access that is mainly utilized to implement lock mechanisms, and they cannot access the PPB region to which accesses must be privileged regardless of the MPU configuration. Therefore, we need to thoroughly analyze the entire code, find all these unconvertible instructions, and leave those instructions as the original types. However, these unconverted loads/stores in the program binary resulted in the other challenges, **C2**, **C3** and **C4**.

- **C2. Malicious indirect branches:** In §5.3, we assumed that attackers are capable of altering the control flow at runtime by manipulating function pointers or return addresses. Therefore, attackers can deliberately jump to the unconverted loads/stores and exploit them. Unlike unprivileged loads, the unconverted ones can access the code region. Thus, the attackers are now able to read the code without a permission fault. Furthermore, the attackers can also use the unconverted stores to manipulate memory-mapped system registers in the PPB. For example, they can configure the MPU to enable unprivileged access to the code region, completely neutralizing the protection offered by *u*XOM.
- **C3. Malicious exception returns:** This challenge is similar to **C2** in that attackers can hijack control flow and eventually exploit the unconverted loads/stores to thwart *u*XOM. As explained in §5.2.4, Cortex-M employs a hardware-based context save and restore mechanism for fast exception entry and return. The problem is that as the context is stored in the stack, attackers can exploit a vulnerability while in the exception handling mode to corrupt any context on the stack. In particular, the context includes a return address that represents the program point at the moment the exception is taken. If the attackers corrupt the return address and then trigger an exception return by assigning `EXC_RETURN` value to the `pc`, they will be able to execute any instruction in the program including the unconverted loads/stores.
- **C4. Malicious data manipulation:** As stated in §5.3, the attackers can perform arbitrary memory read/write, and as a result, they have full control over all kind of

program data, such as globals, heap objects, and local variables on the stack. With such control, they can exploit the unconverted loads/stores even while following a legitimate control flow. For example, they can call a MPU configuration function with a crafted argument to neutralize *u*XOM by compromising the necessary memory access permissions.

- **C5. Unintended instructions:** An attacker capable of manipulating control flow may be able to compromise *u*XOM by executing unintended instructions that are not found at compile-time. Concretely, Cortex-M processors targeted in this work support Thumb-2 instruction set architecture [47] that intermixes 16-bit and 32-bit width instructions with 16-bit alignment. Therefore, the attackers can execute unintended instructions by jumping into the middle of a 32-bit instruction. The attackers can also execute unintended instructions through immediate values embedded in code, whose bit-patterns can coincidentally be interpreted as a valid instruction.

5.5 *u*XOM

In this section, we describe the comprehensive details of *u*XOM. We first explain the basic design of *u*XOM for realizing the XO permission (§5.5.1). We then discuss our techniques for overcoming the challenges **C1-C5** (§5.5.2). Next, we present the optimizations applied to reduce performance penalty imposed by *u*XOM (§5.5.3). Lastly, we perform a security analysis to demonstrate that *u*XOM contains no security hazard (§5.5.4).

5.5.1 Basic Design

Before digging into the design details, we briefly describe how *u*XOM works on the system. As illustrated in Figure 5.2, *u*XOM is implemented as a compiler pass in the LLVM framework and a binary verifier. During compilation, *u*XOM performs static analyses and code instrumentation to generate a *u*XOM-enabled binary (i.e.,

Case	Original Instruction	Converted Instructions
1	LDR rt, [rn, #imm5]	LDRT rt, [rn, #imm8]
2	LDR rt, [rn, #imm12]	(ADD rx, rn, #imm12) LDRT rt, [rx, (#imm8)]
3	LDR rt, [rn, #-imm8]	SUB rx, rn #imm8 LDRT rt, [rx]
4	LDR rt, [rn, #+/-imm8]! (pre-indexed)	ADD/SUB rx, rn, #imm12 LDRT rt, [rx]
5	LDR rt, [rn], #+/-imm8 (post-indexed)	LDRT rt, [rn] ADD/SUB rx, rn, #imm12
6	LDR rt, [rn, rm]	ADD rx, rn, rm LDRT rt, [rx]

7	LDRD rt, rt2, [rn, #+/-imm8]	(ADD/SUB rx, rn, #imm8)
		LDRT rt, [rx, (#imm8)]
		LDRT rt2, [rx, (#imm8)+4]

Table 5.1: Basic instruction conversion (only shown for load word instruction)

firmware). Now, when the binary is flashed on to the board and the system boots, *u*XOM automatically enforces the XO permission on the running code.

Instruction Conversion

As RWX or RX is a mandatory permission for code execution on ARMv7-M, executable code regions are always readable and, as a result, are subject to disclosure attacks. Unfortunately, we cannot omit the read permission to implement XOM because the read permission is required for the processor to fetch instructions from memory. Therefore, our strategy for XOM is to deprive all memory instructions of the access capability for code regions. Briefly put, we convert the memory instructions into unprivileged ones and set the code regions to be accessible only with a privileged manner.

Converting the type of the memory instruction may seem to be a trivial task, but not all memory instructions can be readily converted as unprivileged. The unprivileged loads/stores only support one addressing mode with a base register and an immediate

offset which must be positive and fit within 8 bits. On the other hand, the original memory instructions vary in addressing modes, such as register-offset addressing and pre/post-indexed addressing, which updates the base register. Also, there are unprivileged counterparts to the load/store byte and load/store halfword instructions, but there are no corresponding unprivileged instructions for load/store dual (`LDRD`/`STRD`) and load/store multiple (`LDM`/`STM`), which respectively load/store two or multiple registers. To correctly convert all the memory instructions while preserving the program semantics, we sometimes need extra instructions.

Table 5.1 summarizes the conversions we apply to different types of load instructions. Cases 3-6 always need an extra `ADD` or `SUB` instruction for calculating the memory address. We omit an extra instruction for other cases if we can fit the immediate in the unprivileged instruction. Note that we may need an extra register for storing the calculated address if `rn` is used again in other instructions. We implement our conversion before the register allocation phase so that we do not have to worry about the physical registers and let the compiler choose the best register for the temporary results. `LDM`/`STM` instructions are not shown in the table because they only appear during an optimization pass after register allocation. Therefore, when the optimization pass tries to create `LDM`/`STM` instructions, we disable the optimization to prevent the generation of those instructions.

Permission Control

In order for the XO permission based on the unprivileged load/store instructions to take effect, *u*XOM has to configure the MPU to enforce certain memory access permissions. Figure 5.3 shows the default MPU configuration for *u*XOM. Recall that when multiple regions overlap, the permission setting for the higher-numbered region is applied. We create Region 0 covering the entire address space with RW permission for both privileged and unprivileged modes. This is needed to allow unprivileged instruc-

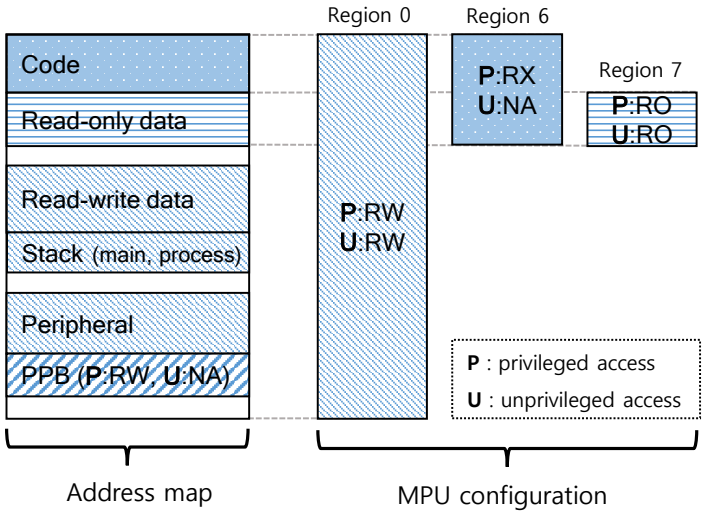


Figure 5.3: *u*XOM-specific memory permission. Unlabeled regions (white-colored regions) in the address map indicate the unused regions where the memory access generates data abort. The PPB region has a default memory permission (P:RW, U:NA) regardless of the MPU configuration.

tions to access the SRAM and the peripheral region. Otherwise, unprivileged access to those regions is not permitted due to the processor's default permission setting. We assign several higher-numbered regions to *u*XOM protection. (Here, we assumed that the number of MPU regions is 8.) Region 6 covers the entire flash region and assigns RX for privileged accesses and NA for unprivileged accesses. Since flash also contains read-only data, we configure Region 7 to let the unprivileged load instruction access the read-only data. To determine the base and size of this region, we need to know the size of the read-only data. To do this, we first compile, find out the read-only data size and generate an include file that is fed back into the MPU configuration code. The linker script is also modified to take this information and place the read-only data appropriately. The configurations are done in the early stage of the reset handler, which is called upon processor reset. In this way, the *u*XOM-specific permission is activated at the early stage of the system boot before attackers can seize control of the system.

5.5.2 Solving the Challenges

So far we have explained the basic design of *u*XOM for activating the XO permission. In the following, we describe how *u*XOM addresses the challenges presented in §5.4.

Finding Unconvertible Memory Instructions

Unprivileged memory instructions do not provide exclusive memory accesses and they cannot access the PPB region. As stated in **C1**, therefore, we need to identify the memory instructions that must not be converted to unprivileged ones and leave them as they are. We simply exclude exclusive memory loads/stores (e.g., `LDREX` and `STREX`) from the conversion candidate. We perform compiler analysis to find loads/stores accessing the PPB. Our analysis of the code base reveals that accesses to the PPB involve calculating the base address from a hard coded address pointing to the PPB region. This is consistent with the claims made in previous work [28]. We conduct a similar backward slicing technique to track how the base address of each memory instruction is calculated. If its address is a constant with the value corresponding to the PPB region, or if it is calculated by adding some offset to that constant value, we identify it as an access to the PPB region and leave it as an original form. For our test platform, intra-procedural analysis suffices to identify all PPB accesses. If a PPB address is passed through a function argument and used in a memory access, we can manually identify those particular cases and add annotations to prevent the compiler from converting the memory instructions as done in previous work [28]. Fortunately, most PPB accesses tend to be performed by the hardware abstraction layer (HAL) provided by the device manufacturer, so no significant amount of annotations are required to complement the static analysis.

