SafeDS: Safe Data Structures for C++

by

Seyedeh Setareh Ghorshi

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2022

Abstract

Memory corruption vulnerabilities in low-level languages such as C/C++ have been a problem in computer security for a long time. Accordingly, there has been a wide variety of proposed solutions for detecting or preventing memory corruption attacks. Due to the constantly evolving nature of such attacks and the importance of achieving high performance in most applications, no comprehensive solution has yet been developed to efficiently secure all data in the program memory and mitigate such attacks. Nevertheless, solutions that only protect critical data in memory provide a balance between security and efficiency that could be practical in many applications.

Accordingly, we introduce *SafeDS*, an extension to the C++ standard library containers that provides secure design and implementation for three frequently-used data structures. We assume a powerful adversary with arbitrary read/write access to the memory but unable to access and modify reserved registers. Data integrity is ensured by SafeDS within the data structures in the presence of such adversary through calculating a Message Authentication Code (MAC) for each element, which can then be used to validate data when reading from the data structure.

Our secure data structures are implemented as a part of the *gcc-11.1.0 C++ Standard Library* and are compatible with both *ARM* and *x86* architectures. We use the Pointer Authentication (PA) hardware extension on ARM-v8 and Intel AES-NI instruction set to calculate MACs on ARM and x86 architectures, respectively.

By testing our prototype against applications that use the data structure APIs in the C++ standard library, such as OpenCV, we show that switching to the secure version of data structures requires minimal changes to the applications' original source code. Our secure data structures use a Merkle tree to securely store one MAC for each instance of them in the program. Therefore, we can theoretically estimate that an overhead of order $O(log(i))$ will be added to the data structure operations, where i is the number of data structure instances in the program. However, since the design for the secure data structures ties the MAC calculation and verification to the normal steps of the operations, the rest of the MAC related operations only add a constant overhead. The performance of our prototype has been evaluated using the provided performance tests in OpenCV, and our results show that the secure data structures introduce an overall overhead of 3.4% compared to the baseline. Furthermore, we present *game-based* proofs to prove the security of our designed data structures against data corruption attacks.

Acknowledgements

I would like to thank all the people who made this thesis possible.

Dedication

This work is dedicated to Pouneh, Arash, and other students on the flight PS752 whose journeys ceased.

Table of Contents

List of Figures

List of Tables

List of Acronyms

AES	Advanced Encryption Standard.
ASLR	Address Space Layout Randomization.
CFG	Control Flow Graph.
CFI	Control-Flow Integrity.
CMAC	Cipher-based Message Authentication Code.
CPI	Code-Pointer Integrity.
DFI	Data-Flow Integrity.
DOP	Data-Oriented Programming.
FIFO	First-In-First-Out.
LBC	Light-weight Bounds Checking.
LIFO	Last-In-First-Out.
MAC	Message Authentication Code.
MPK	Memory Protection Keys.
PA	Pointer Authentication.
PAC	Pointer Authentication Code.
ROP	Return Oriented Programming.
STL	Standard Template Library.
XOM	Execute-only Memory.

Chapter 1

Introduction

Preventing or detecting the attacks that exploit memory errors in C/C++ programs has been a popular topic in security-related research for the past decades. Spatial memory errors allow the attacker to use an out-of-bound pointer to overwrite objects illegitimately, while temporal errors allow using pointers to freed memory addresses to modify or leak newly allocated objects [83]. Exploiting such memory errors provides a way for an attacker to modify the program data as desired to gain control over the program and, for instance, perform privilege escalation attacks. Since many popular applications have been developed in C/C++ over the years, switching to safe programming languages is not a feasible option for the already developed applications. Accordingly, defending against such attacks is a popular research topic among security researchers. Despite the proposed solutions, the attacks corrupting program data remain a crucial threat to the security of C/C++ programs.

Return Oriented Programming (ROP) [70] is an example of such attacks which uses small instruction sequences, called gadgets, in the program code to perform attacker-defined tasks. ROP attacks leverage memory errors to modify control data, more specifically return addresses, and make the program jump to the desired gadgets. Another category of attacks, the Data-Oriented Programming (DOP) [43] attacks, have been known for a long time, but recent research has shown that they can achieve the same expressability as the ROP attacks. These attacks modify the non-control data in the program to achieve a similar goal to the ROP attacks, but they don't alter the program's control flow. Both types of attacks can be mitigated by protecting the corresponding data.

The integrity and the flow of program data (including control and non-control data) can be preserved using solutions categorized as memory error detection, Data-Flow Integrity

(DFI) and Control-Flow Integrity (CFI) methods, data isolation, and data or control plane randomization [83]. Memory error detection defenses attempt to detect the errors before being exploited, which prevents the attacks in the first place, but they might not always be practical due to their high overhead [83]. DFI [24] and CFI [9] approaches use the correct flow of the program to validate the changes to the control or non-control data and prevent unexpected control or data flows. However, none of these approaches provide complete security on their own despite introducing relatively high overhead [83]. Data isolation techniques suffer from the same overhead issue as they separate critical data in memory and validate any access to them. Finally, randomization techniques make the attacks harder by deterring the attackers from finding the desired data but do not sufficiently eliminate the chance of them succeeding in performing the attacks. For instance, Address Space Layout Randomization (ASLR) [67] has been shown to be breakable by CAIN [16] through using the memory page deduplication side-channel to leak the Randomized Base Addresses (RBAs) of libraries that are loaded within user space processes in neighboring VMs.

Accordingly, most of the solutions in the mentioned categories suffer from high overhead or do not provide sufficient security against attacks such as ROP or DOP. One way to reduce the high overhead of such solutions is to limit the scope of protection to a specific type of critical data in the program, such as return addresses in PACStack [49] or code pointers [47]. Limiting the scope of protection to a specific type of data introduces more flexibility when securing a program, as the user (the developer who uses the security solution) can decide which types of data are more vulnerable in their application and apply proper protection mechanisms accordingly.

Data structures provide specific ways of organizing data, so efficient protection of their integrity may require custom designs tailored to their distinct functionality, which could be challenging in some cases. For instance, in a stack or queue data structure, the order in which elements were inserted should be preserved as the elements are read from it. Accordingly, the protection mechanism needs to tie the elements to their position in the data structure. On the other hand, the organizability feature makes them suitable choices to store the access-control or other types of decision-making data in a program. Consequently, data structures will be valuable targets for attackers, making them worthy of being protected against memory corruption attacks.

Accordingly, we present *SafeDS*, an approach for verifying the integrity of elements in data structures through calculating Message Authentication Codes (MACs) over them. *SafeDS* ensures the integrity of elements while stored inside the data structures, meaning that the element that is read from the data structure is identical to what was originally inserted into the data structure. We detect any illegitimate data alteration while inside the data structure. A legitimate modification is changing the elements through the defined operations or using valid pointers to the elements inside the data structures, while any other data alteration, such as by exploiting memory errors, is considered illegitimate. We provide the mentioned promises in the presence of a powerful adversary who has random read/write access to memory but cannot access or modify reserved registers.

To provide the described security guarantees, we calculate a MAC for each element and then bind the MACs into a single MAC (called *top MAC*) for each data structure so that only securing the top MAC is enough to verify all the other MACs and their corresponding element. Furthermore, we use a Merkle tree [80] to store the top MACs for data structures and use a reserved register to securely store the root of the Merkle tree. The use of the reserved register in our design prevents the attacker from replacing MACs with previously seen MACs. This exploit is called a *MAC reuse* attack, and has been an issue in previous work such as CCFI [54].

Finally, we implement our described mechanism and demonstrate its practicality by leveraging Pointer Authentication (PA) [79] on ARM and AES-NI [40] on x86 architectures. PA was introduced as a hardware extension in ARMv8.3-A and was initially used for pointers as the name suggests. PA has been used in various schemes to verify the integrity of pointers or return addresses, as in [50], [34], [30], and [49]. Nevertheless, this feature can also be used for creating MACs over data to verify its integrity. However, the previous work does not cover protecting the integrity of data structures using this feature. The x86 architectures provide a different set of instructions, AES-NI, which allows for performing efficient encryption and decryption operations. This instruction set can be used in a similar manner to ARM PA to calculate MACs over data in x86 architectures.

Our main contributions are as follows:

- A custom secure design for three data structures including **stack**, **queue**, and **red-black tree** which is the underlying container for data structures such as *map*.
- A prototype of the implemented secure data structures (*secure version*) **as an extension to the gcc-11.1.0 C++ standard library containers** which is **compatible with both ARM and x86 architectures**, and requires **minimal change to the program code using the C++ data structures' API** when switching to use the secure version.
- **Game-based security proofs** for our secure designs **proving their resistance to illegitimate data modifications**.

The following chapters are structured as follows: An overview of the background information is presented in Chapter 2, which covers the details about the memory errors, attacks, and defenses, along with more detail on PA and AES-NI instructions. Chapter 3 presents the details of our adversary model and generality, security, and performance requirements. We present the design and implementation details for the secure data structures in Chapter 4 and Chapter 5. Our security proofs are presented in Chapter 6 along with an overview of how our design and implementation satisfy the mentioned requirements from Chapter 3. We describe the limitations of our work along with the path for future work in Chapter 7 and include an overview of the related work in Chapter 8. Finally, we finish with the conclusion in Chapter 9.

Chapter 2

Background

2.1 C++ Programming Language

In this section, some of the related aspects of the C++ are described. For comprehensive documentation, refer to [1].

2.1.1 Data Structures

The C++ Standard Template Library (STL) implements various data structures including stack, queue, map, list, and several other data structures. As the name suggests, these data structures are implemented as generic template classes that support arbitrary data types. Each data structure provides several functions that perform operations such as adding elements to the data structures, reading elements, or removing them. Some of the mentioned functions, such as `top()` in stack, allow the programmer to get a reference to the elements inside the data structures. Hence, although stack functions only allow insertion or removal at the top of the stack, the programmer can use the mentioned reference to access the elements already inside the data structure at any time and potentially modify them.

It is also worth noting that the mentioned implementation of data structures might differ from other definitions of them (e.g., in other programming languages) in some cases. For instance, as the C++ reference [1] describes, the C++ implementation of stack includes two distinct functions `top()` and `pop()`. The `top()` function returns a reference to the top element in the stack without removing the element. On the other hand, the `pop()` function removes the top element from the stack without returning any reference to it. However, the Java definition of a stack [37] assumes that the `pop()` function removes and returns the top element as an output.

In this work, we extend the C++ implementation of data structures in gcc libstdc++ and hence, follow its defined semantics.

2.1.2 Constructors and Assignment Operators

C++ provides various functionalities for objects. One important part of an object is its constructors and assignment operators. A C++ object can have different types of constructors, including copy and move constructors.

In C++, default assignment operators and constructors including copy and move, and deconstructors are automatically generated for a class unless they are explictily defined or deleted [6]. Since the secure data structures require a specific design for these functions, we describe the general properties of each constructor and operator in the following subsections.

Copy Constructor

A copy constructor allows for initializing a new object using another object of a compatible type [2]. Examples of when the copy constructor is called are as follows:

1. Initializing a using b both of which are of the same type T: `T a = b; or T a(b);`
2. Passing an object as a function argument: `f(a);`
3. Returning an object, which has no move constructor, from a function: `return a;`

The copy constructor can be explicitly deleted or defined by the programmer. Otherwise, the compiler will define one implicitly. Similar to the copy constructor, a copy assignment operator initializes an object from another. However, the constructor is called for creating a new object, while the assignment operator is called to assign a new value to an already existing object.

Move Constructor

There are various value categories in C++. In a move constructor, an object is initialized from an *rvalue* with the same type [3]. An *rvalue* is an expression that establishes the identity of an object or function when evaluated, but its resources cannot be reused [1]. Examples of cases that call a move constructor are described below:

1. Initializing a using b both of which are of the same type T: `T a = std::move(b);` `T a(std::move(b));`
2. Passing an object as a function argument: `f(std::move(a));`
3. Returning an object, which has a move constructor, from a function: `return a;`

Similar to the copy constructor, the move constructor can be explicitly deleted or defined by the programmer or the compiler will define it implicitly. Moreover, C++ includes move assignment operators that differ from the move constructors in that they are called on already existing objects, whereas the move constructors are used to initialize new objects.

2.2 Memory Vulnerabilities

Memory corruption vulnerabilities have been known as a serious issue in computer security for a long time. C/C++ languages are among the main programming languages suffering from memory vulnerabilities. Memory corruptions can occur using two types of memory errors. The first type of error is dereferencing a pointer that goes out of the bounds of its pointee object and is called a spatial error. The second type, a temporal error, is dereferencing a pointer that points to a deleted object and is called a dangling pointer [78]. Both out-of-bounds or dangling pointers can be exploited by an attacker for various goals, such as violating the integrity of the internal data or leaking sensitive information.

Various programming bugs allow an attacker to force a pointer to go out of the object's bounds, for instance, a lack of bound checking allows the attacker to index an array out of its bounds or increment the array's pointer until it passes the array's bounds. Similarly, the attacker can create a dangling pointer in different ways, such as using an incorrect exception handler that deallocates an object but does not reinitialize the pointer to it [78].

Buffer overflows are an example of such exploits that allow the attacker to overwrite values in memory. In a buffer overflow attack, the attacker uses out-of-bound access to a buffer to overwrite the objects that are adjacent to it. Figure 2.1 shows how this attack affects memory. *a* and *b* are two adjacent data. If the attacker is able to exploit a bug such as usage of `strcpy()` without bounds check (e.g. `strcpy(a,"OVERFLOW")`), they can write past the bounds of *a* and change the value of *b* as shown in the figure.

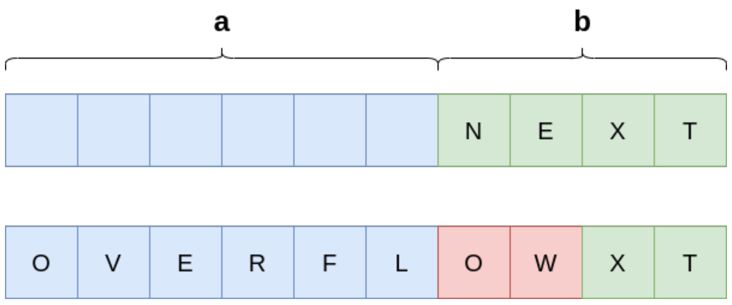

Figure 2.1: Buffer overflow example: using `strcpy()` without proper bound checking (e.g., `strcpy(a,"OVERFLOW")`) can overwrite the adjacent memory (in this case the variable `b`).

Another well-known exploit is a format string attack [62], in which a user-input that is being treated as a command in the program is used to perform malicious tasks. Format functions such as `printf()` are vulnerable to such attacks in cases where their format string specifier is missing. These attacks can be used to create invalid pointers and read data from arbitrary memory locations or write to them. An example of a format string attack that writes to memory is presented below:[78] The %n format string writes the size of the input value to the address pointed by %n. Accordingly, in this code, if the `argv[1]` is for instance `AAA%n`, then the value 3 will be written at the address pointed by %n.

```
int main (int argc, char *argv[]) {
...
printf(argv[1]);
}
```

The buffer overflow and format string attacks take advantage of the spatial errors. However, a temporal error similarly allows an attacker to read from or write to memory. For instance, the attacker can read or corrupt the value of a newly allocated sensitive object in the memory address pointed by a dangling pointer. An example of such an exploit is described below:

```
char* data = (char *)malloc(3);
free (data); //Could accidentally be freed.
/* The allocator might allocate the
same memory previously allocated to data to the
```

```
sensitive value.*/
char* sensitive = (char *)malloc(1);
sensitive[0] = 'N';
 /* Attacker can overwrite the value of sensitive here.
For instance change sensitive[0] from 'N' to 'Y' by inputting
"Yxx" here.*/
scanf("\%s", data);
if (sensitive[0] == 'Y') {
    Do privileged tasks }
```

An attacker can use the above methods to attack a program in different ways. Next, we discuss the two main attack categories that are related to this work.

2.2.1 Control-Flow Hijacking Attacks

In this category of attacks, the goal is to gain control over the program by altering its control flow. An attacker can achieve this goal by modifying code pointers and loading them into the instruction pointer. For example, the attacker can overwrite a return address using either a buffer overflow or one of the previously discussed attacks that allow overwriting memory. In order to successfully complete the attack, the attacker must find the valid address of another target that they might want to make the program jump to. This target could be the address of attacker-specified malicious code or an existing function that performs the attacker's desired instructions.

However, one generally used method to prevent the attacker from injecting and executing malicious code is by enforcing $W \oplus X$(Write XOR Execute) [11] that allows a page to either be writable or executable, but not both. This defense limits the attacker to reusing existing code. Return-to-libc attack [31] uses already existing code (usually libc). Moreover, the approach that attempts to use small instruction sequences is called Return Oriented Programming (ROP) [70].

There are different proposed techniques for preventing control-flow hijacking attacks. Address Space Randomization techniques [67] are known as a mitigation method against this class of attacks, which add randomization to memory addresses and make it harder for the attacker to find the desired target addresses. Regarding ROP attacks, examples of proposed solutions include eliminating gadgets as in [65] and ensuring control flow integrity as in [19].

There have also been solutions for detecting control-flow hijacking attacks. One popular solution is called Control-Flow Integrity (CFI) [9] which enforces policies to verify the indirect control transfers. For instance, CFI suggests performing a check for indirect calls, jumps, and return addresses against a set of valid targets determined based on the correct control flow of the program [78]. Different versions of CFI have been proposed to adjust to the security-efficiency trade-off. CFI policies have two types. The first type of CFI is forward-edge CFI which protects the forward edges in the Control Flow Graph (CFG) of the program such as indirect function calls. The second type is backward-edge CFI which protects the backward edges of the CFG such as returning from a function [69]. Stateful Forward-Edge CFI [87] introduces a mechanism for enforcing forward-edge CFI using Intel Memory Protection Extensions (MPX) [64], and RAGuard [86] is an example of a proposed hardware-assisted mechanism for enforcing backward-edge CFI.

Nevertheless, research shows that even fully-precise static CFI [22] is not completely secure against Control-Flow Bending attacks [22] that are a generalized form of non-control-data attacks. Moreover, in order to be reasonably secure, even fully-precise static CFI requires the use of shadow stacks which adds to the overhead of such solution.

2.2.2 Data-Only Attacks

This class of attacks does not corrupt data explicitly related to the control flow, like code pointers. Instead, they modify data that can indirectly affect the logic of the program and give the attacker more control or higher privilege [78]. A simple case of such an attack is presented below:

```cpp
/*In the below example, a queue data structure is used
to store the privilege level for each user. If an attacker
changes the stored value for a normal user and replaces it
with "ROOT" then they can make the program perform the
privileged operations illegitimately.*/

std::queue<string> access_level_queue;
if (access_level_queue.front() == "ROOT"){
    //Perform privileged operations
}
```

One solution against these attacks is to detect any data corruption before the data is used. Data-Flow Integrity (DFI) [25] proposes a way of performing such checks by comparing the last instruction used to write to a memory location against a set of possible valid instructions based on the program code. DFI can also be used to mitigate control-flow hijacking attacks to some extent since it can ensure that, for instance, a return address has been lastly written by the corresponding call instruction. Similar to CFI, DFI also makes use of a shadow memory to keep track of the writing instructions that wrote to each memory location [78].

However, despite solutions that attempt to prevent or detect memory corruption attacks, new attacks are constantly being introduced that can bypass these solutions. As an example, there are attacks that perform an exploit using intra-object corruption even in the presence of allocation protection defenses [36].

2.3 Message Authentication Code

A Message Authentication Code (MAC) is a checksum that is used to verify the integrity and authenticity of the message for which the MAC is calculated. A MAC algorithm uses a secret key to calculate a MAC over an input message. As long as the secret key and algorithm don't change, the calculated MAC over a specific message will always be the same [51]. Therefore, by calculating a MAC over a message again and comparing it to a previously calculated MAC over the same message, it is possible to verify that the message is the same as it was when the first MAC was generated. A Cipher-based Message Authentication Code (CMAC) is an example of a MAC algorithm based on a symmetric key block cipher [33].

2.4 Merkle Tree

A Merkle tree is a type of tree data structure in which each non-leaf element (all elements except the ones at the lowest level) is the result of calculating a hash over the concatenation of its left and right child elements [80]. The structure of a Merkle tree is shown in Figure 2.2.

A Merkle tree is practical in applications where data integrity verification is required since it allows verifying the elements by recalculating the hashes up to the root and checking if the newly calculated root matches the root of the Merkle tree at some previous time. The key property of this structure that makes the verification process efficient is the fact that based on the architecture of the Merkle tree, in order to verify an element, we only require a small part of the Merkle tree, called a Merkle proof, to recalculate the hashes up to the root. Accordingly, since the verification requires recalculating one hash per level, the overhead of element verification in a Merkle tree is of order $O(log(i))$ where i is the number of leaf elements.

