

Assessing the Potential Impact of Memory-Safe Languages for System Software Security



Fall 2023

Investigators: Daniel Molsbarger, George Kotti,
Kenna Henkel, Kaneesha Moore,
Prathyusha Mustiyala, Wilson Patterson

Faculty Mentor: Dr. Sudip Mittal
Mississippi State University

Problem Mentor: Neal Ziring, National Security Agency



PROJECT SUMMARY

Project Title: Assessing the Potential Impact of Memory-Safe Languages for Software Security

Project Date: August 16, 2023 - December 3, 2023

Investigators:

Kenna Henkel - *Mississippi State University* - kbb269@msstate.edu

Daniel Molsbarger - *Mississippi State University* - dhm88@msstate.edu

George Kotti - *Mississippi State University* - ghk24@msstate.edu

Wilson Patterson - *Mississippi State University* - wep104@msstate.edu

Prathyusha Mustiyala - *Mississippi State University* - pm1112@msstate.edu

Kaneesha Moore - *Mississippi State University* - kkm267@msstate.edu

Problem Title:

Assessing the Potential Impact of Memory-Safe Languages for System Software Security

Target Area:

This project seeks to evaluate the potential impact of memory-safe languages on system software security, exploring their effectiveness in identifying and mitigating vulnerabilities across diverse applications in both theoretical and practical contexts.

Keywords:

Buffer overflow, memory-safety, vulnerabilities, user-after-free, double-free, memory leak, memory allocation, garbage collector, and memory management.

Project Description:

There are several initiatives underway to encourage system programmers and writers of low-level software to reduce the use of C/C++ and adopt memory-safe programming languages. Many members of the security community believe that this shift will reduce the incidence of memory-based vulnerabilities in critical software. This project is a research on the different types of vulnerabilities that can arise from memory handling and memory access errors (e.g., buffer overflow, use-after-free, heap overflow, etc.), survey and categorize salient vulnerabilities in the National Vulnerability Database (NVD), and estimate the potential benefits of adopting memory-safe languages. This project will also examine, and attempt to quantify, how effective various popular memory-safe languages are at eliminating the categorized vulnerabilities, including both the ability to detect when such vulnerabilities are present and the ability to prevent them from being exploitable. A particularly important aspect of this examination will include effects at different stages of the software life cycle (e.g., during coding, at compile time, and during execution).

EXECUTIVE SUMMARY

Project Title: Assessing the Potential Impact of Memory-Safe Languages for Software Security

Project Date: August 16, 2023 - December 3, 2023

Investigators:

Kenna Henkel - *Mississippi State University*

Daniel Molsbarger - *Mississippi State University*

George Kotti - *Mississippi State University*

Wilson Patterson - *Mississippi State University*

Prathyusha Mustiyala - *Mississippi State University*

Kaneesha Moore - *Mississippi State University*

Problem Title:

Assessing the Potential Impact of Memory-Safe Languages for System Software Security

Target Area:

This project seeks to evaluate the potential impact of memory-safe languages on system software security, exploring their effectiveness in identifying and mitigating vulnerabilities across diverse applications in both theoretical and practical contexts.

Keywords:

Buffer overflow, memory-safety, vulnerabilities, user-after-free, double-free, memory leak, memory allocation, garbage collector, and memory management.

Summary:

Over time, the increasing complexity of attacks on software infrastructure has progressively threatened user safety across multiple sectors. As a result, continuous research and improvements in the protections are essential. Although new security technologies can add layers to the security "fortress", inherent weaknesses in older programming languages combined with the preference for these languages hinder these efforts. Particularly, memory-based vulnerabilities in popular programming languages like C and C++ have significantly compromised many software programs [1]. Recent software development has focused on these systemic vulnerabilities. New languages have been developed to address the risks commonly associated with manual memory management. These languages aim to prevent memory-related vulnerabilities through various methods.

The scope of this project contains three milestones: search the computer science literature for analyses of memory-safe languages, present vulnerabilities often listed in the Common Weakness Enumeration (CWE) and the National Vulnerability Database (NVD), and test if the same security vulnerabilities exist in memory-safe languages. The languages chosen for this project are Rust, Go, Kotlin, and Python, while the non-memory-safe counterparts are C and C++. Eventually, these languages will be used in programs projected to emulate the deliberate implementation of vulnerabilities in C languages to test the responses of each memory-safe language.

TABLE OF CONTENTS

I	Introduction	6
I-A	Motivation	6
II	Literature Review	6
II-A	Background	6
II-B	Evolution of Memory-Safe Languages	7
II-C	Characteristics of Memory-Safe Languages	8
II-D	Case Studies of Memory-Safe Languages	8
II-E	Comparison with Non-Memory-Safe Languages	9
II-F	Real-world Implications of Memory Safety	10
II-G	Future Directions	10
III	Methods and Procedures	11
III-A	Vulnerability Execution Considerations for Memory-Safe Languages	11
III-B	Analysis Procedures	12
III-B1	AddressSanitizer	12
III-B2	Rust Compiler	12
III-B3	htop	13
IV	Findings	13
V	Issues	13
V-A	Project Limitations	13
V-A1	Funding /or Time /or Tooling Limitations	13
V-A2	Learning Curve	13
V-B	Unforeseen Circumstances	13
V-B1	Environmental Challenges	13
V-B2	Code Analysis	13
V-C	Language-Specific Issues	14
V-C1	Memory-Safe Vs Non-Memory-Safe languages	14
V-C2	Limitations of Memory-Safe Languages	14
V-C3	Compatibility with Legacy Code	14
V-C4	Community	15
VI	Conclusions and Recommendations	15
VII	Acknowledgements	15
	References	15
	Appendix A: Biographical Sketches of the Investigators	17
	Appendix B: Code Examples	19
	Buffer Overflow	19
	C example	19
	C++ example	19
	Rust example	19
	Go example	19
	Python example	20
	Use-After-Free	21
	C example	21
	C++ example	21
	Rust example	22
	Go example	22
	Kotlin example	22
	Python example	22

Double Free	23
C example	23
C++ example	23
Rust example	23
Go example	24
Python example	24
Memory Leak	25
C example	25
C++ example	25
Rust example	26
Monitor Process	27

I. INTRODUCTION

In software development, different programming languages have unique properties that influence the design phase in the Software Development Lifecycle (SDLC). Because of this, software developers, in optimal circumstances, can choose the programming language best suited for their needs. However, security in manually managed programming languages has increasingly shown a vulnerability to specific forms of attacks that exploit mistakes common to manually managed programming languages. An intuitive solution to avoid these vulnerabilities is to use memory-safe programming languages.

A. Motivation

Low-level system programmers are urged to adopt newer memory-safe programming languages, stemming from the ongoing battle in memory-related issues due to current development tactics. According to Meza et al. [2], memory-related vulnerabilities have remained a pressing issue for decades. C/C++ are notoriously known to be breeding grounds for these vulnerabilities. Due to the familiarity and performance capacity of these languages, developers take on the risk of manually managing memory. However, recent developments have proven this can change. A recent empirical study by Pirelli et al [3] suggests the feasibility of attaining safety without introducing significant runtime overhead. This revelation places an emphasis on the crucial need to examine and further develop memory-safe languages by exploring their ability to detect and prevent said vulnerabilities to counteract this massive computer-security-related issue.

II. LITERATURE REVIEW

This literature review explores memory-safe languages, their effectiveness in preventing the aforementioned vulnerabilities, and the consequences of choosing these memory-safe languages while emphasizing their importance in mitigating software security risks and ensuring operational consistency. Other resources used for a better understanding of memory-related vulnerabilities were considered and studied as well.

A. Background

Memory safety, although a technical term, has ramifications that affect even the most casual computer user. From data breaches to system crashes, memory vulnerabilities can lead to significant disruptions. Memory-safe languages are programming languages designed to prevent encoding memory vulnerabilities into software systems, and this is often accomplished by preventing the developers from writing code that would inherently create vulnerabilities in the system. For example, Rust will not allow code that could result in use-after-free or double-free vulnerabilities [4]. To better appreciate the solutions presented by memory-safe languages, it's vital to understand the definition of memory safety and the common memory-related vulnerabilities developers encounter. Two programming languages formally verified to be memory-safe are **Rust** [5], [6] and **Go** [7]. Throughout this work, we will assess these languages and highlight the innate differences between

the two previously defined categories - manually managed and memory-safe programming languages.

a) *The C Family*: The C language, a procedural-oriented programming language developed in the early 1970s by Bell Laboratories' Dennis Ritchie, holds a unique place in the history of computing as the 'mother of all programming languages'. Originally intended for the development of micro-computer operating systems, it quickly became evident that C's utility extended far beyond its initial purpose [8]. With its roots in the development of the Unix operating system, initially implemented in assembly language, C remains indispensable in modern times for building kernels, operating systems, and low-level applications. This has to do with the fact that C is a minimalist language with the ability to communicate closely with low-level hardware and memory, while also having very little overhead due to the lack of a garbage collection system and full automatic memory management system. This combination of attributes makes C the unwavering top choice for traditional systems programming. C's sibling language C++ was introduced in the 1980s as a powerful and versatile extension of C designed to bridge the gap between low-level systems programming and high-level application development¹.