Instruction Type	Verification Details

Ordinary stores (`STR`)	if $\text{Target}_{\text{address}}$ points to MPU,
	  $\text{Target}_{\text{value}}$ must not violate *u*XOM-specific memory permissions.
	else if $\text{Target}_{\text{address}}$ points to `VTOR`,
	  $\text{Target}_{\text{value}}$ must have one of the valid `VTOR` values.
	else,
	  $\text{Target}_{\text{address}}$ must point to the PPB region excluding MPU region and VTOR region

Exclusive stores (`STREX`)	$\text{Target}_{\text{address}}$ must not point to the PPB region.
Ordinary loads (`LDR`) Exclusive loads (`LDREX`)	$\text{Target}_{\text{address}}$ must not point to the code region.

Table 5.2: Verification details by the type of unconverted memory instructions. $Target_{address}$ denotes the memory address accessed by load/store instructions and $Target_{value}$ denotes the value to be written by the store instructions.

Atomic Verification Technique

Our solution to deal with **C1** is necessary but may endanger the system. The problem is that, as stated in **C2**, **C3** and **C4**, the strong attackers assumed in §5.3 can easily exploit the unconverted instructions to neutralize *u*XOM. To address this problem, we devise a *atomic verification* technique inspired by the concept of the reference monitor [37, 92]. The key of our technique is to verify memory accesses by the unconverted loads/stores. More specifically, it inserts a routine that performs verification as described in Table 5.2 before every unconverted load/store so that we can confirm whether or not the instruction tries to access code regions or manipulate system configuration necessary for *u*XOM, such as *u*XOM-specific memory permission (solve **C4**). At this point, however, the inserted verification may be bypassed by the attackers who can divert control flow. To prevent this, therefore, the technique enforces the atomic execution of the instruction sequence composed of the verification routine and the following untrusted load/store instruction, ensuring that the attackers cannot execute the unconverted loads/stores without a proper verification (solve **C2** and **C3**). Our basic strategy for atomic verification is to (1) allocate a dedicated register as a base regis-

ter of every unconverted load/store, and then (2) enforce the following two invariant properties regarding the dedicated register.

- **Invariant 1**: The dedicated register must be set to a target address of each unconverted load/store immediately before the associated verification routine. The set value will be maintained only during the execution of the atomic instruction sequence due to **Invariant 2**.
- **Invariant 2**: The dedicated register must hold a non-harmful address (i.e., not a code or the PPB address) when the atomic instruction sequence is not executed.

Now, the accessible memory of the unconverted loads/stores is limited by the value of the dedicated register, which is used as their base register. **Invariant 1** allows the unconverted loads/stores to be executed for their original purpose (e.g., access to the PPB) only through the atomic instruction sequence with a verification. Also, **Invariant 2** prevents any attempt to execute the unconverted loads/stores to access code or the PPB without going through the atomic instruction sequence. As a result, the atomic verification is achieved and the challenges, **C2**, **C3** and **C4**, are addressed successfully. Unfortunately, this implementation strategy decreases the number of available registers by exclusively allocating one register for the PPB access, which may incur additional register spills and occasionally cause a performance drop in some code with a high register pressure.

Therefore, we employ an alternative strategy that is similar to the basic strategy but differs in that it uses the `sp` as a base register of every ordinary load/store rather than using the dedicated register. Now, we can achieve the atomic verification if we are able to enforce on the `sp` the same invariant properties as the dedicated register. Enforcing **Invariant 1** is straightforward, but enforcing **Invariant 2** is challenging because it can cause side effects on the program as the `sp` is used throughout the program, unlike the dedicated register, which is exclusively used only in the atomic instruction sequence. Fortunately, recall that the `sp` is a special purpose register that should always point to

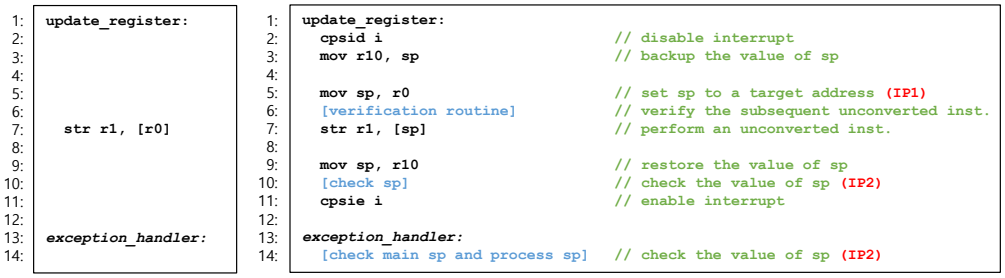


Figure 5.4: An unconverted store before and after applying the atomic verification technique. In the update_register functions `r0` and `r1` are used to pass arguments that will be used as unconverted store's base register and source register, respectively.

the stack, so **Invariant 2** can be safely enforced without worrying about side effects.

Enforcing Invariant 2 on `sp`. We achieve this by adopting the idea suggested by the previous work on SFI [18, 86, 106]—we check the value of the `sp` whenever the attackers could have modified it to point to the outside of the valid region (i.e., the stack region). There are three kinds of program points where we need to insert the `sp` check routines: (1) when the `sp` is modified by a non-constant (i.e., register), (2) when the `sp` is increased or decreased by a constant, and (3) at the entry of an exception handler.

We can usually find the first case when the `alloca` function is called, the variable size array is used, or a stack environment stored by the `setjmp` function is restored by the `longjmp` function, which involves an assignment from a general register to the `sp`. As these cases are rare, we insert the `sp` check routines at all the corresponding points. [2]

The second case is very frequently found in the prolog and epilog of a function when the `sp` is adjusted according to the frame size of the function. The attackers could, although not easily, find a suitable gadget consisting of such an instruction and

[2]Currently, *u*XOM can handle only C code, so we manually insert the `sp` check routine for the `longjmp` function written in assembly language.

repeatedly execute the gadget until the sp is set to a certain value. As pointed out in the previous SFI work [106], if there is a memory instruction based on the sp following the sp modification, the sp can be regulated by placing redzones (i.e., non-accessible memory regions) around the valid stack region. If the redzones are larger than the changes in the value of the sp, the following sp-based memory instruction ensures that any attempt to use the gadget to jump over the redzones will be detected. Fortunately, the address map illustrated in Figure 5.3 shows that there already exist large unused regions that can do the role of redzones. This is because in most cases, the stack, code and PPB reside in a separate memory space, such as SRAM, flash memory and system bus, respectively. Therefore, we create redzones only when the stack is created adjacent to the code and PPB without unused regions in between. Note that redzones can detect the corruption of the sp only if there is an actual memory access using sp. It implies that if, after the sp is corrupted, an indirect branch is executed prior to a sp-based memory instruction, attackers may be able to evade the execution of the memory instruction by manipulating control flow. Therefore, to ensure the success of this method, we implement an analysis that explores all path from each constant sp modification. The analysis checks if there are any sp-based memory instructions before a potentially exploitable indirect branch is encountered. According to our experiments, there are some sp-based memory instructions preceding indirect branches most of the time. However, we sometimes fail to find any sp-based memory instructions or encounter a function call that disables further analysis, and in this case, we insert sp check routines because we can no more guarantee the sp corruption can be detected by the redzones.

Lastly, the attackers can try to avoid all the checks for sp mentioned above by triggering an interrupt right after they corrupt the sp. To neutralize this attempt, we have to validate the sp by inserting another sp check routine at the entry of the exception handlers. Note that as explained in §5.2.4, there are two sps in Cortex-M, and differ-

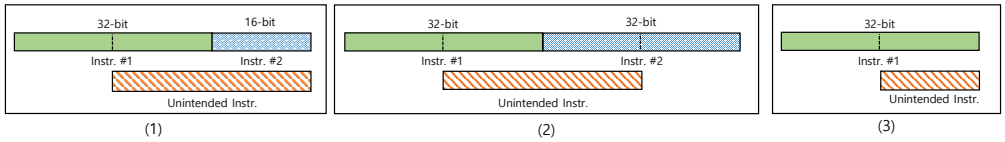


Figure 5.5: The generation of an unintended instruction by an unaligned execution of a 32-bit instruction.

ent `sp` may be activated before and after the exception, so the `sp` check routine at the entry of the exception handler checks the validity of both `sp`s as shown in Figure 5.4. The attackers may try to avoid the `sp` check routine by modifying `VTOR` to alter the exception handlers. To avert this attempt, we identify at compile-time the valid values of `VTOR`, and regulate `VTOR` at run-time so that it does not deviate from the identified values, as described in Table 5.2.

Fulfillment of the Atomic Verification Technique. Now, as both **Invariant 1** and **Invariant 2** can be enforced on the `sp`, we can implement the atomic verification technique using the `sp` without allocating a dedicated register. Figure 5.4 shows an example code on how the atomic verification technique is applied to harden an unconverted store. The original value of the `sp` is backed up while it is used in the unconverted store instruction (Line 3 and 9). The `sp` is assigned a target address (Line 5) and the verification routine verifies the subsequent unconverted store by checking the validity of its target address and target value (Line 6). If the verification is passed, the unconverted store performs memory access (Line 7). Note that because **Invariant 2** is enforced by instrumenting `sp`-update instructions and exception handlers (Line 10 and 14), the `sp` always is forced to point to the stack region except when it is used for the unconverted loads/stores. Therefore, to execute the unconverted store for its original purpose (i.e., accessing the PPB), storing the target address (i.e., the address of the PPB) to the `sp` must be preceded (Line 5), which in turn ensures that the verification routine will be performed (Line 6). At the same time, as the `sp` is used for the unconverted loads/stores and may point to out of the stack region, we temporarily dis-

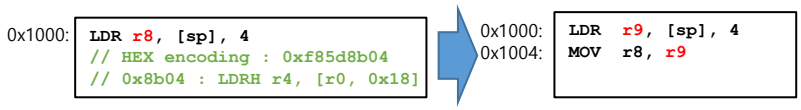
able interrupts (Line 2 and 11), thereby preventing the register from being erroneously checked at the exception handler.

