Survey of Language Level Memory Safety

1st Xinming Guo
Department of Computer Science
Fudan University
Shanghai, China
25213050169@m.fudan.edu.cn

2nd Guangliang Yang
Department of Computer Science
Fudan University
Shanghai, China
yanggl@fudan.edu.cn

3rd Haoxiang Zhang
Department of Computer Science
Fudan University
Shanghai, China
25213050471@m.fudan.edu.cn

4th Can Li
Department of Computer Science
Fudan University
Shanghai, China
25213050216@m.fudan.edu.cn

5th Yi Wang
Department of Computer Science
Fudan University
Shanghai, China
25213050388@m.fudan.edu.cn

6th Min Yang
Department of Computer Science
Fudan University
Shanghai, China
m_yang@fudan.edu.cn

***Abstract*—Memory safety is essential for programming low-level system software. Over the past decades, numerous language-level efforts, such as C dialects and Rust, have aimed to provide strong safety guarantees for programmers. In this paper, we systematically study the landscape of language-level memory safety and address two key questions: (1) What are the core design principles underlying language-level memory safety? (2) What are the primary challenges in achieving it? We taxonomize related work along two dimensions: syntactic and semantic enforcement mechanisms. Our analysis reveals that existing solutions remain imperfect, often facing inherent trade-offs between safety and performance. Guided by these findings, we outline promising future directions to advance memory safety.**

I. Introduction

Memory safety is essential for the programming of low-level system software, such as operating system kernels and web browsers. Over the past decades, numerous serious memory safety vulnerabilities have affected system software, leading to significant security breaches and data loss. Industry reports indicate that approximately 70% of security vulnerabilities stem from memory safety issues [1], [2]. Consequently, several governments and organizations [3], [4], [5], have established roadmaps for safe programming and reached a consensus that memory safety is a mandatory requirement for future software programming.

To improve and ensure memory safety, substantial efforts have been made. These include StackGuard [6] for stack protection, Address Space Layout Randomization (ASLR) [7] and Data Execution Prevention (DEP) [8] for exploit mitigation, and Control Flow Integrity (CFI) [9] to prevent unauthorized code execution. These techniques have been widely deployed to mitigate memory safety vulnerability exploits. However, memory corruption attacks are becoming increasingly sophisticated and can bypass these defenses.

Another line of defense enhances security at the language level. First, several features have been introduced in the C and C++ standards. For instance, C99 [10] introduced the `restrict` keyword to indicate that a pointer is not aliased with any other pointer, which can help prevent buffer overflows. C++11 [11] added several safety-related features, such as `constexpr` for compile-time constant expressions and smart pointers (`unique_ptr`, `shared_ptr`, and `weak_ptr`) to manage pointer ownership and sharing for enhancing temporal memory safety. Nevertheless, developers must still exercise caution when writing code.

Second, various language dialects (with extensions) have been proposed to enhance the memory safety of C and C++ programs. For example, Cornell's Cyclone [12] enforces memory safety through region-based ownership and mandatory bounds checking. Berkeley's CCured [13] and Microsoft's Checked C [14] use pointer annotations to detect and prevent unsafe memory operations.

Third, several entirely new programming languages have been designed with memory safety as a core principle. Mozilla's Rust [15] restricts raw pointer usage and implements a robust ownership and lifetime management system. Google's Go [16] limits pointer arithmetic and relies on a garbage collector for automatic memory management.

These language-level solutions appear promising and have demonstrably reduced the risk of memory safety bugs. For example, the Android OS adopted Rust starting in Android 12 [17], and reported a dramatic decrease in memory safety vulnerabilities [18]. However, memory safety issues are still not entirely eliminated, even in Rust.

In this paper, we review language-level security techniques and focus on the key mechanisms that underpin them. We discuss the challenges in achieving memory safety guarantees at the language level, and classify existing approaches according to how they address spatial (e.g., out-of-bounds accesses) and temporal (e.g., use-after-free and double-free) memory safety. Specifically, we examine how language extensions enforce spatial and temporal memory safety through syntactic enforcement, compile- and run-time checks, ownership models, and type systems. We analyze the trade-offs these techniques make between safety, performance, expressiveness, and compatibility with legacy code.