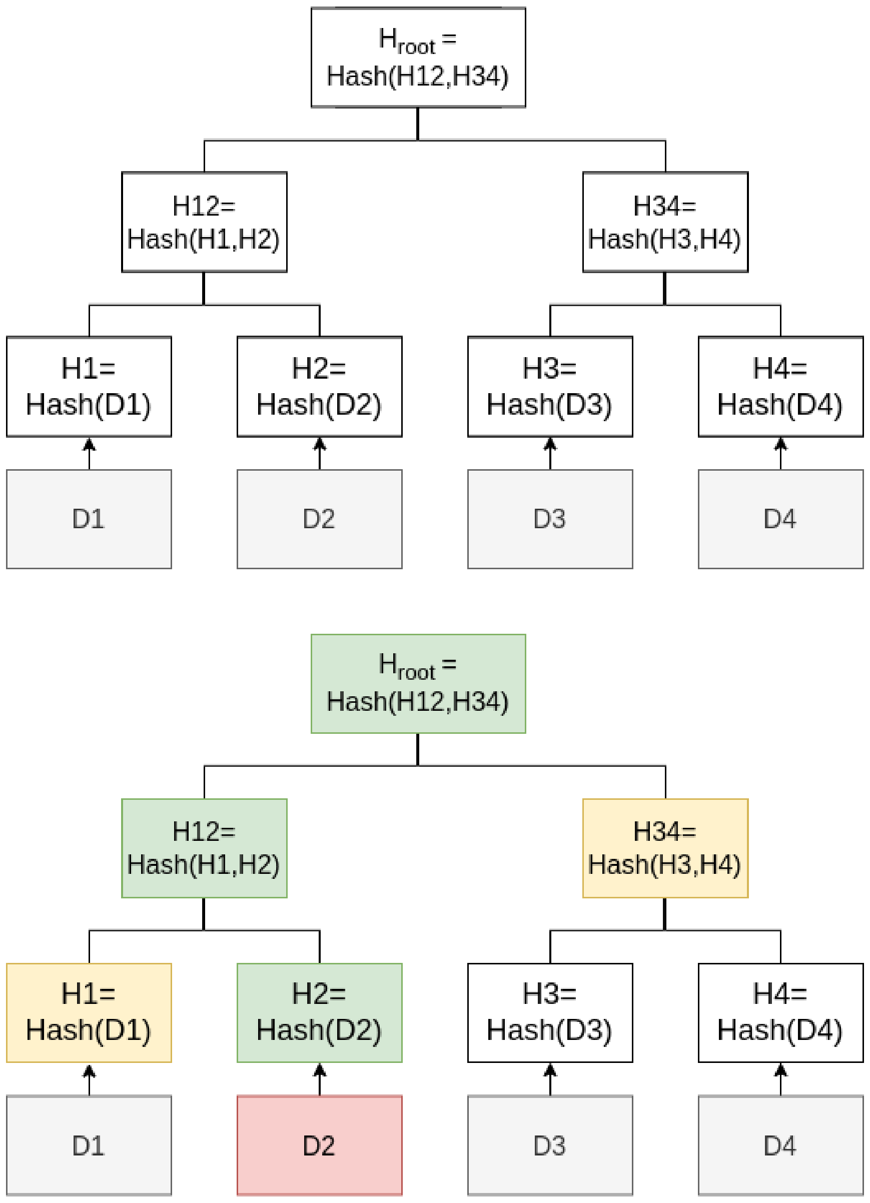

Figure 2.2: Structure of a Merkle tree: Each element's value is the hash of its two child elements. As an example, in order to verify D2, we only require D2, H1, and H34 to recalculate the H2, H12, and H_{root} and verify that the newly calculated root hash matches the previously calculated value.

2.5 Game-based Cryptographic Proofs

Game-based cryptographic proofs show the security of cryptographic primitives through defining a sequence of games played between an adversary who wants to perform an attack and a challenger who is a hypothetical benign entity. Both adversary and challenger are probabilistic processes communicating with each other, and the goal is to show that the probability of some specific event (shown as S, e.g., the adversary successfully breaking the protocol) is close to a target value [75].

In order to achieve this goal, a sequence of several games is defined where the first game describes the original attack, the last game's probability is equal to the target value, and the games in between define events (S_i) related to the main event (S) but with small differences. Then, the proof shows that the probability of the event described in each game ($Pr[S_i]$) is negligibly close to the probability of the next game's event ($Pr[S_{i+1}]$). Therefore, by transforming each game into the next one, we will ultimately show that the probability of the first game is very close to the probability of the last game, which is the target value, and the goal has been achieved.

2.6 ARM Pointer Authentication

Pointer Authentication (PA) [79] is an extension of ARMv8.3, which supports calculating and verifying a Pointer Authentication Code (PAC). PA uses the fact that, in 64-bit architectures, the actual address space is less than 64 bits, meaning that there are several unused bits that can be used to store a MAC.

The way that PA works is by introducing a set of *pac* instructions and their corresponding *aut* instructions, which take a pointer value and a 64-bit modifier and calculate a PAC using a PA key. The instruction set *aut* is used for verification through recalculating the PAC and comparing it with the stored PAC [49]. PA provides five different keys; two keys for data pointers(APDAKey and APDBKey), two keys for code pointers(APIAKey and APIBKey), and one for generic use(APGAKey)[72].

In addition to the previous instructions, PA introduces an instruction, *xpac*, for stripping the PAC, and *pacga*, in order to compute a 32-bit MAC over two 64-bit inputs. The pacga instruction is useful when trying to protect data structures in memory [79]. This instruction uses $APGAKey$, which is the third type of key provided by ARM besides the code pointers keys and data pointers keys.

2.7 Intel AES-NI

Intel AES-NI is a set of encryption instructions based on the Advanced Encryption Standard (AES) algorithm [58], created as an extension to the x86 instruction set architecture to provide high performance and better security. AES is a symmetric block cipher that performs data encryption and decryption in several rounds. The plaintext used in AES encryption is a 128-bit block that is encrypted to a 128-bit block of ciphertext. The key used in AES encryption can have a length of 128, 192, or 256 bits. The number of iterations for AES encryption depends on the length of the key and can be 10, 12, or 14 for 128,192, or 256-bit keys, respectively. Each round of encryption has two 128-bit inputs called *State* and *Round Key*. The Round Key is different in each round and is generated through performing a *Key Expansion* algorithm on the main cipher key.

The encryption process consists of one round of XOR-ing the data block with the cipher key which generates the *State*, and then the *State* is used in the mentioned following rounds of transformations along with the *Round Key*s. The result of the final round is the ciphertext [40].

Chapter 3

Problem Description

3.1 Problem Statement

Low-level languages such as C and C++ can contain memory errors (spatial or temporal errors) that can be exploited to compromise data integrity, leading to attacks like control-flow hijacking or data-only attacks, as outlined in Section 2.2.

There have been numerous attempts to improve the security of C/C++ programs either by introducing ways to detect memory errors or mitigating the mentioned attacks. In the past, control-flow hijacking attacks were considered a more serious threat than non-control-data attacks, so there has been more research done on this category of attacks. Solutions such as Control-Flow Integrity (CFI) techniques are now widely implemented and used. There have also been solutions that enforce Data-Flow Integrity (DFI) that attempts to mitigate both categories of attacks. Nevertheless, as discussed in Section 2.2, there are limitations and downsides to these solutions.

In addition, the overhead and feasibility of such solutions are critical factors to consider when defending against memory corruption attacks. A comprehensive data protection solution would ensure the protection of all data in a program. However, this would lead to high overhead which makes the solution impractical. Accordingly, protecting only critical data is a more reasonable solution. A data structure provides a way to organize data and store it in memory. It can also be used to store sensitive information that can be a target for memory corruption attacks. Consequently, protecting the integrity of data structures is valuable in preventing the mentioned attacks.

In this work, our main goal is to design and implement a *secure (authenticated) version* of gcc C++ standard library data structures (more specifically, stacks, queues, and red-black trees), which guarantees the integrity of elements while stored in memory inside the secure data structures.

3.2 Adversary Model

The adversary model assumes a powerful adversary with arbitrary read and write access to the memory. However, we assume the presence of the following security mechanisms in the system:

- $W \oplus X$ [11]
- Coarse grained control-flow integrity [29]

$W \oplus X$ prevents an adversary from altering code memory pages, eliminates code injection attacks, and limits the adversary to code reuse attacks. Since $W \oplus X$ is implemented in all major processors, it is reasonable to assume its presence.

The second assumption prevents the transfers to arbitrary addresses in the code, meaning that the adversary will not be able to make the program jump to invalid targets (e.g. call functions out of order). This assumption can be satisfied using existing coarse grained CFI solutions with acceptable overhead, such as ROPecker [27].

We also assume that the OS and the underlying hardware are trusted; and that the adversary is limited to userspace and has no control over the kernel space. Moreover, the adversary is unable to access and modify reserved registers.

Since the presence of coarse grained CFI prevents transfers to attacker chosen destinations as explained above, an adversary with the goal of corrupting the secure data structures will not be able to legitimately invoke the implemented API outside the correct flow of the program. Consequently, the adversary is limited to modifying the memory and more specifically, the elements inside the data structures. Our goal is to prevent such element alterations by the adversary. Attacks such as side channel and fault induction attacks are out of the scope of this work.

3.3 Requirements

3.3.1 Generality

Our goal is to provide a drop-in implementation for the secure data structures with the following properties:

G-R1 Requiring minimal change to the program code in applications when switching to use the secure data structures: This requirement ensures maximum usability by minimizing the overhead of switching to the secure version described in Section 3.1.

G-R2 Providing the same functionality as the unmodified data structures: Similar to the previous requirement, this requirement ensures maximum usability since the switch to the secure data structures does not introduce any new limitations.

G-R3 Being compatible with both ARM and x86 architectures: Finally, this requirement reduces the limitations in using the secure version by being available for different architectures.

3.3.2 Security

In order to mitigate the attacks through the data structures, we plan to provide the following security promises:

S-R1 Any corruption of secure data structures will be detected when an element is read from the data structure. This requirement ensures that any element successfully read from a data structure will match the element placed into it.

S-R2 The time windows within the update operations are short enough to make it infeasible to corrupt the element when it's in transit.

3.3.3 Performance

In order for the secure data structures to have acceptable performance overhead, they must have the following property:

P-R1 The basic operations for each secure data structure have the same asymptotic time complexity as the unmodified data structures.

Chapter 4

Design

The general idea for securing the data structures is to use Message Authentication Codes (MACs) to verify the integrity of elements inside data structures. More specifically, we calculate a MAC over each element in a data structure and store the MACs along with the elements so that we can later verify the integrity of the elements. The data verification is done by recalculating the element's MAC and checking that the result is identical to the previously calculated MAC. Furthermore, we assign each secure data structure instance a unique *nonce*, which serves as an identifier for that instance and its elements.

In order to provide maximum protection and minimum overhead, the design details for each data structure are slightly different. We cover the shared and case-specific design details in the following sections. The term *unmodified* data structure refers to the unmodified implementation of data structures in gcc libcstdc++ and is being used in the rest of the sections and chapters.

4.1 Securing the MACs

Our designed secure data structures use MACs to verify the integrity of elements when reading from the data structure. However, since we assume a powerful adversary with read/write access to the entire memory as described in Section 3.2, MACs alone will not provide complete protection against attacks such as MAC reuse attack. In a MAC reuse attack, the attacker uses previously calculated MACs (e.g. the MAC of another element in the data structure instance) and replaces the MAC and value of the target element with them and their corresponding data. Since the reused MAC is a valid MAC, it will

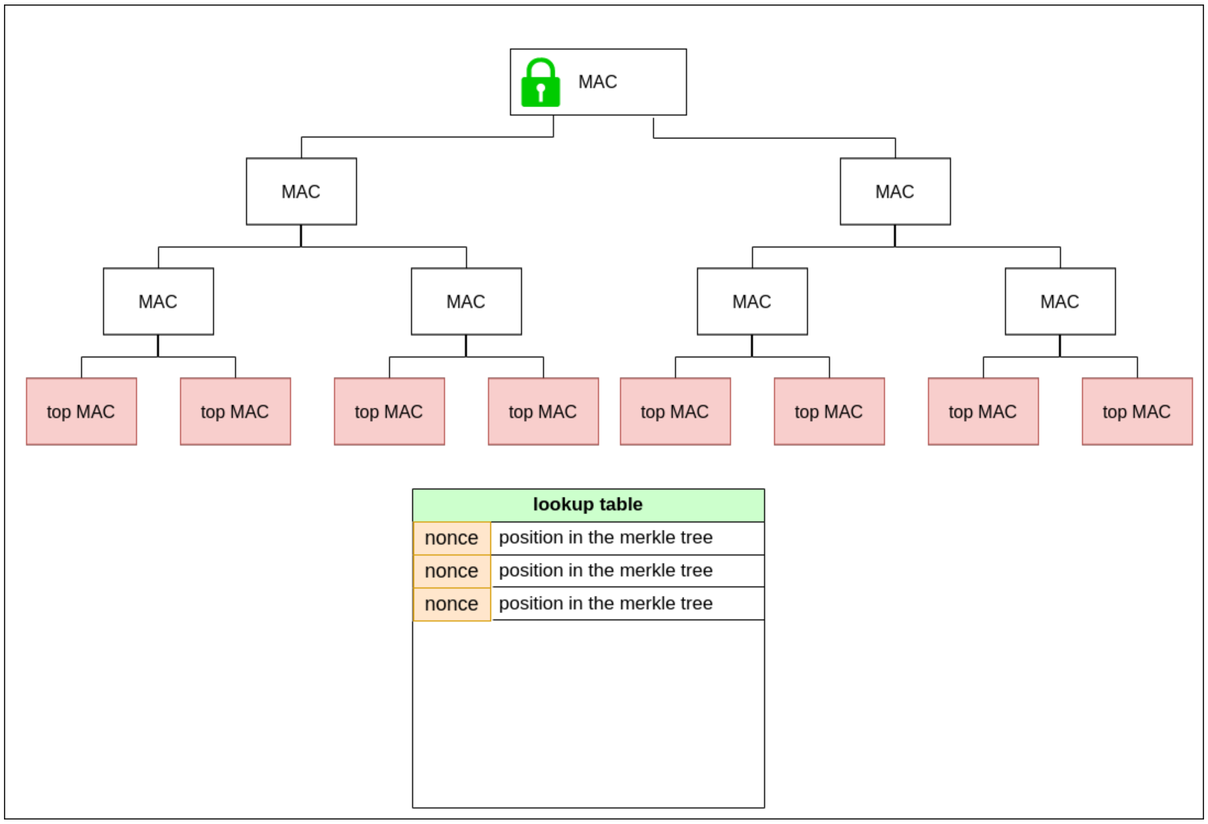

Figure 4.1: Global Merkle tree and lookup table used for mapping each secure data structure instance's nonce to the position of its corresponding top MAC in the Merkle tree. Securely storing the root of the Merkle tree guarantees the integrity of the stored data (the top MACs of the data structures in our case). The lookup table is also used to increase the efficiency of finding the corresponding top MAC of each data structure instance in the Merkle tree.

be verified successfully and hence allows the attacker to replace an element with another element without being detected. This attack can be mitigated by preventing an attacker from replacing MACs.

Reserved registers provide a secure way to store the MACs and prevent the attacker from replacing them. However, keeping all of the MACs in reserved registers is neither a practical nor scalable solution since there are not enough registers to dedicate to storing MACs. Accordingly, our design for the secure data structures is based on two main features: First, for each data structure, we calculate and securely store one specific MAC (which we call the *top MAC* of that data structure instance) that allows for verifying the element MACs. Secondly, a global Merkle tree is used to store the top MAC for each data structure while its own root is kept in a reserved register. This approach minimizes the cost of using reserved registers by requiring only a single register no matter the number of data structure instances. As shown in Figure 4.1, our design consists of a global Merkle tree and a lookup table that maps each data structure instance's nonce to the index of its corresponding top MAC in the Merkle tree.

4.2 Secure Data Structures Design

4.2.1 Secure Stack

A stack is a data structure designed to work in a Last-In-First-Out (LIFO) manner. There is only one point of access in the stack since the data is both pushed to and popped from the top of the stack. The basic operations for the gcc libstdc++ implementation of stack data structure are as follows:

1. push: Adds an element to the top of the stack
2. pop: Removes the top element of the stack without returning the element to the caller
3. top: Returns a reference to the element at the top of the stack
4. size: Returns the number of elements in the stack

As shown in Figure 6.1, we introduce *secure-stack* (out designed secure stack data structure) which is composed of two unmodified stacks, one for storing the elements and

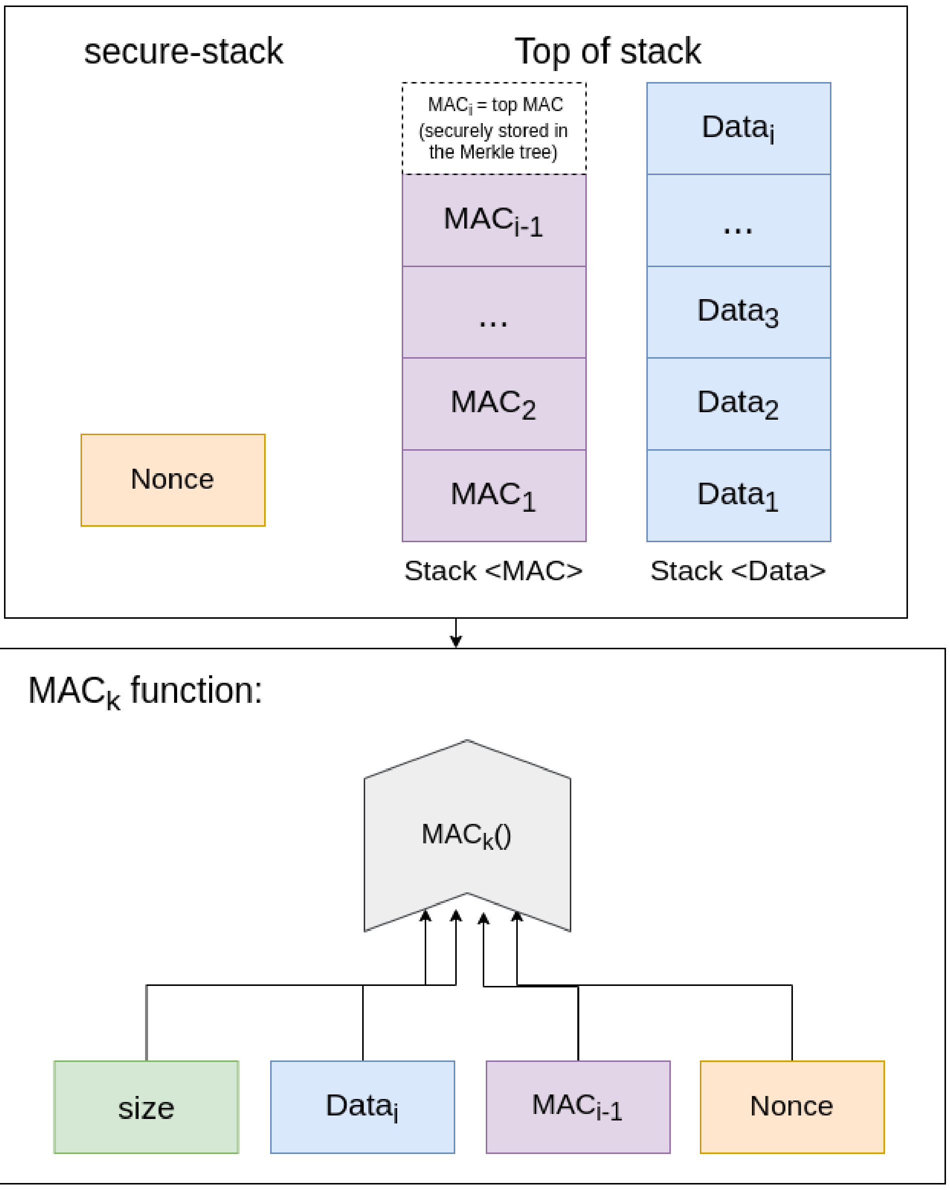

Figure 4.2: Design of a secure stack data structure: the secure design of a stack uses two unmodified stacks to store the elements and their MACs. Moreover, each instance of the secure data structures is assigned a unique nonce as an identifier. Since the nonce is included in the MACs, it also prevents an attacker from replacing elements with elements from other data structures.

one for the MACs that are used to verify the integrity of the elements. The nonce assigned to the secure-stack is used to retrieve its top MAC from the global Merkle tree.

For each element in a stack, a MAC is calculated based on: the element's value, the current size of the stack, the nonce, and the MAC of the previous (top) element in the secure-stack. Accordingly, the MAC for the element i is calculated as follows (assuming that the value of i increases from bottom to top of the stack):

$$MAC_i = MAC_k(\text{element's value, nonce, size}, MAC_{i-1})$$

Based on the MAC calculation formula, each MAC depends on the MAC of the previous element. Therefore, the MACs are chained together, and the most recent MAC alone is enough to verify the integrity of all the other MACs. Therefore, the most recent MAC will serve as the top MAC for a secure-stack instance and is kept in the global Merkle tree. However, while a secure-stack instance is empty, its nonce is stored as its top MAC.

Since in a secure-stack, elements are always read from the top, we can verify the integrity of the accessed elements using the top MAC. Moreover, if an element is removed directly from the top of the secure-stack, we can verify the integrity of the MAC of the next element using the MAC of the removed element (previous top MAC) and then set it as the new top MAC for the data structure instance. Finally, when pushing new elements into a secure-stack, the MAC of the new element will replace the previous top MAC. The details of how each operation works in a secure-stack is presented in Algorithms 1, 2, 3 and 4.

`secure-stack.push(x)`

As shown in Algorithm 1, first, the top MAC is retrieved from the Merkle tree, and then the element is pushed into the data-stack. Furthermore, before calculating and storing the MAC for the new element, the previous top MAC is pushed into the MAC-stack, and then its value is replaced by the MAC of the new element. Finally, the updated top MAC is stored in the Merkle tree.

`secure-stack.pop()`

This operation is presented in Algorithm 2. When popping values from a secure-stack, the next element will be the new top element and its MAC will be the new top MAC. Therefore, we need to verify that it is valid. Consequently, we get the top of the mac-stack and the data-stack, recalculate the top MAC using them and verify that this value matches the top MAC from the Merkle tree. If the top MAC verifies, we remove the top element from the data-stack, remove the top of the mac-stack, and store it as the new top MAC.