b) *Common Memory-Related Vulnerabilities*: Memory safety is a critical concern in software development, especially systems programming, due to its direct implications for security and reliability. Even the most experienced programmers are not immune to writing memory-related vulnerabilities, which makes for a topic of concern. A classic example is buffer overflow - a program writes data beyond the boundaries of a buffer or intended container (see Figure 1b). This can overwrite memory areas outside the program's scope, causing unexpected behavior from the machine. The C family is infamous for these types of vulnerabilities, though it has been kept alive in systems programming due to its exceptional runtime performance; this attribute makes it ideal for low-level application development, such as the development of operating systems (kernels) and embedded systems. Though many unsafe functions in C have been depreciated, they can still be seen in recent code. Attackers make use of these functions to develop exploits and attack machines. Functions used in buffer overflow attacks (as well as format string vulnerabilities) can be separated into standard input reading functions and string manipulation functions (see Table I for the list of these buffer overflow vulnerable functions) [9]. Buffer overflow is not limited to system-based applications; Cross-Site Scripting (XSS), or internet buffer overflow, is a version of buffer overflow that occurs in web-based applications and can be triggered by malicious code or Structured Query Language (SQL) attacks. Other memory-related vulnerabilities include use-after-free, which occurs when a program attempts to access deallocated memory (see Figure 1a); double-free, attempting to free the same memory region more than once (see Figure 2b); and heap-stack exploitation, the manipulation of input data to

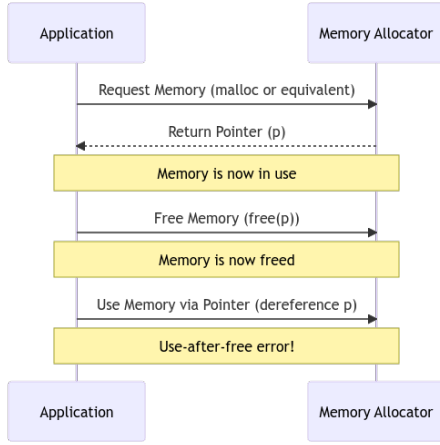
¹For the project's development, we defined high-level application development as having the addition of object-oriented capabilities.

trigger vulnerabilities such as the previously discussed buffer overflows (see Figure 2c).

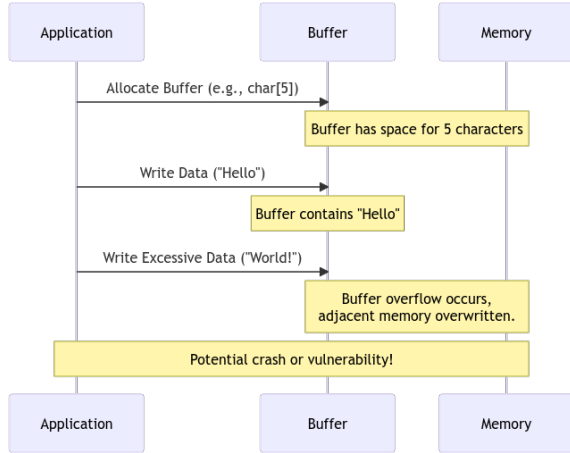
TABLE I: Functions Vulnerable to Buffer Overflow Attacks

Standard Input Reading Functions	String Manipulation Functions
fscanf()	strcpy()
scanf()	strcmp()
gets()	strlen()
sscanf()	strtok()

Fig. 1: Sequence diagrams illustrating two different types of memory errors.



(a) A sequence diagram of an use-after-free error. An application requests memory and receives a pointer. Next, the application frees the memory and then, erroneously attempts to re-access it.



(b) A sequence diagram of a buffer overflow error. An application allocates a buffer and writes data exceeding its capacity, causing an overflow.

B. Evolution of Memory-Safe Languages

Historically, many programming languages left the responsibility of memory management to developers, a choice that offers both benefits and pitfalls. Over time as challenges and

vulnerabilities were uncovered, the software community began to move towards languages that offered intrinsic memory safety features. This section traces the trajectory of these languages, from their rudimentary beginnings to their sophisticated modern iterations. One can assume the trail of programmers implementing and considering memory management can be traced back to the beginning of computing.

a) Rust: Rust, a multi-paradigm, general-purpose programming language, aims to bridge high-level languages with low-level systems programming by offering robust static guarantees in memory and thread safety. It achieves this by enforcing a garbage collection system without utilizing run-time memory management, while still giving the programmer control over data layout and memory management, essential aspects of systems programming [10], [11]. Rust's unique ownership system, lifetimes, and borrowing model provide a robust foundation for memory safety. Rust's syntax style, reminiscent of C and C++ in terms of control flow, keywords, and expression blocks, makes it familiar to those versed in those languages [12].

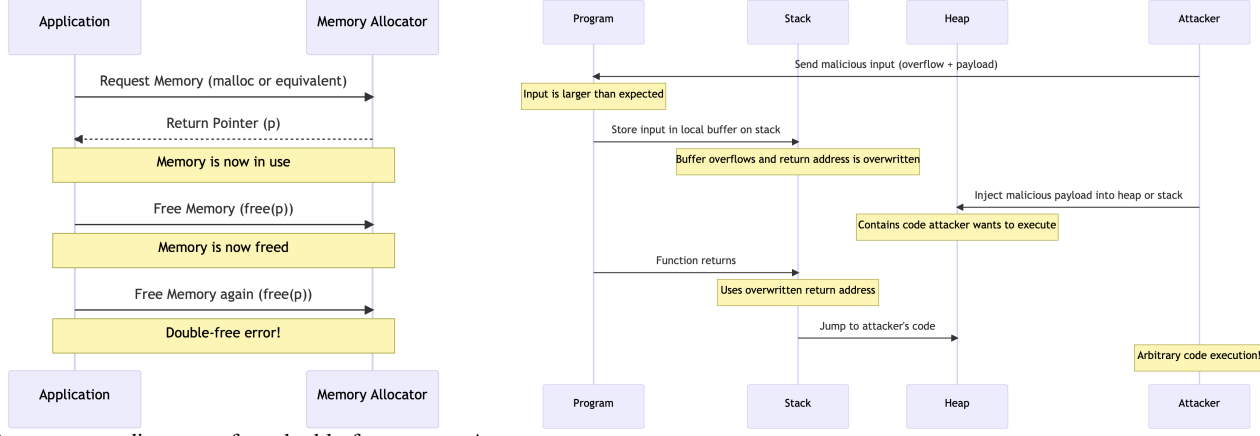
Unlike the C family, Rust enforces type safety and memory safety, catching known memory vulnerabilities such as use-after-free and double-free. Furthermore, Rust offers a powerful 'unsafe' keyword that allows controlled access to low-level operations and facilitates the porting of existing C/C++ code to Rust (i.e. pointer arithmetic, accessing uninitialized memory, the calling of unsafe functions). This flexibility allows developers to gradually refactor and secure existing code bases while maintaining compatibility with their system-level or performance-sensitive components. Rust effectively eliminates common errors that often afflict many other safe programming languages, such as iterator invalidation—when an iterator becomes invalid due to alterations during the data structure's traversal within the iteration [10].

Originally sponsored by Mozilla, Rust is now overseen by the Rust Programming Language Project, which is a collaborative effort involving multiple organizations and individual contributors.

b) GO: The GO programming language, also known as Golang, originated at Google in 2007 and was officially introduced in 2009. As an open-source language, GO aims to provide a simple, efficient, and modern solution for software development [13]. GO quickly gained traction, gaining recognition from performance-critical domains and attracting adoption from companies like Uber, Dropbox, Docker, and Netflix [14].

GO's race detector is a powerful tool integrated into the Go programming language that automatically detects and reports data races in concurrent Go programs [15]. It works by dynamically analyzing the execution of goroutines and identifying instances where multiple threads access shared variables without proper synchronization, helping developers catch and resolve concurrency issues early in the development process. Notably, Uber has actively delved into the tool's application within their source code, showcasing its effectiveness in enhancing the reliability and performance of their Go-based systems [16]. In 2022, Go 1.17 further refined the language, offering im-

Fig. 2: Sequence diagrams illustrating two different types of memory errors.



(a) A sequence diagram of a double-free error. An application attempts to free previously freed memory.

(b)

(c)

provements to the garbage collector, new language features, and enhancements to the standard library, notably introducing the `”//go:embed”` directive for embedding static files [17].

Go’s built-in concurrency model is a standout feature of the language. Goroutines, lightweight threads managed by the Go runtime, allow for concurrent execution without the need for external libraries or complex synchronization mechanisms.

The outlined features make it remarkably simple to write highly concurrent programs, which is particularly beneficial for applications that require parallel processing, such as web servers and distributed systems.

c) Kotlin: Kotlin is a versatile, statically typed programming language designed to interoperate seamlessly with existing Java code, making it an excellent choice for both Android app development and server-side applications. Developed by JetBrains in 2011, Kotlin was open-sourced in 2012, and it gained official support from Google for Android development in 2017 [18].

Kotlin aims to enhance productivity, conciseness, and safety in coding, offering features like null safety, extension functions, and smart casts. Its concise syntax and interoperability with Java have made it a popular choice for Android development, leading many major companies such as Netflix, AirBnB, and Pinterest to adopt it for their mobile apps [19], [20], [21]. Additionally, Kotlin’s multi-platform capabilities allow developers to write shared code for Android, iOS, and back-end services, reducing development time and maintenance overhead [22].

C. Characteristics of Memory-Safe Languages

While the concept of a memory-safe language might seem straightforward, the features constituting such safety are diverse and multifaceted. Memory-safe languages come equipped with mechanisms aimed at reducing or eliminating memory-related errors [23]. This section dissects these features to provide clarity on what truly defines a memory-safe language.

A language is considered to be memory-safe when it automates memory allocation and garbage collection, with popular examples being Python, Java, and Go. The choice between memory-safe and manually managed languages depends on the specific use case and application requirements, balancing safety and control with performance and efficiency [24].