The remainder of this paper is organized as follows. Sec-

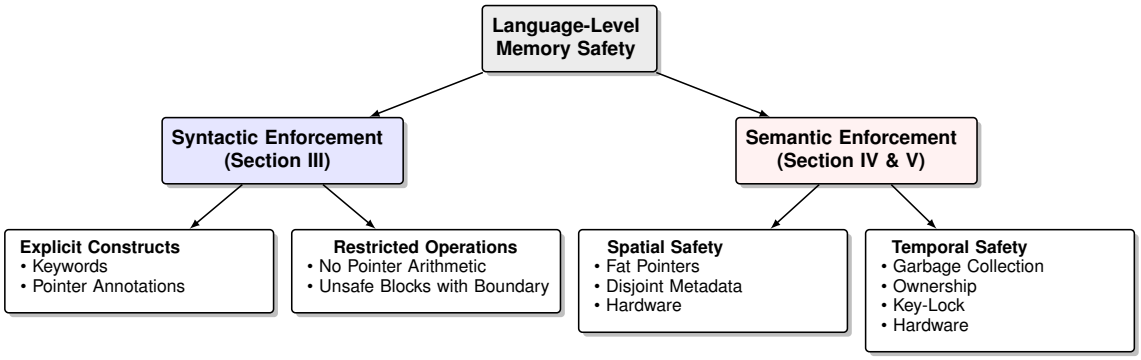

Fig. 1: Taxonomy of language-level memory safety mechanisms.

tion II provides a high-level overview and taxonomy of the underlying techniques. Section III discusses the syntactic design principles. Sections IV and V present detailed descriptions of techniques for addressing spatial and temporal memory safety. Section VI highlights the security enforcement in type systems and runtime. Section VII discusses the implications of these approaches and Section VIII depicts the future work directions in this area. Last, Section IX concludes the paper by summarizing key insights.

II. Technique Taxonomy and Overview

Memory safety is critical for system programming. It primarily encompasses two dimensions: spatial safety and temporal safety. Spatial safety refers to the prevention of out-of-bounds memory accesses. Temporal safety ensures that memory is not accessed after it has been deallocated (e.g., use-after-free or double-free). Ideally, language-level safety should provide safety guarantees at the language level. In such a system, the language itself is responsible for achieving spatial and temporal safety by design, thus guaranteeing these properties irrespective of the programmer's intent or mistakes.

Achieving memory safety remains a significant challenge for systems programming languages, including C/C++, C dialects (e.g., Cyclone, CCured, and Checked C), and Rust. C and C++ grant programmers direct control over memory management. This enables high performance, but places the full burden of memory safety on developers. Over time, various security-enhancing features have been introduced into C/C++ language standards (e.g., C++11) to mitigate these risks. In contrast, C dialects extend C with the goal of eliminating memory vulnerabilities by design. Rust, on the other hand, enforces strict memory safety guarantees through language-level mechanisms such as ownership and borrowing.

In this paper, we systematically study language-level memory safety and aim to understand how far existing language improvements are from providing comprehensive security guarantees. To this end, we focus on mapping the landscape of underlying security techniques employed by systems programming languages. We examine the challenges they face and discuss their strengths and limitations, highlighting the gap between different enforcement mechanisms and the classes of vulnerabilities they target. For this goal, we collected relevant literature from top-tier security and programming language conferences spanning from the 1990s to the present. We then applied a snowballing methodology, reviewing the references of key papers to identify additional related work. This iterative process culminated in nearly 50 papers that form the foundation of our systematization.

Through this systematization, we taxonomize language-level memory safety from two key perspectives: syntactic and semantic enforcement mechanisms, as illustrated in Figure 1. Syntactic enforcement refers to the use of explicit language constructs (such as type annotations) and the disabling of unsafe operations (e.g., unrestricted pointer arithmetic), which are designed to prevent memory errors at the source level. Semantic enforcement, by contrast, relies on deeper language semantics (such as compile-time type checking, ownership rules, or runtime memory access policies) enforced by the languages to guarantee spatial and temporal safety.

The remainder of this paper first presents syntactic enforcement mechanisms in systems languages. Then, we discuss semantic enforcement approaches.

III. Syntactic Enforcement

In this section, we explore the role of syntactic enforcement in systems programming languages. A programming language serves as the interface between programmers and the underlying system. Syntactic enforcement refers to the introduction of explicit security-related language constructs (such as type annotations) and the disabling of risky operations (e.g., unrestricted pointer arithmetic). These mechanisms regulate the channels through which programs interact with low-level system resources and thus constitute the first line of defense in the memory safety landscape.

Syntactic enforcement primarily centers on pointer types and management. A huge number of C/C++ projects have demonstrated that pointers are an effective and efficient means of manipulating low-level memory resources. However, pointer operations are also a common source of memory safety vulnerabilities. Specifically, spatial safety is often violated through pointer arithmetic and offset computations, while temporal safety is frequently compromised by the use of uninitialized or dangling pointers.