`secure-stack.top()`

The `top()` operation represented in Algorithm 3 is almost identical to the `pop()` operation except that the `top()` returns a reference to the element instead of removing it from the stack. Therefore, after verifying the top MAC, a reference to the top element is returned as the output and nothing gets removed.

`secure-stack.size()`

In order to verify the size of the secure stack, we call the `top()` function since it performs the required MAC verification that includes the size (the current size value is included in the top MAC as an input). However, if the secure-stack is empty, we cannot call the `top()` function for this purpose since calling `top()` on empty stack could result in undefined behavior. Instead, since for an empty secure-stack the top MAC is equal to the nonce of the data structure, we can verify the size by checking if the top MAC matches the nonce. These steps are represented in Algorithm 4.

Algorithm 1 secure-stack.push(x)

```
1: top-mac = get-top-mac()  ▷ The top MAC is read from Merkle tree to be pushed into
   the mac-stack so that the new top MAC can be stored in the Merkle tree.
2: data-stack.push(x)
3: size = size + 1
4: mac-stack.push(top-mac)                 ▷ Previous top MAC goes into mac-stack.
5: top-mac = $\mathsf{MAC}_k$(x, nonce, size, top-mac)
6: insert-top-mac(top-mac)  ▷ The old top MAC in the Merkle tree is replaced with the
   MAC of the newly pushed element.
```

4.2.2 Secure Queue

A queue is a data structure that works in a First-In-First-Out (FIFO) format, where elements are enqueued from the back of the queue and dequeued from the front. The basic operations for the gcc libstdc++ queue data structure are:

Algorithm 2 secure-stack.pop()

```
1: x = data-stack.top()
2: top-mac = get-top-mac()
3: if top-mac == $\mathsf{MAC}_k$(x, nonce, size, mac-stack.top()) then    ▷ This checks the correctness of the next MAC since it will later replace the top MAC in the Merkle tree.
4:     data-stack.pop()
5:     insert-top-mac(mac-stack.top())
6:     mac-stack.pop()
7: else
8:     exception("MAC authentication error.")
9: end if
```

Algorithm 3 secure-stack.top()

```
1: x = data-stack.top()
2: top-mac = get-top-mac()
3: if top-mac == $\mathsf{MAC}_k$(x, nonce, size, mac-stack.top()) then    ▷ This check verifies the correctness of the top element before being returned as the output. return x
4: else
5:     exception("MAC authentication error.")
6: end if
```

Algorithm 4 secure-stack.size()

```
1: if size $\neq 0$ then
2:     top()    ▷ Calling the top() function verifies the size as part of the MAC verification. return size
3: else
4:     if nonce == get-top-mac() then    ▷ When the size is 0, it cannot be verified using the top() function. However, in this case, the top MAC should be equal to the nonce and hence it can be verified by this check.
5: return size
6:     else
7:         exception("MAC authentication error.")
8:     end if
9: end if
```

1. enqueue: Inserts elements to the back of the queue
2. dequeue: Removes the element at the front of the queue
3. front: Returns a reference to the element at the front of the queue
4. back: Returns a reference to the element at the back of the queue
5. size: Returns the number of the elements in the queue

The queue is different from a stack mainly due to having two points of access for the stored elements. In a stack, an element is always pushed to, read, and popped from the top. However, in a queue, the element is enqueued from one end and dequeued from the other end. It also allows reading elements from both front and back. Therefore, the chaining strategy used in the secure-stack is not practical in a queue due to compromising efficiency. The reason is that when chaining the MACs if a new MAC is added or removed from the bottom (or top, depending on the direction of the stacking mechanism), all MACs need to be updated since each of them is calculated using the MAC before (or after) it.

Consequently, in order to maintain efficiency, we introduce *secure-queue* (our designed secure queue data structure) with a slightly different design from the secure-stack, as presented in Figure 4.3. As shown in the figure, the secure-queue uses two indices, back-index and front-index, and the top MAC is calculated over them along with the MACs of the front and back elements. The two described indices are initialized as a secure-queue is constructed. The initial values are 0 for the back-index and 1 for the front-index. Once a new element is pushed to the back of secure-queue, its index will be the previous back-index plus one. The value of the back-index itself will also increase by one. When an element is dequeued from the front of the secure-queue, the value of the front-index is updated to front-index+1 so that it is equal to the index of the new front element. We calculate the MAC for the front and back element differently from the rest of the elements. The MAC for the elements with index i, except front and back elements, is calculated as presented below:

$$MAC_i = MAC_k(\text{element value}, \text{nonce}, \text{i})$$

The MAC for the front and back elements does not include their indices. Therefore, their MACs are calculated as follows:

$$\text{front or back MAC} = MAC_k(\text{element value}, \text{nonce})$$

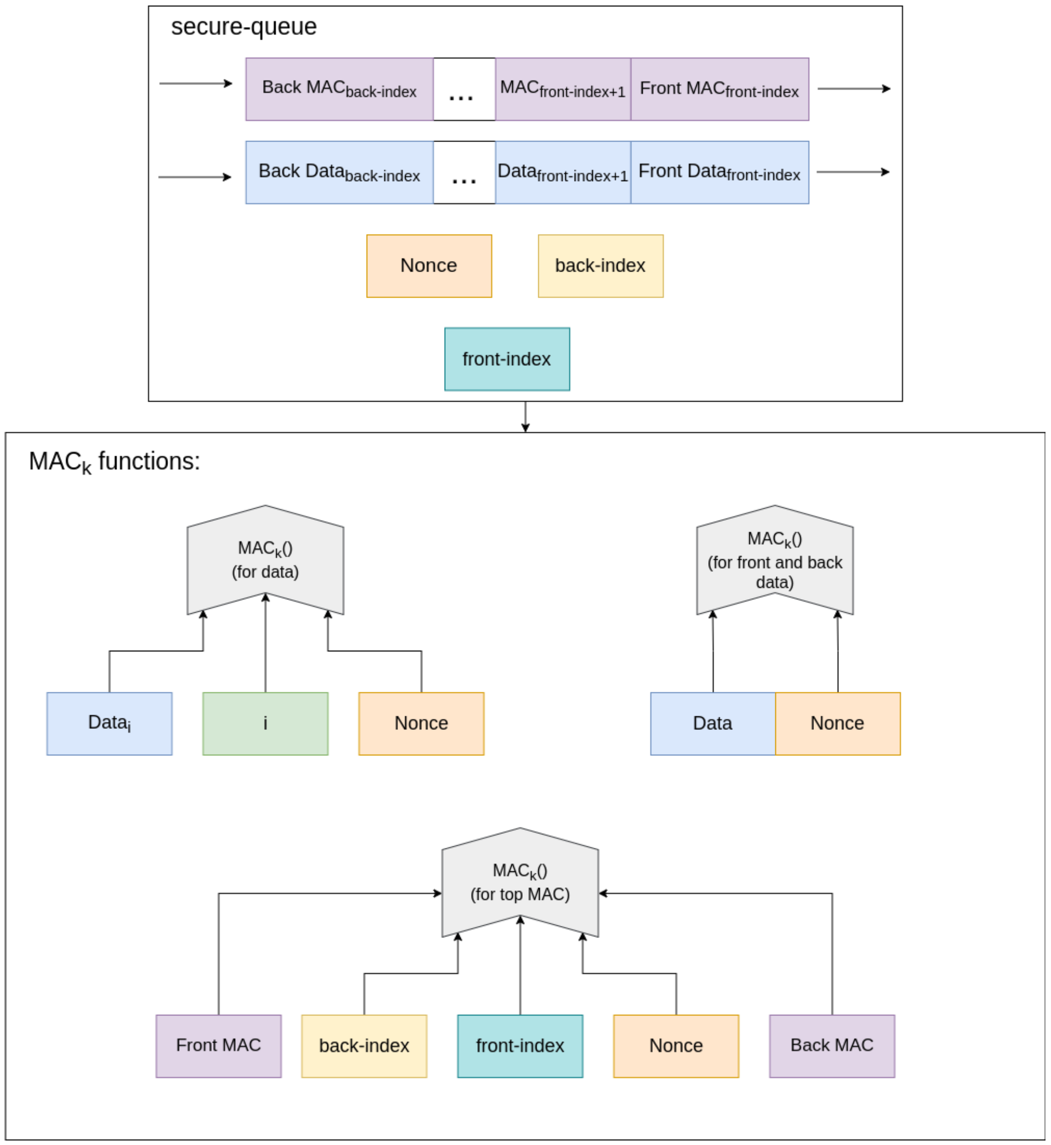

Figure 4.3: Design of a secure queue data structure: the secure design of a queue uses two unmodified queues to store the elements and their MACs. Moreover, each instance of the secure data structures is assigned a unique nonce as an identifier. Since the nonce is included in both types of MACs, it also prevents an attacker from replacing elements with elements from other data structures.

Based on this design, for each element in a secure-queue, there is an index that stays unique during the whole lifetime of the secure-queue instance. The top MAC that is calculated over the nonce, the front and back indices, and the MAC of the front and back elements can be used to validate the integrity of the indices and elements at both ends. Consequently, the attacker will not be able to illegitimately (outside the provided API) change the element and reuse MACs calculated over other elements.

$$\text{top MAC } = MAC_k(\text{nonce}, \text{back-index}, \text{front-index}, \text{back MAC}, \text{front MAC})$$

When an element is read from the front or back, its MAC can be recalculated and verified. The values of the back and front MACs are verified using the top MAC and then they can be used to verify the front or back elements. The algorithms for each operation in a secure-queue are illustrated in Algorithms 5, 6, 7, 8 and 9.

`secure-queue.enqueue(x)`

The enqueue operation is presented in Algorithm 5. The first step is to verify the indices and front and back MACs using the top MAC. After ensuring that the indices and MACs are correct, the MAC of the current back element is updated since now it should include the element index (unless it is also the front element). Then, the element is inserted into the element queue. Next, the back-index will be updated, and the MAC for the new element will be calculated. Finally, the top MAC is updated to realize the changes in the back-index value and back MAC.

`secure-queue.dequeue()`

In the dequeue operation, similar to the `enqueue()`, first, the top MAC is used to verify the indices and front and back MACs. Then the front element and its MAC are dequeued from the element and MAC queues. The next step is to update the front-index to the index of the next element. If the new front element has previously been a middle element, it is first verified and then its MAC will be updated to not include the index anymore. Finally, the top MAC should be updated to realize this change. These steps are represented in Algorithm 6.

`secure-queue.front()` **and** `secure-queue.back()`

The design for the front and back operations is almost the same. Similar to the other operations, the top MAC is used in both operations to verify indices and front and back MACs. Then, in the `front()`, the MAC of the front element is calculated and compared to the one stored in the MAC queue. Similarly, the same steps are performed in the `back()` to verify the back element. If the MAC verification passes, the front or back element is returned. These steps are shown in Algorithms 7 and 8.

`secure-queue.size()`

As shown in Algorithm 9, the `size()` operation in a secure-queue uses the indices to measure the size. The top MAC is first used to verify the indices, then the size of the secure-queue is calculated by subtracting the indices. Since the integrity of the indices is verified by the top MAC , this approach guarantees that the returned size is always correct.

Algorithm 5 secure-queue.enqueue(x)

```
1: if $\mathsf{MAC}_k$(nonce, front-index, back-index, mac-queue.back(), mac-queue.front()) $\neq$ get-top-mac() then
2:     exception("MAC authentication error.")   ▷ Verifying the indices and MACs using top MAC.
3: end if
4:     ▷ Updating the previous back element's MAC, inserting the element, updating the back-index and then calculating the element MAC and updating the top MAC.
5: if size() $> 1$ then
6:     x = element-queue.back()
7:     if $\mathsf{MAC}_k$(x, nonce) $\neq$ mac-queue.back() then   ▷ Verifying the back element before updating its MAC.
8:         exception("MAC authentication error.")
9:     end if
10:    mac-queue.back() = $\mathsf{MAC}_k$(x, nonce, back-index)
11: end if
12: element-queue.enqueue(x)
13: back-index = back-index + 1
14: mac-queue.enqueue($\mathsf{MAC}_k$(x, nonce))
15: insert($\mathsf{MAC}_k$(nonce, front-index, back-index, mac-queue.back(), mac-queue.front()))
```

Algorithm 6 secure-queue.dequeue()

```
1: if $\mathsf{MAC}_k$(nonce, front-index, back-index, mac-queue.back(), mac-queue.front()) $\neq$ get-top-mac() then
2:     exception("MAC authentication error.")    ▷ Verifying the indices and MACs using top MAC.
3: end if
4:   ▷ Removing the element and its MAC, updating the front-index and then updating the top MAC.
5: element-queue.dequeue()
6: mac-queue.dequeue()
7: front-index = front-index + 1
8: if size() > 1 then
9:     x = element-queue.front()
10:    if $\mathsf{MAC}_k$(x, nonce, front-index) $\neq$ mac-queue.front() then    ▷ Verifying the new front element before updating its MAC.
11:        exception("MAC authentication error.")
12:    end if
13:    mac-queue.front() = $\mathsf{MAC}_k$(x, nonce)
14: end if
15: insert($\mathsf{MAC}_k$(nonce, front-index, back-index, mac-queue.back(), mac-queue.front()))
```

Algorithm 7 secure-queue.front()

```
1: if $\mathsf{MAC}_k$(nonce, front-index, back-index, mac-queue.back(), mac-queue.front()) $\neq$ get-top-mac() then
2:     exception("MAC authentication error.")    ▷ Verifying the indices using top MAC.
3: end if
4:   ▷ Reading and verifying the front element with its MAC and returning the element.
5: x = element-queue.front()
6: if mac-queue.front() == $\mathsf{MAC}_k$(x, nonce) then return x
7: else
8:     exception("MAC authentication error.")
9: end if
```

Algorithm 8 secure-queue.back()

```
1: if $\mathsf{MAC}_k$(nonce, front-index, back-index, mac-queue.back(), mac-queue.front()) $\neq$ get-top-mac() then
2:     exception(“MAC authentication error.”)    ▷ Verifying the indices using top MAC.
3: end if
4:     ▷ Reading and verifying the back element with its MAC and returning the element.
5: x = element-queue.back()
6: if mac-queue.back() == $\mathsf{MAC}_k$(x, nonce) then return x
7: else
8:     exception(“MAC authentication error.”)
9: end if
```

Algorithm 9 secure-queue.size()

```
1: if $\mathsf{MAC}_k$(nonce, front-index, back-index, mac-queue.back(), mac-queue.front()) $\neq$ get-top-mac() then
2:
3:     exception(“MAC authentication error.”)    ▷ Verifying the indices using top MAC.
4: else
       return back-index - front-index + 1    ▷ Calculating the size using the indices.
5: end if
```

4.2.3 Secure Red-Black Tree

A red-black tree is a type of self-balancing binary search tree that allows for storing comparable data. In this data structure, every element has a key and a value and is assigned a black or red color which is used for rebalancing the tree. The rebalancing process modifies the position of the elements to ensure that the height of the left and right sides of the tree do not differ by more than one. Basic red-black tree operations in gcc libstdc++ include:

1. insert: Inserts a new element into the tree
2. erase: Removes an element from the tree
3. find: Finds and returns a specified element in the tree

As shown in Figure 4.4, we introduce *secure-rb-tree* (our designed secure red-black tree data structure), in which a MAC is being calculated for each element and stored along with them, and the top MAC in the secure-rb-tree is the root's MAC. The MAC for each element is calculated using the nonce, the data, MAC of its left child, and MAC of its right child as follows:

$$MAC = MAC_k(\text{nonce, data}, \text{left-child.MAC}, \text{right-child.MAC}))$$

In all operations, as the algorithm goes down in a secure-rb-tree to find an element or find the proper place to insert a new one, all the elements on the path will be verified sequentially using their MACs. The verification starts with the root, which is being verified based on the top MAC. Moreover, adding or removing an element requires updating the MACs from that element up to the root along with the rebalancing process.

The details of each operation in the secure-rb-tree are included in the Chapter 9 since they are more complex than the stack or queue operations and take up a considerable amount of space. Below, we describe the main details of the mentioned operations (the `find()` operation is explained within the two other operations):

`secure-rb-tree.insert(x)`

The insert operation consists of three parts. First, we need to go down the tree and compare the new element's key with the current elements in the tree to find the correct place for the new element. While going down the tree starting from the root, each element

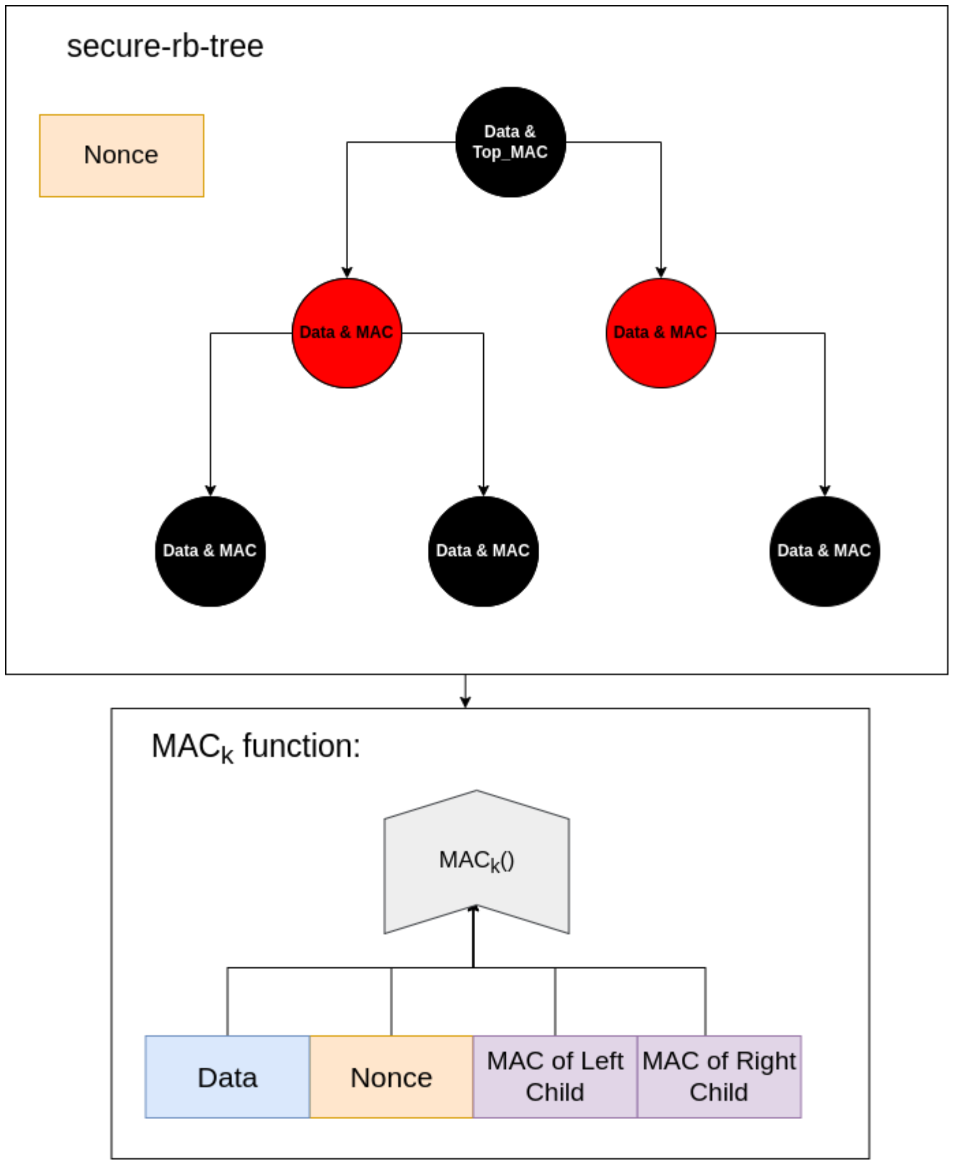

Figure 4.4: Design of a secure red-black tree data structure: The secure-rb-tree stores MACs in the elements themselves since elements are defined as *structs* and there is no need for dedicating another data structure to the MACs.

is verified using its MAC. Since the root's MAC is retrieved from the Merkle tree, and the MAC of each element is verified when authenticating its parent's MAC, all the accessed elements and their MACs will be verified according to the top MAC. After finding the correct spot, the new element is created and stored along with its MAC. The tree is then rebalanced. The rebalancing process goes up from the newly inserted element and checks the color of the elements for inconsistencies caused by adding the new element. We take advantage of this process to update the MACs while rebalancing the tree. However, the rebalancing does not necessarily go up to the root (the tree might be balanced from the beginning or become balanced after going up a few levels). Accordingly, we perform an updating process that continues updating the MACs all the way to the top of the tree.

`secure-rb-tree.erase(x)`

This operation is similar to the `insert(x)` operation. The first step is finding the element that needs to be erased. The process of finding the element is very similar to finding the insertion spot in the `insert(x)` operation and requires the MAC verification for the whole path. After finding the element and removing it from the secure-rb-tree, the tree is rebalanced and the MACs are updated along the path from the path from the removed element to the top of the tree.

4.3 Object Wrappers

When reading elements from stack, queue, or red-black tree data structures, a reference to the element is returned. More specifically, the gcc libstdc++ implementation of these data structures returns a reference to the element when calling functions such as `top()` in the stack, `front()` or `back()` in the queue, and `find()` in the red-black tree. This reference allows for legitimately modifying the element's value while it is inside the data structure. In a secure-stack or secure-queue, this reference can allow for modifying an element in the middle of the data structure and not only the top element in the secure-stack, or the front and back elements in the secure-queue. The reason is that for instance, after returning a reference to the top element in the secure-stack, new elements could be pushed to the top. In this case, the reference will then point to the element which is now in the middle of the secure-stack. The same thing can happen in a secure-queue. In a secure-rb-tree, the reference generally allows modifying an element in the tree, which can be at any position in the data structure.