One key characteristic of memory-safe languages is their approach to memory management. While some languages employ automatic memory management, others opt for manual memory management for specific reasons. Memory-safe languages implement a range of various mechanisms to enhance safety like bounds checking, type safety, safe concurrency mechanisms, block identifiers, and other safety measures, collectively designed to prevent memory misuse. Block identifiers play a crucial role in ensuring memory safety within this language. They act as capabilities, restricting pointers to access memory associated with their respective identifiers. To prevent the disclosure of sensitive information, these identifiers are intentionally hidden from programs, ensuring they can only be used for referencing memory values and nothing more. Memory-safe languages are characterized by a diverse and multifaceted set of features. These features encompass memory management strategies, safety mechanisms, and error-handling techniques [25].

D. Case Studies of Memory-Safe Languages

Theory is most effective when grounded in practical examples. This section delves into several case studies of prominent memory-safe languages, like Rust, Python, and Go. Through an examination of their unique features, strengths, and weaknesses, this section offers a tangible understanding of how different languages approach the challenge of memory safety.

Highlighting the increasing reliance on Google web applications, Staniloiu et al. emphasize the need for memory-safe languages and the lack of safe offerings by Google [26]. They present a proof of concept integration of the memory-safe programming language D. The authors demonstrate that

D provides more security for users interacting through the graphical user interface (GUI) or the application programming interface (API) while also being straightforward to utilize with a library they created to syntactically resemble Java. Automated solutions for enforcing safety when mixing a memory-safe with a non-memory-safe language also exist, see this work by Kirth et al. [27].

The case study portion of the project investigates the practical achievement of memory safety in programming languages without incurring runtime overhead. It begins with C as a performance baseline and compares it with three memory-safe languages. The choice of programming languages for this study is pivotal in evaluating their efficacy in memory safety, shedding light on their suitability for real-world applications. It starts by comparing C as a performance baseline with three memory-safe languages, Ada, Rust, and C#, and evaluates their effectiveness for real-world applications. Ada stands out for its rich safety features, including non-null references and custom bounds for arrays. Rust was chosen due to its unique memory safety features and widespread usage. It enforces memory safety at compile-time and offers an "unsafe" dialect for advanced scenarios. C# was selected for its balance between safety through garbage collection and low-level features for flexibility [3].

Ada and Rust achieve memory management without runtime overhead through language features, enabling efficient memory pools. While Ada involves runtime checks for cross-pool memory operations, these can be avoided by using a single pool consistently. Rust relies solely on compile-time checks for memory safety, enhancing efficiency. In contrast, C# introduces runtime overhead in memory operations, even without active garbage collection. This trade-off prevents impractical full heap scans for large heaps. C# uses compiler-assisted reference tracking to pinpoint modifications. To address this trade-off, garbage-collected languages like C# can implement a basic form of arena-based memory management. It allows developers to declare stack-only types and a unique reference type for these stack-based types. This approach balances safety and speed, akin to Rust, with simplified usage [3].

a) Porting unsafe C code to Rust: The communal movement in software engineering to begin porting existing C code to Rust rose out of a need for enhancing safety and reliability while retaining compatibility with legacy systems. Rust's powerful memory safety guarantees make it an appealing choice, particularly in the realm of systems programming, where memory-related vulnerabilities can lead to severe consequences. The process involves the transformation of the existing C code base into Rust, capitalizing on Rust's memory safety features to address vulnerabilities inherent in the original code. However, in cases where the C code relies on low-level memory manipulations, direct hardware access, or intricate pointer arithmetic, Rust's 'unsafe' keyword emerges as a valuable resource. The use of 'unsafe' code sections permits controlled access to low-level operations, corresponding to the practices common in C programming. Furthermore, tools like c2rust streamline the conversion process by automating parts of the code translation

[28]. To enhance memory safety in Rust while securing existing C code, solutions such as X Rust can be employed [29]. This approach ensures a seamless transition and effectively mitigates most memory vulnerabilities (FC is also a valid tactic in this regard [30]). The combination of these strategies is helping to pave the road to a more natural and seamless transition from unsafe programming languages to memory-safe programming languages.

b) Software Hardening: When dealing with vulnerabilities in software development, particularly memory-related vulnerabilities, software hardening plays a pivotal role in bolstering applications against potential security threats. Software hardening is used to protect code from memory corruption, without changing the software's model away from unrestricted memory access [11]. This includes but is not limited to, protection from buffer overflow, stack overwriting, pointer masking, binary stirring, and control flow randomization. Modern compilers implement such tactics in the form of Control-Flow Integrity and SafeStack.

E. Comparison with Non-Memory-Safe Languages

Not all programming languages prioritize memory safety, often for related to performance, flexibility, or historical context reasoning [31]. This section juxtaposes memory-safe languages against their non-memory-safe counterparts, specifically C and C++. By exploring the trade-offs, challenges, and benefits of each, readers will gain a holistic understanding of the broader landscape of programming language design concerning memory management.

An additional definition for a memory-safe language is one where memory management is abstracted from the developer, reducing the risk of memory errors. Languages like Python, Java, and Go fall into this category. While this abstraction enhances safety, it may come at the cost of reduced efficiency and performance. Additionally, the garbage collection process in memory-safe languages can be unpredictable, affecting program responsiveness. On the other hand, manually managed languages provide developers with enhanced efficiency and performance as they have direct control over system memory with some exceptions. However, this control comes at the price of being more prone to memory-related failures. Without automated garbage collection, developers must explicitly manage memory allocation and deallocation, making it easier to introduce memory errors if not handled carefully [24]. Many efforts are underway to enhance the safety guarantees and capabilities of C, from basic semantic reinforcement to entire re-conceptualization of the heap functionality [32].

a) Efforts to Enhance Memory Safety in C/C++: Memory safety in C and C++ is an ongoing area of research, and there have been numerous attempts and strategies to address the challenges presented. The fundamental issue is these languages, while offering flexibility and high performance, do not inherently prevent out-of-bounds errors, which can lead to vulnerabilities and crashes. The following overview presents a

few chronological examples from research aiming to address these prevalent concerns.

Backwards Compatible Bounds Checking: In 2009, Akritidis et al. proposed a technique to safeguard C and C++ programs from out-of-bounds errors. Their method, unlike previous ones, efficiently checks bounds by constraining the sizes and alignment of allocated memory regions. This approach reduced performance overhead considerably, marking a significant improvement over earlier techniques [33].

Memory-Safe Interpretation of the C Abstract Machine: In 2015, Chisnall et al. re-examined C's memory model. They identified and addressed challenges in implementing a memory-safe version of C due to prevalent old idioms. By merging the experimental hardware features with fat pointers, they report a new interpretation using a modified C machine, capable of running legacy C code with an added guarantee of enhanced memory safety [34].

C/C++11 Memory Model Rectification: In 2017, Lahav et al. introduced the Repaired C11 (RC11) model, which refined sequential consistency access semantics. Establishing the need for their model, they underscored flaws in C/C++11's semantics of SC atomic accesses; this new model ensures sound compilation to the power architecture and offers stronger guarantees for SC fences [35].

b) MS-Wasm: WebAssembly is a byte-code language designed to enable not just C/C++ but various high-level languages to be compiled for execution in web browsers. Due to this versatility, WebAssembly has recently seen a massive growth in popularity. Disselkoen et al. observed that when compiling C/C++, the inherent memory vulnerabilities are retained; therefore, they developed a WebAssembly extension, called MS-Wasm, to encapsulate low-level semantics at compile time to enforce memory safety [36]. The extension provides programmers with configurable security-perform trade-offs and empowers them to strengthen and future-proof their code.

F. Real-world Implications of Memory Safety

Every code base in the digital sphere has repercussions in the real world. Whether it's a banking application managing millions or a simple utility tool, memory safety issues can have dire consequences, ranging from data breaches to substantial financial losses. In Ralf et al., Microsoft and the Chrome team at Google report that around 70% of the security vulnerabilities in their products are caused by memory safety violations [10], [37]. Potentially, this substantial percentage of devastating vulnerabilities could be decreased by writing vital pieces of the communication trail in a memory-safe language. Emmerich et al. discuss this idea in their 2019 work [38]. This section unveils the tangible repercussions of memory vulnerabilities by studying past incidents and emphasizing the pivotal role memory-safe languages play in a world increasingly reliant on digital solutions.

Theoretical security properties identified in memory safety discussions may not always apply to actual systems. This discrepancy arises partly due to inherent physical limitations,

such as finite memory, and the intricate ways real systems interact with users, often transcending simple input/output. There is a trade-off between integrity and secrecy. Some relaxation in memory safety may allow observers to gain insight into aspects of a program's global state. It is noted that the distinction between integrity and secrecy is not always clear-cut, as violations of secrecy can sometimes escalate into integrity breaches, particularly when security mechanisms depend on secrecy [25].

a) Attacks and Bugs: It is important to differentiate between the attacks that more easily exploit manually managed languages and to briefly discuss current offensive methods that bypass both manually managed languages and memory-safe languages. The list below details some of the attacks and errors commonly encountered in languages that are not memory-safe [39]:

- Use-after-free errors are caused by the dereferencing of dangling pointers, allowing users to manipulate the freed memory previously associated with the pointer.
- Double-Free errors are caused by "freeing a pointer twice" [39]. This error also allows the user to access the freed memory similarly to use-after-free errors.
- Buffer overflows occur when a system attempts to access memory beyond its allocated boundary.
- Uninitialized memory access occurs when an attempt to execute instructions in memory is made without initializing the memory.
- Concurrency bugs via memory corruption can occasionally allow data racing conditions, and this can lead to any of the errors mentioned above.
- Mixed binary code is the result of multiple compilers writing machine code with varying memory models. Mixed binary code can be vulnerable even when compiled in Rust if it is also compiled in C++. Papaevriptides et al, discuss methods of exploiting mixed binary code compiled in combinations of either C++ and Rust or C++ and Go [40].