TABLE I: Comparison of Different Security Enforcement Approaches

Category	Language/Dialect	Syntactic Enforcement	Spatial Safety	Temporal Safety
New Security Enforcement	C99	restrict		
	C++11	constexpr		smart pointer
C Dialects	Cyclone	annotation	full pointer	regions
	CCured	annotation	full pointer	
	Checked-C	annotation	full pointer	key lock
	SoftBounds+CETS		disjoint metadata	key lock
New Language	Rust	disable pointer derefer arithmetic unsafe boundary	full pointer	ownership borrowing

C/C++ Security Standards. In C and C++, language syntax is defined and maintained by their respective standardization committees [19], [20]. New language standards are typically released every few years, guiding the evolution of the languages. A significant milestone was the introduction of smart pointers in C++11, namely `unique_ptr`, `shared_ptr`, and `weak_ptr`. Smart pointers aim to replace raw pointers and help developers write safer code. A `unique_ptr` represents exclusive ownership of a resource and automatically deallocates it when going out of scope. A `shared_ptr` enables shared ownership through reference counting, while a `weak_ptr` observes a resource without affecting its lifetime.

Progress in C Dialects. C dialects, such as Cyclone, CCured, and Checked C, are variants of C that introduce type annotations to support memory safety. These syntactic extensions enable compilers to infer and verify pointer types at compile time, facilitating static type checking and enforcement. As a result, pointers are classified into distinct categories (e.g., safe vs. unsafe, or region-annotated types). For instance, CCured categorizes pointers into three kinds: SAFE pointers, whose usage is statically verifiable and disallows arithmetic; SEQUENCE pointers, which permit pointer arithmetic for array access but require runtime bounds checks; and DYNAMIC pointers for all other cases, which are protected by extensive runtime checks. Conversions between different pointer types are strictly checked and enforced by the compiler. To reduce the manual burden of annotation, CCured employs static analysis to automate this process. Consequently, approximately 97% of pointers can be automatically classified. For the few remaining pointers whose safety cannot be statically verified, rigorous runtime checks are enforced.

Safety Guarantees and Side Effects. Rust is one of the few widely adopted systems programming languages that provides strong memory safety guarantees without relying on garbage collection. Its syntax system introduces a safety boundary through the `unsafe` keyword. Code within `safe` regions is guaranteed by the compiler to be free of memory safety violations. This is a foundational innovation in the Rust language design. Within safe Rust, raw pointers are fully restricted. They can only be created and used as references, and other operations (such as dereferencing, arithmetic, or comparison) are disallowed. This syntactic discipline effectively eliminates spatial violations and significantly reduces the risk of dangling pointers.

However, this strictness also introduces practical limitations. Safe Rust prohibits pointer-intensive idioms, such as pointer-based doubly linked lists or low-level memory traversal (e.g., for I/O streaming buffers). As a compromise, Rust allows `unsafe` blocks, where risky operations (including raw pointer manipulation, union accesses, and calls to other unsafe functions) are permitted. The `unsafe` mechanism thus represents a deliberate trade-off between safety and expressiveness.

Recent studies [21], [22] have illustrated that `unsafe` regions are the primary sources of memory safety vulnerabilities in Rust. Some highly critical vulnerabilities have even been discovered in the Rust standard library, for example, in string handling routines. These findings indicate that the `unsafe` keyword can introduce security holes, and developers must exercise extreme caution when writing code within `unsafe` blocks. Another security concern is the potential abuse of `unsafe`. Because direct pointer manipulation can yield performance benefits, many low-level systems projects opt to use `unsafe` extensively instead of adhering to safe Rust.

Other New Languages. Beyond Rust, several newer languages, such as Go, Zig [23], Odin [24], and V [25], have been designed with memory safety in mind. These languages introduce syntactic features that differ from Rust's approach. Most adopt garbage-collected memory models, which effectively address temporal safety but incur runtime overhead and reduced control. We discuss garbage collection and its implications in Section V.

IV. Spatial Safety

Spatial memory safety guarantees that all memory accesses are confined within the legitimate boundaries of the target object. Violations of this property are the root cause of canonical vulnerabilities such as buffer overflows, which have historically plagued software systems. The key challenge in achieving language-level spatial safety lies in accurately identifying the boundaries of memory objects. Based on where and how bounds information is stored and managed, existing approaches can be broadly categorized into three classes:

disjoint metadata, fat pointers, and hardware-assisted pointer extensions. We detail each below.

A. Disjoint Metadata

The disjoint metadata approach stores pointer bounds information in a separate memory region, commonly referred to as *shadow space*. This shadow space is logically decoupled from the program's main memory layout. In this model, a pointer p does not directly hold the target memory address. Instead, it serves as a handle to a corresponding *shadow pointer* p' in shadow space, which contains both the actual memory address and associated bounds metadata.

When performing pointer arithmetic (e.g., $p + \text{offset}$), the runtime system uses p' to retrieve the base address and bounds of the underlying object. It then checks whether the computed offset falls within the valid range. If the check passes, the real memory address is derived as $p'_{\text{addr}} + \text{offset}$; otherwise, the access is rejected as out-of-bounds.