Handling Unintended Instructions

As stated in **C5**, our strong attackers capable of manipulating the control flow of the program can execute unintended instructions to bypass the security of *u*XOM. The unintended instructions are mainly caused by the unique property of Thumb-2 instruction set architecture that intermingle 16-bit and 32-bit instructions. Specifically, as shown in Figure 5.5, when the attackers deliberately jump into the middle of a 32-bit instruction, unintended 16-bit or 32-bit instructions can be decoded and executed. Unintended instructions can also appear in the immediate values in code memory that match the bit patterns of some valid instructions, as illustrated in Figure 5.6.(b). As such, a number of unintended instructions are lurking in code. Fortunately, however, only a minority of them that can be interpreted as ordinary memory instructions or `sp`-modifying instructions can actually be exploited to compromise *u*XOM.

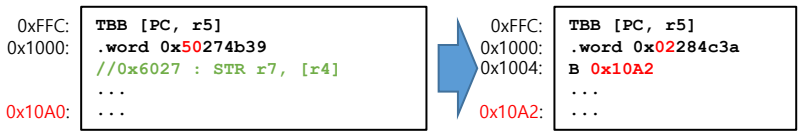
Against this problem, we have implemented the code instrumentation technique based on the idea in the previous work [9] that replaces each exploitable unintended instruction into safe instruction sequences that serve the same function as the original instruction. There was one complication in solving the problem that not all exploitable unintended instructions can be identified at compile time. Many of the exploitable unintended instructions result from immediate values (i.e., symbol addresses) in instructions which are not resolved until all the object files are linked by the linker. Simply transforming all those instructions that use unresolved symbol addresses will result in unacceptable overhead in both performance and code size. Thus, it is preferable to implement the transformation inside the linker or use the static binary transformation tool. However, adding extra instructions at this stage is almost impossible because it will require us to adjust all the `pc`-relative offsets that are used in many ARM instruc-

(a) Unintended instruction originating from a 32-bit MOVW instruction



(b) Unintended instruction originating from a 32-bit LDR instruction

(c) Unintended instruction originating from an immediate value in the code region



(d) Unintended instruction originating from a jump table

Figure 5.6: Examples of unintended instructions and code transformations to remove them.

tions. Adding this capability to current ARM GNU linker implementation will require significant engineering effort.[3] As a work around, we implemented a binary verifier that scans the binary executable for exploitable unintended instructions and records the position of each instruction inside the function. With that information, the program is then recompiled and the exploitable unintended instructions are replaced into alternative instruction sequences. Sometimes, new exploitable unintended instructions are revealed after this process, as code and object layouts are changed and offsets and addresses embedded in the code are changed accordingly. Thus, the interaction between the compiler and the verifier is repeated until there are no exploitable unintended instructions in the binary.

[3]This capability is available in the linker for some architectures like RISC-V which implements aggressive linker relaxation. For those architectures, the `pc`-relative offset resolution is deferred until the linking time to enable linker optimizations that reduce instructions and thus may change the `pc`-relative offsets in the code.

Figure 5.6 demonstrates a few examples showing how the transformation is applied to remove exploitable unintended instructions. Figure 5.6.(a) shows the case where an exploitable unintended instruction (`STR`) is generated from the immediate value of 32-bit instruction (`MOVW`). To remove the exploitable instruction, we divide the original immediate value into two numbers A and B. Then we replace the original 32-bit instruction to use A and add an extra instruction (e.g., `ADDW`) to add B to the register written by the original instruction. Note that for 32-bit instructions whose immediate value is only determined at link time, we only add the extra instruction at compile time and make sure that the linker puts value A and B instead of the original immediate value. Figure 5.6.(b) shows another example that the destination register of the 32-bit instruction (`LDR`) generates the exploitable unintended instruction (`LDRH`). We solve this case by putting the value loaded from memory into the other register and then use an extra `MOV` instruction to copy the value into the original destination register. We have also implemented an optimization in the register allocation pass to prefer invulnerable registers over the others for the destination of these 32-bit instructions so that exploitable unintended instructions can be avoided as much as possible. This saves the use of extra instructions and reduces the performance and code size overhead. Figure 5.6.(c) shows an unintended instruction that exists in a constant embedded in a code region to be loaded by a `pc`-relative load. To sanitize it, we remove the constant value and replace the associated `pc`-relative load with two move instructions. If the resulting `MOVT` or `MOVW` instruction creates new exploitable unintended instructions, it is further transformed similarly to the example in Figure 5.6.(a). Finally, Figure 5.6.(d) shows the case where the offsets in a jump table embedded in the code create an exploitable unintended instruction. In the example, the value 0xA0 (0x50 * 2) is added to `pc` and the control is transferred to 0x10A0. To remove the unintended instruction in this case, we add a trampoline code right after the jump table for the targets with the problematic offsets.

5.5.3 Optimizations

According to our experiments (see §5.6.1), unprivileged memory instructions consume the same CPU cycles as ordinary memory instructions. However, unprivileged instructions are 32-bits in size while many ordinary memory instructions have a 16-bit form. Also, extra instructions that are added as described in §5.5.1 can increase both the code size and the performance overhead. Since code size is another critical factor in an embedded application due to its scarce memory, it can be beneficial to leave the memory instructions in their original form if we can ensure that this does not harm the security guarantees of *u*XOM. In fact, a large number of the instructions do not need to be converted either because they are safe by nature or because they can be made safe through some additional effort. For example, ARM supports `pc`-relative memory instructions which access a memory location that is a fixed distance away from the current `pc`—i.e., the address of the current instruction. As these instructions can only access certain data embedded in the code region, attackers cannot exploit them to access other memory locations. Therefore, we do not need to convert these instructions, so we leave them as long as it is not exploitable as unintended instructions (§5.5.2). We also do not convert stack-based ordinary memory instructions. Numerous instructions use the `sp` as the base address. Almost all of them are 16-bits in size since Cortex-M provides special 16-bit encoding for stack-based memory instructions. Converting all of these as the unprivileged will significantly add to the code size of the final binary. Most of the `LDM`/`STM` instructions, including all the `PUSH`/`POP` instructions, are also based on `sp`. Converting them would require multiple unprivileged instructions which would further increase the code size and even the performance overhead. Luckily, recall that *u*XOM already enforces the invariant properties noted in §5.5.2 on the `sp`. Therefore, attackers cannot exploit the ordinary memory instructions based on `sp`, and we can safely leave `sp` based memory instructions in their original forms.

5.5.4 Security Analysis

*u*XOM builds on the premise that there remains no abusable instructions in a firmware binary. *u*XOM satisfies this through its compiler-based static analysis (§5.5.1 and §5.5.2) that (1) identifies all abusable instructions, such as ordinary memory instructions and unintended instructions, and (2) converts them into safe alternative instructions. This conservative analysis does not make false negative conversions, so *u*XOM is fail-safe in terms of security. In the following, we show that attackers we assumed in the threat model (§5.3) will not be able to compromise *u*XOM.

At Boot-up

As noted in §5.3, we trust the integrity and confidentiality of the firmware image. The firmware image will be distributed and installed with the *u*XOM-related code instrumentation applied. As soon as the system is powered up, the reset exception handler starts to run and the code snippet that *u*XOM inserted at the start of the handler is executed to enforce *u*XOM-specific memory access permissions. Note that the firmware has started its execution from a known good state and the attackers have not yet injected any malicious payloads. Therefore, we can guarantee that *u*XOM will safely enable XOM without being disturbed by the attackers.

At Runtime

Once *u*XOM enables XOM, the attackers are completely prevented from accessing the code. They cannot use unprivileged loads/stores to bypass *u*XOM, so they have to resort to the unconverted loads/stores. Through the instruction conversions and optimizations of *u*XOM, only three types of unconverted loads/stores remain in the binary: stack-based loads/stores, exclusive loads/stores and ordinary loads/stores for the PPB access.

Stack-based loads/stores. *u*XOM 's optimization excludes `sp` based loads/stores

from the conversion candidates. The attackers may be able to execute these loads/stores, but they cannot access the PPB region or code regions. This is because the `sp` is forced to point to the stack regions due to the invariant property (**Invariant 2** in §5.5.2) enforced on the `sp`.

Exclusive loads/stores and ordinary loads/stores for the PPB access. These unconverted loads/stores are protected by the atomic verification technique. Verification routines are inserted just before each unconverted load/store and the atomic execution of the inserted routine and the corresponding unconverted load/store is guaranteed. Of course, the attacker may jump into the middle of the atomic instruction sequence to directly execute the unconverted load/store without a proper verification. However, as the unconverted loads/stores use the `sp` as their base register, the attackers still cannot access the code and the PPB regions.

5.6 Evaluation

*u*XOM transformations are implemented in LLVM 5.0, and *u*XOM's binary verifier is implemented using the Radare2 binary analysis framework [82]. We used the RIOT-OS [10] version 2018.10 as the embedded operating system. As the whole binary, including the OS, runs in a single physical address space at the same privilege level, *u*XOM compiler transformations are applied to the OS code as well as the application code to enable complete protection. We also applied our transformations to the C library (newlib) included in arm-none-eabi toolchain, which had to be patched in a few places to compile and run correctly with LLVM.