In our designed secure data structures, the MACs are only calculated and updated during the main operations described in the previous sections. Therefore, if an element inside the secure data structures is modified using the reference, the previously calculated MAC will not match the MAC of the element's new value and will cause an authentication failure which is a false positive.

In order to overcome this issue, we added an object wrapper mechanism to the design. Accordingly, when reading an element, instead of returning a plain reference to the element, an object wrapper is returned that includes a reference to the element and performs the necessary MAC updates as soon as the element is modified through the object wrapper interface. The MAC update process will update all the MACs affected by the change, which is not always just the element's MAC. For instance, in a secure-stack, since each MACs is calculated based on the MAC of the element below it, if an element in the middle of the stack changes, all of the MACs up to the top need to be updated.

Chapter 5

Implementation

5.1 General Implementation Details

We extended the gcc-11.1.0 C++ Standard library to include our implementation for secure data structures. The gcc C++ standard library implements the data structures as template classes that use a container as their underlying storage class. The stack and queue templates use the *deque* container by default as a template parameter. Therefore, in order to preserve the API of the standard library data structures, we implemented the secure data structures as containers that can replace the deque. We offer an option, *Secure* macro, which can be set at compile-time to switch the default template to use secure data structures. Accordingly, the default container for the stack and queue will be the secure-stack and secure-queue if the *Secure* option is enabled. The container can also be manually chosen by the developer regardless of whether the *Secure* option is enabled. However, the rb-tree is not directly exposed by the standard library API, but is instead used as the underlying container for data structures such as *map*. Therefore, we used a similar approach as the secure-stack and secure-queue to create a secure map using our secure-rb-tree implementation.

As mentioned in Chapter 4, we use Message Authentication Codes (MACs) to verify the integrity of the data structures. In order to provide a generic interface for the MAC calculations, we implemented an interface class called top-MAC-store, which defines the MAC calculation functions for all data structures. There are derived classes that implement the MAC related functions for each of ARM and x86 architectures and are passed to the secure data structures as template parameters. Therefore, although the implementation ensures that the suitable class will be chosen by default using pre-defined compiler macros, the developer can also choose their desired MAC calculation class when using secure data structures. For the ARM version, the MACs are calculated using Pointer Authentication (PA) feature on ARMv8.3. Specifically, we used the *pacga* instruction which computes a 32-bit MAC over two 64-bit inputs. Regarding the x86 version, we used the Intel AES-NI encryption instructions described in Section 2.7. The keys for AES-NI encryption are stored in reserved registers (*xmm5-xmm15*) to prevent an attacker from modifying the keys.

Finally, we calculate the MACs over the hash of the element to easily incorporate different data types into MAC instructions. Accordingly, we allow the developer to use a default hash function which treats objects as flat memory structures or provide their own hash function which could support complex data types. We use an open-source SHA256 hash generator [63] as the default option for all data structures.

Listing 5.1 presents an example of how the mentioned template options are defined in secure data structures. This example specifically represents the secure-stack; however, a similar approach has been used for the rest of the secure data structures.

```
template <typename _Tp, typename _Sequence = std::deque<_Tp>, typename
    HashType = SecureHash<_Tp>, typename MACStoreType = MACStore>
class secure_stack : std::unsafestack<_Tp>
```

Listing 5.1: Secure-stack template parameters: The default MACStoreType, MACStore, will be defined as an ARM or x86 architecture class using the macros depicting the architecture for which the library is used.

5.2 Securing the MACs

We use an open-source implementation of a Merkle tree [56] for our prototype. We created a Merkle tree class that stores an instance of the Merkle tree along with a lookup table and required functions to get, verify, and update the top MACs. This Merkle tree class is created as a global object that is accessible to all derived classes from the top-MAC-store abstract class. The root of the Merkle tree is securely stored in a reserved register as stated in Chapter 4. The register used for storing the root of the Merkle tree is *x28* for the ARM version and *r13* for the x86 version.

The implemented lookup table is a simple `std::map` data structure (the unmodified version). The map data structure used for the lookup table cannot be the secure version using the secure-rb-tree. The reason is that the secure-rb-tree itself uses the Merkle tree class and hence the lookup table. Therefore, if the lookup table is a secure map, it will make an infinite loop which does not work. However, since the top MAC for each data structure is tied to its nonce and can be verified that it belongs to the correct data structure, modifying the lookup table will not create any new attack opportunities for the adversary. Accordingly, the implementation does not require using secure storage for the lookup table in the first place.

Moreover, in order to decrease the overhead of verifying the Merkle tree, the Merkle tree class maintains a list of indices of the emptied elements in the Merkle tree. These elements belong to the deconstructed data structure instances which no longer exist. Therefore, their corresponding elements in the Merkle tree can be reassigned to the new data structures. Accordingly, upon the creation of a new data structure, the Merkle tree class checks the list of empty indices and assigns an empty element to the new data structure instead of immediately creating a new element in the Merkle tree. This approach eliminates the need for fully deleting the elements from the Merkle tree when they are emptied, which can require changing the structure of the whole tree, and improves the performance as it assigns new values to the empty elements (taking $O(log(i))$) instead of leaving them unused or removing them from the Merkle tree which could be of order $O(i)$ in the worst case.

5.3 Secure Data Structures Implementation

5.3.1 Secure Stack

As described in Section 4.2, the stack data structure provides four main operations: push, pop, top, and size. The template implementation of stack in the Standard Template Library (STL) also provides the `empty()`, `swap()`, and `emplace()` operation. The `empty()` function returns a boolean showing whether or not the stack is empty. `swap()` and `emplace()` functions are only provided since C++11 and swap the content of two stacks, and construct an element in-place at the top of the stack, respectively. Although our secure stack is used as a container replacing the deque as the default container, it is implemented with the same functionality as the stack itself. Accordingly, the secure-stack does not implement all operations provided by a container such as a deque. However, the implemented operations are sufficient to replace the deque container in a stack data structure based on the requirements stated by the STL.

Our secure-stack implementation uses two deque-based stacks (which is the unmodified version) to store the elements and their MACs. Moreover, each instance of the secure-stack stores a pointer to its own top-MAC-store derived class, which is used for the MAC calculations and storing the top MAC .

Furthermore, the stack data structure implements the copy and move constructors, which are also included in our secure version. For a move constructor, we move the MACs, nonce, and the top-MAC-store to the new secure-stack using the *std::move()* operation. However, the copy constructor is more complicated because the copy and the original data structures are two distinct data structures and will have two different nonce values since no two instances should have the same nonce. Since the nonce is included in all MACs, the previous MACs are not usable in the copied version. Accordingly, the MACs need to be verified first to ensure the integrity of the data, and then, they need to be recalculated for all the elements. Note that the MACs and elements do not need to be validated in the move constructor since they will not change during the construction, and the validation process will happen when reading the element from the new data structure. The details of how the copy constructor works are presented in Listing 5.2. The copy constructor creates a copy of the fields in the original secure-stack to be used for the verification process. A *temp* vector is also used to store the verified elements. In the first loop, the elements are verified one by one, starting from the top. Then, once all elements are verified, in the second loop, the elements will be pushed into the new secure-stack using its *push()* function, which handles the MAC generation too.

In addition, we customize the deconstructor for the secure-stack to call the proper functions that remove its data from the Merkle tree and add its corresponding element in the Merkle tree to the empty elements list as mentioned in Section 5.2.

```
explicit secure_stack(const secure_stack<_Tp, _Sequence, HashType,
   MACStoreType> &__c) {
    top_mac_store = *new MACStoreType();
    nonce = Nonce::next_nonce();
    // Making copies from the fields of the original secure-stack (__c)
   since it is a constant and should not be modified.
    MACStoreType copy_top_mac_store = __c.top_mac_store;
    int nonce_copy = __c.nonce;
    std::unsafestack<MAC> src_macs{*__c.macs};
    std::unsafestack<_Tp> src_data{__c};
    int size_copy = src_data.size();
    // This function verifies and returns the top MAC from the merkle
   tree.
    MAC c_top_mac = *copy_top_mac_store.get_top_mac(nonce_copy);

    macs = std::unique_ptr<std::unsafestack<MAC>>(new std::unsafestack<
   MAC>());
    std::vector<_Tp> temp;
    int size = size_copy;
    // Verifying the MACs of the original secure-stack
    for (int j = 0; j < size; j++)
    {
        _Tp top_data = src_data.top();
        std::string dataHash = HashType::hash(top_data);
        // The dataHash is a string but is transformed into a long int to calculate the MAC over it.
        verify_mac(
            top_mac_store.calculate_mac_stack(stol("000" + dataHash.substr(3, dataHash.length() - 1)), nonce_copy, this_address, size_copy - j, src_macs.top()),
            c_top_mac);
        c_top_mac = src_macs.top();
        src_macs.pop();
        src_data.pop();
        temp.push(top_data);
    }
    // Stores the initial top MAC, which is its nonce, for the new secure-stack in the merkle tree.
    init_mac((MAC)nonce);
    // Pushes the values into the new secure-stack. The push function performs the required MAC calculations.
    for (int i = temp.size() - 1; i > -1; i--)
    {
        _Tp top = temp[i];
        push(top);
    }
}
```

Listing 5.2: secure-stack copy constructor

5.3.2 Secure Queue

The queue data structure includes basic operations as discussed in Section 4.2.2 which are: enqueue, dequeue, front, back, and size. The implemented queue data structure in STL also includes `empty()`, `swap()`, and `emplace()` operations similar to the stack. The secure-queue is also added to the libstdc++ *stl-queue* implementation as an underlying container and implements the same operations as the template queue implementation in STL.

The secure-queue uses unmodified queues to store the elements and MACs. The top-MAC-store and nonce field in the secure-queue is the same as in the secure-stack. Moreover, the move and copy constructors for the secure-queue are very similar to the secure-stack. In the copy constructor, we verify all the MACs and then recalculate the new MACs by calling the *enqueue* function for each element in the original secure-queue. The move constructor uses *std::move()* to move the fields from the original secure-queue to the new one. The deconstructor for the secure-queue also calls the proper functions to remove its data from the Merkle tree class and the lookup table.

5.3.3 Secure Red-Black Tree

The secure-rb-tree is implemented separately from the *stl_tree* implementation in libstdc++ but has been added as the underlying container to *stl_map* to create a secure map data structure. As explained in the Section 4.2.3, the operations in a secure-rb-tree require verifying and updating a whole path from top to bottom and vice versa. Accordingly, this might introduce an opportunity for the attacker to change values in between the verification and update operations. This type of attack can be prevented by limiting the time window for attacks on the elements while updating by using an additional reserved register for securing a MAC over the updated value and old value of the subtree root. In this approach, we store a MAC over the new and old value of the local root of the updated sub-tree to ensure its security. This approach satisfies our security requirement Item **S-R2**.

Moreover, the unmodified rb-tree implementation makes use of an additional element, called *header*, which stores the root, the leftmost, and rightmost elements in the tree for easier access. The secure-rb-tree uses the same implementation. In order to preserve the integrity of the *header*, we created a second type of MAC for the header. Since the header MAC includes the root MAC as part of it, in the secure-rb-tree implementation, the *header* MAC is stored as the top MAC instead of the root MAC.

$$headerMAC = MAC_k(\text{nonce}, \text{leftmost.MAC}, \text{rightmost.MAC}, \text{root.MAC}))$$

The STL implementation for the rb-tree includes various functions for simplifying the operations on the tree. We have added the required MAC validation and updates to those functions but preserved their general format in the secure-rb-tree. Examples of such functions include `minimum(x)` which returns the leftmost descendant, and `maximum()` which returns the rightmost descendant of x. In both functions, a loop has been implemented that goes down on the left or right side of x until the last leaf is reached. We have implemented a validation check for each step of the loop to validate the corresponding element at that level as depicted in Listing 5.3.

```
element minimum(element __x)
{
  while (__x->_M_left != 0)
  {
    __x = __x->_M_left;
    // mac() returns the stored MAC for the element.
    verify_mac(top_mac_store.calculate_mac_tree(nonce, mac(__x->_M_left),
    mac(__x->_M_right), hash_calculation(__x)), mac(__x));
  }
  return __x;
}
```

Listing 5.3: minimum(x) function in red black tree which returns the leftmost descendant.

5.4 MAC Calculation

The MAC calculation functions have been defined as pure virtual functions in top-MAC-store abstract class and then implemented in separate derived classes for the ARM and x86 architectures. An example of how the ARM and x86 MACs are calculated is shown in Listings 5.4 and 5.5.

```
   __asm__ volatile(
             "pacga %[data], %[data], %[nonce]\n\t"
             "pacga %[result], %[index], %[data]\n\t"
             : [result] "=r"(result)
             : [data] "r"(data),
               [nonce] "r"(nonce),
               [index] "r"(index)
             :);
```

Listing 5.4: ARM MAC calculation for secure-queue data: We use inline assembly to calculate the MAC. Also, since the ”pacga” instruction only uses two values, we need to perform the MAC generation two times to include all three inputs.

```
#define DO_ENC_BLOCK(m,k) \
  do{\
        m = _mm_xor_si128       (m, k[ 0]); \
        m = _mm_aesenc_si128    (m, k[ 1]); \
        m = _mm_aesenc_si128    (m, k[ 2]); \
        m = _mm_aesenc_si128    (m, k[ 3]); \
        m = _mm_aesenc_si128    (m, k[ 4]); \
        m = _mm_aesenc_si128    (m, k[ 5]); \
        m = _mm_aesenc_si128    (m, k[ 6]); \
        m = _mm_aesenc_si128    (m, k[ 7]); \
        m = _mm_aesenc_si128    (m, k[ 8]); \
        m = _mm_aesenc_si128    (m, k[ 9]); \
        m = _mm_aesenclast_si128(m, k[10]);\
    }while(0)

void AES_CMAC ( unsigned char *key, unsigned char *input, int length,
 unsigned char *mac )
{
     unsigned char       X[16],Y[16], M_last[16], padded[16];
      unsigned char       K1[16], K2[16];
      int         n, i, flag;
      generate_subkey(key,K1,K2);
      n = (length+15) / 16;       /* n is number of rounds */
      if ( n == 0 ) {
          n = 1;
          flag = 0;
      } else {
          if ( (length%16) == 0 ) { /* last block is a complete block */
              flag = 1;
          } else { /* last block is not complete block */
              flag = 0;
          }
      }
      if ( flag ) { /* last block is complete block */
          xor_128(&input[16*(n-1)],K1,M_last);
      } else {
          padding(&input[16*(n-1)],padded,length%16);
          xor_128(padded,K2,M_last);
      }
      for ( i=0; i<16; i++ ) X[i] = 0;
      for ( i=0; i<n-1; i++ ) {
          xor_128(X,&input[16*i],Y); /* Y := Mi (+) X  */
          aesencrypt(Y, X);
      }
      xor_128(X,M_last,Y);
      aesencrypt(Y, X);
      for ( i=0; i<16; i++ ) {
          mac[i] = X[i];}}
```

Listing 5.5: x86 MAC calculation adopted from open-source implementations of CMAC [45]: a CMAC is calculated, over a message (*unsigned char *input*) generated by concatenating all inputs, using the AES-NI encryption instructions.

5.5 Object Wrappers

The object wrappers required for each secure data structure are implemented as *struct*s that store the wrapped element and provide the same interface as a plain reference. The object wrapper implements the `operator=()` (pseudo-code shown in Listing 5.6) for the element and calls the MAC update functions when the element is modified using this operator. The object wrappers can be disabled using compiler macros regardless of the *Secure* option. In this case, the application can avoid utilizing the object wrappers when it is not required to modify any elements inside the data structures. Moreover, we created a manually callable function for performing MAC updates. The programmer can manually call this function where the code changes the elements inside secure data structures to perform required MAC updates. This function can be used in cases not covered by the object wrappers.

```
Secure_Rb_wrapper &operator=(typename value_type::second_type other)
{
    // Check whether the new value is different
    if (*second != other)
    {
        // Update the element
        node->second = other;
        // Check if the node is the root
        if (node != tree->root)
        {
            // If the node is not the root, the new MAC is calculated for
 the node and then the MACs will be updated from that node up to the
root.

            MAC old = node->mac;
            node->mac = tree->top_mac_store.calculate_mac_tree(tree->
nonce, node->left->mac, node->right->mac, hash(node));
            // The update function needs to know whether the node is the
left child or the right child of its parent.

            if (node == node->parent->left)
                tree->Secure_Rb_tree_update_macs(node->parent, old, node
->mac, true);
            else
            {
                tree->Secure_Rb_tree_update_macs(node->parent, old, node
->mac, false);
            }
        }
    else
    {
      // If the node is the root, aside from its own MAC, the header MAC needs to be updated.
      MAC new_mac = tree->top_mac_store.calculate_mac_tree(tree->nonce, node->left->mac, node->right->mac, hash(node));
      if (tree->leftmost() == node)
        tree->top_mac_store.update_top_mac(tree->top_mac_store.calculate_header_mac(tree->nonce, new_mac, tree->rightmost()->mac, new_mac), tree->nonce);
      else
        tree->top_mac_store.update_top_mac(tree->top_mac_store.calculate_header_mac(tree->nonce, tree->leftmost()->mac, tree->rightmost()->mac, new_mac), tree->nonce);
      node->mac = new_mac;
    }
  }
  return *this;
}
```

Listing 5.6: Assignment operator in object wrappers: after checking whether the newly assigned value is different with the previous one, a process of updating the MACs is performed.

Chapter 6

Evaluation

6.1 Generality

We built the secure data structures as a part of gcc-11.1.0. Our prototype has been tested with OpenCV as an example of real-world software. The test results indicate that real-world applications that use the standard library container APIs can switch to using the secure data structures with no or relatively small changes to their source code. There are some limitations that might require changing the program code in specific cases. For instance, as described in Section 5.5, the object wrapper classes replace the plain reference when accessing values inside the data structures. The object wrapper implements the assignment and element access operations similar to a reference to the element but it requires a different syntax to access its fields. Moreover, the secure data structures themselves implement some basic operations mentioned in Chapter 4 as the Standard Template Library (STL) data structures but leave out some of the utility functions that are typically not used directly and will not work with the secure design, for instance, static functions in the red-black tree. These functions are incompatible with the secure design since they do not require any instance of the data structure to be created for performing their tasks while generating and verifying Message Authentication Codes (MACs) necessarily requires an instance of the data structure. These limitations might require changes to the program code in some cases. Nevertheless, since the API of the secure data structures is similar to the unmodified data structures for the operations described in Section 4.2, the program code does not require any changes while using those basic functions unless in specific cases where data is modified inside the data structures. In general, we can consider three cases:

- **Constant data of any type:** In this case, the data is not legitimately modifiable and therefore does not require the use of object wrappers. Accordingly, no change to the source code is required.

- **Simple data types (E.g., Integer):** This category includes the data types that are only modifiable using the assignment operator. Object wrappers can perform the required MAC updates in this case but require changes to the source code of the program. The required changes include the case where the code expects the reference as the returned type, but instead, object wrappers are returned (can be fixed by using `auto` instead of specifying the expected type).

- **Complex data types (E.g., Class):** This category of data types allows for the modification of the data in various ways, such as function calls. Our implemented object wrappers partially cover this case meaning that they can perform the required MAC updates if the data is modified through assignment operators but are unable to do so when the data is modified using methods such as function calls. This issue can be resolved by using a manual MAC update function. The details of our proposed solution are described in Section 7.2. Nevertheless, using object wrappers and manual MAC update function in this case also require changes to the source code. In addition to the previously discussed changes in the source code, calling functions or accessing fields of the complex data types through object wrappers require changes to the source code to call the MAC update function.

The above cases either require no change to the source code or require changes to make the code compatible with object wrappers or MAC update function. This partially satisfies our generality requirement **G-R1**.

Furthermore, as described in Chapter 5, the secure data structures provide the same basic operations as the unmodified data structures, which satisfies our generality requirement **G-R2**. The secure data structures leave out several functions provided by unmodified data structures such as static functions in the red-black tree as mentioned before.

Finally, we implemented two different MAC functions, one using the ARM architecture features and one using x86 features to satisfy the compatibility requirement **G-R3**. The compatibility has also been tested through successfully building the library for both architectures and using it in OpenCV.

The current prototype is however limited to single-threaded usage and does not work if secure data structure instances are accessed by threads other than the one that created them. The reason for this limitation is that in multi-threaded programs, each thread has its own set of local registers separate from other threads. Consequently, the root of a global Merkle tree cannot be stored in a single register accessible by all threads; hence, each thread will require to have its own local Merkle tree with its root stored in reserved registers. Therefore, each thread can only use and verify its own secure data structures, and any cross-thread access will result in MAC authentication failure. This issue and its possible solutions will further be discussed in Chapter 7.

6.2 Security

We demonstrate the security of our prototype by showing how both our design and implementation satisfy our security requirements in Section 3.3.2. As mentioned in the security requirements **S-R1**, any corruption of the elements in the secure data structures should be detected as soon as that element is read from them. As far as this requirement is concerned, it is assumed that the data will remain intact until it is stored in secure data structures. Accordingly, we don't make any assumption about the correctness of data before being stored in secure data structures but guarantee that the data that is read from the data structure is identical to what was inserted in. However, since the data is hashed before being used for the MAC calculation (as explained in Section 5.1), it is crucial to ensure that the data is not modified by the adversary during the hashing process. This can be ensured, for instance, using in-process isolation techniques such as Intel MPK [82]. Since the secure data structures specifically guarantee the integrity of the data while stored inside the data structures, the following proof sections assume that the input data is correct.