Approaches such as SoftBound [26] employ disjoint metadata to detect out-of-bounds accesses. This shadow space based design offers significant advantages. It is agnostic to the underlying memory model and can be applied to languages like C/C++ without requiring source code modifications, thereby preserving strong compatibility.

Nevertheless, this design has notable limitations. First, given a pointer p, the system must locate its corresponding shadow entry p'. When the metadata table is small, the lookup overhead is low; Then as the pointer number increases, the overhead grows significantly. Although several works [26], [27] have proposed tree-based or hash-based indexing structures to improve performance, the overhead remains non-negligible. Second, the shadow metadata is globally shared. The concurrent accesses to the space require synchronization (e.g., via mutexes) to prevent data races. Thus, it can degrade performance in multi-threaded programs.

B. Fat Pointer

In contrast to the disjoint model, the fat pointer approach embeds bounds metadata directly within the pointer representation itself. This transforms a traditional single-word raw pointer into a multi-word structure, i.e., 'fat pointer'. To enforce spatial safety, a fat pointer typically comprises three fields: the `base address` of the memory object, its `bounds` (i.e., size), and an `offset` indicating the current position relative to the base. When a fat pointer $p = \langle \text{base}, \text{bounds}, \text{offset} \rangle$ undergoes arithmetic (e.g., $p + \Delta$), the system checks whether $\text{offset} + \Delta$ satisfies $0 \leq \text{offset} + \Delta <$ bounds. If not, the operation is flagged as out of bounds.

Most C dialects (such as Cyclone, CCured, and Checked C) rely on fat pointers to prevent spatial violations. However, their adoption typically requires extensive source code modifications to replace raw pointers with custom fat pointer types, limiting practical deployability. Additionally, modern languages also utilize fat pointers. in Rust, slices are represented as fat pointers, which inherently prevent out-of-bounds accesses.

Despite their safety benefits, fat pointers introduce performance overhead. Expanding a pointer from one word to two or three words increases memory footprint and affects calling conventions. In particular, function arguments that are fat pointers cannot be passed entirely in CPU registers for optimizations. This forces spilling to the stack and degrades performance. Recent works [28] have proposed 'thin' pointer optimizations to reduce fat pointer size. However, since essential metadata (base and bounds) must still be preserved, the scope for such optimizations is inherently limited.

On the concurrency scenario, fat pointers have an advantage over global shadow tables. Because each fat pointer carries its own metadata, accesses are inherently thread-local and do not require global synchronization, making them naturally thread-safe.

C. Hardware-Assisted Pointer Extension

CHERI [29], [30], [31] is a hardware-assisted capability based architecture available on several platforms, including RISC-V and ARM. CHERI extends traditional pointers, normally one machine word, into two-word *capabilities*. The first word encodes a capability structure containing the base address, length (bounds), permissions, and other metadata; the second word holds the current offset (i.e., the effective address). At runtime, CHERI works like fat pointers but with its own instructions for accessing its capabilities.

Due to hardware support, CHERI enforces spatial safety with significantly lower overhead than software-based fat pointers. However, it comes with several limitations. First, CHERI requires specialized hardware and is only available on select architectures. Second, it is not compatible with existing C/C++ binaries while providing protection. Third, CHERI currently focuses primarily on heap-allocated memory and does not completely support stack memory. This limitation stems from the high frequency and dynamic nature of stack allocations, which strain CHERI's metadata management resources.

V. Temporal Safety

Temporal memory safety ensures that memory objects are not accessed after they have been deallocated. Violations of this property commonly manifest as use-after-free and double-free bugs. The key challenge in enforcing temporal safety at the language level lies in tracking the allocation status of memory objects, i.e., determining whether a given object is currently allocated or has already been freed. This is particularly difficult because when an object's status changes (e.g., from allocated to freed), existing pointers pointing to it remain unaware of this change. At the language level, temporal safety is typically achieved by imposing strict controls over the entire lifecycle of memory objects. We systematize existing approaches into four dominant technical paradigms: garbage collection, ownership-based management, metadata-based management, and hardware-assisted validation.

A. Garbage Collection

Garbage Collection (GC) is a memory management technique that automatically reclaims memory no longer reachable from living program variables. It enforces temporal safety by delegating deallocation entirely to the runtime system, ensuring that memory is freed only when it is provably unreachable. Consequently, dangling pointer accesses (i.e., attempts to use memory after deallocation) are inherently prevented.