Many of these bugs can be addressed using memory-safe languages. However, some attacks can bypass the mechanisms in memory-safe languages designed to safely handle memory. A speculative execution attack begins with an initial attack sending an input to the target to trigger the memory-safe properties deliberately [41]. The speculative execution attack then "overwrites the control-flow data" to "obtain a speculative control-flow hijack". This is done using Speculative Architectural Control-Flow Hijacks (SPEAR) to target vulnerable, or SPEAR-vulnerable as defined in Mambretti et al., code that eventually yields to the control-flow hijack [41]. From here, data is loaded from the side channel to eventually gain more speculative access to the system.

G. Future Directions

The field of software development is in perpetual motion, continuously evolving to meet the demands of a rapidly changing technological landscape. As a component of this evolution, memory safety is bound to see further innovations and shifts.

This section forecasts the future of memory-safe languages, discussing emerging techniques, potential integration with traditionally unsafe languages, and the rise of hardware-assisted memory safety methodologies.

The Go programming language, while equipped with robust memory protection mechanisms, does include an “unsafe” package to offer developers a degree of flexibility for specialized tasks. The use of this package can introduce an array of potential vulnerabilities in its frequent real-world usage. Wickert et al. [42] present Automated Classification of Unsafe Usages in Go (UNGOML), an automated classification system developed to identify the contexts and intentions behind the use of the unsafe package in Go. Utilizing deep learning classifiers that operate on enriched control-flow graphs (CFGs), UNGOML aids in contextualizing the employment of unsafe practices. Better understanding the project-specific underlying reasoning or need for developers to use an unsafe package enables quicker feasibility judgments and more efficient refactoring.

Utilizing the Memory Tagging Extension (MTE) recently implemented by major chip manufacturer arm®, known for and named after their architecture: Advanced RISC Machine, Seo et al. present a novel approach to deterministic spatial safety: Zone-Based Memory Tagging (ZOMETAG) [43]. Through hardware integration, ZOMETAG addresses spatial memory violations by assigning each object with permanent and unique tags that are distributed across zones, which are regions in memory. Enforcing a mechanism, deemed *two-layer isolation* to combine tags introduced by ARM with the zones defined in their approach, the authors report an increase in spatial safety.

As more developers transition to memory-safe programming languages, it becomes increasingly important that the tools used to prevent vulnerabilities are compatible with these languages. GitLab, a comprehensive DevOps platform, offers a robust feature set for securing application source code, particularly in the context of memory-safe languages. Some of the memory-safe languages supported by GitLab include Python, Go, Java, JavaScript, and Ruby. For each language, GitLab offers a combination of scanning tools to identify potential vulnerabilities and security issues. For Python, GitLab employs tools like Semgrep with GitLab-managed rules and Bandit. Similarly, Go is supported with Semgrep and GoSec. Java benefits from Semgrep and SpotBugs with the find-sec-bugs plugin, along with MobSF in beta. JavaScript code is scanned using Semgrep and ESLint security plugin, while Ruby code is checked using brakeman [24]. GitLab recognizes the shift toward memory-safe languages and provides a versatile and adaptable set of scanning tools to support developers in securing their code bases effectively.

Even though there’s a growing push toward the adoption of memory-safe languages, it remains impractical to eliminate manually managed languages from a developer’s toolbox. For the C language, GitLab utilizes Semgrep with GitLab-managed rules. This powerful tool helps pinpoint vulnerabilities related to bounds checking, ensuring the use of length-limiting functions, and validating that sizes are sufficient to handle maximum possible lengths. When it comes to C++, GitLab relies on Flawfinder to scan for potential issues. This complementary

tool contributes to a robust security posture by identifying vulnerabilities unique to the C++ language. These scanning capabilities in GitLab empower DevSecOps teams to detect and rectify security issues at an early stage of development, well before code reaches production environments [24]. Other innovations in this realm include data-flow analysis to detect bugs in Rust, specifically regarding false deallocation of memory [44] and the application of Rust to Zero Trust Architecture Development [45].

III. METHODS AND PROCEDURES

Our research focused on the impact of different programming languages on memory, so we needed a consistent way to monitor program behavior. To capture every step of a memory vulnerability, we incorporated an input monitoring loop in each program that required user interaction to trigger the vulnerability. This process involved obtaining the program’s process ID, initializing the monitor process script with this ID, and then returning to the compiler to begin the memory test while collecting all relevant data.

The monitoring script was a pivotal tool in our methodology. The script systematically recorded performance metrics of the target program, such as timestamp, CPU usage, and memory allocation, at regular intervals. These metrics were logged for subsequent analysis. When the program terminated, the script compiled the collected data into a comprehensive log file for detailed examination.

For our initial memory vulnerability analysis, we identified common vulnerabilities in languages with manual memory management, such as buffer overflows, use-after-free errors, and memory leaks, to create basic replications of them for analysis. Each of these vulnerabilities posed unique challenges:

- Buffer overflows occur when a program writes more data to a buffer than it can hold. This can overwrite adjacent memory locations, potentially corrupting data or even executing malicious code.
- Use-after-free errors occur when a program uses a pointer to memory that has already been freed. This can also lead to data corruption or (potentially malicious) code execution.
- Memory leaks occur when a program fails to release dynamically allocated memory back to the system. This can eventually lead to the program running out of memory and crashing.

To mitigate these risks, developers should adopt careful programming practices, such as utilizing bounds-checking functionality and avoiding dangling pointers. We supply full code for these vulnerabilities in Appendix B, with inline comments to detail functionality.

A. Vulnerability Execution Considerations for Memory-Safe Languages

Memory-safe programming languages are designed with built-in protections against common memory-related vulnerabilities, such as buffer overflows, use-after-free errors, and memory leaks. These languages prioritize correct memory usage,

with the compiler and runtime systems enforcing strict memory safety checks. This approach significantly alleviates the programmer's burden to anticipate potential errors, leading to more secure and reliable software. One of the key advantages of memory-safe languages is their robust defense against vulnerabilities that can compromise software security and reliability. For instance, buffer overflows, where a program writes data beyond the allocated buffer size, and use-after-free errors, where freed memory is still in use, are effectively countered. Additionally, these languages tackle memory leaks, where allocated memory is not freed, preventing the exhaustion of system resources. However, while the enhanced safety measures in memory-safe languages offer substantial benefits, they also introduce certain challenges. Testing and benchmarking for memory-related vulnerabilities can become more complex. The advanced safety features can obscure the identification and reproduction of memory-related issues during testing, sometimes leaving programmers in the dark about the underlying causes of bugs. Nevertheless, it is important to recognize that the safety improvements often outweigh these challenges, especially when considering the long-term security and reliability gains.

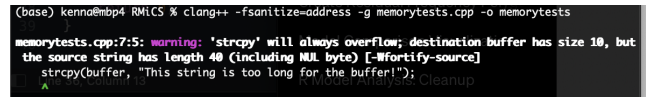
Memory-safe languages use techniques like garbage collection to reclaim unused memory and prevent leaks. They also use data structures and runtime checks to avoid buffer overflows and use-after-free errors. These features can complicate the testing process, but they play a critical role in reducing the likelihood and impact of vulnerabilities, thus enhancing overall software security [10], [11]. Memory-safe languages provide versatile and secure options for programmers, businesses, and organizations to incorporate into their development stack, particularly when updating legacy software. We present our procedures and tools used to analyze memory-related vulnerabilities and their varying presentations and manifestations in different memory-safe languages. This provides a comprehensive understanding of the strengths and limitations of these languages in the context of modern software development.

B. Analysis Procedures

In this section, details of the tools used for the test framework are given. Our approach is multi-faceted and incorporates powerful tools to understand memory management across various programming environments. The following subsections provide more insight into a few tools to identify, observe, and ultimately mitigate memory safety in software development.

1) *AddressSanitizer*: AddressSanitizer, a tool for C++, specializes in identifying and resolving memory safety violations, such as buffer overflows and use-after-free errors [46]. By instrumenting the program's memory accesses, AddressSanitizer monitors for out-of-bounds accesses and attempts to use already-freed memory. Upon detecting an error, it halts the program and presents a comprehensive error report, improving the understanding of memory safety issues in C++ code 3.

The utilization of AddressSanitizer extends across various prominent C++ applications and systems. Its application ranges from uncovering buffer overflows in critical software like the Linux kernel, OpenSSL library, and GNU Compiler Collection

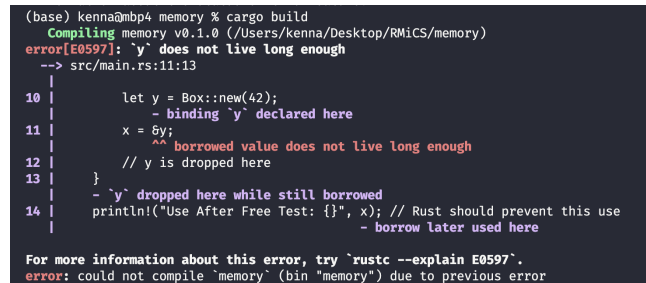


```
(base) kenna@mbp4 RMIcs % clang++ -fsanitize=address -g memorytests.cpp -o memorytests
memorytests.cpp:7:5: warning: 'strcpy' will always overflow; destination buffer has size 10, but
the source string has length 40 (including NUL byte) [-Wformat-source]
strcpy(buffer, "This string is too long for the buffer!");
      ^~~~~~
```

Fig. 3: AddressSanitizer catching a buffer overflow.