6.2.1 Cryptographic Security Proofs

In this section, we present a detailed game-based proof of the security of the secure stack and queue data structures' design. We leave the red-black tree out of the game-based proofs due to its high complexity but briefly discuss the security of the red-black tree by comparing it to a Merkle tree.

Accordingly, through a high-level cryptographic analysis, we demonstrate how our design provides data integrity for the stored elements.

The secure data structures are based upon a series of MAC calculations and verifications, and their security depends significantly on this. Consequently, we introduce a security game called MAC-Collision-Game$^{\mathcal{A}}_{\mathsf{MAC}_k}(q)$, which serves as a basis for security proofs of all three data structures.

```
MAC-Collision-Game$^{\mathcal{A}}_{\mathsf{MAC}_k}(q)$
1 : for $i \in 1, ..., q$
2 :     $(x, y) \leftarrow \mathcal{A}.choose()$
3 :     $\mathcal{A}.receive(\mathsf{MAC}_k(x, y))$
4 : endfor
5 : $(x'', x', y) \leftarrow A$.gen-collision()
6 : if $x' \neq x'' \wedge \mathsf{MAC}_k(x', y) = \mathsf{MAC}_k(x'', y)$
7 :     return 1
8 : endif
9 : return 0
```

As described in Chapter 5, the MAC function used in our library uses the Pointer Authentication (PA) (*pacga* instruction) for ARM implementation and AES-NI encryption for x86 Implementation. Assuming that both functions are pseudo-random with respect to their keys, the only possible way for the attacker to find a collision is through brute force. According to [55], we can show that based on the birthday paradox, the probability of the attacker finding a collision after collecting q MACs is as follows: (b is the length of the MACs)

$$Pr[\text{MAC-Collision-Game}^{\mathcal{A}}_{\mathsf{MAC}_k}(q) = 1] = 1 - \frac{2^b!}{(2^b - q)!2^{q.b}}$$

Accordingly, as shown in [55], on average, the attacker would find a collision after collecting q MACs, where:

$$q = \sqrt{\frac{\pi 2^b}{2}}$$

Therefore, if the length of the MACs (b) is long enough, we can assume that the probability of the attacker $\mathcal{A}$ finding a collision in MAC-Collision-Game$^{\mathcal{A}}_{\mathsf{MAC}_k}(q)$ is small. The value of b is 32 for ARM PA and 128 for AES-NI.

Now, we show that the probability of conducting a successful corruption attack on the secure data structures is also small since it can be reduced to finding a collision. Adversary $\mathcal{A}$ in all security games has the same capabilities as in the MAC-Collision-Game$^{\mathcal{A}}_{\mathsf{MAC}_k}(q)$.

```
Stack-Game-MAC$_1^{\mathcal{A}}(q)$
// The following 4 steps show the secure-stack initialization. Data and MAC stacks are unprotected stacks.
macs-stack $\leftarrow$ []
data-stack $\leftarrow$ []
nonce $\leftarrow_{\$} \{0,1\}^{\ell}$
mac-in-register $\leftarrow$ nonce
// pushed-values is a list that stores the values pushed into secure-stack to be later used in the attack loop.
pushed-values = []
// In the following loop, the adversary experiments with the secure-stack through pushing $n$ values and
// receiving the corresponding top MAC. After each round of $n$ push operations, the stack will be emptied to
// allow the same process again. The MAC validation steps have been omitted since the adversary's
// goal is not to attack at this stage.
for $i \in 1, ..., q$ do
    $n \leftarrow \mathcal{A}$.stack-choose-n()
    for $j \in 1, ..., n$ do
        $x \leftarrow \mathcal{A}$.stack-choose()
        // The following 4 steps show secure-stack.push(x).
        data-stack.push($x$)
        size $\leftarrow$ size + 1
        mac-stack.push(mac-in-register)
        mac-in-register $\leftarrow \mathsf{MAC}_k(x, \text{nonce}, \text{size}, \text{mac-in-register})$
        // x is inserted in the pushed-values list to keep track of the pushed values.
        pushed-values.insert($x$)
    endfor
    // The adversary receives the top MAC and then the secure-stack is emptied back to initial state.
    $\mathcal{A}$.stack-receive(mac-in-register)
    macs-stack $\leftarrow$ []
    data-stack $\leftarrow$ []
    mac-in-register $\leftarrow$ nonce
    pushed-values = []
endfor
```

Stack-Game-MAC$_1^{\mathcal{A}}(q)$

```
// In the following loop, the adversary attempts to violate the integrity by replacing data and its MAC.
// If the returned data is different from what was originally pushed, and the MACs verify, the
// attacker wins the game; otherwise,
// they lose. Since the focus is on the data value, we show the combination of the rest of the inputs by $y$.
foreach $x' \in$ pushed-values
  $(x", y) \leftarrow \mathcal{A}$.stack-attack()
  if $x' \neq x"$
    if $\mathsf{MAC}_k(x", y) =$ mac-in-register
      return 1
    else
      return 0
    endif
  endif
  // The next line updates the state and has no effect on the probability of the adversary winning the game.
  mac-in-register $\leftarrow$ macs-stack.pop()
endfor
return 0
```

```
Stack-Game-MAC$_2^{\mathcal{B}^{\mathcal{A}}}(q)$

$\mathcal{B}^{\mathcal{A}}$.stack-init()
pushed-values = []
// Here, $\mathcal{A}$ is replaced with $\mathcal{B}^{\mathcal{A}}$ which performs the updating steps for the data structure but could not
// calculate the MACs. Therefore, the MAC calculation steps are performed outside the $\mathcal{B}^{\mathcal{A}}$ operations.
for $i \in 1, ..., q$ do
    $n \leftarrow \mathcal{B}^{\mathcal{A}}$.stack-choose-n()
    for $j \in 1, ..., n$ do
        $(x, y) \leftarrow \mathcal{B}^{\mathcal{A}}$.stack-choose()
        mac-in-register $\leftarrow \mathsf{MAC}_k(x, y)$
        pushed-values.push($x$)
    endfor
    // In the next step, $\mathcal{B}^{\mathcal{A}}$ receives the top MAC and also performs the reset steps.
    $\mathcal{B}^{\mathcal{A}}$.stack-receive(mac-in-register)
endfor
// Again, $\mathcal{A}$ is replaced with $\mathcal{B}^{\mathcal{A}}$ who performs the attack and state updating steps.
$(x", y, \text{mac}) \leftarrow \mathcal{B}^{\mathcal{A}}$.stack-attack()
if $\mathsf{MAC}_k(x", y) = \text{mac}$
    return 1
else
    return 0
endif
return 0
```

```
$\mathcal{B}^{\mathcal{A}}$.stack-init()

macs-stack $\leftarrow$ []
data-stack $\leftarrow$ []
nonce $\leftarrow_\$ \{0,1\}^\ell$
mac-in-register $\leftarrow$ nonce

$\mathcal{B}^{\mathcal{A}}$.stack-choose()

$x \leftarrow \mathcal{A}$.stack-choose()
data-stack.push($x$)
size $\leftarrow$ size $+ 1$
mac-stack.push(mac-in-register)
return ($x$, (nonce, size, mac-in-register))

$\mathcal{B}^{\mathcal{A}}$.stack-choose-n()

$\mathcal{A}$.stack-choose-n()

$\mathcal{B}^{\mathcal{A}}$.stack-receive(mac)

$\mathcal{A}$.stack-receive(mac)
macs-stack $\leftarrow$ []
data-stack $\leftarrow$ []
mac-in-register $\leftarrow$ nonce
pushed-values = []

$\mathcal{B}^{\mathcal{A}}$.stack-attack()

foreach $x' \in$ pushed-values
  ($x$”, $y$) $\leftarrow \mathcal{A}$.stack-attack()
  if $x' \neq x$”
    return ($x$”, $y$, mac-in-register)
  endif
  if mac-stack.size() $> 0$
    mac-in-register $\leftarrow$ mac-stack.pop()
  endfor
  // If the adversary does not attempt to attack, at the end, the initial value that was pushed in the stack is
  // returned with the correct values which will verify and the adversary loses the game.
  return ($x'$, (nonce, size, nonce), mac-in-register)
```

Secure-Stack

We create a series of games to prove the security of the secure-stack. These games demonstrate the scenario in which the adversary attempts to change the values in a secure-stack without being recognized. To prove that the probability of success is negligible in the described scenario, we then reduce the game to a MAC-collision game by defining a second adversary. Showing the incapability of the adversary to undetectably modify the content of the secure-stack proves that the security requirement **S-R1** is being satisfied in this data structure. Below is a more detailed description of each game and the final proof of security. For simplicity, in these games, the MAC is calculated over the data instead of the hash of the data. Moreover, in all following security games, the nonce of the data structures is a random value with the length of ℓ bits.

Stack-Game-MAC$_\mathbf{1}^{\mathcal{A}}$(q). We define Stack-Game-MAC$_1^{\mathcal{A}}(q)$, an attack game against the integrity of the secure-stack. In this game, we assume an adversary $\mathcal{A}$ who has arbitrary read/write access to the memory as described in Section 3.2. The game consists of two parts as shown in Stack-Game-MAC$_1^{\mathcal{A}}(q)$. In the first two loops, the adversary chooses values to be pushed to the secure-stack and receives the corresponding top MAC (which is securely stored in a reserved register). The attacker can also empty the secure-stack as needed to start again from an empty stack (to get different calculated MACs for each stack state). This step of the game allows the adversary to collect the corresponding MACs for different data values, which can later be used for the attack.

The second loop demonstrates the attack. The attacker attempts to replace at least one of the elements in the secure-stack with another value while the MACs still verify successfully. If the attacker does not succeed in performing such an attack, they lose the game.

Stack-Game-MAC$_\mathbf{2}^{\mathcal{B}^{\mathcal{A}}}$(q). We now introduce a second game, Stack-Game-MAC$_2^{\mathcal{B}^{\mathcal{A}}}(q)$, which can be reduced to the MAC-Collision-Game$_{\mathsf{MAC}_k}^{\mathcal{A}}(q)$ game. For this purpose, we introduce a new attacker, $\mathcal{B}^{\mathcal{A}}$, defined in $\mathcal{B}^{\mathcal{A}}$ functions, and replace $\mathcal{A}$ in Stack-Game-MAC$_1^{\mathcal{A}}(q)$ with $\mathcal{B}^{\mathcal{A}}$.

6.2.1 Lemma.

$$Pr[\textit{Stack-Game-MAC}_1^{\mathcal{A}}(q) = 1] \leq Pr[\textit{Stack-Game-MAC}_2^{\mathcal{B}^{\mathcal{A}}}(q) = 1].$$

Proof. The transition from the first game to the second game only includes wrapping adversary $\mathcal{A}$ with $\mathcal{B}^{\mathcal{A}}$ who has the same functionality as $\mathcal{A}$ but also performs the computations required for secure-stack operations. For instance, steps such as initializing or updating the state of the data structure are performed by $\mathcal{B}^{\mathcal{A}}$. These steps are required for the correct functionality of the data structure but have no effect on the probability of the adversary winning the game. Accordingly, $\mathcal{A}$ winning the first game implies that $\mathcal{B}^{\mathcal{A}}$ can also win the second game (since $\mathcal{B}^{\mathcal{A}}$ uses $\mathcal{A}$ in order to perform the attack). Consequently, we can conclude the given bound. $\square$

6.2.2 Lemma.

$$Pr[\textit{Stack-Game-MAC}_2^{\mathcal{B}^{\mathcal{A}}}(q) = 1] \leq Pr[\textit{MAC-Collision-Game}_{\mathsf{MAC}_k}^{\mathcal{A}}(q) = 1]$$

Proof. We can reduce the Stack-Game-MAC$_2^{\mathcal{B}^{\mathcal{A}}}(q)$ to MAC-Collision-Game$_{\mathsf{MAC}_k}^{\mathcal{A}}(q)$. From lines 16–17 of Stack-Game-MAC$_2^{\mathcal{B}^{\mathcal{A}}}(q)$, winning Stack-Game-MAC$_2^{\mathcal{B}^{\mathcal{A}}}(q)$ requires that the adversary ($\mathcal{B}^{\mathcal{A}}$) finds a collision in the top MACs so that the authentication can pass successfully when comparing the MAC of the proposed top element by the adversary with the MAC in the register. Moreover, since adversary $\mathcal{B}^{\mathcal{A}}$ winning the game implies that $\mathcal{A}$ has found a collision (we can instantiate the collision game with $\mathcal{B}^{\mathcal{A}}$ with corresponding calls from MAC-Collision-Game$_{\mathsf{MAC}_k}^{\mathcal{A}}(q)$ adversary interface instead of $\mathcal{A}$ and win the game), we can conclude the above bound. $\square$

Theorem 1 (Security of the Secure Stack)**.**

$$Pr[\textit{Stack-Game-MAC}_1^{\mathcal{A}}(q) = 1] \leq Pr[\textit{MAC-Collision-Game}_{\mathsf{MAC}_k}^{\mathcal{A}}(q) = 1]$$

Proof. We can conclude this by applying Lemma 6.2.2 to Lemma 6.2.1 to get the above bound.

Accordingly, the probability of the adversary $\mathcal{A}$ corrupting the secure stack data structure is less than the probability of wining the MAC-Collision-Game$_{\mathsf{MAC}_k}^{\mathcal{A}}(q)$, and hence as long as the probability of finding a collision is negligible, the secure stack data structure is secure. $\square$

```
Queue-Game-Data-MAC$_1^{\mathcal{A}}(q)$
// The following 6 steps show the secure-queue initialization. The data and MAC queues are unprotected.
macs-queue $\leftarrow$ []
data-queue $\leftarrow$ []
nonce $\leftarrow_{\$} \{0,1\}^{\ell}$
back-index $\leftarrow 0$
front-index $\leftarrow 1$
mac-in-register[] $\leftarrow \mathsf{MAC}_k$(nonce, back-index, front-index)
// enqueued-values is a queue that keeps track of the values enqueued to be used in the attack loop.
enqueued-values = []
// In the following loop, the adversary experiments with the secure-queue through enqueue
// operation in order to collect corresponding MACs for different data values. The MAC validation steps
// have been omitted since the adversary's goal is not to attack at this stage.
for $i \in 1, ..., q$ do
    $x \leftarrow \mathcal{A}$.queue-choose-data-attack()
    // The following 4 steps show secure-queue.enqueue(x).
    data-queue.enqueue($x$)
    back-index $\leftarrow$ back-index + 1
    mac-queue.enqueue($\mathsf{MAC}_k$($x$, (nonce, back-index)))
    mac-in-register $\leftarrow \mathsf{MAC}_k$(nonce, back-index, front-index)
    enqueued-values.enqueue($x$)
    // The adversary receives the corresponding MAC
    $\mathcal{A}$.queue-receive(mac-queue.back())
endfor
// In the following loop, the adversary attempts to violate the integrity by replacing data, and its mac.
// . If the returned data is different from what was originally enqueued, and the MAC verifies, the
// attacker wins the game, otherwise, they lose. Since the adversary is not attacking the indexes in
//  this game, we have omitted the verification for the index mac.
```

Queue-Game-Data-MAC$_1^{\mathcal{A}}(q)$

```
foreach $x' \in$ enqueued-values
  $(x", mac) \leftarrow \mathcal{A}$.queue-data-attack()
  if $x' \neq x"$
    if $mac = \mathsf{MAC}_k(x", (\text{nonce}, \text{front-index}))$
      return 1
    else
      return 0
    endif
  endif
  // Updating the front-index for the data MAC validation in the next iteration.
  front-index $\leftarrow$ front-index $+ 1$
endfor
return 0
```

```
Queue-Game-Data-MAC$_2^{\mathcal{A}}(q)$
macs-queue $\leftarrow$ []
data-queue $\leftarrow$ []
nonce $\leftarrow_\$ \{0,1\}^\ell$
back-index $\leftarrow 0$
front-index $\leftarrow 1$
mac-in-register $\leftarrow \mathsf{MAC}_k$(nonce, back-index, front-index)
enqueued-values = []
for $i \in 1, ..., q$ do
  $x \leftarrow \mathcal{A}$.queue-choose-data-attack()
  data-queue.enqueue($x$)
  back-index $\leftarrow$ back-index + 1
  // We replaced the MAC function from previous games with RO which is a Random Oracle.
  mac-queue.enqueue($RO(x$, (nonce, back-index)))
  mac-in-register $\leftarrow RO$(nonce, back-index, front-index)
  enqueued-values.enqueue($x$)
  $\mathcal{A}$.queue-receive(mac-queue.back())
endfor
foreach $x' \in$ enqueued-values
  ($x$", $mac$) $\leftarrow \mathcal{A}$.queue-data-attack()
  if $x' \neq x$"
    if $MAC = RO(x$", (nonce, front-index))
      return 1
    else
      return 0
    endif
  endif
  front-index $\leftarrow$ front-index + 1
endfor
return 0
```

```
$\mathcal{B}^{\mathcal{A}}$.queue-init()

macs-queue $\leftarrow$ []
data-queue $\leftarrow$ []
$nonce \leftarrow_{\$} \{0,1\}^{\ell}$
back-index $\leftarrow 0$
front-index $\leftarrow 1$
enqueued-values = []

$\mathcal{B}^{\mathcal{A}}$.queue-choose()

($x$, enqueue) $\leftarrow \mathcal{A}$.queue-choose-index-attack()
if enqueue
   data-queue.enqueue($x$)
   back-index $\leftarrow$ back-index + 1
   enqueued-values.enqueue($x$)
   return $x$, (nonce, back-index),
   (nonce, back-index, front-index)
else
   data-queue.dequeue()
   mac-queue.dequeue()
   front-index $\leftarrow$ front-index + 1
   enqueued-values.dequeue()
   return data-queue.front(), (nonce, front-index), (nonce, back-index, front-index)
endif

$\mathcal{B}^{\mathcal{A}}$.queue-receive(mac)

$\mathcal{A}$.queue-receive($mac_1$, $mac_2$)
```

```
$\mathcal{B}^{\mathcal{A}}$.queue-attack()
foreach $x' \in$ enqueued-values
    $(x'', mac, k_1, k_2) \leftarrow \mathcal{A}$.queue-index-attack()
    if $x' \neq x''$
        return $(x'', mac,$ mac-in-register, (nonce, $k_1$),
        (nonce, $k_1, k_2$))
    endif
    // Updating the state of the authenticated queue.
    front-index $\leftarrow$ front-index $+ 1$
endfor
return (enqueued-values.front(), mac-in-register, (nonce, front-index), (nonce, back-index,
front-index))
```

```
Queue-Game-Index-MAC$_1^{\mathcal{A}}(q)$
// The following 6 steps show the secure-queue initialization. Data and MAC queues are unprotected.
macs-queue $\leftarrow$ []
data-queue $\leftarrow$ []
nonce $\leftarrow\!\!\$\ \{0,1\}^\ell$
back-index $\leftarrow 0$
front-index $\leftarrow 1$
mac-in-register $\leftarrow \mathsf{MAC}_k$(nonce, back-index, front-index)
// enqueued-values is a list that stores the values enqueued into secure-queue to be used in the attack loop.
enqueued-values = []
// In the following loop, the adversary experiments with the secure-queue through enqueue and dequeue
// operations in order to collect corresponding MACs for different index values.
for $i \in 1, ..., q$ do
  ($x$, enqueue) $\leftarrow \mathcal{A}$.queue-choose-index-attack()
  if enqueue
    // The following 4 steps show secure-queue.enqueue(x).
    data-queue.enqueue($x$)
    back-index $\leftarrow$ back-index + 1
    mac-queue.enqueue($\mathsf{MAC}_k$($x$, (nonce, back-index)))
    mac-in-register $\leftarrow \mathsf{MAC}_k$(nonce, back-index, front-index)
    enqueued-values.enqueue($x$)
    $\mathcal{A}$.queue-receive(mac-queue.back(), mac-in-register)
  else
    // The following 4 steps show secure-queue.dequeue().
    data-queue.dequeue()
    mac-queue.dequeue()
    front-index $\leftarrow$ front-index + 1
    mac-in-register $\leftarrow \mathsf{MAC}_k$(nonce, back-index, front-index)
    enqueued-values.dequeue()
    $\mathcal{A}$.queue-receive(mac-queueu.front(), mac-in-register)
  endif
endfor
```

Queue-Game-Index-MAC$_1^{\mathcal{A}}(q)$

```
// In the following loop, the adversary attempts to violate the integrity by replacing data, its mac, and the
// indexes. If the returned data is different from what was originally enqueued, and the MAC s verify, the
// attacker wins the game, otherwise, they lose.
foreach $x' \in$ enqueued-values
  $(x", mac, k_1, k_2) \leftarrow \mathcal{A}$.queue-index-attack()
  if $x' \neq x"$
    if $mac = \mathsf{MAC}_k(x", (\text{nonce}, k_1)) \wedge \mathsf{MAC}_k(\text{nonce}, k_2, k_1) = \text{mac-in-register}$
      return 1
    else
      return 0
    endif
  endif
  front-index $\leftarrow$ front-index $+ 1$
  mac-in-register $\leftarrow \mathsf{MAC}_k(\text{nonce}, \text{back-index}, \text{front-index})$
endfor
return 0
```

```
Queue-Game-Index-MAC$_2^{\mathcal{B}^{\mathcal{A}}}(q)$
1 :  $\mathcal{B}^{\mathcal{A}}$.queue-init()
2 :  // In this loop, we replace the enqueue and dequeue steps with the $\mathcal{B}^{\mathcal{A}}$.queue-choose()
3 :  // function which performs the same steps.
4 :  for $i \in 1, ..., q$ do
5 :      $(x, y, y') \leftarrow \mathcal{B}^{\mathcal{A}}$.queue-choose()
6 :      $\mathcal{B}^{\mathcal{A}}$.queue-receive($\mathsf{MAC}_k(x, y), \mathsf{MAC}_k(y')$)
7 :  endfor
8 :  // In this part, similar to the previous steps $\mathcal{A}$ is replaced with $\mathcal{B}^{\mathcal{A}}$ who also performs the
9 :  // authenticated-queue related steps and the loop. The step for updating the mac-in-register
10 : // is removed since it's just a state update and doesn't affect the probability.
11 : $(x'', mac_1, mac_2, y, y') \leftarrow \mathcal{B}^{\mathcal{A}}$.queue-attack()
12 : if $mac_1 = \mathsf{MAC}_k(x'', y) \wedge \mathsf{MAC}_k(y') = mac_2$
13 :     return 1
14 : else
15 :     return 0
16 : endif
17 : return 0
```

Secure-Queue

We prove the security of the queue by considering the front and back elements separately from the elements in the middle. Regarding the front and back elements, since their MACs are included in the top MAC, the only way for the adversary to modify them is to find another element with the same MAC or a set of a new element and its corresponding MAC which will result in the same top MAC. Accordingly, modifying the back and front elements requires finding a collision in MACs and therefore, the probability of such an attack is negligibly small.