GC has been widely adopted in modern managed languages such as Go, owing to its simplicity and strong safety guarantees. However, it has notable limitations. First, GC is unsuitable for many low-level programming contexts, such as operating systems and language runtime implementations, which require precise, deterministic control over memory layout, and also their own memory allocation and deallocation management. Second, GC introduces runtime overhead and non-deterministic pause times during collection cycles, which can be problematic for performance-sensitive or real-time applications.

B. Ownership-Based Allocation Management

Several systems programming languages avoid garbage collection in favor of explicit and compile-time memory management. Notable examples include Cyclone and Rust. Cyclone, a safe dialect of C, employs a region-based memory model to enforce temporal safety. Memory is partitioned into regions, each with a designated owner responsible for allocating and deallocating all objects within it. When a region goes out of scope, its owner deallocates the entire region, ensuring no dangling references persist. While Cyclone pioneered the use of ownership for temporal safety, its coarse-grained region model limits expressiveness and practical applicability.

Rust adopts a more fine-grained ownership system based on compile-time lifetime tracking. Each memory object has a single owner, and ownership may be transferred (moved) between variables. When the owner goes out of scope, the value is automatically deallocated. To support shared or temporary access without transferring ownership, Rust enforces strict borrowing rules. At any time, there can be either one mutable reference or any number of immutable references, but not both.

C++11 also introduced ownership based pointer management, i.e., smart pointers. `unique_ptr` manages the lifetime of a memory object, ensuring that it is deallocated when the pointer goes out of scope. For `shared_ptr` and `weak_ptr`, C++ applies auto reference counting (garbage collection) to track the number of shared owners.

Different from C++ smart pointers, Rust's ownership model is more strict and does not rely on GC for memory management. The Rust ownership model is highly effective. It eliminates entire classes of temporal memory bugs without runtime overhead, as all checks are performed statically at compile time. Moreover, it enables fine-grained control over resource management, often improving performance compared to GC.

However, the uniqueness constraint of ownership complicates certain common programming patterns. For instance, pointer-intensive data structures like doubly linked lists inherently violate Rust's aliasing rules. To implement such structures, Rust programmers must resort to raw pointers within `unsafe` blocks. As discussed in Section IV, the use of `unsafe` bypasses the compiler's safety guarantees, potentially reintroducing temporal (and spatial) vulnerabilities and undermining overall system security.

C. Metadata-Based Allocation Management

Metadata can also be used to track the temporal status of memory objects at runtime. A common technique is the *key-lock* mechanism. When a memory object o is allocated, a unique `lock` (e.g., a random integer) is associated with it. Any pointer p that refers to o stores a corresponding `key` in its metadata, where key $=$ lock. Upon deallocation, the object's `lock` is invalidated (e.g., cleared). At dereference time, the runtime compares the pointer's `key` with the current `lock` of the target object. Access is permitted only if key $==$ lock; otherwise, it is rejected as a temporal violation.

The C dialect Checked C [32] adopts this key-lock mechanism to enforce temporal safety. Experimental results show that it can detect and report temporal errors at runtime with low overhead. Similarly, the disjoint-metadata system CETS [33] also implements key-lock validation. However, as Zhou et al. [32] demonstrated, CETS incurs significantly higher performance overhead compared to Checked C, primarily due to its global metadata table and synchronization requirements.

D. Hardware-Assisted Validation

Hardware extensions can also help enforce temporal safety. CHERI implements lazy address reuse. When memory is deallocated, its address is not immediately returned to the allocator but is instead marked as invalid in hardware. Subsequent accesses to such addresses are blocked by the capability system, preventing use-after-free exploits.

Another hardware-based approach is Pointer Authentication (PAC), used in ARM. PAC embeds a cryptographic signature (or 'tag') into the upper bits of a pointer. This tag is verified on every indirect jump or load/store operation. If the tag does not match the expected value (e.g., because the pointer points to freed memory), the access is aborted. While originally designed for control-flow integrity, PAC can be adapted to detect certain temporal violations by binding pointer validity to object lifetime.

Both CHERI and PAC shift part of the temporal safety burden to hardware, reducing software overhead. However, they require specialized architectures and offer only partial coverage (e.g., CHERI currently lacks full stack support), limiting their applicability in heterogeneous or legacy environments.

VI. Type and Runtime Enforcement

The type system is a cornerstone of programming languages, enabling correctness checks at compile time. Although C and

C++ are statically typed, they are not type-safe. First, they introduce `void*` to facilitate programming. However, this convenience comes at the cost of erasing the type information of the pointed object. Second, C/C++ permit low-level pointer-based casting. For example, in the code snippet '`char* p1; ... int* p2 = (int*)p1;`', the same memory region is interpreted as `char` typed when accessed via p_1, and as `int` typed via p_2. Third, pointers are treated as raw addresses i.e., unsigned integers. C allows conversion between pointers and integers, which also erases type information. Such type-punning through pointer casts may lead to memory accesses with mismatched type sizes, often resulting in spatial memory safety violations. Similarly, unions in C/C++ allow multiple types to alias the same memory location, enabling type confusion and undefined behavior. Consequently, the C/C++ type system is fairly fragile, making it difficult to perform rigorous type inference and enforcement. As a result, many memory access errors cannot be reliably detected and prevented at compile time.

