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

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

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 have 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. 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.

II. Background

Memory safety, although a technical term, has ramifications that affect even the most casual computer users. From data breaches to system crashes, memory vulnerabilities can lead to significant disruptions. Memory-safe languages are programming languages that are designed to prevent encoding memory vulnerabilities into software systems, and this is often done 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. [1] To better appreciate the solutions presented by memory-safe languages, it's vital first to understand the definition of memory safety and the common memory-related vulnerabilities that developers encounter. Two programming languages that are formally verified to be memory safe are **Rust** [2], [3] and **Go** [4]. Throughout this work, we will discuss these languages and a selection of others in detail.

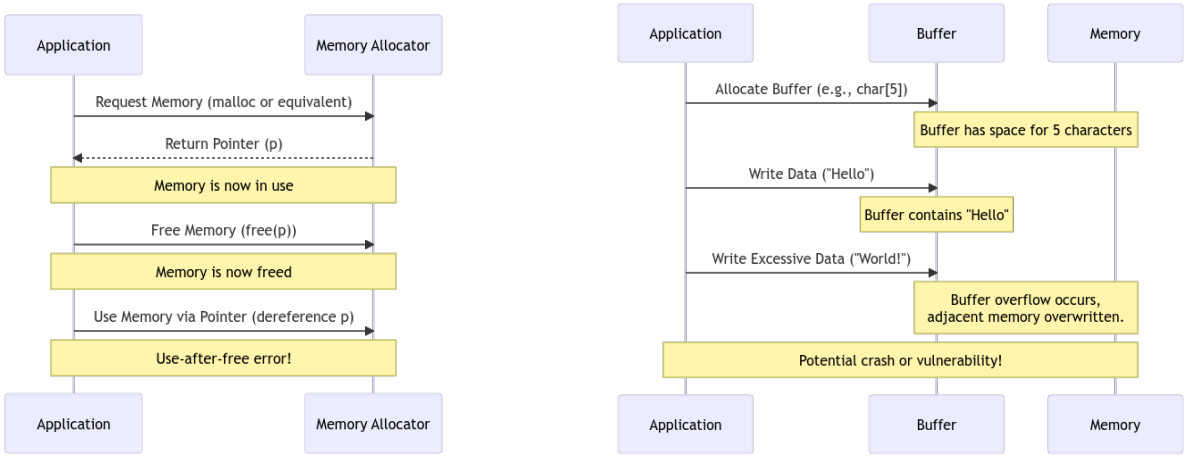
A. The C Family

The C language, a procedural-oriented programming language developed in the early 1970s by Dennis Ritchie at Bell Laboratories, holds a unique place in the history of computing, referred to as the 'mother of all programming languages.' Originally intended for the development of microcomputer operating systems, it quickly became evident that C's utility extended far beyond its initial purpose [5]. 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 able 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 top choice for traditional systems programming to this day. C's sibling language, C++, was introduced in the 1980's as a powerful and versatile extension of C, designed to bridge the gap between low-level systems programming and high-level application development (the addition of Object-Oriented capabilities).

B. Common Memory-Related Vulnerabilities

Memory Safety is a critical concern in software development, especially systems programming, due to its direct implications with security and reliability. These vulnerabilities can show up even with the most experienced programmers, which makes for a topic of concern. A classic example is the buffer overflow - a program writes data beyond the boundaries of a buffer or intended container (see Figure 1b). This can overwrite critical memory areas causing unexpected behavior from the machine it is running on. The C family is infamous for these types of vulnerabilities, though it has been kept alive in systems programming due to its exceptional run-time performance as 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 types of functions to develop exploits and