To better show the merits of our approach, we also implemented and evaluated SFI-based XOM to compare against *u*XOM. Originally, SFI is developed to sandbox an untrusted module in the same address space. It restricts the store and indirect branch instructions (i.e., by masking or checking the store/branch address) in the untrusted module so that the untrusted module cannot corrupt or jump into the trusted mod-

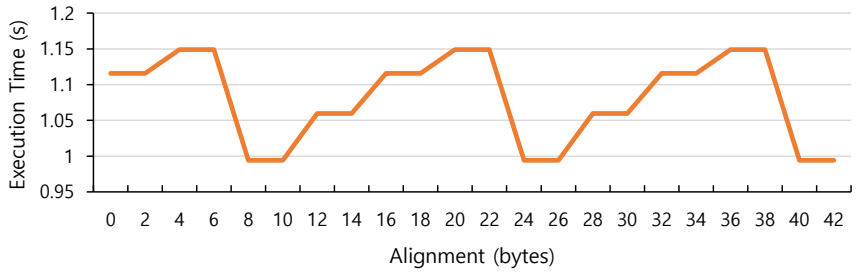


Figure 5.7: Execution time of `bitcount` according to the different alignments of the code region.

ule. It also bundles the checks with the store/branch instructions and prevents jumps into the bundle so that the restrictions applied to the store or branch address cannot be skipped. Capitalizing on the SFI's access control scheme, some studies [18, 80] have implemented the SFI-based XOM that instruments every load instructions with masking instructions to prevent them from reading the code region. However, as these studies focus on high-end devices like smartphones and desktop PCs, we adapted the SFI-based XOM to work on Cortex-M based devices. As our target device do not use virtual memory, code and data must reside in a specific memory region. This prevents us from using simple masking to restrict load addresses and forces us to use a compare instruction to validate the address. Furthermore, the instruction set of Cortex-M requires us to insert additional `IT` (If-Then) instruction to make load instruction execute conditionally on the comparison result. Next, we place the compare and load inside a 16-byte aligned bundle and make sure that they do not cross the bundle boundary. We insert `NOP`s in the resulting gaps. Lower bits of indirect branch targets are masked (cleared) to prevent control flows into the bundle. We also make sure that all possible targets of an indirect branch (i.e., functions and call-sites) are aligned. `POP` instructions used for function returns are converted to masking and return sequence as described in the previous work on SFI [86]. Following the optimization done in the paper [106], the memory load instructions based on the `sp` are not checked and the `sp` is regulated in the same way as in *u*XOM.

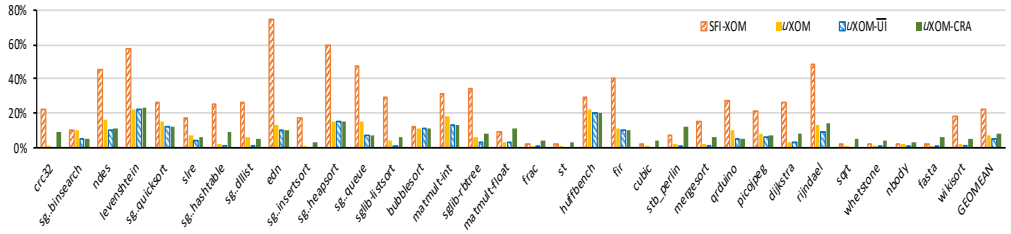


Figure 5.8: Runtime overhead on BEEBs benchmark suite.

To evaluate *u*XOM and the SFI-based XOM, we used the publicly available BEEBs benchmark suite (version 2.1) [78]. We selected 33 benchmarks that are claimed to have relatively long execution time [28][4]. We ran each benchmark on an Arduino Due [5] board which ships with an Atmel SAM3X8E microcontroller based on the Cortex-M3 processor. During the experiment, we found that the program runs give very inconsistent timing results depending on how the code is aligned, even though there are no caches in the processor. After some investigation, we found that the reason is due to the flash memory. The Arduino Due core runs at 84MHz in the default setting, which makes it necessary to wait for 4 cycles (called flash wait state) to get stable results from the flash memory. SAM3X3E chips are equipped with a flash read buffer to accelerate sequential reads [8], which gave us variable results depending on where the branches are located. As a preliminary experiment, we measured the execution time while changing the displacement of the entire code region for `bitcount` benchmark. As shown in Figure 5.7, the changes in execution time show a pattern that is repeated every 16-byte, which corresponds to the size of the flash read buffer. Because of this result, to get a consistent result, we decreased the core frequency to 18.5MHz in all our experiments.

[4]Some of the benchmarks have been dropped in the newest version due to the license problem.

5.6.1 Runtime Overhead

Figure 5.8 shows the runtime overhead of *u*XOM and SFI-based XOM. The geomean overhead of all benchmarks is 7.3% for *u*XOM and 22.7% for SFI-based XOM. The worst case overhead for *u*XOM is 22.3% for `huffbench` benchmark and that for SFI-based XOM is 75.1% for `edn` benchmark. Note that the performance overhead of SFI reported in the previous work [86] for a high-end ARM device (Cortex-A9) is 5%. In the paper, they mention that overhead induced by additional instructions for SFI can be hidden by cache misses and out-of-order execution. Based on this, we presume that the large overhead of SFI-based XOM for Cortex-M3 observed in our experiment is due to the low-power and cache-less processor implementation. This strongly shows the need for an efficient low-end device oriented XOM implementation like *u*XOM.

To inspect the sources of overhead, we built and ran multiple partially instrumented versions of binaries with different kinds of transformations applied. First, to examine the performance impact of removing exploitable unintended instructions, we measured the runtime overhead for *u*XOM-$\overline{\text{UI}}$—a variation of *u*XOM that does not handle unintended instructions. As a result, we measured that the geomean overhead for *u*XOM-$\overline{\text{UI}}$ is 5.2%, which shows that removing unintended instructions incurs 2.1% of overhead in *u*XOM. We then gathered the statistics on the number of conversions and check codes inserted in *u*XOM-$\overline{\text{UI}}$ (Table 5.3). We also measured the overhead ratio in terms of code size and execution time according to the type of conversions and checks (Figure 5.9). In Table 5.3 and Figure 5.9, `no extra instr.` denotes the case where a memory instruction is converted to an unprivileged one without an additional instruction. `imm. offset` denotes the case where an additional instruction is required because the immediate offset is too large or is negative. `pre/post idx.` represents the pre/post-indexed addressing mode and `reg. offset` represents the register-register addressing mode. `double/multiple mem. ops.` represents LDRD/STRD/LDM/STM instructions. For the `sp` check part, `non-const`

Cases	Count (ratio %)
Instruction conversion	
no extra instr.	25932 (77.0)
imm. offset	2547 (7.6)
pre/post idx.	1671 (4.9)
reg. offset	2891 (8.6)
double/multiple mem. ops.	641 (1.9)
`sp` check	
non-const `sp` mod.	18 (0.7)
const `sp` mod. (checked)	769 (28.8)
const `sp` mod. (no check)	1881 (70.5)

Table 5.3: Statistics for instruction conversion and `sp` check instrumentation.

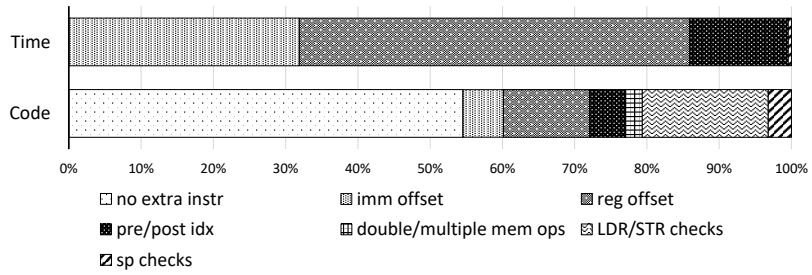


Figure 5.9: Performance overhead breakdown for the different components of *u*XOM-$\overline{\text{UI}}$ transformation.

`sp mod.` is the case where the `sp` is modified by the non-constant (and the check is required). `const sp mod. (checked)` is the case where the `sp` is modified by the constant and requires checking since no load/store based on the `sp` is found afterwards. `const sp mod. (no check)` is the case where the `sp` is modified by the constant but does not need to be checked. Finally, `LDR/STR checks` denotes the instructions inserted for the atomic verification technique.

The statistics shown in Table 5.3 are gathered while compiling the C standard library, RIOT-OS, and each of the benchmarks. Note that although the numbers do not represent those executed at runtime, we can expect some correlation between them. Among the converted memory instructions, the majority of the cases is the one where a memory instruction is directly converted to a single unprivileged memory instruction without any extra instruction (`no extra instr.` accounts for 77% of all conver-

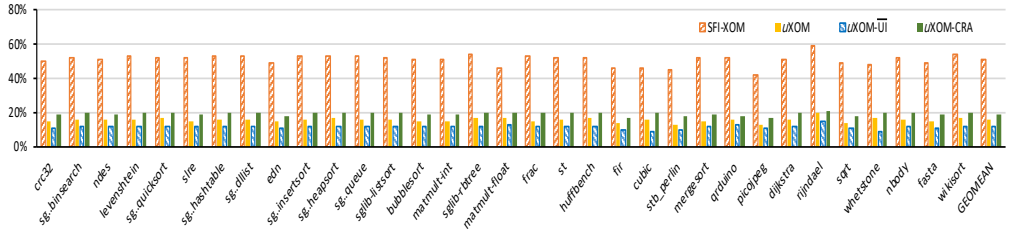


Figure 5.10: Code size overhead on BEEBs benchmark suite.

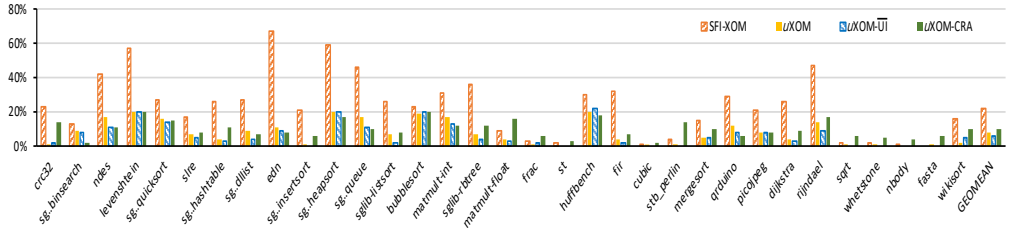


Figure 5.11: Energy overhead on BEEBs benchmark suite.

sions). This tells us that most of the load/store instructions are using an immediate-offset addressing mode and the offset is usually small so that it fits in the immediate field of the unprivileged instructions. As we can see, instructions converted in this way do not contribute to the runtime overhead albeit being the majority. Even though the unprivileged instructions are 32-bits long, they do not increase the overhead unless additional instructions are inserted. This is a big advantage for *u*XOM, and it is the main reason why *u*XOM can be much more efficient than SFI-based XOM.