C dialects, which extend C with additional type qualifiers and restricted pointers, build their safety mechanisms atop this inherently weak and permissive type system. This fragile foundation poses significant challenges for constructing robust and strict type enforcement. Consequently, such dialects turn to runtime checks to guarantee memory safety.

In contrast, the modern systems language, Rust, adopts a strict type system that severely restricts implicit or unsafe type conversions, permitting only explicit ones. Built upon this type discipline, Rust further performs compile-time ownership and lifetime analysis, enabling developers to detect and eliminate most memory safety bugs before execution.

Furthermore, Rust employs runtime enforcement where necessary. For instance, it performs bounds checking on the accesses of slices, which are fat pointers. Rust supports dynamic dispatch through trait objects, which are implemented using virtual function tables (vtables). Crucially, Rust enforces correct trait implementations at compile time, preventing runtime errors caused by missing or mismatched method definitions.

VII. Discussion

In this paper, we systematically summarize key memory safety paradigms along two primary dimensions: syntactic and semantic enforcement. Syntactic enforcement concerns language-level constructs that directly restrict unsafe operations, e.g., disabling pointer arithmetic or requiring explicit type annotations in safe code. Semantic enforcement, on the other hand, relies on compile- and runtime analyses grounded in program semantics, such as metadata propagation, ownership and lifetime tracking, and type inference and validation.

Over the past decades, significant efforts have been made to address memory safety in C/C++ standards, C dialects, and new languages like Rust. Guided by this landscape, we answer two critical research questions:

(1) *What are the key technical paradigms and inherent trade-offs in achieving memory safety?* We break down the underlying techniques and explore their pros and cons. Table II presents a comprehensive comparison of these mechanisms regarding their limitations. In spatial safety, disjoint metadata based approaches have strong compatibility with existing C/C++ code, but they are not very efficient. Fat pointers are more efficient, but existing fat pointer based solutions require the modification of the existing code. In temporal safety, GC is automatic, but it is not suitable for all applications and has performance concerns. Ownership based approaches are fast and deterministic at compile time, but they cannot handle pointer-based data structures, posing side efforts (e.g., the introduction of 'unsafe').

(2) *How far have we come in providing language-level safety guarantees?* We define such a guarantee as follows: regardless of how a programmer writes code, the underlying language automatically prevents all memory corruption vulnerabilities. Our analysis reveals that no existing language is perfect. First, C++ introduced smart pointers to mitigate memory safety issues, but their opt-in nature and coexistence with raw pointers limit their effectiveness. Second, C dialects attempt to enforce safety using fat pointers, which require extensive source-code annotations and manual refactoring. Moreover, fat pointers incur runtime overhead due to the extra memory needed for metadata storage. These approaches essentially try to construct a robust type system on top of C's inherently fragile foundation, a fundamentally challenging endeavor.

Finally, Rust stands out as the only widely adopted systems language that provides a solid and strictly enforced type system without GC. Through compile-time ownership and borrowing rules, Rust eliminates most memory safety bugs before execution. However, to enable and accommodate low-level operations that cannot be statically verified, Rust introduces the `unsafe` keyword. Code within `unsafe` blocks bypasses the compiler's safety checks and memory safety guarantees. Empirical studies [22], [34] have shown that `unsafe` code is a primary source of memory vulnerabilities in Rust programs, a direct consequence of the trade-offs necessitated by its otherwise strict enforcement model.

VIII. Future Work

Guided by the above discussion, several promising directions for future work emerge.

Enhancing Legacy C/C++ Programs. Despite the growing adoption of Rust for systems programming, a vast amount of legacy C/C++ code remains in use. One approach is to automatically translate C/C++ code into safe Rust. The C2Rust project [35], [36], [37], [38] represents a notable step in this direction. However, much of the translated code ends up in `unsafe` blocks due to the expressiveness gap between C and Rust's safety model. Consequently, developing translation techniques that minimize or eliminate `unsafe` usage is a compelling research goal.