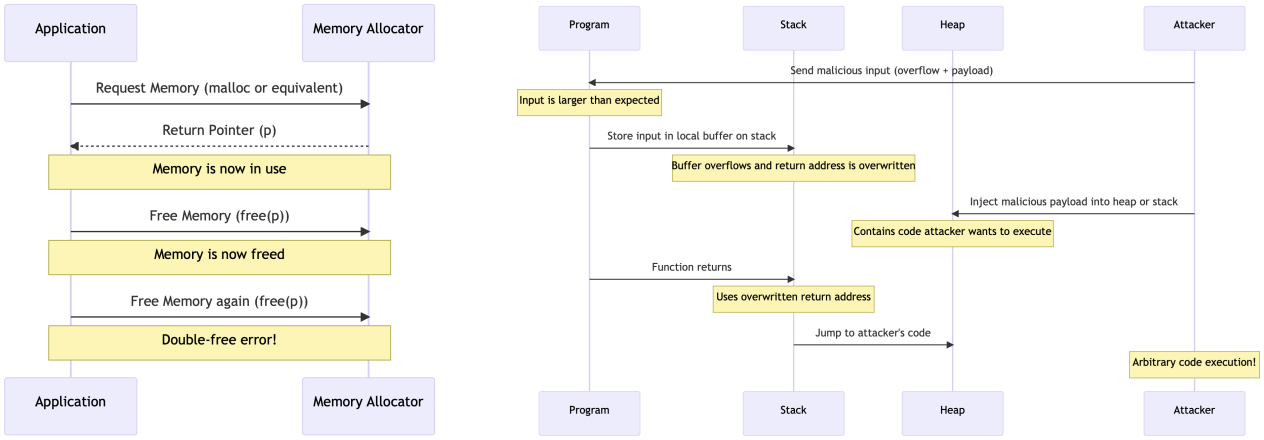
to attack machines. Functions used in buffer overflow attacks (as well as format string vulnerabilities) can be separated into standard input reading functions such as fscanf(), scanf(), gets(), and sscanf(), as well as string manipulation functions like strcpy(), strcmp(), strlen(), and strtok() [6]. Buffer overflow is not limited to system based applications, XSS or internet buffer overflow, is a version of web based buffer overflow that occur in web-based applications and can be triggered by malicious code or SQL attacks. Other memory-related vulnerabilities include: use-after-free - which occurs when a program attempts to access memory that has already been deallocated (see Figure 1a), Double-Free - attempting to free the same memory region more than once(see Figure 2a), and Heap-Stack Exploitation - the manipulation of input data to trigger vulnerabilities such as the previously discussed buffer overflows (see Figure 2b).



(a) A sequence diagram of a Use-After-Free error. An application requests memory and receives a pointer. Next, the application frees the memory, 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.

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



(a) (b)

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

III. Evolution of Memory-Safe Languages

Historically, many programming languages left the responsibility of memory management to developers, a choice that had both benefits and pitfalls. Over time, with the challenges and vulnerabilities this presented, 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. The trail of programmers implementing and considering memory management can, as one may assume, be traced back to the beginning of computing.

Though the official memory model for the Java programming language was introduced in 2005 [7], there are also older publications discussing execution order and its implications [8]. As technology progressed, the research discussions and publications begin to shift focus to efficiency and safety [9].

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 actually using 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 to that of C and C++ in terms of control flow, keywords, and expression blocks, makes it familiar to those versed in these 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 codebases while still 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 in the data structure it is traversing during 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 own 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 improvements 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. This makes 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].

IV. Characteristics of Memory-Safe Languages

While the concept of a memory-safe language might seem straightforward, the features that constitute such safety are diverse and multifaceted. Memory-safe languages come equipped with mechanisms like automatic memory management, type safety, bounds checking, and even safe concurrency mechanisms—all 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 memory-safe language is one that automates memory allocation and garbage collection, such as 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 mechanisms to enhance safety like bounds checking, 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]

V. 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, Go, and Swift. 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 and non-memory-safe language also exist, see this work by Kirth et al. [27]

The study 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, aiming to evaluate 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. [28]

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 tradeoff prevents impractical full heap scans for large heaps. C# uses compiler-assisted reference tracking to pinpoint modifications. To address this tradeoff, 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. [28]

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 codebase 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 [29]. To enhance memory safety in Rust while securing existing C code, solutions such as XRust can be employed [30]. This approach ensures a seamless transition and effectively mitigates most memory vulnerabilities (FC is also a valid tactic in this regard [31]). The combination of these strategies are 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 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, the 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.

VI. Comparison with Non-Memory-Safe Languages

Not all programming languages prioritize memory safety, often for valid reasons related to performance, flexibility, or historical context [32]. This section juxtaposes memory-safe languages against their non-memory-safe counterparts, like 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.

A memory-safe language is one where memory management is abstracted away 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, such as C and C++, 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 [33].

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 that C and C++, 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 [34].

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 [35].