(GCC), to detecting use-after-free errors in large-scale projects like the Chromium web browser and the Android operating system [47].

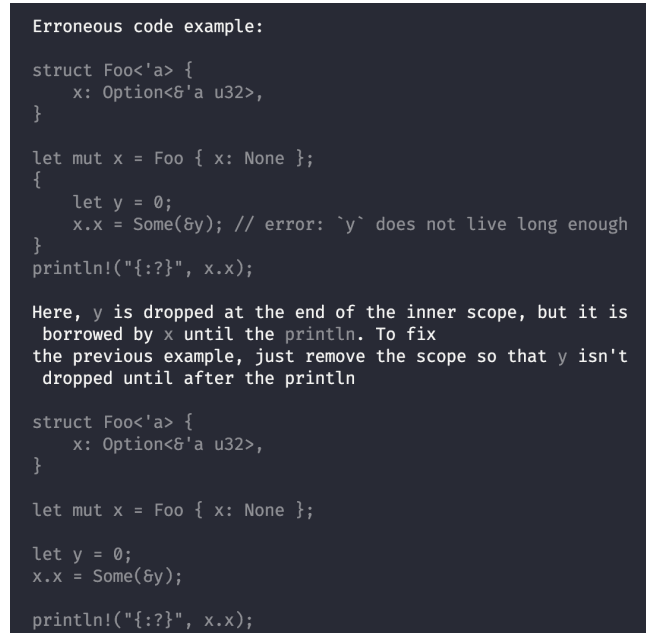
2) *Rust Compiler*: The Rust programming language's compiler played a major role in our analysis. Its ability to detect potential errors, such as buffer overflows and use-after-free incidents during the compilation phase, is particularly noteworthy. The compiler not only identifies these issues but also suggests corrections and provides erroneous code examples, guiding programmers toward memory-safe solutions 45. This proactive approach of the Rust compiler is a significant advantage in ensuring memory safety.



```
(base) kenna@mbp4 memory % cargo build
Compiling memory v0.1.0 (/Users/kenna/Desktop/RMIcs/memory)
error[E0597]: 'y' does not live long enough
--> src/main.rs:11:13
10 |         let y = Box::new(42);
    |         - binding 'y' declared here
11 |         x = &y;
    |         ^^ borrowed value does not live long enough
12 |         // y is dropped here
13 |     }
    |     - 'y' dropped here while still borrowed
14 |     println!("Use After Free Test: {}", x); // Rust should prevent this use
    |                                           - borrow later used here

For more information about this error, try `rustc --explain E0597`.
error: could not compile 'memory' (bin "memory") due to previous error
```

Fig. 4: Rust compiler communicating potential use-after-free error



```
Erroneous code example:

struct Foo<'a> {
    x: Option<&'a u32>,
}

let mut x = Foo { x: None };
{
    let y = 0;
    x.x = Some(&y); // error: 'y' does not live long enough
}
println!("{:?}", x.x);

Here, y is dropped at the end of the inner scope, but it is
borrowed by x until the println. To fix
the previous example, just remove the scope so that y isn't
dropped until after the println

struct Foo<'a> {
    x: Option<&'a u32>,
}

let mut x = Foo { x: None };

let y = 0;
x.x = Some(&y);

println!("{:?}", x.x);
```

Fig. 5: When prompted, the Rust compiler provides example erroneous code and suggests solutions.

The Rust compiler uses several techniques to detect potential memory safety errors. One technique is to use a type system that tracks the ownership of memory. This allows the

compiler to ensure memory is never freed while still in use. Another technique is to use a borrow checker that checks for references to deallocated memory. The borrow checker prevents dangling pointers, which can lead to use-after-free errors. Furthermore, the compiler incorporates additional safety mechanisms, such as stack guards and implementations of safe integer arithmetic, thereby providing a comprehensive defense against a broad spectrum of memory safety errors, including but not limited to buffer overflows and integer overflows.

3) *htop*: The *htop* tool is an interactive process viewer for Unix systems, offering a more user-friendly and visually appealing alternative to the classic *'top'* command. *htop* provides a dynamic, real-time view of a running system, displaying the list of processes and their resource usage. Its interface is color-coded and provides a visual representation of the system's CPU, memory, and swap usage, making it easier to understand the performance metrics at a glance.

IV. FINDINGS

Software security and reliability are paramount considerations in modern programming. The choice of programming language can significantly impact a project's vulnerability to common issues like the ones previously discussed. Each programming language has its own mechanisms and practices for addressing these concerns. Throughout our research, we have explored how different programming languages, including C, C++, Rust, Go, and Java (in some cases, Kotlin), handle these issues and the tools and features they employ to enhance the security and robustness of software applications.

In the C programming language, there is no inherent protection against issues like use-after-free, buffer overflows, and memory leaks. It relies on manual memory management and good coding practices. Tools like Valgrind can be used for detection. C is prone to buffer overflows due to the lack of bounds checking, and manual memory management can lead to memory leaks if not properly handled by the programmer.

C++ shares similarities with C but offers some improvements. It still lacks *inherent* protection, but the use of smart pointers, such as `std::unique_ptr`, can help prevent use-after-free and buffer overflow issues. Additionally, Standard Template Library (STL) containers and algorithms reduce the risk of buffer overflows. Smart pointers and the concept of RAII (Resource Acquisition Is Initialization) in C++ also contribute to preventing memory leaks.

In Rust, the ownership model is designed to prevent use-after-free errors by enforcing strict ownership rules. Safe Rust includes bounds checking, which effectively prevents buffer overflows. Rust's ownership and borrowing rules are instrumental in preventing memory leaks, although leaks can still occur in unsafe Rust code.

Go addresses these issues through its garbage collection mechanism, which helps mitigate memory-related problems; however, similar to Rust, improper use of unsafe pointers *can* still lead to vulnerabilities. Slice-and-array bounds checking in Go help prevent buffer overflows. The garbage collector is also beneficial in preventing memory leaks, although leaks may still occur if references are held longer than necessary.

In Java, the Java virtual machine's (JVM) garbage collection plays a significant role in mitigating memory-related issues. Use-after-free is rare due to the JVM's memory management. Array bounds checking in the JVM helps prevent buffer overflows. The garbage collector is effective at preventing memory leaks, although they can still occur if references are held longer than necessary.

Kotlin, managed by the JVM or using automatic reference counting in Kotlin/Native, reduces the risk of use-after-free, buffer overflows, and memory leaks. The JVM's garbage collection and bounds checking in Kotlin/Native contribute to this protection.

For a comprehensive comparative analysis of how these various programming languages tackle memory-related issues, refer to table II below. In it, the approaches employed by these languages in addressing these concerns are explored. This comparative overview can aid developers in making informed decisions when selecting the most suitable programming language for their specific software development needs.

V. ISSUES

A. Project Limitations

1) *Funding /or Time /or Tooling Limitations*: One limitation constantly at the forefront of our team's mind was time or lack thereof. Since the duration of this project was constrained to a semester, the research was subjected to a fast pace. Granted, if time was not a constraint, a more in-depth analysis regarding adopting memory-safe languages for systems programming would have been the overarching goal.

2) *Learning Curve*: Established programmers with experience in manually managed programming languages are presented with a learning curve when considering program design. The complexity of Rust and the paradigm shift in coding practices required additional time for upskilling and adjusting to new methodologies such as Rust's ownership-based resource management model. While practice was conducted to facilitate this transition, the learning curve greatly influenced the initial project velocity.

B. Unforeseen Circumstances

1) *Environmental Challenges*: The project encountered unexpected impediments associated with configuring and maintaining the requisite development environments for multiple programming languages. These complexities led to delays in establishing consistent and stable environments across the development team. Configuring tools, libraries, and a testing environment (via Docker) for the languages posed unanticipated difficulties, impacting the initial stages of code analysis and migration efforts. Consequently, this necessitated additional time allocation and collaborative problem-solving to ensure a uniform and functional environment conducive to effective code assessment.

2) *Code Analysis*: The code analysis phase confronted challenges within the scope of extracting data from sections of code that contained vulnerabilities preventing compilation. Identifying and isolating these vulnerable segments posed difficulties in gathering comprehensive data for vulnerability assessment. These issues hampered the evaluation process and

TABLE II: Comparison of Memory Issues in Different Programming Languages

Language		Memory Issues	
C	<ul style="list-style-type: none"> • No inherent protection • Relies on manual management and good coding practices • Can use tools like Valgrind for detection • Prone to buffer overflows due to lack of bounds checking • Manual memory management can lead to leaks if not properly handled 	C++	<ul style="list-style-type: none"> • Similar to C, but smart pointers can help prevent issues • STL containers and algorithms reduce the risk of buffer overflows • Smart pointers and RAII (Resource Acquisition Is Initialization) help prevent leaks
Rust	<ul style="list-style-type: none"> • Ownership model prevents use-after-free by design • Bounds checking in safe Rust prevents buffer overflows • Ownership and borrowing rules help prevent memory leaks (unsafe Rust can still have leaks) 	Go	<ul style="list-style-type: none"> • Garbage collection mitigates memory issues • Improper use of unsafe pointers can lead to vulnerabilities • Slice and array bounds checking help prevent buffer overflows • Garbage collector helps prevent memory leaks, but leaks can occur due to long-held references
Kotlin	<ul style="list-style-type: none"> • Managed by the JVM or automatic reference counting in Kotlin/Native • Reduces the risk of memory issues • Garbage collection in JVM and bounds checking in Kotlin/Native help prevent leaks 	Java	<ul style="list-style-type: none"> • Garbage collection mitigates memory issues • Use-after-free is rare due to JVM's memory management • Array bounds checking in JVM helps prevent buffer overflows • Garbage collector helps prevent memory leaks, but leaks can occur due to long-held references

necessitated alternative methods for data extraction, impacting the overall analysis of specific vulnerabilities.