Next, we can prove the security of the secure-queue regarding the middle elements by considering them as a queue on their own. For simplicity, we can assume that the indices of the front and back elements in the inside queue are included in the top MAC (since they are the same as the actual front and back indices but transformed by 1). In this case, we can consider two scenarios. In the first scenario, Queue-Game-Index-MAC, the adversary attempts to change the indices in order to substitute one entry with another. The second set, Queue-Game-Data-MAC games, covers the case in which the attacker tries to violate the integrity of the secure-queue by replacing an element and its MAC with a different value without changing the indices. Similar to the secure-stack, proving the inability of the adversary in modifying the content of the secure-queue shows that the security requirement **S-R1** is satisfied for this data structure. In these games, the MAC is calculated over the data instead of the hash of the data.

Queue-Game-Index-MAC$_1^{\mathcal{A}}$(q). We define Queue-Game-Index-MAC$_1^{\mathcal{A}}(q)$ to be an attacking game against the integrity of a secure-queue. Similar to the secure-stack, we assume an adversary $\mathcal{A}$ who has arbitrary read/write access to memory as described in the Section 3.2. Accordingly, we define the game such that $\mathcal{A}$ chooses values to be enqueued in or dequeued from the secure-queue and then receives the corresponding MACs. After q rounds of performing this process, the attacker tries to attack the secure-queue by replacing at least one of the elements with a different value without being noticed. This is demonstrated by the second half of the game, in which the adversary chooses values to replace the actual secure-queue's content with them. If the attacker can replace a value such that the check passes successfully, they win. The game is shown in Queue-Game-Index-MAC$_1^{\mathcal{A}}(q)$.

Queue-Game-Index-MAC$_2^{\mathcal{B}^{\mathcal{A}}}$(q). Similar to the approach used for the secure-stack, we create a second game (shown in Queue-Game-Index-MAC$_2^{\mathcal{B}^{\mathcal{A}}}(q)$) which can be reduced to the MAC-Collision-Game$_{\mathsf{MAC}_k}^{\mathcal{A}}(q)$ game. For this purpose, we create a new adversary, $\mathcal{B}^{\mathcal{A}}$, defined in $\mathcal{B}^{\mathcal{A}}$ functions, and replace $\mathcal{A}$ in Queue-Game-Index-MAC$_1^{\mathcal{A}}(q)$ with $\mathcal{B}^{\mathcal{A}}$.

6.2.3 Lemma.

$$Pr[\textit{Queue-Game-Index-MAC}_1^{\mathcal{A}}(q) = 1] \leq Pr[\textit{Queue-Game-Index-MAC}_2^{\mathcal{B}^{\mathcal{A}}}(q) = 1].$$

Proof. The transition from the first game to the second game only includes replacing adversary $\mathcal{A}$ with $\mathcal{B}^{\mathcal{A}}$ who has the same functionality as $\mathcal{A}$ but also performs the computations required for secure-queue operations such as initializing or updating the state. These steps only represent the correct functionality of the data structure but have no effect on the probability of the adversary winning the game. Accordingly, we can conclude the given bound since $\mathcal{A}$ winning the first game implies that $\mathcal{B}^{\mathcal{A}}$ can also win the second game ($\mathcal{B}^{\mathcal{A}}$ uses $\mathcal{A}$ in order to perform the attack). $\square$

6.2.4 Lemma.

$$Pr[\textit{Queue-Game-Index-MAC}_2^{\mathcal{B}^{\mathcal{A}}}(q) = 1] \leq Pr[\textit{MAC-Collision-Game}_{\mathsf{MAC}_k}^{\mathcal{A}}(q) = 1]$$

Proof. We can reduce the Queue-Game-Index-MAC$_2^{\mathcal{B}^{\mathcal{A}}}(q)$ to MAC-Collision-Game$_{\mathsf{MAC}_k}^{\mathcal{A}}(q)$. From lines 13-15 of Queue-Game-Index-MAC$_2^{\mathcal{B}^{\mathcal{A}}}(q)$, winning Queue-Game-Index-MAC$_2^{\mathcal{B}^{\mathcal{A}}}(q)$ requires the adversary ($\mathcal{B}^{\mathcal{A}}$) to find a collision in index MACs. The reason is that for the authentication to pass successfully, the MAC of the proposed indices by the adversary should be equal to the MAC in the register (in this game, the indices proposed by the adversary are necessarily different from the correct indices since this will allow the adversary to replay previous elements' MACs). Moreover, since $\mathcal{B}^{\mathcal{A}}$ wining the game implies that $\mathcal{A}$ has found a collision (we can replace $\mathcal{A}$ with $\mathcal{B}^{\mathcal{A}}$ in the collision game and win the game), we can conclude the above bound. □

Queue-Game-Data-MAC$_1^{\mathcal{A}}$(q). This game is similar to Queue-Game-Index-MAC$_1^{\mathcal{A}}(q)$ with one minor difference. In this game, removing elements from the secure-queue does not produce new data MACs. Hence, the first loop only enqueues the values chosen by the adversary, and there is no need for a dequeue operation. The game is shown in Queue-Game-Data-MAC$_1^{\mathcal{A}}(q)$.

Queue-Game-Data-MAC$_2^{\mathcal{A}}$(q). We now transform the Queue-Game-Data-MAC$_1^{\mathcal{A}}(q)$ game to Queue-Game-Data-MAC$_2^{\mathcal{A}}(q)$ by replacing the MAC function with a random oracle and keeping the rest of the game exactly the same. Queue-Game-Data-MAC$_2^{\mathcal{A}}(q)$ shows the details of this game.

6.2.5 Lemma.

$$Pr[\textit{Queue-Game-Data-}MAC_2^{\mathcal{A}}(q) = 1] = Pr[\textit{Queue-Game-Data-}MAC_1^{\mathcal{A}}(q) = 1].$$

Proof. In this scenario, the adversary attempts to replace an element with another element but with the same index. Accordingly, in order to pass the authentication, the adversary needs to find the corresponding MAC over the nonce, the new element's value, and the index. Considering the fact that each instance of the data structure has its own unique nonce, and each index only appears once in the lifetime of the data structure, we can conclude that the MAC required by the attacker has not been previously calculated (the MAC is freshly sampled from a known distribution, and this is the definition of a random oracle). Therefore, the attacker is not able to replay a previous MAC with a higher probability than a random guess, and the probability of the adversary winning both games is the same. □

6.2.6 Lemma.

$$Pr[\textit{Queue-Game-Data-}MAC_2^{\mathcal{A}}(q) = 1] = 2^{-b}.$$

Proof. The adversary needs to find the corresponding output for a query from a random oracle over input values that have not been seen before. Therefore, since the adversary does not have direct access to such a random oracle outside the game structure, the only way is to guess the output which leads to the above probability. □

Theorem 2 (Security of the Secure Queue)**.**

$$Pr[\textit{Queue-Game-Data-MAC}_1^{\mathcal{A}}(q)] + Pr[\textit{Queue-Game-Index-MAC}_1^{\mathcal{A}}(q)]$$
$$\leq Pr[\textit{MAC-Collision-Game}_{\mathsf{MAC}_k}^{\mathcal{A}}(q) = 1] + 2^{-b}$$

Proof. We can replace the Queue-Game-Data-MAC$_1^{\mathcal{A}}(q)$ with Queue-Game-Data-MAC$_2^{\mathcal{A}}(q)$ based on Lemma 6.2.5. Then we can replace Queue-Game-Data-MAC$_2^{\mathcal{A}}(q)$ with 2^{-b} according to Lemma 6.2.6. Therefore, we can conclude that the probability of the adversary winning Queue-Game-Data-MAC$_1^{\mathcal{A}}(q)$ is at most 2^{-b}.

Similarly, we replace Queue-Game-Index-MAC$_1^{\mathcal{A}}(q)$ with Queue-Game-Index-MAC$_2^{\mathcal{B}^{\mathcal{A}}}(q)$ according to Lemma 6.2.3 , then apply the bound from Lemma 6.2.4. Consequently, the probability of the adversary winning the Queue-Game-Index-MAC$_1^{\mathcal{A}}(q)$ is at most equal to winning the MAC-Collision-Game$_{\mathsf{MAC}_k}^{\mathcal{A}}(q)$.

Therefore, the probability of the attacker corrupting the data structure either through data MACs or index MACs is less than Pr[MAC-Collision-Game$_{\mathsf{MAC}_k}^{\mathcal{A}}(q) = 1] + 2^{-b}$. As a result, the secure queue data structure is secure as long as the probability of the adversary $\mathcal{A}$ finding a collision is negligible. □

Secure-Tree

The security of the secure-rb-tree, which is the basis for creating a secure map data structure, can be proven similarly to the stack and queue. However, due to the random access property of the tree, which requires MAC updates for multiple elements when performing operations, proving the security of a tree is noticeably more complicated than the stack or queue.

Accordingly, we can instead conclude the security of the secure red-black tree since it is cryptographically a Merkle tree. As proved by [28], we can assume that a Merkle tree is secure as long as the probability of the adversary finding a collision in the hash function it uses is reasonably low. Consequently, we can conclude the security of the red-black tree assuming the probability of finding a collision in MACs is small.

6.2.2 Implementation Security

We have implemented the secure data structures considering the security requirements mentioned in Chapter 3.

The security games in the Section 6.2.1 assume that the top MAC for each data structure is stored securely. As mentioned in Chapter 5, we use a Merkle tree along with reserved registers to provide such secure storage for the top MACs. Storing the top MACs in the Merkle tree and the root of the Merkle tree in a reserved register ensures the integrity of the top MACs assuming the Merkle tree implementation is secure.

Furthermore, as mentioned in Section 5.5, we added object wrappers to our implementation, which allow for updating the MACs once the returned objects from data structures' API calls are modified. Object wrappers are not included in the cryptographic security proofs since they are specific to the implementation, not the high-level protocol.

Our security games for secure-queue assume that the internal elements in the queue are immutable. This assumption prevents the attacker from performing a MAC reuse attack on the updated elements inside the queue using their previous state. However, the elements at the front and back of the queue are modifiable using object wrappers since our design prevents MAC reuse attacks on them due to including their MACs in the top MAC. The unmodified implementation of the queue allows for modifying any element inside the queue using the references. We could design the secure-queue such that it would hold the same property but with the cost of sacrificing the performance. One efficient way to achieve this property is to use a Merkle tree to store the elements of the queue, which will increase the overhead of the operations from $O(1)$ in the unmodified version to $O(log(n))$ where n is the number of elements in the queue. However, in order to achieve a similar performance overhead to the unmodified queue ($O(1)$), we limited the modification of the stored elements in our prototype to the front and back elements only. This assumption matches the functionality of a queue data structure.

Finally, as mentioned in the security requirements **S-R2**, the implementation of the secure data structures should minimize the time windows among the operation steps which can be exploited by an attacker. We minimize the attack window in our implementation by performing MAC verifications before any update to each element or its MAC. Additionally, we store the new and old roots of the recently updated sub-tree in a reserved register when updating the MACs, allowing us to safely verify the upper elements before updating their MACs. This ensures that it would be infeasible for an attacker to corrupt data while in transfer between states.

6.3 Performance

We have tested the performance of our implementation as part of gcc-11.1.0 by testing the built library with Google benchmark and real-world applications such as OpenCV.

We allow the developers to choose between secure and unmodified data structures in each instance of declaring a stack, queue, or red-black tree (map). This option allows them to increase efficiency by using the unmodified data structures where they are not storing sensitive or valuable data. Below we present the details for the performance test.

6.3.1 Microbenchmarks

We used the Google benchmark library [38] along with the LLVM container benchmarks [52] to test the performance of individual operations in the secure data structures outside real-world applications. The reason for using LLVM benchmarks is that it implements specific benchmark tests for the containers using the Google benchmark library which were similar to what we required for testing our prototype while the test suites in gcc did not implement such benchmarks.

Google benchmarks allow for creating test functions and measuring the execution time. The tests will be iterated through several times to make sure that the result will be statistically stable. We set the number of iterations to the fixed number of 1000 for all our microbenchmark tests to make it easier to compare the results. Based on several previous test runs without fixing the iteration number, we concluded that 1000 iterations are enough to get stable results.

In order to test the overhead of a single operation, we measured the overhead of several sequential operations in both secure and unmodified data structures. The reason for performing the operation more than one time is to reduce the noise in the results by increasing the execution time. Moreover, we performed a warm-up round of executing the operations to eliminate the effect of initial memory allocations on the results. Each benchmark can also be set to be executed multiple times, and the benchmark library reports the mean and standard deviation, which better reflects the overall behavior of the benchmark compared to the result of a single run. We set the repetition number to 10 for our benchmarks.

The microbenchmark results for the stack are shown in Table 6.1. In these tests, 500 pushes and 500 pops have been performed. The comparison shows that operations of the secure data structures are slower than those of the unmodified ones, and the overhead varies from 153 to 1502 times slower based on the input type. The overhead for the *Integer*

Benchmark	Mean Time	Standard Deviation
secure-stack (Integer input)	16853.65 μs	1554.99 μs
stack (Integer input)	11.21 μs	0.03 μs
secure-stack (String input)	17288.43 μs	1530.19 μs
stack (String input)	22.61 μs	0.4 μs
secure-stack (Vector$< int >$ input)	18124.69 μs	1564.43 μs
stack (Vector$< int >$ input)	116.94 μs	0.62 μs

Table 6.1: Microbenchmark results for a single stack data structure performing 500 push and 500 pop operations. The Google benchmark library computes the average time over 1000 repetitions (10^6 operations in total). We then report the mean and standard deviation calculated over ten such averages.

input type is higher than the two other tested types. The reason for this difference is that the *push* operations create a copy of the data to store, and the copy process is much faster for the *Integer* type compared to the *String* or *Vector*. Therefore, we notice that the execution time for the unmodified data structures is noticeably lower for the *Integer* type. The secure data structures are also affected by this; however, since the mean time is already much higher for them, the difference is less significant. Accordingly, the relative overhead is higher in this case.

Since the tree data structure is an internal data structure and does not provide a public API, we created tests for a map data structure using the tree. The microbenchmarks for the queue and map (using red-black tree) were tested similarly to the stack, and the results are presented in Tables 6.2 and 6.3. The secure-queue has been tested through 500 enqueues and 500 dequeues and shows an overhead of 142 to 1508 times the baseline. The results for performing 10 operations(5 inserts and 5 removes) on secure and non-secure maps also suggest that a secure-map generally has a higher overhead than the two other data structures when considering the same input type (the secure-map takes 3676 times as long as the unmodified map to perform these operations), which is expected because, for each operation in the tree, several MACs need to be calculated and verified, which is a costly operation.

Moreover, the microbenchmark results for the case of having i instances of the stack data structures indicate a growing overhead of order $O(log(i))$ for the operations in secure data structures. This overhead was expected since secure data structures use the Merkle tree to store their top MACs, and all operations on data structures rely on obtaining that MAC from the Merkle tree, which causes the $O(log(i))$ overhead. The microbenchmark results are displayed in Table 6.4. Even though the microbenchmarks show considerable

Benchmark	Mean Time	Standard Deviation
secure-queue (Integer input)	16793.65 μs	1216.12 μs
queue (Integer input)	11.13 μs	0.05 μs
secure-queue (String input)	17060.97 μs	1224.29 μs
queue (String input)	23.24 μs	0.08 μs
secure-queue (Vector$< int >$ input)	17868.11 μs	1261.28 μs
queue (Vector$< int >$ input)	124.76 μs	1.73 μs

Table 6.2: Microbenchmark results for a single queue data structure which performs 500 enqueue and 500 dequeue operations. The Google benchmark library computes the average time over 1000 repetitions (10^6 operations in total). We then report the mean and standard deviation calculated over ten such averages.

Benchmark	Mean Time	Standard Deviation
secure-map (String key and data)	553959.23 μs	85048.1 μs
map (String key and data)	150.63 μs	0.91 μs

Table 6.3: Microbenchmark results for a single map data structure which uses a tree and performs 5 insert and 5 remove operations. The Google benchmark library computes the average time over 1000 repetitions (10^4 operations in total). We then report the mean and standard deviation calculated over ten such averages.

Benchmark	Number of Instances	Mean Time	Standard Deviation
secure-stack	1	17008.72 μs	1524.04 μs
secure-stack	2	18439.05 μs	3379.15 μs
secure-stack	4	21897.05 μs	73.03 μs
secure-stack	8	26055.28 μs	39.62 μs
secure-stack	16	31020.69 μs	82.63 μs
secure-stack	32	36777.37 μs	3609.44 μs
secure-stack	64	47938.77 μs	65.24 μs
secure-stack	128	65405.72 μs	131.06 μs
stack	1	12.03 μs	0.19 μs
stack	2	11.95 μs	0.21 μs
stack	4	11.82 μs	0.18 μs
stack	8	11.92 μs	0.15 μs
stack	16	11.92 μs	0.14 μs
stack	32	11.93 μs	0.19 μs
stack	64	11.90 μs	0.19 μs
stack	128	11.90 μs	0.20 μs

Table 6.4: Microbenchmark results measuring the overhead caused by the Merkle tree. Number of instances indicate the number of created data structures.

Testsuite	Number of maps
opencv_perf_calib3d	1
opencv_perf_dnn	1531
opencv_perf_dnn_superres	2
opencv_perf_features2d	1
opencv_perf_gapi	9791
opencv_perf_imgcodecs	1
opencv_perf_imgproc	1
opencv_perf_line_descriptor	4
opencv_perf_objdetect	122
opencv_perf_optflow	2
opencv_perf_photo	1
opencv_perf_reg	1
opencv_perf_rgbd	1
opencv_perf_stereo	2
opencv_perf_stitching	1
opencv_perf_superres	2
opencv_perf_tracking	2
opencv_perf_video	2
opencv_perf_videoio	3
opencv_perf_xfeatures2d	1
opencv_perf_ximgproc	2
opencv_perf_xphoto	1
opencv_perf_core	3

Table 6.5: Data structure instances in OpenCV performance tests

overhead, they reflect the worst case where a program only creates and uses multiple data structures and does no other task. Accordingly, these results do not reflect the overhead caused by secure data structures in real-world applications that use the data structures more sparsely while performing other tasks. In the next section, we present the results of overhead measurements when using secure data structures in such applications.

6.3.2 Real-World Applications

OpenCV is a Computer Vision library implemented in C++ which uses the standard library data structures. We built OpenCV from source code with our modified C++ standard

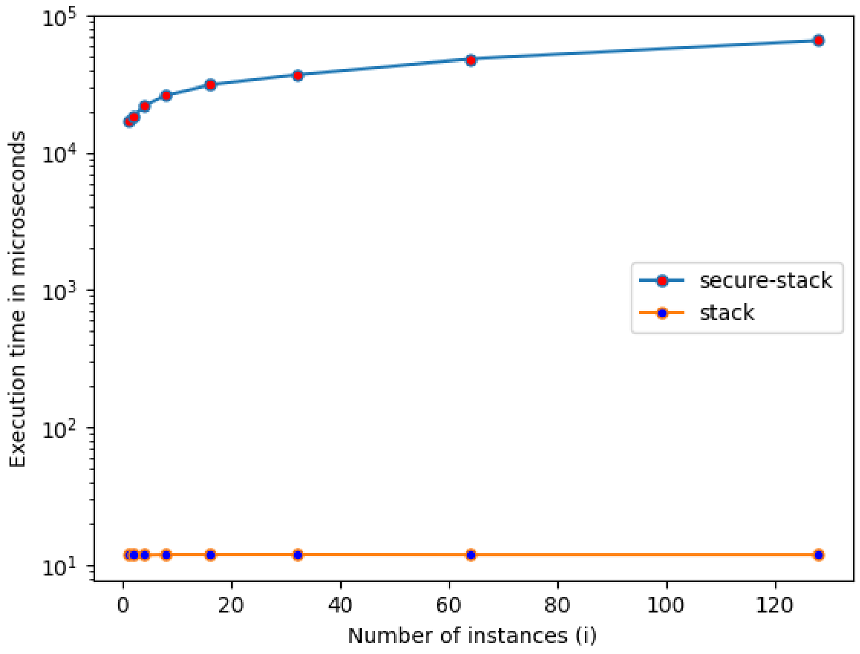

Figure 6.1: The microbenchmark results for performing 500 pushes and 500 pops on a single stack instance with total number of instances ranging from 1 to 128. The graph shows how the execution time increases with order O(log(i)) for a secure-stack while it is fairly constant in the unmodified stack.

library in gcc. The OpenCV source code provides a set of performance tests that could be useful for testing the performance of our secure data structures. None of the performance tests in OpenCV uses queues and only one of them (*opencv_perf_gapi*) uses 9789 stacks. However, most of the performance tests use the tree data structure. The number of trees used in each performance test is shown in table Table 6.5.