As illustrated in Figure 5.9, the type of instruction conversions that contributes the most of the overhead is the one for the register-register addressing mode (`reg. offset`). Even though they represent only 8.6% of all conversions, they cause 54% of the total overhead for *u*XOM-$\overline{\text{UI}}$. The reason would be that they are frequently used in time-consuming loops, for example, to index array variables. `imm. offset` and `pre/post idx.` take up the other half of the overhead. Memory instructions that load/store multiple registers (`double/multiple mem. ops.`) cause a negligible runtime overhead; they are rare in number and also, although they are converted into

multiple unprivileged instructions, the original instruction also takes up extra cycles to load/store multiple registers. The `sp` checks that are inserted for stack modification have an only negligible impact on performance as our analysis finds that the `sp` checks are only needed for less than 30% of `sp`-based memory instructions.

5.6.2 Code Size Overhead

To see the impact of instruction insertion by *u*XOM, we measured the size of the code in the final binary, excluding the data size. Figure 5.10 shows the result for both *u*XOM and SFI-based XOM. For *u*XOM, code size is increased by 15.7%, and for SFI-based XOM, it is increased by 50.8%. It shows that *u*XOM can implement XOM with much less code size overhead compared to SFI-based XOM. In addition, we measured that the geomean overhead of *u*XOM-$\overline{\text{UI}}$ is 11.6%, which indicates the amount of increased code for removing unintended instructions is 4.1%. Figure 5.9 shows the source of the overhead that is caused by instruction conversions and checks. First, `no extra inst.` accounts for 54.5% of the code size overhead for *u*XOM-$\overline{\text{UI}}$, differently from the impact that it had on the runtime performance. This is because the original 16-bit load/store instructions are converted to 32-bit unprivileged instruction, and they are large in number, too. Other types of instructions that need additional instructions also increases the code size to some degree. The instructions added for the atomic verification technique (`ldr/str check`) accounts for 17.4% of the code size overhead for *u*XOM-$\overline{\text{UI}}$. Although there are not many instructions accessing the PPB region, around ten instructions are inserted for each of those points, which adds some overhead to the code size especially since the benchmark code size are only around 30KB. We expect the overhead from the atomic verification to be a smaller percentage in the real program with a larger code base.

5.6.3 Energy Overhead

Since many embedded devices running on Cortex-M processors often operate based on constrained battery, energy efficiency is one of the important performance factors for these devices. To measure the impact of *u*XOM on energy consumption, we recorded the power while running the individual benchmarks using the ODROID Smart Power [76]. For the convenience of measurement, the benchmarks were repeatedly executed to run for at least 30 seconds. Figure 5.11 shows the results. For *u*XOM, the geometric mean of all benchmarks is 7.5%, which is slightly larger than 5.8% of *u*XOM-$\overline{\text{UI}}$ but much lower than 22.3% of SFI-based XOM. The results share a similar trend with the execution time since the energy is also affected by the execution time.

5.6.4 Security and Usability

Other than its excellence for performance, we also need to mention the security and flexibility benefits of *u*XOM over SFI-based XOM. *u*XOM provides a better security guarantee against privileged attackers than SFI-based XOM. SFI-based XOM, including the existing studies, focus only on the code disclosure through memory read instructions, because they assume that W$\oplus$X policy is assured by a Trusted Computing Base (TCB) such as the OS kernel. However, as described in §5.3, *u*XOM cannot assume any TCB in the bare-metal environment in which all software components are running with privileges in a single address space. The privileged attacker could neutralize W$\oplus$X by manipulating the MPU configuration register using memory vulnerabilities in the code. To prevent such an attack, SFI-based XOM for Cortex-M would also have to regulate memory write instructions to protect memory-mapped registers for the MPU. However, this would undoubtedly lead to more severe performance overheads, and even worse, SFI-style masking of write instructions would still leave the system vulnerable against attacks through the exception handler (**C3** of §5.4). In addition, the current implementation of SFI-based XOM is vulnerable to unintended instructions. To

defend this, it should eliminate all exploitable unintended instructions either by using the instruction replacement technique similar to *u*XOM or selectively aligning 32-bit instructions so that jump into the middle of those instructions can be prevented by the masking of indirect jump addresses. Either way, additional performance overhead will be unavoidable.

*u*XOM is also more flexible in placing the code and data. For *u*XOM, the XOM region can be placed anywhere in the address space. For example, *u*XOM can be applied for the code placed in SRAM for performance or firmware updates [49]. Also, *u*XOM can set multiple XOM regions as long as the number of MPU regions supports it. However, SFI-based XOM must place the code at one end and the data on the other to simplify code instrumentation. Moreover, SFI-based XOM needs a guardzone between the code and the data region [106] which further restricts the code and data placement and also causes the memory to be wasted for the guardzone.

5.6.5 Use Cases

*u*XOM can be used to hide sensitive information in the code region, such as secret keys and code layout. We describe two use cases to illustrate how *u*XOM can be applied to a security solution.

Secret key protection. In tiny devices, secret keys are frequently used for various purposes, such as device authentication and communication channel protection. *u*XOM can protect these keys against arbitrary memory read vulnerabilities by embedding them inside the code. For example, consider the following code that defines the constant global `key`.

```
const unsigned char key[32] =
    {0xcb, 0x21, 0xad, 0x38, ...};
```

The code that reads the first 4-byte of this value is compiled to the assembly code composed of `MOVW` and `MOVT` as follows:

```
MOVW r0, #0x21cb
MOVT r0, #0x38ad
```

Now, if we use *u*XOM to apply the XO permission to this code, attackers cannot access the key value by arbitrary memory reads. As an example, we applied *u*XOM to `rijndael` benchmark, which uses a symmetric key for encryption. By declaring the key as a global constant, we could confirm that the key is embedded in the code protected by *u*XOM. Such a protection offered by *u*XOM can further be combined with in-register computation techniques [64] for a secure computation robust against memory vulnerabilities.

CRA defense. To date, many researchers have proposed code diversification-based CRA defense techniques [18,28,29,79]. They randomize code layout to prevent attackers from using the existing gadgets for CRA. As the code disclosure attack emerged as a serious threat to randomization-based defenses, XOM has been proposed as an effective solution to fortify these defenses.

As another use case of *u*XOM, we implemented a CRA defense solution based on Readactor [29], which is a representative code diversification based CRA defense with resistance to code disclosure attacks. Readactor aims to defend against two classes of code disclosure attacks: direct disclosure where the attackers disclose code layout by directly reading the code and indirect disclosure where attackers indirectly infer the code layout through the value of the code pointers. Readactor first places all code in XOM to prevent the direct disclosure attacks. It then replaces all code pointers with pointers to *trampolines* so that all indirect control transfers must go through the trampoline. In this way, code pointers containing the original code location are never stored in a register or memory, thereby preventing the indirect disclosure attacks. To demonstrate this use case, we implemented function reordering and the trampoline mechanism. Every function call is replaced with a direct branch to the trampoline followed by the call to the original function. When the original function returns, another direct

branch takes the control flow back to the original callsite. Also, every function pointer is replaced with a pointer to the corresponding function trampoline. We implemented this use case on top of *u*XOM-$\overline{\text{UI}}$ because the code diversification based CRA defense mitigates control flow hijacking, and consequently hinders an attacker from exploiting unintended instructions. The experimental results of our CRA defense are presented together with the results for *u*XOM, *u*XOM-$\overline{\text{UI}}$ and SFI-based XOM. It imposes average runtime overhead of 8.6%, the code size overhead of 19.3%, and the energy overhead of 9.7%. The runtime overhead is only slightly larger than that for original Readactor implementation (6.4%) which shows the applicability of *u*XOM technique in low-end embedded devices.

5.7 Discussion

Cortex-M Processors based on ARMv8-M Architecture. ARMv8-M [48] is a recently introduced instruction set architecture for the microcontroller profile. Basically, ARMv8-M provides backward compatibility with ARMv7-M, so *u*XOM is also applicable to ARMv8-M based Cortex-M(23/33/35) processors. Here, we list several possible changes in *u*XOM implementation due to the newly added hardware feature in ARMv8-M. First of all, ARMv8-M includes the stack pointer limit register (`SPLR`) that defines a lower limit for the stack pointer and prevents the stack pointer from pointing below the limit. When enabling `SPLR`, therefore, *u*XOM only needs to ensure that the stack pointer does not point to the PPB region. Secondly, load-acquire and store-release memory instructions are newly added in ARMv8-M. Since these instructions do not have unprivileged counterparts, they should be protected by the atomic verification technique.

False Positive Conversion. When it comes to the instruction conversion of *u*XOM, false positive cases could happen where unconvertible instructions are converted to unprivileged ones. The false positive conversion does not harm the security aspect

of *u*XOM but may cause an unexpected system fault. For instance, if PPB-accessing memory instructions are converted to unprivileged ones, it would not expose the PPB to attackers but raise a memory access fault when executed. To avoid an unexpected system halt due to the fault, uXOM can install a custom fault handler, which in turn may invoke the fail-safe handler already implemented in the existing system (e.g., emergency landing in drones).

Dynamic Data Protection. Although the current *u*XOM implementation aims to defeat the code disclosure attacks, it may be extended to provide protection for the dynamic data as well. To be concrete, *u*XOM can be expanded to implement a data isolation scheme [54,92,93] that minimizes the possibility of exposures of critical data by only allowing access through authorized instructions. More specifically, we may allow only authorized instructions (i.e., ordinary loads/stores that are not converted into unprivileged types) to access critical data (e.g., return addresses/session keys) by placing the data on a certain memory region marked as "privileged". To implement such an extension, some modifications to *u*XOM are required. First of all, authorized instructions should be predetermined through the help of programmers or compilers and prevented from being converted to unprivileged ones. Since attackers can exploit these data-accessing instructions to compromise *u*XOM, usage of these instructions should be regulated in a way similar to PPB-accessing instructions through the atomic verification technique with a new verification routine that confines memory access target to the memory region of the critical data.