C. Language-Specific Issues

1) *Memory-Safe Vs Non-Memory-Safe languages:* Memory-safe languages offer robust safeguards against common memory-related vulnerabilities, contrasting with non-memory-safe languages like C and C++. However, despite their advantages, memory-safe languages present certain limitations. One notable limitation lies in their performance overhead. While safety mechanisms, such as bounds checking and automatic memory management, significantly enhance security, they often incur runtime costs, impacting execution speed and resource utilization. This overhead becomes more pronounced in systems programming scenarios where microseconds matter.

In contrast, non-memory-safe languages like C and C++ prioritize performance and low-level control over memory safety. They allow direct memory manipulation and lack built-in memory safety features, making them susceptible to memory vulnerabilities like buffer overflows or dangling pointers. Yet, they often outperform memory-safe languages due to their minimal runtime overhead. The trade-off between security and performance remains a division in the transition to memory-safe languages.

2) *Limitations of Memory-Safe Languages:* The limitations of memory-safe languages can be broken up into three sections: Performance Overhead, Learning Curve and Adoption Challenges, and Limited Tools and Features.

Some memory-safe languages incur a performance overhead due to the complexity of garbage collection features and runtime checks, the reason they are memory-safe. This can clash with some of the original reasons systems programming languages like C and C++ even exist; they are meant to be fast as microseconds begin to matter when dealing with system features and OS programming.

Transitioning from non-memory-safe languages to newer memory-safe languages in itself requires a learning curve, though the degree of this depends on various factors such as documentation, community support, and pre-built tools and libraries required to use for certain tasks such as network programming. The paradigm shift to memory-safe languages often demands a rethinking of programming methodologies, introducing new concepts like ownership, borrowing, and lifetimes. This learning curve can hinder the adoption of these languages within teams or organizations already established in non-memory-safe languages.

Compared to older, more established languages, the environment around memory-safe languages may be a relatively immature one. This can result in a scarcity of libraries, tools, or specialized frameworks, making it more challenging when specific development requirements are needed.

3) *Compatibility with Legacy Code:* Depending on the language, integrating memory-safe languages with existing legacy code bases, particularly earlier systems programming, can raise challenges in terms of compatibility. Legacy code often includes practices that lack memory safety. The conversion of legacy code to a memory-safe language will require refactor-

ing, at varying degrees, to adhere to new language paradigms. Although there are already certain tools to aid in this process, some of which show promising results (C2Rust), the issue still stands for memory safety as well as explicitly needing a built-in feature that allows 'unsafe' code to bypass security checks.

4) *Community*: The knowledge and size of a programming language's community, including the availability of forums, user groups, active contributors, and company use, greatly affects the success of transitioning and usage of that language. If proper documentation is too difficult to obtain or a language lacks vast community support, there is much less appeal to use these memory-safe languages over well-established counterparts. A flourishing community fosters collaboration, offers support for newcomers, and contributes to the development of much-needed tools, libraries, and frameworks, which all contributes to the adoption of these languages.

VI. CONCLUSIONS AND RECOMMENDATIONS

Although it is optimal to choose the most applicable programming language based on the goal, the team highly encourages the consideration of implementing a memory-safe language before deciding to implement a non-memory-safe programming language. In this paper, our team discussed some challenges related to manually managing memory and provided evidence that supports the inherent security of memory-safe languages without introducing additional overhead. As this special type of language continues to grow in popularity, improved functionalities and additional extensions will expand the scope of the languages and can aid in the effort to eradicate preventable memory-related vulnerabilities.

In conclusion of the project, our research findings support the migration towards utilizing more memory-safe languages. The duration of this project allowed the team to collect evidence that shows memory-safe languages have the inherent capabilities to reduce memory-related vulnerabilities in user-developed programs. By participating in this research project, our team has gained more awareness regarding safe coding practices in hopes of decreasing the amount of exploitable vulnerabilities within source code.

VII. ACKNOWLEDGEMENTS

Our research team would like to express gratitude to all those involved with the development of our project. We would like to thank Neal Ziring for choosing to be our problem mentor as well as Dr. Sudip Mittal for leading our team as faculty lead. We would like to extend our thanks to Justine Ho and those involved with the INSURE Program and the Centers of Academic Excellence (CAE) Community. We would also like to thank you, the reader, for reading our final report. Without these listed individuals, this learning opportunity would have not been revealed to us.

REFERENCES

- [1] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Softbound: Highly compatible and complete spatial memory safety for c," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 245–258. [Online]. Available: <https://doi.org/10.1145/1542476.1542504>
- [2] J. Meza, Q. Wu, S. Kumar, and O. Mutlu, "Revisiting memory errors in large-scale production data centers: Analysis and modeling of new trends from the field," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2015, pp. 415–426.
- [3] S. Pirelli, "Safe low-level code without overhead is practical," in *Proceedings of the 45th International Conference on Software Engineering*, ser. ICSE '23. IEEE Press, 2023, p. 2173–2184. [Online]. Available: <https://doi.org/10.1109/ICSE48619.2023.00183>
- [4] J. Caballero, G. Grieco, M. Marron, and A. Nappa, "Undangle: Early detection of dangling pointers in use-after-free and double-free vulnerabilities," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ser. ISSTA 2012. New York, NY, USA: Association for Computing Machinery, 2012, p. 133–143. [Online]. Available: <https://doi.org/10.1145/2338965.2336769>
- [5] V. Astrauskas, A. Bílý, J. Fiala, Z. Grannan, C. Matheja, P. Müller, F. Poli, and A. J. Summers, "The prusti project: Formal verification for rust," in *NASA Formal Methods*, J. V. Deshmukh, K. Havelund, and I. Perez, Eds. Cham: Springer International Publishing, 2022, pp. 88–108.
- [6] Y. Matsushita, X. Denis, J.-H. Jourdan, and D. Dreyer, "Rusthornbelt: A semantic foundation for functional verification of rust programs with unsafe code," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 841–856. [Online]. Available: <https://doi.org/10.1145/3519939.3523704>
- [7] F. A. Wolf, L. Arquint, M. Clochard, W. Oortwijn, J. C. Pereira, and P. Müller, "Gobra: Modular specification and verification of go programs (extended version)," in *International Conference on Computer Aided Verification*, 2021.
- [8] D. M. Ritchie, "The development of the c language." ACM Sigplan Notices, 1993. [Online]. Available: <https://www.bell-labs.com/usr/dmr/www/chist.pdf>
- [9] S. A. Ebad, "Investigating the input validation vulnerabilities in c programs input validation in c," *International Journal of Advanced Computer Science and Applications*, vol. 14, 2023.
- [10] R. Jung, "Understanding and evolving the rust programming language," Ph.D. dissertation, Universität des Saarlandes, 2020.
- [11] M. Papaevripides and E. Athanasopoulos, "Exploiting mixed binaries," *ACM Trans. Priv. Secur.*, vol. 24, no. 2, jan 2021. [Online]. Available: <https://doi.org/10.1145/3418898>
- [12] N. D. Matsakis and F. S. Klock, "The rust language," in *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, ser. HILT '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 103–104. [Online]. Available: <https://doi.org/10.1145/2663171.2663188>
- [13] R. Pike, "Go at google: Language design in the service of software engineering. online," URL: <https://go.dev/talks/2012/splash.article>, 2012.
- [14] B. E. Team, "Bairesdev blog: Insights on software development," *BairesDev Blog: Insights on Software Development & Tech Talent*, Aug 2022. [Online]. Available: <https://www.bairesdev.com/blog/companies-using-golang/>
- [15] "Security best practices for go developers." [Online]. Available: <https://go.dev/security/best-practices>
- [16] M. Chabbi and M. K. Ramanathan, "A study of real-world data races in golang," in *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2022, pp. 474–489.
- [17] "Go 1.17 release notes." [Online]. Available: <https://go.dev/doc/go1.17>
- [18] F. Lardinois, "Five years later, google is still all-in on kotlin," Aug 2022. [Online]. Available: <https://techcrunch.com/2022/08/18/five-years-later-google-is-still-all-in-on-kotlin/>
- [19] D. Jemerov and S. Isakova, *Kotlin in action*. Simon and Schuster, 2017.
- [20] A. Ginani, "What is kotlin and why use it for app development? - 2023 guide," May 2023. [Online]. Available: https://medium.com/@elijah_williams_agc/what-is-kotlin-and-why-use-it-for-app-development-2023-guide-187661aa2f97
- [21] D. Henry and M. Yahya, "Netflix android and ios studio apps-now powered by kotlin multiplatform," Oct 2020. [Online]. Available: <https://netflixtechblog.com/netflix-android-and-ios-studio-apps-kotlin-multiplatform-d6d4d8d25d23>
- [22] A. Skantz, "Performance evaluation of kotlin multiplatform mobile and native ios development in swift," 2023.