An alternative is to define a safe subset of C/C++ [39], [40] that retains most of the original syntax while integrating Rust-like safety mechanisms, such as ownership, borrowing,

TABLE II: Comparison of Memory Safety Mechanisms

Mechanism Category	Technique	Runtime Cost	Pros	Cons
Spatial Safety	**Fat Pointers**	Medium	• Inherently thread-safe • Efficient for local access	• Breaks binary ABI • High memory footprint • Degrades cache locality
	Disjoint Metadata	High	• Preserves binary compatibility • Agnostic to memory model	• High lookup latency • Requires synchronization for multi-threading
	Hardware	Low	• High efficiency via hardware acceleration	• Requires specialized hardware • Incompatible with existing binaries
Temporal Safety	**Garbage Collection**	Variable*	• Simple for developers • Strong safety guarantees	• Non-deterministic pauses • Introduces runtime overhead
	Ownership	None	• Zero-cost abstraction • Deterministic resource management	• Steep learning curve • Restricts data structures • May force 'unsafe' usage
	Key-Lock	Medium	• Deterministic detection • No GC pauses	• Memory fragmentation • Runtime check overhead on every access
	Hardware	Low	• Minimal software burden • Fast validation	• Hardware dependency • Partial coverage (e.g., stack issues)

* Variable indicates that runtime overhead fluctuates between negligible cost during allocation and significant latency spikes during non-deterministic collection cycles

and restricted pointer arithmetic. This idea is already under active discussion within the C++ standards committee [41].

Eliminating or Mitigating Rust `unsafe` Blocks. Since `unsafe` code is a known root cause of memory safety vulnerabilities in Rust, reducing its footprint is crucial. One direction is to design new languages that provide strong safety guarantees without any `unsafe` escape hatches. Another is to sandbox `unsafe` code: even if unsafe operations are permitted, their effects are confined within a secure boundary, ensuring that memory safety is preserved at the system level.

Unsafe Code Detection and Repair. Automated detection and repair of unsafe code usage is another important avenue. Unsafe code is often buried deep within libraries, and its misuse may only manifest in client programs. Static analysis alone may capture problematic usages. Therefore, developing fine-grained, context-aware tools for detecting unsafe code and suggesting repairs is essential [37], [42].

AI-Assisted Code Security Analysis and Enhancement. Artificial intelligence (AI) shows great promise in understanding code semantics and assisting developers. AI models can help identify unsafe code patterns, suggest safe alternatives, and even automate refactoring to eliminate `unsafe` blocks. AI-assisted code analysis may significantly improve the efficiency and scalability of memory safety assurance, making it a highly promising direction for future research.

IX. Conclusion

C and C++ are widely used low-level programming languages. They provide direct access to memory and enable programmers to write high-performance code. However, they introduce significant spatial and temporal memory safety challenges. Over the past decades, various language-level memory safety mechanisms have been proposed and deployed to address these issues, aiming to provide strong safety guarantees for programmers. These mechanisms include syntactic enforcement, strict type systems, pointer annotations and restrictions, and runtime checks.

Through a systematic study of existing work, we find that current solutions remain imperfect, often facing inherent trade-offs between safety and performance. We examine the underlying techniques of these approaches and highlight the key challenges they encounter. By analyzing the limitations of existing efforts and outlining potential future directions, we hope our survey and analysis can facilitate further research and inspire the design of effective and efficient memory-safe system languages.

Acknowledgement

We would like to thank the anonymous reviewers for their insightful comments that helped improve the quality of the paper. Guangliang Yang is the corresponding author. This work was supported in part by the National Natural Science Foundation of China (62202106, 62172104, 62172105, 62472096, 62302101, 62402114, 62402116).

References

[1] "Secure by Design: Google's Perspective on Memory Safety," https://research.google/pubs/secure-by-design-googles-perspective-on-memory-safety/.

[2] "We need a safer systems programming language," https://www.microsoft.com/en-us/msrc/blog/2019/07/we-need-a-safer-systems-programming-language.

[3] "NSA Releases Guidance on How to Protect Against Software Memory Safety Issues." https://www.nsa.gov/Press-Room/News-Highlights/Article/Article/3215760/nsa-releases-guidance-on-how-to-protect-against-software-memory-safety-issues/.

[4] "CISA Releases Joint Guide for Software Manufacturers: The Case for Memory Safe Roadmaps," https://www.cisa.gov/news-events/alerts/2023/12/06/cisa-releases-joint-guide-software-manufacturers-case-memory-safe-roadmaps.

[5] "The Urgent Need for Memory Safety in Software Products," https://www.cisa.gov/news-events/news/urgent-need-memory-safety-software-products.

[6] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "StackGuard: Automatic adaptive detection and prevention of Buffer-Overflow attacks," in *Proceedings of the 7th Conference on USENIX Security Symposium*, 1998.

[7] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM Conference on Computer and Communications Security*, 2004.

[8] "Data Execution Prevention," https://learn.microsoft.com/en-us/windows/win32/memory/data-execution-prevention.

[9] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, 2005.