5.8 Related Work

Hardware-assisted Execute Only Memory. Due to the compelling security guarantee provided by XOM, today's high-end processor architectures (e.g., x64 and AArch64) provide the XO permission setting in the MMU [19,24]. Apart from that, various works have attempted to implement XOM in the system with the help of the hardware. David

et al. [61] implemented XOM by encrypting the code in memory and decrypting it only when it is executed. However, since it requires significant processor redesign, it is not suitable for wide adoption. In subsequent works, XOM has been implemented by capitalizing on the built-in hardware features. Shadow Walker [95] and HideM [42] presented an implementation of XOM using the split translation lookaside buffer (TLB) architecture, which separates the TLB for instruction fetches and data accesses. They configure the two TLBs so that the same virtual address is translated into different physical addresses for data access and instruction fetch, preventing the data accesses to the code region. XOM-switch [63] implemented XOM using Intel Memory Protection Keys (MPK), which can be used to set memory pages execute-only. Shadow Walker, HideM and XOM-switch are not applicable to Cortex-M based devices because they rely on specific hardware features (i.e., split-TLB or Intel MPK) that do not exist in the Cortex-M processor.

Software-based Execute Only Memory. On the other hand, there have been attempts to emulate XOM in software for processors that do not have the above hardware supports. XnR [11] sets all code pages as non-accessible except for the currently executed code pages called *sliding window* and detects illegal memory reads and writes for non-accessible pages by augmenting the MMU page fault handler. For Cortex-M/R processors, since MPU also provides non-accessible permission setting for memory regions, XOM can be implemented in a similar way. However, this approach cannot detect memory reads for code pages in the sliding window, and also, the performance overhead becomes larger as the sliding window size is reduced. LR^2 [18] and kRˆX [80] realize XOM by SFI-inspired techniques [106, 113]. They prevent code reads by masking load instructions, instead of stores as done in the SFI technique. As shown in our evaluation, however, such SFI-based XOM implementation can be bypassed and is inefficient in low-end devices.

Security Solutions using XOM. Many researchers have proposed various security

solutions based on XOM. Early works [70] proposed XOM for the purpose of protecting intellectual properties and preventing tampering or leakage of sensitive information stored in the code. Since the advent of code disclosure attacks (i.e., JIT-ROP), a number of works [18, 29, 41, 80] have utilized XOM to prevent the attackers from reading code to learn code layout and launch CRAs. In §5.6.5, we have shown that these solutions can be implemented with *u*XOM.

Security for Tiny Embedded Devices. Recently, much research has been done on enhancing the security of tiny embedded devices. Mbed uvisor [6], MINION [51], uSFI [9] and ACES [27] proposed memory isolation techniques for software modules based on MPU. At compile time, they define memory views (stack, heap, and peripherals) for each of the software modules, and at runtime, MPU enforces one of the memory views according to the active software module. Epoxy [28] and AVRAND [79] developed diversification based security solutions for tiny embedded devices. As with these solutions, *u*XOM also seeks to enhance the security of tiny devices. *u*XOM is the first to implement efficient execute-only memory in Cortex-M processors.

5.9 Summary

XOM is a prominent protection mechanism that can be used in various security purposes such as intellectual property protection and CRA defense. However, for a low-end embedded processor such as Cortex-M, there has been no efficient way to implement XOM. In this paper, we present *u*XOM, a novel technique to realize XOM in a way that is secure and highly optimized to work on Cortex-M processors. *u*XOM achieves this by leveraging hardware features (i.e., unprivileged load/store instructions and MPU) in Cortex-M processors. Our evaluation shows that not only *u*XOM is more efficient than SFI-based XOM in terms of execution time, code size and energy consumption, and that *u*XOM is compatible with existing XOM-based security solutions.

Chapter 6

Conclusion and Future Work

In this thesis, I have introduced three code transformation techniques for achieving different security goals for the protection of computer memory. First, I proposed a compiler technique to insert special push/pop instructions that manage t_{PC} stack to track implicit information flows through conditional branches. By using special hardware instructions, our technique can efficiently keep track of implicit flows, which could only be done with significant performance overhead with previous software approaches. With careful analysis of the control flow graph, our technique can correctly handle complicated cases involving nested conditional branches and loops. Second, I proposed CRCount, which is a compiler based technique to mitigate use-after-free errors in legacy C/C++ programs by automatically keeping track of reference counts for the heap objects. In order to accurately keep track of generation and deletion of pointers, which is essential for reference counting mechanism, CRCount uses pointer footprinting to track the location of pointers. To minimize performance overhead, CRCount carefully analyzes the program to instrument reference count managing code only in the places where it is required. CRCount shows reasonable performance/memory overhead across single and multi-threaded benchmarks, compared to previous works which show significant overhead in either performance or memory. The increased memory

consumption due to delayed free is negligible for the most benchmarks. Finally, I proposed uXOM, a code transformation technique to enable execute-only memory in ARM Cortex-M processors. uXOM converts all load/store instructions to special unprivileged load/store instructions and runs the code in privileged mode so that code region can be executed but cannot be read. By using compiler analysis, uXOM identifies the load/store instructions which should not be converted. To prevent the unconverted instructions from exploited by the attackers, uXOM adds verification code in front of these instructions and transforms the code to use specific registers so that the verification code cannot be bypassed. uXOM efficiently implements XOM for ARM Cortex-M processors utilizing unprivileged load/store instructions, compared to the existing software fault isolation based technique. Overall, the code transformation techniques presented in this thesis improves the existing art for achieving each of the security objects.

6.1 Future Work

In this section I will discuss some possible future research directions. One of the problems of current implicit information flow tracking technique is when implicit information flow causes to many data to be tagged. Although the case shown in the security evaluation shows that only the memory locations with sensitive data are tagged, there is a chance that too many data that is control dependent on the sensitive value is tagged and propagated to large region of memory. Although, strictly speaking, we can say that there is an implicit flow, it is worth examining how much of a help the data leaked that way is for the attacker. In other approach, a debugging tool can be developed that notifies the developers of possible implicit flows in the program and help rewrite the code to get rid of them. Another interesting research direction is how to handle other kinds of implicit information flows such as those generated from side channels. One example of the existing research in that direction is constant time code generation approaches

to prevent leakage through timing channels [102].

For the use-after-free prevention, one of the remaining issues is how to deal with custom memory allocators. Many programs use their own memory allocator which allocates a large chunk of memory using the standard allocator and separately manages the given memory area using the special allocator. CRCount and most other use-after-free mitigations cannot deal with these custom allocators. It is worth exploring the potential threat of custom memory allocator-based use-after-free vulnerabilities and how to detect and handle custom memory allocators automatically. Second, research is needed to find a way to boost the performance even further, for example by reducing the precision of reference counting and use garbage collection to make up for the lost precision or by introducing some optimization for pointers stored in stack. Third, it is worthwhile to develop a tool to guide developers about the locations where they can nullify the pointers in order to avoid possibility of use-after-free errors or to reduce memory consumption due to delayed frees in CRCount. Lastly, CRCount could be combined with Intel MPX to provide protection against out-of-bounds memory access in addition to the use-after-free errors. Intel MPX uses a bounds table which can be used to figure out the locations of pointers. It will be interesting to see if CRCount can utilize the table as an optimization.

For uXOM, one of the issue is that we need to compile and run binary analysis several times to get rid of exploitable unintended instructions. Since inserting code at the binary level messes up the relative offsets for branches and calls, we need a binary instrumentation tool that can completely understand the relative offsets and fix them correctly. Second, uXOM is only interested in protecting secrets stored in the code region. Research is needed to protect against data-only attacks targeted for the low-end embedded devices efficiently. Randomization based and memory access control based approaches [28, 51] have been proposed but research is still needed to improve security and performance.

Bibliography

[1] the heartbleed bug. http://heartbleed.com/.

[2] Periklis Akritidis. Cling: A memory allocator to mitigate dangling pointers. In *USENIX Security Symposium*, pages 177–192, 2010.

[3] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing memory error exploits with wit. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 263–277. IEEE, 2008.

[4] Andrei Alexandrescu. *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley, 2001.

[5] Arduino. arduino-due. https://store.arduino.cc/usa/arduino-due.

[6] ARM. The mebed os uvisor. https://www.mbed.com/en/technologies/security/uvisor/.

[7] ARM. *Embedded Trace Macrocell Architecture Specification*, 2011.

[8] Atmel. Atmel-11057c-atarm-sam3x-sam3a-datasheet, 2015.

[9] Zelalem Birhanu Aweke and Todd Austin. usfi: Ultra-lightweight software fault isolation for iot-class devices. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1015–1020. IEEE, 2018.

[10] Emmanuel Baccelli, Oliver Hahm, Mesut Gunes, Matthias Wahlisch, and Thomas C Schmidt. Riot os: Towards an os for the internet of things. In *Computer Communications Workshops (INFOCOM WKSHPS), 2013 IEEE Conference on*, pages 79–80. IEEE, 2013.

[11] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. You can run but you can't read: Preventing disclosure exploits in executable code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1342–1353. ACM, 2014.

[12] Michael Backes and Stefan Nürnberger. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 433–447, 2014.

[13] Brian Belleville, Hyungon Moon, Jangseop Shin, Dongil Hwang, Joseph M Nash, Seonhwa Jung, Yeoul Na, Stijn Volckaert, Per Larsen, Yunheung Paek, et al. Hardware assisted randomization of data. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 337–358. Springer, 2018.

[14] Emery D Berger and Benjamin G Zorn. Diehard: probabilistic memory safety for unsafe languages. In *Acm sigplan notices*, volume 41, pages 158–168. ACM, 2006.