- [23] B. Qin, Y. Chen, Z. Yu, L. Song, and Y. Zhang, "Understanding memory and thread safety practices and issues in real-world rust programs," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 763–779. [Online]. Available: <https://doi.org/10.1145/3385412.3386036>
- [24] F. Diaz, "How to secure memory-safe vs. manually managed languages," 3 2023. [Online]. Available: <https://about.gitlab.com/blog/2023/03/14/memory-safe-vs-unsafe/>
- [25] A. Azevedo de Amorim, C. Hrițcu, and B. C. Pierce, "The meaning of memory safety," in *Principles of Security and Trust*, L. Bauer and R. Küsters, Eds. Cham: Springer International Publishing, 2018, pp. 79–105.
- [26] E. Staniloiu, R. Nitu, R. Aron, and R. Rughinis, "Extending client-server api support for memory safe programming languages," in *2021 20th RoEduNet Conference: Networking in Education and Research (RoEduNet)*, 2021, pp. 1–5.
- [27] P. Kirth, M. Dickerson, S. Crane, P. Larsen, A. Dabrowski, D. Gens, Y. Na, S. Volckaert, and M. Franz, "Pkru-safe: Automatically locking down the heap between safe and unsafe languages," in *Proceedings of the Seventeenth European Conference on Computer Systems*, ser. EuroSys '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 132–148. [Online]. Available: <https://doi.org/10.1145/3492321.3519582>
- [28] M. Emre, R. Schroeder, K. Dewey, and B. Hardekopf, "Translating c to safer rust," *Proc. ACM Program. Lang.*, vol. 5, no. OOPSLA, oct 2021. [Online]. Available: <https://doi.org/10.1145/3485498>
- [29] P. Liu, G. Zhao, and J. Huang, "Securing unsafe rust programs with xrust," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 234–245. [Online]. Available: <https://doi.org/10.1145/3377811.3380325>
- [30] H. M. J. Almhori and D. Evans, "FideliuS charm: Isolating unsafe rust code," in *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 248–255. [Online]. Available: <https://doi.org/10.1145/3176258.3176330>
- [31] T. Weis, M. Waltereit, and M. Uphoff, "Fyr: A memory-safe and thread-safe systems programming language," in *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, ser. SAC '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1574–1577. [Online]. Available: <https://doi.org/10.1145/3297280.3299741>
- [32] S. Hong, J. Lee, J. Lee, and H. Oh, "Saver: Scalable, precise, and safe memory-error repair," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 271–283. [Online]. Available: <https://doi.org/10.1145/3377811.3380323>
- [33] P. Akritidis, M. Costa, M. Castro, and S. Hand, "Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors," in *USENIX Security Symposium*, 2009.
- [34] D. Chisnall, C. Rothwell, R. N. Watson, J. Woodruff, M. Vadera, S. W. Moore, M. Roe, B. Davis, and P. G. Neumann, "Beyond the pdp-11: Architectural support for a memory-safe c abstract machine," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 117–130. [Online]. Available: <https://doi.org/10.1145/2694344.2694367>
- [35] O. Lahav, V. Vafeiadis, J. Kang, C.-K. Hur, and D. Dreyer, "Repairing sequential consistency in c/c++11," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 618–632. [Online]. Available: <https://doi.org/10.1145/3062341.3062352>
- [36] C. Disselkoben, J. Renner, C. Watt, T. Garfinkel, A. Levy, and D. Stefan, "Position paper: Progressive memory safety for webassembly," in *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: <https://doi.org/10.1145/3337167.3337171>
- [37] J. Liu and C. Xu, "Pwning microsoft edge browser: From memory safety vulnerability to remote code execution," 2018.
- [38] P. Emmerich, S. Ellmann, F. Bonk, A. Egger, E. G. Sanchez-Torija, T. Gunzel, S. D. Luzio, A. Obada, M. Stadlemeier, S. Voit, and G. Carle, "The case for writing network drivers in high-level programming languages," in *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2019. [Online]. Available: <https://doi.ieeecomputersociety.org/10.1109/ANCS.2019.890189>
- [39] H. Xu, Z. Chen, M. Sun, Y. Zhou, and M. Lyu, "Memory-safety challenge considered solved? an in-depth study with all rust cves," 2021.
- [40] M. Papaevripides and E. Athanasopoulos, "Exploiting mixed binaries," *ACM Trans. Priv. Secur.*, vol. 24, no. 2, jan 2021. [Online]. Available: <https://doi.org/10.1145/3418898>
- [41] A. Mambretti, A. Sandulescu, A. Sorniotti, W. Robertson, E. Kirda, and A. Kurmus, "Bypassing memory safety mechanisms through speculative control flow hijacks," in *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2021, pp. 633–649.
- [42] A.-K. Wickert, C. Damke, L. Baumgärtner, E. Hüllermeier, and M. Mezini, "Ungoml: Automated classification of unsafe usages in go," in *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, 2023, pp. 309–321.
- [43] J. Seo, J. You, D. Kwon, Y. Cho, and Y. Paek, "Zometag: Zone-based memory tagging for fast, deterministic detection of spatial memory violations on arm," *IEEE Transactions on Information Forensics and Security*, vol. 18, pp. 4915–4928, 2023.
- [44] M. Cui, C. Chen, H. Xu, and Y. Zhou, "Safedrop: Detecting memory deallocation bugs of rust programs via static data-flow analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 4, may 2023. [Online]. Available: <https://doi.org/10.1145/3542948>
- [45] D. Hardin, "Hardware/software co-assurance for the rust programming language applied to zero trust architecture development," *ACM SIGAda Ada Letters*, vol. 42, no. 2, pp. 55–61, 2023.
- [46] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *USENIX ATC 2012*, 2012. [Online]. Available: <https://www.usenix.org/conference/usenixfederatedconferencesweek/addresssanitizer-fast-address-sanity-checker>
- [47] "The llvm compiler infrastructure project." [Online]. Available: <https://llvm.org/>

APPENDIX A

BIOGRAPHICAL SKETCHES OF THE INVESTIGATORS

Daniel Molsbarger, a graduate student set to graduate this semester, has experience in C, C++, Python, and JavaScript. He has knowledge in Parallel Programming, Database Management, Machine Learning, Artificial Intelligence, and Software Engineering, with a foundational understanding of security. Currently, he's studying Secure Software Engineering and is enrolled in his third Machine Learning course. His research focuses on Human-Computer Interactions, specifically Virtual Reality.

George Kotti is an undergraduate at Mississippi State University in the Computer Science and Engineering program. He is familiar with both Windows and UNIX-based operating systems and has completed courses in Operating Systems, Computer Networking, Foundations of Comp Sci, Secure Software Dev, and Memory Management (C). He is working towards his Security+ Certification. George is proficient in C, C++, C#, Rust, Python, Relational Algebra (SQL), and Robotics. He has developed applications that address Concurrency/Parallel programming, Network Scanning, and Device Driver mimicry. He has also worked as a Full-Stack developer in team projects.

Kenna Henkel holds a bachelor's degree in software engineering and is in her graduate studies focusing on human-centered computing. She is involved in the design and development of a social robot at Mississippi State University. Her interest lies in memory-safe languages due to the importance of system security. During her time at MSU, she completed various software and security courses, including the 2019 Secure Software Engineering course by Facebook and Codepath, which led to a scholarship for Black Hat/DEFCON.

Kaneesha Moore is a senior in computer science with a strong academic background. She has completed courses such as Data Structures and the Analysis of Algorithms, Intro to Algorithms, Computer Organization, Systems Programming, and Operating Systems. Currently, she is expanding her knowledge by studying the Theory and Implementation of Programming Languages. Kaneesha's proficiency in C has been honed through various academic courses, and her expertise in C++ has been sharpened through her involvement in multiple research projects.

Prathyusha Mustiyala is a senior in computer science. She has completed courses in Systems Programming, Operating Systems, Artificial Intelligence, Data Structures, and Intro to Algorithms. Last semester, she studied Intro to Software Engineering and Theory and Implementation of Programming Languages. She is proficient in C, C++, Python, and HTML, skills she acquired through coursework and research projects.

Wilson Patterson a senior majoring in Cybersecurity, plans to pursue a master's degree next academic year. He has conducted research in Machine Learning, AI, and Cyber Physical Systems. He has also contributed to research on Digital Twins and Digital Twin Security. His academic background includes systems programming and operating systems, primarily in C and C++.

APPENDIX B CODE EXAMPLES

Buffer Overflow

C example

```

1
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <string.h>
5 #define BUFSIZE 16
6
7 int main(void) {
8     // Allocate 16 bytes of memory to 'buffer'
9     char* buffer = (char*)malloc(BUFSIZE * sizeof(char));
10    memset(buffer, 0, sizeof(buffer));
11
12    // Assign a string larger than 16 bytes to 'msg'
13    char msg[64] = "This is a string larger than the size of buffer";
14
15    // Attempt to copy 'msg' into 'buffer'
16    strncpy(buffer, msg, strlen(msg)); // Buffer Overflow occurs here
17
18    printf("Buffer String: %s \n", buffer); // Will produce unexpected results
19
20    free(buffer);
21    return 0;
22 }
```

Listing 1: C example

C++ example

```

1 #include <iostream>
2 #include <vector>
3
4 void bufferOverflow() {
5     int arr[10];
6     // Intentionally accessing out of bounds
7     arr[15] = 1; // Buffer overflow
8 }
9
10 int main() {
11     std::cout << "Press Enter to start the memory tests..." << std::endl;
12     std::cin.ignore();
13
14     bufferOverflow();
15
16     std::cout << "Memory tests completed. Press Enter to finish..." << std::endl;
17     std::cin.ignore();
18     return 0;
19 }
```

Listing 2: C++ example

Rust example

```

1
2 fn buffer_overflow() {
3     let mut vec = vec![0; 10];
4     // Attempting unsafe access - Rust will prevent this at compile time
5     vec[15] = 1;
6 }
7
8 fn main() {
9     // Prompt for input before starting the tests
10    println!("Press Enter to start the memory leak test...");
11    let mut input = String::new();
12    std::io::stdin().read_line(&mut input).unwrap();
13
14    buffer_overflow();
15
16
17    // End of program
18    println!("Memory leak test completed. Press Enter to finish...");
19    std::io::stdin().read_line(&mut input).unwrap();
20 }
```