[10] "C99." https://en.cppreference.com/w/c/99.html.

[11] "C++11." https://en.cppreference.com/w/cpp/11.html.

[12] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of c," in *USENIX Annual Technical Conference*, 2002.

[13] G. C. Necula, S. McPeak, and W. Weimer, "Ccured: type-safe retrofitting of legacy code," in *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002.

[14] A. S. Elliott, A. Ruef, M. Hicks, and D. Tarditi, "Checked c: Making c safe by extension," in *IEEE Cybersecurity Development*, 2018.

[15] "Rust," https://github.com/rust-lang/rust, 2006.

[16] "Go," https://github.com/golang/go.

[17] "Rust in the Android platform," https://security.googleblog.com/2021/04/rust-in-android-platform.html.

[18] "Memory Safe Languages in Android 13," https://security.googleblog.com/2022/12/memory-safe-languages-in-android-13.html.

[19] " ISO/IEC JTC1/SC22/WG14 - C," https://www.open-std.org/jtc1/sc22/wg14/.

[20] "The C++ Standards Committee," https://isocpp.org/std/the-committee.

[21] H. Xu, Z. Chen, M. Sun, Y. Zhou, and M. R. Lyu, "Memory-safety challenge considered solved? an in-depth study with all rust cves," *TOSEM*, 2021.

[22] S. Höltervennhoff, P. Klostermeyer, N. Wöhler, Y. Acar, and S. Fahl, "unsafe Rust–conscious choice or spiky shortcut," in *43rd IEEE Symposium on Security and Privacy.*, 2022.

[23] "Zig," https://github.com/ziglang/zig.

[24] "Odin," https://github.com/odin-lang/Odin.

[25] "VLang," https://github.com/vlang/v.

[26] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "Softbound: highly compatible and complete spatial memory safety for c," *SIGPLAN Not.*, 2009.

[27] P. Akritidis, M. Costa, M. Castro, and S. Hand, "Baggy bounds checking: an efficient and backwards-compatible defense against out-of-bounds errors," in *Proceedings of the 18th Conference on USENIX Security Symposium*, 2009.

[28] A. Kwon, U. Dhawan, J. M. Smith, T. F. Knight, and A. DeHon, "Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, 2013.

[29] R. N. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera, "Cheri: A hybrid capability-system architecture for scalable software compartmentalization," in *IEEE Symposium on Security and Privacy*, 2015.

[30] R. N. Watson, R. M. Norton, J. Woodruff, S. W. Moore, P. G. Neumann, J. Anderson, D. Chisnall, B. Davis, B. Laurie, M. Roe, N. H. Dave, K. Gudka, A. Joannou, A. T. Markettos, E. Maste, S. J. Murdoch, C. Rothwell, S. D. Son, and M. Vadera, "Fast Protection-Domain Crossing in the CHERI Capability-System Architecture," *IEEE Micro*, 2016.

[31] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The CHERI capability model: revisiting RISC in an age of risk," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, 2014.

[32] J. Zhou, J. Criswell, and M. Hicks, "Fat Pointers for Temporal Memory Safety of C," *Proceedings of the ACM on Programming Languages*, 2023.

[33] S. Nagarakatte, J. Zhao, M. M. Martin, and S. Zdancewic, "CETS: compiler enforced temporal safety for C," in *Proceedings of the 2010 International Symposium on Memory Management*, 2010.

[34] V. Astrauskas, C. Matheja, F. Poli, P. Müller, and A. J. Summers, "How do programmers use unsafe rust?" *Proceedings of the ACM on Programming Languages*, 2020.

[35] "C99-compliant code to Rust," https://github.com/immunant/c2rust.

[36] J. Hong and S. Ryu, "Don't write, but return: Replacing output parameters with algebraic data types in c-to-rust translation," *Proceedings of the ACM on Programming Languages*, 2024.

[37] Y. Bae, Y. Kim, A. Askar, J. Lim, and T. Kim, "Rudra: Finding memory safety bugs in rust at the ecosystem scale," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021.

[38] R. Li, B. Wang, T. Li, P. Saxena, and A. Kundu, "Translating c to rust: Lessons from a user study," *ArXiv*, 2024.

[39] B. Stroustrup, "A rationale for semantically enhanced library languages," 2005. [Online]. Available: https://api.semanticscholar.org/CorpusID:10562147

[40] "C++ Core Guidelines," https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines.

[41] H. Sutter, "Core safety Profiles: Specification, adoptability, and impact," *ISO/IEC JTC1/SC22/WG21 - The C++ Standards Committee*, 2024.

[42] Y. Takashima, R. Martins, L. Jia, and C. S. Păsăreanu, "Syrust: automatic testing of rust libraries with semantic-aware program synthesis," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2021.