C/C++11 Memory Model Rectification: In 2017, Lahav et al. introduced the RC11 (Repaired C11) model, which refined sequential consistency access semantics. Establishing 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 [36].

B. *MS-Wasm*

WebAssembly is a bytecode 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 [37]. The extension provides programmers with configurable security-perform trade-offs and empowers them to strengthen and future-proof their code.

VII. Real-world Implications of Memory Safety

Every codebase 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], [38]. Potentially, this incredibly high percentage of devastating vulnerabilities could be decreased by writing vital pieces of the communication trail in a memory-safe language. Emmerich et al. disccus this idea in their 2019 work [39]. 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 in which real systems interact with users, often transcending simple inputs and outputs. 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].

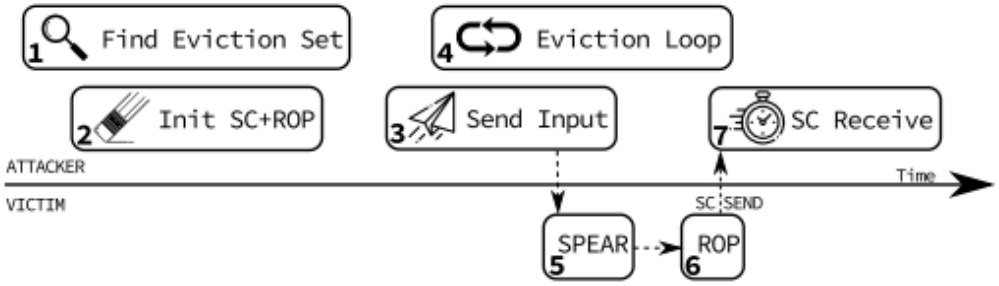


Fig. 3: Phases of a speculative execution attack

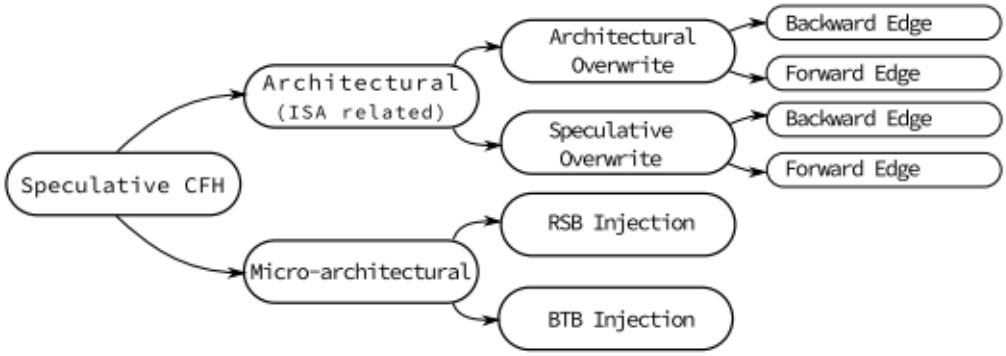


Fig. 4: Speculative Architectural Control-Flow Hijacks(SPEAR) actions

VIII. Attacks and Bugs

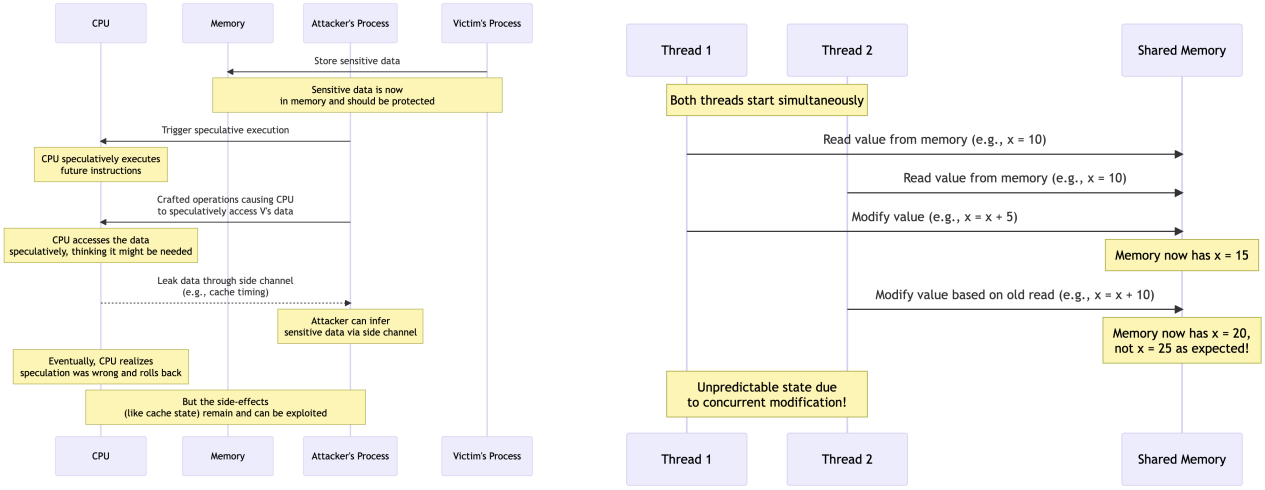
It is important to differentiate between the attacks that more easily exploit manually managed languages, and to briefly discuss current offensive methods that bypass both manually managed languages and memory-safe languages. The list below details some of the attacks and errors commonly encountered in languages that are not memory-safe [40]:

- 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 of a pointer twice” [40]. This error also allows user to access the freed memory similarly to Use-after-free errors.
- Buffer-Overflows occur when a system attempts to access memory beyond it’s allocated boundary.
- Uninitialized Memory Access occurs when an attempt to execute instructions in memory is done without initializing the memory.
- Concurrency bugs via memory corruption can occasional allow data racing conditions, and this can lead to any of the aforementioned errors.
- 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++. Papaevripides et al, discuss methods of exploiting mixed binary code compiled in combinations of either C++ and Rust or C++ and Go. [41]

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 deliberately trigger the memory-safe properties. [42] 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 eventual yields to the control-flow hijack. [42] From here, data is loaded from the side-channel to eventual gain more speculative access to the system.

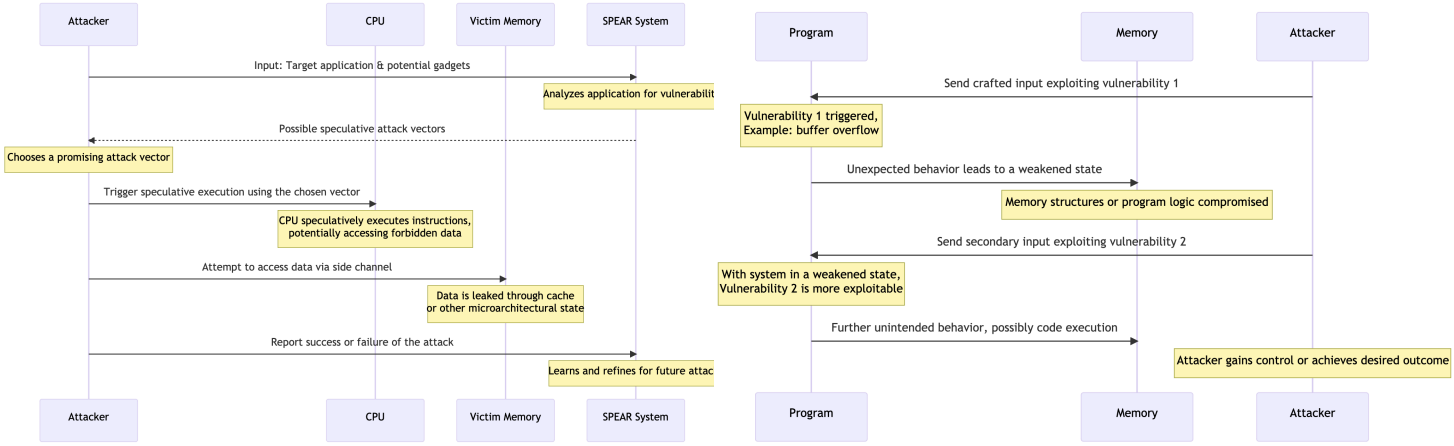
IX. Criticisms and Limitations

While memory-safe languages offer undeniable advantages in ensuring software security and reliability, they are not without their criticisms. Some developers argue that the overhead associated with memory safety features can impede performance, while others feel it may provide a false sense of security. This section delves into these concerns, highlighting potential pitfalls



(a) Speculative (b) concurrent

Fig. 5: Sequence diagrams illustrating two different types of memory errors.