Listing 3: Rust example

Go example

```

1 package main
2 import "fmt"
3
4
5 func main() {
6     // Allocate 5 bytes of memory to 'buffer'
7     buffer := make([]byte, 5)
8
9     // Assign a string longer than the size of 'buffer'
10    input := "String longer than the size of buffer"
11
12    // Attempt to copy data from 'input' over to 'buffer'
13    copy(buffer, []byte(input)) // Buffer Overflow Prevented Here
14
15    // Go's memory safety checks will catch this and throw a runtime panic, due to bounds check
16    // failure, halting this process.
17 }

```

Listing 4: Go example

Python example

```

1 def main():
2     # Create a byte array of size 5
3     buffer = bytearray(5);
4
5     # Assign a string longer than 5 bytes to 'input'
6     input_str = "String longer than 5 bytes"
7
8     # Attempt to copy input into buffer
9     buffer[:len(input_str)] = input_str.encode()
10
11    # Python's memory safety checks will detect this operation and will dynamically resize the string
12    # as needed
13    # allowing the program to safely print 'buffer'
14    print(buffer)
15
16 if __name__ == "__main__":
17     main()

```

Listing 5: Python example

*Use-After-Free**C example*

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(void) {
5     // Allocate memory for an integer
6     int* ptr = (int*)malloc(sizeof(int));
7
8     // Assign a value to the allocated memory
9     *ptr = 10;
10
11     if(ptr != NULL)
12         free(ptr); // Free the allocated memory
13
14     // Use the memory after it has been freed
15     int value = *ptr; // Use-After-Free vulnerability occurs here
16
17     return 0;
18 }

```

Listing 6: C example

C++ example

```

1 #include <iostream>
2 #include <vector>
3
4 void useAfterFree() {
5     int* ptr = new int(42);
6     delete ptr;
7     // Intentionally using after free
8     std::cout << "Use After Free Test: " << *ptr << std::endl;
9 }
10
11 int main() {
12     std::cout << "Press Enter to start the memory tests..." << std::endl;
13     std::cin.ignore();
14
15     useAfterFree();
16
17     std::cout << "Memory tests completed. Press Enter to finish..." << std::endl;
18     std::cin.ignore();
19     return 0;
20 }

```

Listing 7: C++ example

Rust example

```

1 fn use_after_free() {
2     let x;
3     {
4         let y = Box::new(42);
5         x = &y;
6         y is dropped here
7     }
8     println!("Use After Free Test: {}", x); // Rust prevents this use
9 }
10
11 fn main() {
12     // Prompt for input before starting the tests
13     println!("Press Enter to start the memory leak test...");
14     let mut input = String::new();
15     std::io::stdin().read_line(&mut input).unwrap();
16
17     use_after_free();
18
19
20     // End of program
21     println!("Memory leak test completed. Press Enter to finish...");
22     std::io::stdin().read_line(&mut input).unwrap();
23 }

```

Listing 8: Rust example

Go example

```

1 package main
2
3 import "fmt"
4
5 func main() {
6
7     //Allocate memory for an integer
8     data := new(int)
9     *data = 42
10
11     //Access the memory safely
12     fmt.Printf("Value: %d\n", *data)
13
14 }

```

Listing 9: Golang example

Kotlin example

```

1 fun main() {
2
3     //Allocate memory for an integer
4     val data: Int? = 42
5
6     //Access the memory safely
7     println("Value: $data")
8
9 }

```

Listing 10: Kotlin example

Python example

```

1 def main():
2
3     # Allocate memory for an integer
4     data = 42
5
6     # Access the memory safely
7     print(f"Value: {data}")
8
9 if name == "main":
10     main()

```

Listing 11: Python example

Double Free C example

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #define BUFSIZE 16
5
6 void DoubleFree(void){
7     // Allocate 16 bytes to 'buffer'
8     char* buffer = (char*)malloc(BUFSIZE * sizeof(char));
9
10    // Assign and copy 'msg' into 'buffer'
11    char msg[8] = "message";
12    strncpy(buffer, msg, strlen(msg));
13
14    if(buffer != NULL)
15        free(buffer); // Free 'buffer' for the 1st time
16
17
18    free(buffer); // Free 'buffer' again // Here is where the Double Free vulnerability lies, causing
19    // unintended behavior
20 }
21
22 int main(void){
23     puts "... Program Executing ...");
24
25     DoubleFree();
26
27     puts "... Memory Test Run ... ");
28
29     return 0;
30 }

```

Listing 12: C example

C++ example

```

1
2 #include <iostream>
3 #include <cstdlib>
4 using std::cout;
5 using std::endl;
6
7 void DoubleFree(void){
8     // Allocate memory to 'integer_buffer' of size 'int'
9     int *integer_buffer = new int;
10
11    if(integer_buffer != nullptr)
12        delete integer_buffer; // Freeing the memory for the 1st time
13
14    delete integer_buffer; // Freeing the memory for a 2nd time // Double Free Vulnerability occurs
15    // here
16 }
17
18 int main(void){
19     cout << "... Program Executing ... " << endl;
20
21     DoubleFree();
22
23     cout << "... Memory Test Run ..." << endl;
24     return 0;
25 }

```

Listing 13: C++ example

Rust example

```

1
2 fn main(){
3     // Allocate 32 bytes to 'ptr'
4     let ptr = Box::new(32);
5     // 'Box' provides ownership and manages memory accordingly

```

```

6 // No explicit deallocation required
7 // Memory is automatically freed when 'ptr' goes out of scope
8 }

```

Listing 14: Rust example

Go example

```

1 package main
2
3 func main() {
4     //Using 'new' to allocate memory for an int
5     ptr := new(int)
6     *ptr = 1;
7
8     //No explicit deallocation needed;
9     //memory is automatically freed by the garbage collector
10
11 }

```

Listing 15: Golang example

Python3 example

```

1
2 def main():
3     # Assign a string to 'str_memory'
4     str_memory = "Hello Python World!"
5
6     print(str_memory)
7
8     # 'str_memory' is automatically deallocated when it is no longer referenced
9
10 if __name__ == "__main__":
11     main()

```

Listing 16: Python3 example

Memory Leak

C example

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #define BUFSIZE 16
5
6 void MemoryLeak(void){
7     // Allocate memory of size 'BUFSIZE' to 'func_buffer' inside the scope of 'MemoryLeak'
8     char* func_buffer = (char*)malloc(BUFSIZE * sizeof(char));
9
10    // Assign a string for use in 'MemoryLeak' to 'msg'
11    char msg[8] = "message";
12
13    // Copy 'msg' to 'func_buffer'
14    strncpy(func_buffer, msg, strlen(msg));
15
16    // free(func_buffer); // Not freeing 'func_buffer'
17    // This causes 'func_buffer's memory block to remain allocated even after the function's execution
18    // , causing potential issues
19 }
20
21 int main(void){
22     puts("... Program Executing ...");
23
24     MemoryLeak();
25
26     puts("... Memory Test Run ...");
27
28     return 0;
29 }

```

Listing 17: C example

C++ example

```

1 #include <iostream>
2 #include <vector>
3
4 void memoryLeak() {
5     int* leak = new int(1);
6     // Intentionally not freeing 'leak'
7 }
8
9 int main() {
10    std::cout << "Press Enter to start the memory tests..." << std::endl;
11    std::cin.ignore();
12
13    // Loop control
14    const long long iterations = 1000000000;
15    for (long long i = 0; i < iterations; ++i) {
16        memoryLeak();
17    }
18
19    std::cout << "Memory tests completed. Press Enter to finish..." << std::endl;
20    std::cin.ignore();
21    return 0;
22 }

```

Listing 18: C++ example

Rust example

```

1 fn memory_leak() {
2     use std::rc::Rc;
3     use std::cell::RefCell;
4
5     let a = Rc::new(RefCell::new(1));
6     let b = Rc::clone(&a);
7
8     *a.borrow_mut() += 1;
9     *b.borrow_mut() += 1;
10
11     // Intentional reference cycle - memory leak
12     // a and b are not dropped
13 }
14
15 fn main() {
16     // Prompt for input before starting the tests
17     println!("Press Enter to start the memory leak test...");
18     let mut input = String::new();
19     std::io::stdin().read_line(&mut input).unwrap();
20
21     // Loop control
22     let iterations = 1000000000;
23     for _ in 0..iterations {
24         memory_leak();
25     }
26
27     // End of program
28     println!("Memory leak test completed. Press Enter to finish...");
29     std::io::stdin().read_line(&mut input).unwrap();
30 }

```

Listing 19: Rust example

Monitor Process

Upon execution, this bash script takes a PID (process id) as input and begins monitoring that process. It logs usage of system resources and a timestamp to a file for analysis.

```

1 #!/bin/bash
2
3 if [ -z "$1" ]; then
4     echo "Usage: $0 <PID>"
5     exit 1
6 fi
7
8 PID=$1
9 LOGFILE="process_monitor_log_${PID}.txt"
10 INTERVAL=5
11
12 echo "Timestamp, %CPU, %MEM" > $LOGFILE
13
14 get_process_info() {
15     ps -p $PID -o %cpu -o %mem | tail -n 1
16 }
17
18 while true; do
19     if ! ps -p $PID > /dev/null; then
20         echo "Process $PID has exited."
21         break
22     fi
23
24     INFO=$(get_process_info)
25     TIMESTAMP=$(date +"%Y-%m-%d %H:%M:%S")
26     echo "$TIMESTAMP, $INFO" >> $LOGFILE
27     sleep $INTERVAL
28 done
29
30 echo "Monitoring finished. Data logged to $LOGFILE."

```

Listing 20: Bash script