Although the performance tests in OpenCV do not use queues, due to the similarity of the design for the secure stack and queue, we do not expect it to have drastically different performance characteristics. More specifically, since both stack and queue perform a constant number of MAC calculations for each operation, they are expected to cause relatively close overheads. The previously stated microbenchmark results in tables 6.1 and 6.2 confirm the similarity in their overheads.

In order to compare the performance of the secure data structures with the baseline, we used the OpenCV performance tests since they were already created to measure the performance and included a reasonable number of data structures. Each performance test consists of multiple test cases varying from 4 to 10040 tests. The performance tests run the test cases multiple times and report the execution times. We ran all performance tests and calculated the ratio of execution time for secure data structures to the baseline for each test case and then calculated the geometric mean of the overhead ratios as the overall performance overhead of our prototype. The results show an overhead of around **3.4%** compared to the baseline.

The mentioned overhead is calculated without using object wrappers and without enforcing exceptions in case of MAC authentication failures. Therefore, the mentioned overhead is a lower bound estimation of the actual overhead in real-world applications. Nevertheless, we counted the cases in the source code of OpenCV where the references to elements inside data structures are used to modify them or call their functions. Such cases in which the required MAC updates are skipped do not exceed 100 cases in total but could be executed more then once. Almost all mentioned cases happen when using map data structure (In OpenCV, the stack `top()` functions are followed by `pop()` and therefore do not require any MAC updates since the element is no longer inside the stack). Consequently, assuming each MAC update process takes at most as much time as an insertion operation (taking 55.4μ for each insertion based on the microbenchmarking results), we can conclude that the MAC updates would approximately add 5540μ to the whole performance testing process which is negligible compared to the overall time. Accordingly, the estimated lower bound for the overhead is close to the actual overhead, including the use of object wrappers.

We also investigated how the multi-threading issue explained in Section 6.1 affects the usability of our prototype. Since the current prototype's functionality is limited to the cases where each data structure is only accessed by its own thread (the thread that created the data structure), if the applications require several threads accessing the same data structure, they will get false-positive results (MAC authentication failures while there is no malicious modification of the elements). To check whether this limitation affects our prototype's compatibility, we added counters that count how many times each data structure is accessed by other threads. Using the mentioned counters we verified that this limitation is not problematic in OpenCV as the data structures are only accessed by their own threads.

6.3.3 Memory Overhead

Assuming that the element size is not smaller than the MAC size, in the worst case, our approach doubles the program's memory space by storing a MAC for each element in the data structures. However, based on the size of the elements and MACs, the memory overhead could be lower than this value if the data size is larger than the MACs, or higher in systems that don't have alignment requirements.

Moreover, since we maintain a global Merkle tree to store the top MACs for all instances of the data structures, we also require an additional $O(i)$ memory space where i is the total number of data structure instances.

In terms of memory overhead, this approach is comparable with methods that use shadow memory [20] to store metadata.

6.3.4 Comparing SafeDS With Similar Approaches

Secure data structures can be compared with memory isolation approaches such as Intel Memory Protection Keys (MPK) [42]. MPK is a hardware primitive that can be used to set page table permissions from user-space. An example of using MPK for memory isolation is in ERIM [82]. Using MPK is efficient; therefore, applications such as ERIM only add less than 1% overhead. However, MPK has several considerable limitations: First, MPK assigns 4-bit keys to each page table. Therefore, there are only 15 possible keys (0 is used as the default key) which might not be sufficient in some use cases. Moreover, using MPK requires adding specific instructions to set the keys and permissions, meaning that the source code of the programs would require several changes.

SafeDS is comparable to MPK considering the required changes to the source code of the applications. However, our secure data structures provide stronger security guarantees compared to MPK; for example, writable pages are not protected from memory errors when using MPK. In addition, secure data structures do not require separate allocators for the memory, while MPK requires storing data on specified pages. In general, the MPK works on page level and is not specifically concerned about integrity, while secure data structures provide fine-grained integrity check for their stored elements.

Although the performance overhead of SafeDS is higher than MPK, it is still less than the maximum reasonable overhead, which is 5% [78].

Chapter 7

Discussion

7.1 Attacks against ARM Pointer Authentication

There have been several attempts to break the security of ARM Pointer Authentication (PA). One recent example of these attacks is PACMAN [68], which uses speculative execution attacks to create a Pointer Authentication Code (PAC) oracle. The created PAC oracle is able to distinguish between a valid and invalid PAC without crashing the program. Thus, it can be used to brute force PACs and find the correct PAC for a pointer, which can then be used to hijack the program's control flow.

PACMAN uses a gadget including a pointer verification operation and a transmission operation for the verification result through a micro-architectural side channel. However, this attack would not be a serious issue for our case since we use the *pacga* instruction to calculate Message Authentication Codes (MACs) and there is no such gadget consisting of the authentication instruction or leakage of the verification result.

Another another attack on the PA instrumentation of IOS uses the Apple A12 processor, which turns an invalid pointer to a valid pointer and its PAC [81]. However, this attack relies on the *pac* instruction set used for signing pointers which is not used in our case.

Accordingly, although ARM PA has been shown to be prone to several attacks, neither of the current attacks use *pacga* instruction. Moreover, although the possibility of such attacks exists, since the *pacga* instruction is not widely used throughout the code, finding proper gadgets would be harder than with the *pac* instruction set which could be widely used for return addresses and pointers.

7.2 Object Wrappers

As mentioned in Chapter 4, having a reference to an element in data structures allows legitimate modification of the element while inside the secure data structure. Since MAC updates are only performed when modifying the secure data structures through the provided API, we introduced object wrappers as a method to perform MAC updates when objects are modified while inside the data structures.

We implemented object wrappers as part of our prototype and tested their functionality along with our compatibility tests. When trying to use object wrappers, we noticed that there are considerable cases in OpenCV where changes to the program code are required due to the use of object wrappers. For instance, in almost all cases, when calling functions such as `top()` in stack, the OpenCV code was expecting the plain reference to the element to be returned. Therefore, the returned type (object wrapper class) was not matching the expected type, hence causing a compile error. Accordingly, we had to change the OpenCV source code to fix these issues.

Moreover, we noticed that in many of the OpenCV cases where secure data structures were used, the element type itself was a class. Method calls in such classes can change the element, and this change was not captured by the object wrapper since the wrapper only updates MACs on assignment.

Our manually callable function for performing MAC updates is a reasonable alternative in cases where object wrappers would introduce the mentioned obstacles.

7.3 Thread Safety

C++ programming language supports creating several threads in a program that allow for performing functions and tasks concurrently. Since threads in a multi-threaded program can share the same memory, having multiple threads modify the program variables can lead to inconsistencies, such as two threads trying to write to the same object simultaneously, or one thread reading the data while another is writing to it. Accordingly, a thread safe code can be safely accessed from different treads without causing any unintended behavior.

According to C++ standard specification, [4], the C++ standard library does not guarantee correct behavior when multiple threads are writing, or writing and reading an object simultaneously. However, C++ provides mechanisms such as *lock*s that can be used to ensure thread safety, for instance when using data structures.

Although locks provide a method to ensure thread safety, the attacker might still be able to bypass the locks and use the legitimate API of the program to perform unsafe reads and writes. However, based on our adversary model requirements in Section 3.2, we assume the presence of forward-edge Control-Flow Integrity (CFI). Forward-edge CFI ensures that an attacker is unable to make the program jump to arbitrary locations in code. Therefore, the attacker will not be able to bypass locks by directly jumping to the API calls of the data structures.

In the secure data structures, using *lock*s alone will not guarantee thread safety. A thread's local registers can not be accessed by other threads because each has its own set of registers. This can break our secure data structures design since we use a register to store the root of the global Merkle tree as explained in Chapter 5. In a multi-threaded setting, each thread will have its own reserved register for the Merkle tree's root and will not be able to verify the changes to the root value made by other threads. In order to overcome this limitation, we propose two distinct approaches.

The first approach is to use a secure messaging scheme that allows threads to communicate with each other. The communication scheme allows each thread to notify other threads when updating the Merkle tree. Consequently, the rest of the threads will be able to update the root stored in their reserved registers. Nevertheless, this approach requires the communication to be completely secure against the adversary's attempt to modify the messages between threads. For instance, the messages can be passed through the kernel space to ensure that the attacker cannot alter them. However, this solution can lead to high overhead that might not always be practical. The reason is that in order to achieve synchronization across the threads, they need to either constantly wait for the updates from other threads, which prevents the program from actually working, or store messages and check the received updates before performing any operation that requires verifying or updating the top MACs.

The second approach is to calculate and store a MAC for the root of the Merkle in memory. In this approach, each thread stores a state value in its reserved register. The state value for each thread is a counter starting from 0 that is incremented every time the thread updates the Merkle tree. The MAC for the root of the Merkle tree is calculated over the root along with all the state values. This method allows the verification of the root value using the state values and the MAC. However, this approach alone is still vulnerable to reuse attacks. For instance, if a thread updates the root, the attacker can restore the previous state values, root, and its MAC to revert the recent update. Then, unless the thread which updated the tree tries to access the root again, no other thread will detect the attacker's activity. The reason is that each thread only has its own latest state value safely stored in a register, so reverted actions can only be detected by the thread responsible for that action.

Since the MAC generation keys are securely stored, the adversary will not be able to use them to calculate new MACs. Moreover, the adversary is also unable to calculate random MACs by performing arbitrary jumps to MAC functions due to the presence of forward-edge CFI (our adversary model requirements, Section 3.2). Therefore, because the attacker is unable to calculate arbitrary MACs, the main issue with storing the Merkle tree root MAC in memory is an attacker being able to reuse old MACs. We can solve this issue by hiding the root MAC from the attacker. One potential option is to adopt a feature called Execute-only Memory (XOM) [8] which allows making part of the memory executable but not readable. If the root MAC is stored in such a memory area, the attacker will not be able to read the MACs and reuse them to revert the updates.

Because the root MAC needs to be updated as the Merkle tree changes, the executable memory should also be writable. The idea is to encode the root MAC in the instructions and load or update it inside the unreadable memory itself. However, having a writable and executable memory brings back the possibility of attacks such as code injection, which might make this approach impractical unless the mentioned problem can be properly dealt with.

7.4 Use Cases

Data structures can be the target of attacks in various applications. We were able to find real-world attack incidents that could be mitigated by using our secure data structures [5]. As an example, a buffer overflow found in the HMI3 Control Panel contained within the Swisslog Healthcare Nexus Panel allows an attacker to overwrite an internal queue data structure [7]. This issue can lead to malicious remote code execution. Using secure data structures can prevent the successful execution of the attack in this case.

Chapter 8

Related Work

8.1 Defenses Against Data Corruption Attacks

The defense methods against data corruption attacks can be divided into several categories [83]. In the following sections, we describe each category of defense along with examples of the previous work done in that area.

8.1.1 Spatial and Temporal Memory Safety

This category attempts to detect spatial or temporal memory errors and prevent them from being exploited. Examples of proposed defenses in this category are as follows:

CHERI [84] instruction set introduces a hybrid capability-system architecture that implements fine-grained memory protection with various properties, including but not limited to spatial and temporal safety. Most of these protections are managed mainly by the compiler. Light-weight Bounds Checking (LBC) [41] introduces a method to perform bound checking for the objects in memory by using guard zones around each object. The memory addresses that are marked as guard zone should not be accessed in the program. This approach allows the detection of out-of-bound access errors before being exploited. Another similar example from this category is SoftBound [59] which performs bound checks using stored metadata for the base and bound of the pointers in a shadow space. Fat [44] and low-fat [48] pointers, are examples of approaches that store the metadata for a pointer along with it instead of in a separate memory address. The mentioned solutions are also referred to as pointer-based approaches.

CETS [60] is an example of temporal safety enforcement which uses *identifiers* and a *lock-and-key* approach to detect the dangling pointers. In this approach, each pointer has a unique allocation key, and there is a lock location corresponding to that key. Whenever the region pointed by the pointer is deallocated, the lock location is set to invalid. This allows the detection of a dangling pointer since the key will not match the lock anymore. Another example is Undangle [21], which executes the program once inside an execution monitor, tracks the instructions, and creates an allocation log. These traces are then used for the early detection of temporal errors.

Furthermore, since both spatial and temporal safety are required to achieve complete security, there are solutions such as AddressSanitizer [74] which cover both spatial and temporal memory errors. AddressSanitizer uses shadow memory to record and check which memory addresses are safe to access when performing a load or store instruction.

As mentioned, all these solutions attempt to protect the whole program by detecting and preventing memory errors in the first place, while our approach attempts to detect the occurrence of the attack and ensure data integrity even in the presence of memory errors. Even though detecting memory errors before they are exploited provides valuable security promises, these solutions are still vulnerable to intra-object corruption attacks [36]. Moreover, since they attempt to protect the whole memory, most of the solutions in this category suffer from high memory or time overhead. On the other hand, we focus on protecting specific data in the program, which helps achieve lower performance overhead while still providing reasonable security.

8.1.2 Control and Data plane Randomization

This class of defense attempts to make the exploits harder either by randomizing the addresses or the data representation.

Examples of this class are described below:

Address Space Layout Randomization (ASLR) [67] applies randomness to addresses used by a given task. ASLR causes an attacker's attempt to exploit the memory to fail with a quantifiable probability since it limits the attacker to brute-forcing or guessing the addresses. ASLP [46] proposes randomizing code and data segments in the user memory space. Oxymoron [15] is another approach that divides the program code into memory pages and randomizes and shares the corresponding pages among processes. These solutions are however vulnerable to the JIT-ROP attacks since they only perform single randomization. As an example, one solution proposes re-randomization mechanisms to fix this issue [53].

DSR [18] randomizes data representations in memory by using random masking values that are XORed with the variables. HARD [17] improves DSR by using hardware features for shift operations.

This category still attempts to prevent attacks as opposed to our approach, which detects the occurrence of an attack. Both randomizations of control and data plane require high entropy to be secure, which could be costly to achieve. Furthermore, the randomization information itself needs to be protected from leakage. Our solution similarly requires protecting the Message Authentication Code (MAC) generation key from being leaked, which we ensure by using reserved registers.

8.1.3 Data Isolation

This class of defenses isolates critical data in specific parts of memory and enforces safety policies when accessing them. ERIM [82] combines memory protection keys (MPKs), a hardware primitive introduced in x86 allowing protection in userspace, with binary inspection to provide efficient hardware-enforced isolation.

IMIX [35] introduces isolated pages that store the critical data. The data in these pages can only be accessed using a new instruction *smov*, and the rest of the code is unable to access these pages. Execute-no-Read [14] proposes the implementation of execute-only memory, which allows for code execution but prevents load and store instruction to protect against memory disclosure.

This category is similar to our approach as they focus on protecting critical data instead of the whole program data. However, isolating data still renders non-negligible overhead in software-based solutions or requires special hardware features in hardware-based solutions, which can reduce its usability and compatibility. On the other hand, although our solution utilizes hardware features such as Pointer Authentication (PA), we achieve generality by proposing compatible implementation for x86 architectures. Moreover, our design can further be generalized for use in other settings by replacing the MAC calculation mechanism with available secure encryption methods.

8.1.4 CFI and DFI

This class of defenses monitors the control transfers or data load and stores in the program and validates them against a set of valid targets or instructions to prevent exploitation of memory errors for attacks such as Return Oriented Programming (ROP) and Data-Oriented Programming (DOP) attacks.

Control-Flow Integrity (CFI) was originally proposed in [9], but there have been various other versions that attempted to improve upon CFI, such as CCFI [54], which proposes the idea of creating a cryptographic MAC over the objects that affect the control flow of a program. Code-Pointer Integrity (CPI) [47] achieves CFI protection by protecting the integrity of all code pointers (e.g. return addresses) by separately storing them securely. PACstack [49] provides an approach for mitigating illegitimate changes in the function return addresses through calculating and chaining MACs. The MACs allow for discovering any changes to the correct return address hence preventing the adversary from altering the flow of the program and making the program jump to arbitrary addresses.

Similarly, Data-Flow Integrity (DFI) was originally introduced in [24], but was followed by many other proposed solutions to improve it, for instance, through using hardware as in HDFI [76].

Providing CFI or DFI by protecting all critical data can cause high overhead, therefore, most of the proposed practical solutions are approximations. Moreover, both CFI and DFI solutions need to be used at the same time to provide complete security since CFI is vulnerable to DOP attacks and DFI to ROP attacks. Our work uses the same idea as CCFI [54] and PACStack [49] to use cryptographic MACs for integrity verification. However, while the previous work used MACs to protect control data, we use them to specifically protect the integrity of data structures which could be used to store various types of critical data.

8.1.5 Program Anomaly Detection

In this class of defenses, the solutions detect anomalies in programs by a training process at the beginning and then monitoring the program throughout the execution process. One example of this category uses the *Intel Processor Trace* for this purpose [26].

This class of defense lacks compatibility, and its effectiveness depends on the results of the training phase [83]. Therefore, the current solutions from this category are not reliable enough to be generally practiced.

8.2 Authenticated Data Structures

Aside from securing the data structures against memory corruption attacks, verifiable data structures can be practical in networked systems and among untrusted entities. There are however some differences between the secure data structures and authenticated data structures used in networked systems. In the authenticated data structures, we assume the presence of an untrusted *prover* that performs the operations and provides proofs, and a *verifier* who wants to check its authenticity [57].

There have been many proposed designs for the authenticated data structures, such as binary trees, red-black trees, skip lists, and a few other data structures. For instance, there are persistent authenticated dictionaries that provide authenticated answers for queries about the presence of an element in the data structure at a certain time [12]. They introduce two designs, one based on the red-black tree, and one based on the skip-list. Moreover, they use collision-free hash functions for the authentication process.

$\lambda\cdot$ is a language for programming authenticated data structures in general [57]. With this language, the authentication process for any defined data structure is turned into a set of structurally similar steps performed by a prover and verifier at specific key points. The basis of the authentication proofs is a collision-resistance hash function similar to the previous designs for authenticated data structures.

Chapter 9

Conclusion

Memory attacks in low-level programming languages such as C++ are an ongoing problem in computer security. One way to prevent memory attacks that modify the program data is to detect when a malicious adversary changes the data illegitimately. In order to reduce the costs associated with protecting all data in the program, it is more reasonable to protect only specific data. SafeDS introduces secure data structures that ensure the integrity of their stored elements. We present secure designs tailored to each data structure's specific functionality and cryptographically prove how the proposed designs ensure element integrity. Moreover, we presented our prototype, the implementation of the secure data structures as part of gcc-11.1.0, which can be used with no change or minimal changes to the source code. We showed that our prototype can be used in real-world applications with negligible overhead. Finally, we discussed the possible future work for improving the current prototype. Our future work involves providing compatibility with multi-threaded programs and performing automatic Message Authentication Code (MAC) updates when the element is legitimately modified while inside the data structures.

References

[1] C++ reference. https://en.cppreference.com/w/cpp.

[2] C++ reference - copy constructors. https://en.cppreference.com/w/cpp/language/copy_constructor.

[3] C++ reference - move constructors. https://en.cppreference.com/w/cpp/language/move_constructor.

[4] C++11 standard library extensions - concurrency. https://isocpp.org/wiki/faq/cpp11-library-concurrency.

[5] CVE. =https://cve.mitre.org/.

[6] The rule of three/five/zero. https://en.cppreference.com/w/cpp/language/rule_of_three.

[7] swisslog healthcare letterhead. =https://www.swisslog-healthcare.com/.

[8] What is execute-only-memory (XOM)? https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/what-is-execute-only-memory-xom.

[9] Martin Abadi, Mihai Budiu, Ulfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):1–40, 2009.

[10] Steven Van Acker, Nick Nikiforakis, Pieter Philippaerts, Yves Younan, and Frank Piessens. Valueguard: Protection of native applications against data-only buffer overflows. In *International Conference on Information Systems Security*, pages 156–170. Springer, 2010.

[11] Alvinashcraft. Data execution prevention - win32 apps. https://docs.microsoft.com/en-us/windows/win32/memory/data-execution-prevention.

[12] Aris Anagnostopoulos, Michael T Goodrich, and Roberto Tamassia. Persistent authenticated dictionaries and their applications. In *International Conference on Information Security*, pages 379–393. Springer, 2001.

[13] Roberto Avanzi. The QARMA block cipher family. almost MDS matrices over rings with zero divisors, nearly symmetric even-mansour constructions with non-involutory central rounds, and search heuristics for low-latency s-boxes. *IACR Transactions on Symmetric Cryptology*, pages 4–44, 2017.

[14] Michael Backes, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nurnberger, 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, 2014.

[15] Michael Backes and Stefan Nurnberger. Oxymoron: Making fine-grained memory randomization practical by allowing code sharing. In *23rd USENIX security symposium (USENIX security 14)*, pages 433–447, 2014.

[16] Antonio Barresi, Kaveh Razavi, Mathias Payer, and Thomas R Gross. Cain: Silently breaking aslr in the cloud. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, 2015.

[17] 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.

[18] 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.

[19] Tyler Bletsch, Xuxian Jiang, and Vince Freeh. Mitigating code-reuse attacks with control-flow locking. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 353–362, 2011.

[20] Nathan Burow, Xinping Zhang, and Mathias Payer. Sok: Shining light on shadow stacks. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 985–999. IEEE, 2019.