[15] Sandeep Bhatkar and R Sekar. Data space randomization. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 1–22. Springer, 2008.

[16] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceed-*

ings of the 17th international conference on Parallel architectures and compilation techniques, pages 72–81. ACM, 2008.

[17] Hans Boehm, Alan Demers, and Mark Weiser. A garbage collector for c and c++, 2002.

[18] Kjell Braden, Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Stephen Crane, Michael Franz, and Per Larsen. Leakage-resilient layout randomization for mobile devices. In *NDSS*, 2016.

[19] Scott Brookes, Robert Denz, Martin Osterloh, and Stephen Taylor. Exoshim: Preventing memory disclosure using execute-only kernel code. In *Proceedings of the 11th International Conference on Cyber Warfare and Security*, pages 56–66, 2016.

[20] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. 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*, pages 133–143. ACM, 2012.

[21] Cristian Cadar, Periklis Akritidis, Manuel Costa, Jean-Phillipe Martin, and Miguel Castro. Data randomization. Technical report, Technical Report TR-2008-120, Microsoft Research, 2008. Cited on, 2008.

[22] Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive mutational fuzzing. In *2015 IEEE Symposium on Security and Privacy*, pages 725–741. IEEE, 2015.

[23] Xi Chen, Robert P Dick, and Alok Choudhary. Operating system controlled processor-memory bus encryption. In *Design, Automation and Test in Europe, 2008. DATE'08*, pages 1154–1159. IEEE, 2008.

[24] Yaohui Chen, Dongli Zhang, Ruowen Wang, Rui Qiao, Ahmed M Azab, Long Lu, Hayawardh Vijayakumar, and Wenbo Shen. Norax: Enabling execute-only memory for cots binaries on aarch64. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 304–319. IEEE, 2017.

[25] Yu-Yuan Chen. *Architecture for data-centric security*. PhD thesis, Citeseer, 2012.

[26] Yu-Yuan Chen, Pramod A Jamkhedkar, and Ruby B Lee. A software-hardware architecture for self-protecting data. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 14–27. ACM, 2012.

[27] Abraham A Clements, Naif Saleh Almakhdhub, Saurabh Bagchi, and Mathias Payer. Aces: Automatic compartments for embedded systems. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 65–82, 2018.

[28] Abraham A Clements, Naif Saleh Almakhdhub, Khaled S Saab, Prashast Srivastava, Jinkyu Koo, Saurabh Bagchi, and Mathias Payer. Protecting bare-metal embedded systems with privilege overlays. In *Security and Privacy (SP), 2017 IEEE Symposium on*, pages 289–303. IEEE, 2017.

[29] Stephen Crane, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz. Readactor: Practical code randomization resilient to memory disclosure. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 763–780. IEEE, 2015.

[30] Michael Dalton, Hari Kannan, and Christos Kozyrakis. Raksha: a flexible information flow architecture for software security. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 482–493. ACM, 2007.

[31] Thurston HY Dang, Petros Maniatis, and David Wagner. Oscar: A practical page-permissions-based scheme for thwarting dangling pointers. In *26th*

{*USENIX*} *Security Symposium ({USENIX} Security 17)*, pages 815–832, 2017.

[32] Mark Daniel, Jake Honoroff, and Charlie Miller. Engineering heap overflow exploits with javascript. *WOOT*, 8:1–6, 2008.

[33] Lucas Davi, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z Snow, and Fabian Monrose. Isomeron: Code randomization resilient to (just-in-time) return-oriented programming. In *NDSS*, 2015.

[34] Drew Davidson, Benjamin Moench, Thomas Ristenpart, and Somesh Jha. Fie on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *USENIX Security Symposium*, pages 463–478, 2013.

[35] Prachi Deshpande, Subhash Chander Sharma, Sateesh K Peddoju, and S Junaid. Hids: A host based intrusion detection system for cloud computing environment. *International Journal of System Assurance Engineering and Management*, 9(3):567–576, 2018.

[36] Manuel Egele, Christopher Kruegel, Engin Kirda, Heng Yin, and Dawn Xiaodong Song. Dynamic spyware analysis. In *USENIX annual technical conference*, pages 233–246, 2007.

[37] Úlfar Erlingsson. The inlined reference monitor approach to security policy enforcement. Technical report, Cornell University, 2003.

[38] David Gay, Rob Ennals, and Eric Brewer. Safe manual memory management. In *Proceedings of the 6th international symposium on Memory management*, pages 2–14. ACM, 2007.

[39] David Gens, Simon Schmitt, Lucas Davi, and Ahmad-Reza Sadeghi. K-miner: Uncovering memory corruption in linux. In *NDSS*, 2018.

[40] Sanjay Ghemawat and Paul Menage. Tcmalloc: Thread-caching malloc, 2007. *URL {http://goog-perftools. sourceforge. net/doc/tcmalloc. html}*, 2005.

[41] Jason Gionta, William Enck, and Per Larsen. Preventing kernel code-reuse attacks through disclosure resistant code diversification. In *Communications and Network Security (CNS), 2016 IEEE Conference on*, pages 189–197. IEEE, 2016.

[42] Jason Gionta, William Enck, and Peng Ning. Hidem: Protecting the contents of userspace memory in the face of disclosure vulnerabilities. In *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*, pages 325–336. ACM, 2015.

[43] Istvan Haller, Erik Van Der Kouwe, Cristiano Giuffrida, and Herbert Bos. Metalloc: Efficient and comprehensive metadata management for software security hardening. In *Proceedings of the 9th European Workshop on System Security*, page 5. ACM, 2016.

[44] John L Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[45] Matthew Hertz and Emery D Berger. Quantifying the performance of garbage collection vs. explicit memory management. In *ACM SIGPLAN Notices*, volume 40, pages 313–326. ACM, 2005.

[46] Martin Hirzel and Amer Diwan. On the type accuracy of garbage collection. *ACM SIGPLAN Notices*, 36(1):1–11, 2001.

[47] ARM Holdings. Armv7-m architecture reference manual, 2010.

[48] ARM Holdings. Armv8-m architecture reference manual, 2017.

[49] IAR. Execute in ram after copying from flash or rom. https://www.iar.com/support/tech-notes/general/execute-in-ram-after-copying-from-flashrom-v5.20-and-later/.

[50] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. Dta++: Dynamic taint analysis with targeted control-flow propagation. In *NDSS*, 2011.

[51] Chung Hwan Kim, Taegyu Kim, Hongjun Choi, Zhongshu Gu, Byoungyoung Lee, Xiangyu Zhang, and Dongyan Xu. Securing real-time microcontroller systems through customized memory view switching. In *Network and Distributed Systems Security Symp.(NDSS)*, 2018.

[52] Oliver Kömmerling and Markus G Kuhn. Design principles for tamper-resistant smartcard processors. *Smartcard*, 99:9–20, 1999.

[53] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. No need to hide: Protecting safe regions on commodity hardware. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 437–452. ACM, 2017.

[54] Volodymyr Kuznetsov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. Code-pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 147–163, 2014.

[55] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. Sok: Automated software diversity. In *2014 IEEE Symposium on Security and Privacy (SP)*, pages 276–291. IEEE, 2014.

[56] Roland Büschkes Pavel Laskov. Detection of intrusions and malware & vulnerability assessment. 2006.

[57] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. Preventing use-after-free with dangling pointers nullification. In *NDSS*, 2015.

[58] Jinyong Lee, Ingoo Heo, Yongje Lee, and Yunheung Paek. Efficient dynamic information flow tracking on a processor with core debug interface. In *Proceedings of the 52nd Annual Design Automation Conference*, page 79. ACM, 2015.

[59] Jinyong Lee, Yongje Lee, Hyungon Moon, Ingoo Heo, and Yunheung Paek. Extrax: Security extention to extract cache resident information for snoop-based external monitors. In *Design Automation and Test in Europe Conference and Exhibition (DATE)*, 2015.

[60] Keith Lee. Memory management. In *Pro Objective-C*, pages 53–74. Springer, 2013.

[61] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. *ACM SIGPLAN Notices*, 35(11):168–177, 2000.

[62] Alfredo Mazzinghi, Ripduman Sohan, and Robert NM Watson. Pointer provenance in a capability architecture. In *10th {USENIX} Workshop on the Theory and Practice of Provenance (TaPP 2018)*, 2018.

[63] Ravi Sahita Mingwei Zhang and Daiping Liu. executable-only-memory-switch (xom-switch). *Black Hat Asia*, 2018.

[64] Tilo Müller, Felix C Freiling, and Andreas Dewald. Tresor runs encryption securely outside ram. In *USENIX Security Symposium*, volume 17, 2011.

[65] S Nagaraju, Cristian Craioveanu, Elia Florio, and Matt Miller. Software vulnerability exploitation trends. *Microsoft Corporation*, 2013.

[66] Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. Watchdog: Hardware for safe and secure manual memory management and full memory safety. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 189–200. IEEE Computer Society, 2012.

[67] Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. Watchdoglite: Hardware-accelerated compiler-based pointer checking. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, page 175. ACM, 2014.

[68] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. *ACM Sigplan Notices*, 44(6):245–258, 2009.

[69] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. Cets: compiler enforced temporal safety for c. In *ACM Sigplan Notices*, volume 45, pages 31–40. ACM, 2010.

[70] Gleb Naumovich and Nasir Memon. Preventing piracy, reverse engineering, and tampering. *Computer*, 36(7):64–71, 2003.

[71] Nicholas Nethercote and Julian Seward. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd international conference on Virtual execution environments*, pages 65–74. ACM, 2007.

[72] James Newsome and Dawn Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. 2005.

[73] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS*, volume 5, pages 3–4. Citeseer, 2005.

[74] Gor V Nishanov and Sibylle Schupp. Garbage collection in generic libraries. *ACM SIGPLAN Notices*, 34(3):86–96, 1999.

[75] Gene Novark and Emery D Berger. Dieharder: securing the heap. In *Proceedings of the 17th ACM conference on Computer and communications security*, pages 573–584. ACM, 2010.