(a) SPEAR (b) Mixed Binary

Fig. 6: Sequence diagrams illustrating two different types of memory errors.

and challenges while striving for a balanced perspective on the utility of memory-safe languages. As discussed by Nagarakatte et al., C and C++ are so widely proliferated within development stacks, they are unlikely to be phased out [43]. Even with a memory-safe language, to implement certain functionalities developers require some features that are unsafe; as such, the "*unsafe*" tag in Rust is still utilized, which could lead to a false sense of security for users. Futhermore, when using non-native Go libraries, memory vulnerabilities from the source languages (such as C/C++) may remain, as discussed by Sorniotti et al. [44].

A. Performance Overhead

Critics argue that the introduction of memory-safe can lead to an increase in performance overhead due to features like garbage collection and runtime checks (EXPAND - ADD COMPILER). Said critics contend that in low-level software in Operating Systems where microseconds matter, such as high-frequency trading or real-time systems, this overhead can be a limiting factor (to an extent).

B. Learning Curve

Transitioning from unsafe programming languages like C and C++ to memory-safe languages requires an investment of time and effort from individual programmers and companies as a whole. Although, the necessary training required for this transition can vary vastly depending on the individuals programming experience with said languages, companies still might not be willing to make this change due to the budget required to do so.

C. Limited Features and Tools

Addressing the limited landscape of memory-safe language libraries and tools is a crucial aspect in the adoption of these languages. While memory-safe languages offer much needed guarantees in the regards of memory and thread safety, the domain of these languages is still premature compared to that of other unsafe languages.

X. Future Directions

The field of software development is in perpetual motion, continuously evolving to meet the demands of a rapidly changing technological landscape. Memory safety, as a component of this evolution, is bound to see further innovations and shifts. This section forecasts the future of memory-safe languages, discussing emerging techniques, potential integrations 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 afford developers a degree of flexibility for specialized tasks. The use of this package can, however, introduce an array of potential vulnerabilities in its frequent real-world usage. Wickert et al. [45] present 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 judgements and more efficient refactoring.

Utilizing the Memory Tagging Extension (MTE) recently implemented by major chip manufacturer ARM, Seo et al. present a novel approach to deterministic spatial safety: ZOMETAG [46]. 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 tag 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 codebases effectively.

Even though there’s a growing push towards the adoption of memory-safe languages, it remains impractical to entirely 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 [47] and the application of Rust to Zero Trust Architecture Development [48].

XI. Conclusion

As the digital world becomes increasingly complex, the languages we use to communicate with machines play an essential role in ensuring safety, reliability, and efficiency. Through a comprehensive exploration of memory-safe languages, this review underscores their significance in today’s programming landscape. While challenges persist, the move towards memory safety is a testament to the software community’s dedication to creating a safer digital environment for all.

REFERENCES

[1] 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

[2] 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.

[3] 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

[4] 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.

[5] 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

[6] 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.

[7] J. Manson, W. Pugh, and S. V. Adve, "The java memory model," in *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 378–391. [Online]. Available: https://doi.org/10.1145/1040305.1040336

[8] Lamport, "How to make a multiprocessor computer that correctly executes multiprocess programs," *IEEE Transactions on Computers*, vol. C-28, no. 9, pp. 690–691, Sep. 1979.

[9] M. Wegiel and C. Krintz, "Cross-language, type-safe, and transparent object sharing for co-located managed runtimes," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 223–240. [Online]. Available: https://doi.org/10.1145/1869459.1869479

[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. HriŢcu, 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] 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

[29] 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

[30] 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

[31] H. M. J. Almohri 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

[32] 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

[33] 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

[34] 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.

[35] 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

[36] 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

[37] C. Disselkoen, 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

[38] J. Liu and C. Xu, "Pwning microsoft edge browser: From memory safety vulnerability to remote code execution," 2018.

[39] 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

[40] 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.

[41] 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

[42] 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.

[43] 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

[44] A. Sorniotti, M. Weissbacher, and A. Kurmus, "Go or no go: Differential fuzzing of native and c libraries," in *2023 IEEE Security and Privacy Workshops (SPW)*, 2023, pp. 349–363.

[45] 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.

[46] 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.

[47] 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

[48] 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.