[21] 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, 2012.

[22] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R Gross. Control-flow bending: On the effectiveness of control-flow integrity. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 161–176, 2015.

[23] Scott A Carr and Mathias Payer. Datashield: Configurable data confidentiality and integrity. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 193–204, 2017.

[24] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 147–160, 2006.

[25] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 147–160, 2006.

[26] Long Cheng. Program anomaly detection against data-oriented attacks. 2018.

[27] Yueqiang Cheng, Zongwei Zhou, Yu Miao, Xuhua Ding, and Robert H Deng. Ropecker: A generic and practical approach for defending against rop attack. 2014.

[28] Carlos Coronado. On the security and the efficiency of the merkle signature scheme. *Cryptology ePrint Archive*, 2005.

[29] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of Coarse-Grained Control-Flow integrity protection. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 401–416, San Diego, CA, August 2014. USENIX Association.

[30] Remi Denis-Courmont, Hans Liljestrand, Carlos Chinea, and Jan-Erik Ekberg. Camouflage: Hardware-assisted CFI for the ARM linux kernel. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.

[31] Solar Designer. "return-to-libc" attack. *Bugtraq, Aug*, 1997.

[32] Gregory J Duck and Roland HC Yap. Heap bounds protection with low fat pointers. In *Proceedings of the 25th International Conference on Compiler Construction*, pages 132–142, 2016.

[33] Morris Dworkin. Recommendation for block cipher modes of operation: The CMAC mode for authentication, 2016.

[34] Reza Mirzazade Farkhani, Mansour Ahmadi, and Long Lu. PTAuth: Temporal memory safety via robust points-to authentication. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1037–1054, 2021.

[35] Tommaso Frassetto, Patrick Jauernig, Christopher Liebchen, and Ahmad-Reza Sadeghi. IMIX: In-process memory isolation extension. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 83–97, 2018.

[36] Ronald Gil, Hamed Okhravi, and Howard Shrobe. There's a hole in the bottom of the C: On the effectiveness of allocation protection. In *2018 IEEE Cybersecurity Development (SecDev)*, pages 102–109. IEEE, 2018.

[37] Michael T Goodrich, Roberto Tamassia, and Michael H Goldwasser. *Data structures and algorithms in Java*. John Wiley & Sons, 2014.

[38] Google. Google benchmark. `https://github.com/google/benchmark/`, May 2022.

[39] Google. Google benchmark user guide. `https://github.com/google/benchmark/blob/main/docs/user_guide.md`, May 2022.

[40] Shay Gueron. Intel advanced encryption standard (AES) new instructions set, 2010.

[41] Niranjan Hasabnis, Ashish Misra, and R Sekar. Light-weight bounds checking. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 135–144, 2012.

[42] Charly Castes https://twitter.com/CharlyCastes/. Diving into intel mpk, Feb 2020.

[43] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-oriented programming: On the expressiveness of non-control data attacks. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 969–986. IEEE, 2016.

[44] Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. Cyclone: a safe dialect of C. In *USENIX Annual Technical Conference, General Track*, pages 275–288, 2002.

[45] Jicheol Lee Junhyuk Song. Openpana. `https://github.com/OpenPANA/openpana`, 2014.

[46] Chongkyung Kil, Jinsuk Jun, Christopher Bookholt, Jun Xu, and Peng Ning. Address space layout permutation (ASLP): Towards fine-grained randomization of commodity software. In *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*, pages 339–348. IEEE, 2006.

[47] Volodymyr Kuznetzov, Laszlo Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. Code-pointer integrity. In *The Continuing Arms Race: Code-Reuse Attacks and Defenses*, pages 81–116. 2018.

[48] Albert Kwon, Udit Dhawan, Jonathan M Smith, Thomas F Knight Jr, and Andre DeHon. Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 721–732, 2013.

[49] Hans Liljestrand, Thomas Nyman, Lachlan J Gunn, Jan-Erik Ekberg, and N Asokan. PACStack: an authenticated call stack. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 357–374, 2021.

[50] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N Asokan. PAC it up: Towards pointer integrity using ARM pointer authentication. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 177–194, 2019.

[51] Dale Liu, Max Caceres, Tim Robichaux, Dario V. Forte, Eric S. Seagren, Devin L. Ganger, Brad Smith, Wipul Jayawickrama, Christopher Stokes, and Jan Kanclirz. Chapter 3 - an introduction to cryptography. In *Next Generation SSH2 Implementation*, pages 41–64. Syngress, Burlington, 2009.

[52] LLVM. LLVM libcxx benchmarks. `https://github.com/llvm/llvm-project/tree/main/libcxx/benchmarks`, 2022.

[53] Kangjie Lu, Wenke Lee, Stefan Nurnberger, and Michael Backes. How to make ASLR win the clone wars: Runtime re-randomization. In *NDSS*, 2016.

[54] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. CCFI: Cryptographically enforced control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 941–951, 2015.

[55] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 2018.

[56] Microsoft. merklecpp. https://github.com/microsoft/merklecpp, 2021.

[57] Andrew Miller, Michael Hicks, Jonathan Katz, and Elaine Shi. Authenticated data structures, generically. *ACM SIGPLAN Notices*, 49(1):411–423, 2014.

[58] Frederic P. Miller, Agnes F. Vandome, and John McBrewster. *Advanced Encryption Standard*. Alpha Press, 2009.

[59] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. SoftBound: Highly compatible and complete spatial memory safety for C. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 245–258, 2009.

[60] Santosh Nagarakatte, Jianzhou Zhao, Milo MK Martin, and Steve Zdancewic. CETS: compiler enforced temporal safety for C. In *Proceedings of the 2010 International Symposium on Memory Management*, pages 31–40, 2010.

[61] George C Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(3):477–526, 2005.

[62] Tim Newsham. Format string attacks, 2000.

[63] Okdshin. PicoSHA2 - a C++ SHA256 hash generator. https://github.com/okdshin/PicoSHA2, 2021.

[64] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel MPX explained: An empirical study of Intel MPX and software-based bounds checking approaches. *arXiv preprint arXiv:1702.00719*, 2017.

[65] Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. G-Free: defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference*, pages 49–58, 2010.

[66] Aleph One. Smashing the stack for fun and profit. *Phrack magazine*, 7(49):14–16, 1996.

[67] Team PaX. Pax address space layout randomization (ASLR). *http://pax.grsecurity.net/docs/aslr.txt*.

[68] Joseph Ravichandran, Weon Taek Na, Jay Lang, and Mengjia Yan. PACMAN: attacking ARM pointer authentication with speculative execution. In *ISCA*, pages 685–698, 2022.

[69] Christian Fredrik Fossum Resell. Forward-edge and backward-edge control-flow integrity performance in the linux kernel. Master's thesis, 2020.

[70] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1):1–34, 2012.

[71] Nick Roessler and Andre DeHon. Protecting the stack with metadata policies and tagged hardware. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 478–495. IEEE, 2018.

[72] Mark Rutland. Pointer authentication in AARCH64 linux. https://www.kernel.org/doc/html/latest/arm64/pointer-authentication.html, Jul 2017.

[73] Cole Schlesinger, Karthik Pattabiraman, Nikhil Swamy, David Walker, and Ben Zorn. Yarra: An extension to c for data integrity and partial safety, 2011.

[74] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 309–318, 2012.

[75] Victor Shoup. Sequences of games: a tool for taming complexity in security proofs. Cryptology ePrint Archive, Paper 2004/332, 2004. https://eprint.iacr.org/2004/332.

[76] 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.

[77] Zhichuang Sun, Bo Feng, Long Lu, and Somesh Jha. OAT: Attesting operation integrity of embedded devices. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1433–1449. IEEE, 2020.

[78] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62. IEEE, 2013.

[79] Qualcomm Technologies. Pointer authentication on ARMv8.3: Design and analysis of the new software security instructions. 2017.

[80] Alin Tomescu. What is a merkle tree? https://decentralizedthoughts.github.io/2020-12-22-what-is-a-merkle-tree/#fn:consideredtobe.

[81] Unknown. Examining pointer authentication on the iphone xs. https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html, Jan 1970.

[82] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, efficient in-process isolation with protection keys (MPK). In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1221–1238, 2019.

[83] Ye Wang, Qingbao Li, Zhifeng Chen, Ping Zhang, and Guimin Zhang. A survey of exploitation techniques and defenses for program data attacks. *Journal of Network and Computer Applications*, 154:102534, 2020.

[84] Robert N. M. Watson, Peter G. Neumann, Jonathan Woodruff, Michael Roe, Jonathan Anderson, David Chisnall, Brooks Davis, Alexandre Joannou, Ben Laurie, Simon W. Moore, Steven J. Murdoch, Robert Norton, Stacey Son, and Hongyan Xia. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 5). Technical Report UCAM-CL-TR-891, University of Cambridge, Computer Laboratory, June 2016.

[85] Yubin Xia, Yutao Liu, Haibo Chen, and Binyu Zang. CFIMon: Detecting violation of control flow integrity using performance counters. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, pages 1–12. IEEE, 2012.

[86] Jun Zhang, Rui Hou, Junfeng Fan, Ke Liu, Lixin Zhang, and Sally A McKee. RAGuard: A hardware based mechanism for backward-edge control-flow integrity. In *Proceedings of the Computing Frontiers Conference*, pages 27–34, 2017.

[87] Jun Zhang, Rui Hou, Wei Song, Zhiyuan Zhan, Boyan Zhao, Mingyu Chen, and Dan Meng. Stateful forward-edge CFI enforcement with Intel MPX. In *Conference on Advanced Computer Architecture*, pages 79–94. Springer, 2018.

APPENDICES

We present the secure red-black tree functions here:

Algorithm 10 RB-DELETE

```
y = z
if z.left == T.nil then
    x = z.right
else if z.right == T.nil then
    x = z.left
else
    y = y.right
    while $y.left \neq T.nil$ do
        y = y.left
    end while
    x = y.right
end if
if $y \neq z$ then
    z.left.p = y
    y.left = z.left
    if $y \neq z.right$ then
        x-parent = y.p
        if x then
            x.p = y.p
        end if
        y.p.left = x
        y-dummy = y
        //updating the MACs
        while $y - dummy \neq z.right$ do
            y-dummy.p.mac = calculate-mac(nonce, y-dummy.p.left.mac ,
y-dummy.p.right.mac, y-dummy.p)
            y-dummy = y-dummy.p
        **end while**
        y.right = z.right
        y.mac = calculate-mac(nonce, y.left.mac , y.right.mac, y)
        z.right.p = y
    **else**
        y.mac = calculate-mac(nonce, y.left.mac , y.right.mac, y)
        x-parent = y
    **end if**
    **if**  root == z **then**
        root = y
        update-top-mac(calculate-mac(nonce,leftmost.mac,rightmost.mac, root.mac), nonce)
    **else if** z.p.left == z **then**
        z.p.left = y
    **else**
        z.p.right = y
    **end if**
    z.p.mac = calculate-mac(nonce, z.p.left.mac , z.p.right.mac, z.p)
    currently-updated = z.p
    old =z.p.mac
    store-mac(calculate-mac(nonce, old, z.p.mac))
    y.p = z.p
    **if** root == currently-updated **then**
        update-top-mac(calculate-mac(nonce,leftmost.mac,rightmost.mac, root.mac), nonce)
    **end if**
    swap(y.color,z.color)
    y = z
**else**
    x-parent = y.p
    **if** x **then**
        x.p = y.p
    **end if**
    **if** z == root **then**
        root = x
        update-top-mac(calculate-mac(nonce,leftmost.mac,rightmost.mac, root.mac),
nonce)
    else if z.p.left == z then
        z.p.left = x
    else
        z.p.right = x
    end if
    z.p.mac = calculate-mac(nonce, z.p.left.mac , z.p.right.mac, z.p)
    currently-updated = z.p
    old =z.p.mac
    store-mac(calculate-mac(nonce, old, z.p.mac))
    if root == currently-updated then
        update-top-mac(calculate-mac(nonce,leftmost.mac,rightmost.mac, root.mac),
nonce)
    end if
//check if leftmost and rightmost have changed, and if yes, update them along with
the top-mac
end if
```

Algorithm 11 RB-DELETE-FIXUP

```
        // Checking if balancing is required.
while $x \neq T.rootandx.color == BLACK$ do
    if x = currently-updated then
        temp = x.p.mac
        verify-and-update-macs(x,1,old)
        old = temp
        if x != root then
            currently-updated = x.p
        end if
    end if
    if x == x.p.left then
        w = x.p.right
        if w.color == RED then
            w.color = BLACK
            x.p.color = RED
            LEFT-ROTATE(T, x.p)
            currently-updated = currently-updated.p
            store-mac(calculate-mac(nonce, old, currently-updated.mac))
            if  currently-updated = root then
                update-top-mac(calculate-mac(nonce,leftmost.mac,rightmost.mac, root.mac), nonce)
            end if
            w = x.p.right
        end if
        if w.left.color == BLACK and w.right.color == BLACK then
            w.color = RED
            x = x.p
        else
            if w.right.color == BLACK then
                w.left.color = BLACK
                w.color = RED
                verify-mac(calculate-mac(nonce, w.left.left.mac, w.left.right.mac, w.left), w.left.mac)
                verify-mac(calculate-mac(nonce, w.right.left.mac, w.right.right.mac, w.right), w.right.mac)
                dummy-x = RIGHT-ROTATE(T,w)
                while $x - dummy \neq currently - updated$ do
                    x-dummy.p.mac = calculate-mac(nonce, x-dummy.p.left.mac, x-dummy.p.right.mac, x-dummy.p)
                    x-dummy = x-dummy.p
                end while
                store-mac(calculate-mac(nonce, old, currently-updated.mac))
                if  currently-updated = root then
                    update-top-mac(calculate-mac(nonce,leftmost.mac,rightmost.mac, root.mac), nonce)
                end if
                w = x.p.right
            end if
            w.color = x.p.color
            x.p.color = BLACK
            w.right.color = BLACK
            if currently-updated = x.p then
                is-updated = true
            end if
            x-dummy = LEFT-ROTATE(T,x.p)
            if is-updated then
                currently-updated = currently-updated.p
            **else**
                **while** $x - dummy \neq currently - updated$ **do**
                    x-dummy.p.mac = calculate-mac(nonce, x-dummy.p.left.mac , x-dummy.p.right.mac, x-dummy.p)
                    x-dummy = x-dummy.p
                **end while**
            **end if**
            store-mac(calculate-mac(nonce, old, currently-updated.mac))
            **if** currently-updated = root **then**
                update-top-mac(calculate-mac(nonce,leftmost.mac,rightmost.mac, root.mac), nonce)
            **end if**
        **end if**
    **else**(same as **then** clause with "right" and "left" exchanged)
    **end if**
**end while**
**if** x **then**
    x.color = BLACK
**end if**
```

Algorithm 12 RB-Insert

```
y = T.nil
x = T.root
root-mac = get-top-mac()
verify-mac(root-mac, x.mac)
verify-mac(calculate-mac(nonce, x.left.mac, x.right.mac, x), x.mac)
**while** $x \neq T.nil$ **do**
    y = x
    verify-mac(calculate-mac(nonce, y.left.mac, y.right.mac, y), y.mac)
    **if** $z.key < x.key$ **then**
        x = x.left
    **else**
        x = x.right
    **end if**
**end while**
z.p = y
**if** y == T.nil **then**
```

```
    T.root = z
    update-root()
else if z.key < y.key then
    y.left = z
else
    y.right =z
end if
z.left = T.nil
z.right = T.nil
z.color = RED
z.mac = calculate-mac(nonce, z.left.mac, z.right.mac, z)
old = 0
to-save = calculate-mac(old, z.mac)
store-mac(to-save)
(old, z) = RB-INSERT-FIXUP(T, z)
verify-mac(calculate-mac(nonce, old, z.mac), get-mac-from-register())
if z == z.p.left then
    update-macs(z.p,old, z.mac, true)
else
    update-macs(z.p,old, z.mac, false)
end if
```

Algorithm 13 RB-INSERT-FIXUP

```
while z.p.color = RED do
    if z == currently-updated then
        verify-and-update-macs(z,2)
        currently-updated = z.p.p
    else
        if z.p == currently-updated then
            verify-and-update-macs(z,1)
            currently-updated = z.p
        end if
    end if
    if z.p == z.p.p.left then
        y = z.p.p.right
        if y.color == RED then
            z.p.color = BLACK
            y.color = BLACK
            z.p.p.color = RED
            z = z.p.p
        else
            if  z == z.p.right then
                z = z.p
                LEFT-ROTATE(T,z)
                x.p.p.mac = calculate-mac(nonce, z.p.p.left.mac, z.p.p.right.mac,
z.p.p)
                store-mac(old, x.p.p.mac)
            end if
            z.p.color = BLACK
            z.p.p.color = RED
            currently-updated = ROTATE-RIGHT(T, z.p.p)
            store-mac(old, currently-updated.mac)
        end if
    else (same as then clause with ”right” and ”left” exchanged)
    end if
end while
T.root.color = BLACK
return (z, old)
```

Algorithm 14 LEFT-ROTATE

```
y = x.right
x.right = y.left
if $y.left \neq T.nil$ then
    y.left.p = x
end if
y.p = x.p
if x.p == T.nil then
    T.root = y
else if x == x.p.left then
    x.p.left = y
else
    x.p.right = y
end if
y.left = x
```

```
x.p = y
x.mac = calculate-mac(nonce, x.left.mac, x.right.mac, x)
y.mac = calculate-mac(nonce, x.mac, y.right.mac, y)
if  y == root then
    update-top-mac(calculate-mac(nonce,leftmost.mac,rightmost.mac, root.mac),
nonce)
end if
return y
```

Algorithm 15 VERIFY-AND-UPDATE-MACS

```
mac = get-mac-from-register();
verify-mac(calculate-mac(nonce,old,x.mac), mac)
verify-mac(calculate-mac(nonce, x.left.mac, x.right.mac, x), x.mac)
while $level - count > 0$ do
    if  $x \neq root$ then
        if $x.p \neq root$ then
            if x == x.p.right then
                if x.p.left then
                    verify-mac(calculate-mac(nonce, x.p.left.left.mac, x.p.left.right.mac,
x.p.left), x.p.left.mac)
                end if
                verify-mac(calculate-mac(nonce, x.p.left.mac, old-mac, x.p), x.p.mac)
                old = x.p.mac x.p.mac = calculate-mac(nonce, x.p.left.mac,
x.p.right.mac, x.p)
            else
                if x.p.right then
                    verify-mac(calculate-mac(nonce, x.p.right.left.mac,
x.p.right.right.mac,x.p.right), x.p.right.mac)
                end if
                verify-mac(calculate-mac(nonce,old-mac, x.p.right.mac, x.p), x.p.mac)
                old = x.p.mac
                x.p.mac = calculate-mac(nonce, x.p.left.mac, x.p.right.mac, x.p)
            end if
            level-count = level-count - 1
            x = x.p
        else
            if x == x.p.right then
                if x.p.left then
                    verify-mac(calculate-mac(nonce, x.p.left.left.mac,
x.p.left.right.mac,x.p.left), x.p.left.mac)
                end if
                top = get-top-mac(nonce)
                verify-mac(top, calculate-mac(nonce, leftmost.mac,
rightmost.mac, root.mac))
                verify-mac(calculate-mac(nonce,x.p.left.mac, old-mac, x.p), x.p.mac)
                old = x.p.mac
                x.p.mac = calculate-mac(nonce, x.p.left.mac, x.p.right.mac, x.p)
            else
                if x.p.right then
                    verify-mac(calculate-mac(nonce, x.p.right.left.mac,
x.p.right.right.mac,x.p.right), x.p.right.mac)
                end if
                top = get-top-mac(nonce)
                verify-mac(top, calculate-mac(nonce, leftmost.mac, rightmost.mac,
root.mac))
                verify-mac(calculate-mac(nonce,old-mac, x.p.right.mac, x.p), x.p.mac)
                old = x.p.mac
                x.p.mac = calculate-mac(nonce, x.p.left.mac, x.p.right.mac, x.p)
            end if
            level-count = level-count - 1
            update-top-mac(calculate-mac(nonce,leftmost.mac,rightmost.mac,
root.mac), nonce)
            x = x.p
        end if
      end if
end while
store-mac(calculate-mac(old-mac, x.mac, nonce))
```

Algorithm 16 UPDATE-MACS

```
if $x \neq root$ then
    if left then
        verify-mac(new-mac, x.left.mac)
        verify-mac(calculate-mac(nonce, old-mac, x.right.mac, x), x.mac)
    else
```

```
        verify-mac(new-mac, x.right.mac)
        verify-mac(calculate-mac(nonce, x.left.mac, old-mac, x), x.mac)
    **end if**
    old-mac = x.mac
    x.mac = calculate-mac(nonce, x.left.mac, x.right.mac, x)
    **if** x == x.p.left **then**
        update-macs(x.p, old-mac, x.mac, true)
    **else**
        update-macs(x.p, old-mac, x.mac, false)
    **end if**
**else**
    top = get-top-mac(nonce)
    verify-mac(calculate-mac(nonce,leftmost.mac,rightmost.mac, root.mac), top)
    **if** left **then**
        verify-mac(new-mac, x.left.mac)
        verify-mac(calculate-mac(nonce, old-mac, x.right.mac, x), x.mac)
    **else**
        verify-mac(new-mac, x.right.mac)
        verify-mac(calculate-mac(nonce, x.left.mac, old-mac, x), x.mac)
    **end if**
    x.mac = calculate-mac(nonce, x.left.mac, x.right.mac, x)
    update-top-mac(calculate-mac(nonce,leftmost.mac,rightmost.mac, root.mac),
nonce)
**end if**
```