[76] ODROID. smart-power. `https://wiki.odroid.com/old_product/accessory/odroidsmartpower`.

[77] Mikael Olsson. Smart pointers. In *C++ 17 Quick Syntax Reference*, pages 157–160. Springer, 2018.

[78] James Pallister, Simon Hollis, and Jeremy Bennett. Beebs: Open benchmarks for energy measurements on embedded platforms. *arXiv preprint arXiv:1308.5174*, 2013.

[79] Sergio Pastrana, Juan Tapiador, Guillermo Suarez-Tangil, and Pedro Peris-López. Avrand: a software-based defense against code reuse attacks for avr embedded devices. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 58–77. Springer, 2016.

[80] Marios Pomonis, Theofilos Petsios, Angelos D Keromytis, Michalis Polychronakis, and Vasileios P Kemerlis. krˆx: Comprehensive kernel protection against just-in-time code reuse. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 420–436. ACM, 2017.

[81] Feng Qin, Cheng Wang, Zhenmin Li, Ho-seop Kim, Yuanyuan Zhou, and Youfeng Wu. Lift: A low-overhead practical information flow tracking sys-

tem for detecting security attacks. In *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, pages 135–148. IEEE, 2006.

[82] Radare2. unix-like reverse engineering framework and commandline tools. `https://www.radare.org/r/`.

[83] Jon Rafkind, Adam Wick, John Regehr, and Matthew Flatt. Precise garbage collection for c. In *Proceedings of the 2009 international symposium on Memory management*, pages 39–48. ACM, 2009.

[84] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. In *NDSS*, volume 17, pages 1–14, 2017.

[85] Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *Selected Areas in Communications, IEEE Journal on*, 21(1):5–19, 2003.

[86] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting software fault isolation to contemporary cpu architectures. In *USENIX Security Symposium*, pages 1–12, 2010.

[87] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*, pages 309–318, 2012.

[88] Sam Silvestro, Hongyu Liu, Corey Crosser, Zhiqiang Lin, and Tongping Liu. Freeguard: A faster secure heap allocator. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2389–2403. ACM, 2017.

[89] Matthew S Simpson and Rajeev K Barua. Memsafe: ensuring the spatial and temporal memory safety of c at runtime. *Software: Practice and Experience*, 43(1):93–128, 2013.

[90] Kevin Z Snow, Fabian Monrose, Lucas Davi, Alexandra Dmitrienko, Christopher Liebchen, and Ahmad-Reza Sadeghi. Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 574–588. IEEE, 2013.

[91] Patrick Sobalvarro. A lifetime-based garbage collector for lisp systems on general-purpose computers. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB, 1988.

[92] Chengyu Song, Byoungyoung Lee, Kangjie Lu, William Harris, Taesoo Kim, and Wenke Lee. Enforcing kernel security invariants with data flow integrity. In *NDSS*, 2016.

[93] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. Hdfi: Hardware-assisted data-flow isolation. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 1–17. IEEE, 2016.

[94] Alexander Sotirov. Heap feng shui in javascript. *Black Hat Europe*, 2007.

[95] Sherri Sparks and Jamie Butler. Shadow walker: Raising the bar for rootkit detection. *Black Hat Japan*, 11(63):504–533, 2005.

[96] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*, volume 16, pages 1–16, 2016.

[97] G Edward Suh, Jae W Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *ACM SIGOPS Operating Systems Review*, volume 38, pages 85–96. ACM, 2004.

[98] Menasveta Tim, Soubra Diya, and Yiu Joseph. Introducing arm cortex-m23 and cortex-m33 processors with trustzone for armv8-m, 2016.

[99] Mohit Tiwari, Xun Li, Hassan MG Wassel, Frederic T Chong, and Timothy Sherwood. Execution leases: A hardware-supported mechanism for enforcing strong non-interference. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 493–504. ACM, 2009.

[100] Mohit Tiwari, Hassan MG Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T Chong, and Timothy Sherwood. Complete information flow tracking from the gates up. In *ACM Sigplan Notices*, volume 44, pages 109–120. ACM, 2009.

[101] Neil Vachharajani, Matthew J Bridges, Jonathan Chang, Ram Rangan, Guilherme Ottoni, Jason A Blome, George A Reis, Manish Vachharajani, and David I August. Rifle: An architectural framework for user-centric information-flow security. In *Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on*, pages 243–254. IEEE, 2004.

[102] Jeroen Van Cleemput, Bjorn De Sutter, and Koen De Bosschere. Adaptive compiler strategies for mitigating timing side channel attacks. *IEEE Transactions on Dependable and Secure Computing*, 2017.

[103] Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. Dangsan: Scalable use-after-free detection. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 405–419. ACM, 2017.

[104] Victor Van Der Veen, Enes Göktas, Moritz Contag, Andre Pawoloski, Xi Chen, Sanjay Rawat, Herbert Bos, Thorsten Holz, Elias Athanasopoulos, and Cristiano Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 934–953. IEEE, 2016.

[105] Stijn Volckaert, Bart Coppens, Alexios Voulimeneas, Andrei Homescu, Per Larsen, Bjorn De Sutter, and Michael Franz. Secure and efficient application monitoring and replication. In *2016 {USENIX} Annual Technical Conference ({USENIX}{ATC} 16)*, pages 167–179, 2016.

[106] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, 1994.

[107] David Williams-King, Graham Gobieski, Kent Williams-King, James P Blake, Xinhao Yuan, Patrick Colp, Michelle Zheng, Vasileios P Kemerlis, Junfeng Yang, and William Aiello. Shuffler: Fast and deployable continuous code re-randomization. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 367–382, 2016.

[108] Paul R Wilson. Uniprocessor garbage collection techniques. In *Memory Management*, pages 1–42. Springer, 1992.

[109] I Xilinx. Microblaze processor reference guide v13. 4. *reference manual*, 2011.

[110] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 414–425. ACM, 2015.

[111] Toshihiro Yamauchi and Yuta Ikegami. Heaprevolver: Delaying and randomizing timing of release of freed memory area to prevent use-after-free attacks. In *International Conference on Network and System Security*, pages 219–234. Springer, 2016.

[112] Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. Spatio-temporal context reduction: A pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 327–337. IEEE, 2018.

[113] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *2009 30th IEEE Symposium on Security and Privacy*, pages 79–93. IEEE, 2009.

[114] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 116–127. ACM, 2007.

[115] Yves Younan. Freesentry: protecting against use-after-free vulnerabilities due to dangling pointers. In *NDSS*, 2015.

[116] Mingwei Zhang and R Sekar. Control flow integrity for {COTS} binaries. In *Presented as part of the 22nd {USENIX} Security Symposium ({USENIX} Security 13)*, pages 337–352, 2013.

[117] Yu Zhu, Jaeyeon Jung, Dawn Song, Tadayoshi Kohno, and David Wetherall. *Privacy Scope: A precise information flow tracking system for finding application leaks*. PhD thesis, University of California, Berkeley, 2009.

초 록

컴퓨터 메모리는 컴퓨터 시스템의 보안을 위해 보호되어야 하는 중요한 컴포넌트이다. 컴퓨터 메모리는 보안상 중요한 데이터를 담고 있을 뿐만 아니라, 시스템의 올바른 동작을 위해 공격자에 의해 조작되어서는 안되는 중요한 데이터 값들을 저장한다. 따라서 많은 보안 솔루션은 메모리를 보호하여 컴퓨터 시스템에서 중요한 데이터가 유출되거나 컴퓨터 데이터에 대한 불법적인 접근을 방지하는 데 중점을 둔다. 본 논문에서는 메모리 보호를 위한 보안 정책을 시행하기 위한 다양한 코드 변환 기술을 제시한다. 먼저, 프로그램에서 분기문을 통해 보안에 민감한 데이터가 유출되지 않도록 암시적 데이터 흐름을 추적하는 코드 변환 기술을 제시한다. 그 다음으로 C / C ++ 프로그램을 변환하여 use-after-free 오류를 완화하는 컴파일러 기술을 제시한다. 마지막으로, 중요 데이터를 보호하고 코드 재사용 공격으로부터 디바이스를 강화할 수 있는 강력한 보안 정책인 실행 전용 메모리(execute-only memory)를 저사양 임베디드 디바이스에 구현하기 위한 코드 변환 기술을 제시한다.

주요어: 컴퓨터 보안, 메모리 보호, 코드 변환
학번: 2013-20813

ACKNOWLEGEMENT

우선 논문 작성을 지도해 주셔서 결과적으로 좋은 학회에 실을 수 있도록 도와주신 백윤흥 교수님께 감사를 드립니다. 그리고 박사과정을 핑계로 많이 신경써 드리지 못했는데도 물심양면으로 지원해주신 부모님께 감사드립니다. 부모님께서 연구에 집중할 수 있도록 배려해 주시고 매 주말마다 자취생활로 건강이 나빠질 것을 우려해 먹을것들을 챙겨주시는 덕분에 무사히 졸업까지 올 수 있었던 것 같습니다. 긴 연구실 생활 동안 함께한 동료들에게도 감사의 말을 전합니다. 선후배들과 함께 디스커션하며 많은 것을 배울 수 있었습니다. 특히 제가 박사과정동안 참여하였던 논문 및 과제들에 함께했던 분들께 감사드립니다. 그 어떤 논문이나 과제도 혼자서는 순조로이 완성짓는 것이 불가능하였을 것입니다. 마지막으로 기나긴 연구실 생활 동안 힘들고 지칠때에 위로와 힘이 되어준 모든 연구실 및 지인분들께 감사를 드립니다. 7년은 굉장히 긴 시간이었고 그 시간동안 연구라는 것이 즐거울 때도 있었지만 외롭고 좌절감을 느낄때도 많았고 긴 연구에 지쳐버린 때도 있었습니다. 그 때마다 하소연을 들어주는 것만으로도 저에겐 너무나도 큰 힘이 되었습니다. 많은 분들께 받은 사랑을 기억하며 앞으로도 한 사람의 박사로서 겸손하고 치열하게 살아가도록 하겠습니다.