Memory Safety in C

Aman Hamidsha

1 Abstract

This essay examines the concept of memory safety through the C programming language, addressing both its theoretical foundations and practical implications. It begins with a general introduction to C, outlining its history, common uses, and its legacy in systems programming. The discussion then moves to a theoretical outline on memory safety which involves comparing C to an idealized memory safe language developed by Amorim et. al[1]. This sets the precedent for the following section which covers the two main types of memory safety: spatial and temporal memory safety. There will be a discussion on ways to mitigate said memory safety issues with focus being placed on Nagarakatte et al.'s[2] attempt to resolve issues related to spatial memory safety, and Farkhani et al.'s[3] attempt to do the same for temporal memory safety. Next, the essay compares C to other languages such as C++, Rust, and Zig and the ways in which all these languages address problems related to memory safety, focusing on the trade-offs between security, performance and other factors. We conclude with a reflection on our personal experience with C and explain how this research has deepened our understanding of programming language design.

2 An Introduction to C

The C programming language, created by Dennis Ritchie in the early 1970s, remains one of the most influential languages in computing history. It was designed to provide low-level access to memory and system resources while maintaining the structure and readability of a high-level language. C became the foundation of modern operating systems, including UNIX and later Linux, and heavily influenced subsequent languages such as C++ and Java.

While C's influence on systems programming is undeniable, its design philosophy of prioritizing control over safety has also earned it a reputation for insecurity. Many vulnerabilities in modern software, especially in operating systems and networking stacks, can be traced back to unsafe operations in C code. Understanding how C's memory model works and how it can fail provides valuable insight into the broader study of programming language design and safety.

3 Formal Foundations of Memory Safety

Amorim et al., through their paper 'The Meaning of Memory Safety' aim to define what memory safety actually means, not by listing forbidden bugs, but by formalizing the reasoning principles that memory safety enables. Instead of asking whether a language prevents buffer overflows, the authors ask what logical guarantees a programmer gains when memory safety is enforced automatically. Their central claim is that memory safety fundamentally supports local reasoning: code can neither affect nor be affected by parts of memory it cannot reach. This idea is formalized using a small imperative language with explicit allocation and deallocation, where invalid memory accesses result in well-defined runtime errors rather than undefined behavior.

At the mathematical level, memory is modeled as a finite map from block identifiers to arrays, and pointers are represented as pairs (i, n), where i is a block identifier and n an offset. Block identifiers behave like unforgeable capabilities: possession of an identifier is the only way to access the corresponding memory region. The core results of the paper are a series of frame theorems and a noninterference theorem. Informally, if a program initially has access to a set of block identifiers I, then extending the heap with a disjoint set of blocks J, where $I \cap J = \emptyset$, does not change the observable behavior of the program, up to renaming of freshly allocated identifiers. This yields a strong isolation guarantee that can be cleanly expressed as reachability-based noninterference.

The authors then relate these results to separation logic. Traditional separation logic relies on the frame rule, which assumes that programs only access the heap regions described in their specifications. In unsafe languages such as C, this locality must be established manually by proving that every memory access is valid. In the memory-safe language studied in the paper, locality follows automatically from pointer reachability, allowing a weaker but robust variant of the frame rule that remains sound even in the presence of buggy or malicious code. Reasoning about a command c depends only on

the sub-heap reachable from the pointers in its state, while unreachable regions are provably invariant.

When compared to C, the contrast is sharp. In C, pointers are integers, block identity is implicit and observable, and undefined behavior means that program semantics are not even total functions of the form $S \to S \cup \{\text{error}, \bot\}$. Consequently, none of the paper's theorems hold in general: extending memory can change outcomes, dangling pointers can regain access to newly allocated objects, and pointer-to-integer casts collapse reachability into global observability. From a mathematical perspective, C violates the invariants that block identifiers are fresh and unforgeable, breaking the disjointness condition $I \cap J = \emptyset$ on which the frame theorems depend.

A limitation of the paper is that it is descriptive rather than prescriptive. It does not introduce a new enforcement mechanism, but instead provides a formal yardstick for evaluating existing ones. This is also its main strength: it explains why C is fundamentally difficult to reason about and why mechanisms to mitigate memory safety issues (to be covered later) only partially recover the reasoning principles associated with memory safety. Overall, Amorim et al. present a mathematically clean account of memory safety as reachability-based noninterference, and we can use it to show that C fails to satisfy this account in almost every essential respect.

4 Spatial and Temporal Memory Safety

We now analyse memory safety in C by distinguishing between spatial and temporal errors. Spatial memory safety concerns whether a program accesses the correct memory location, while temporal memory safety concerns whether an access occurs at a valid point in an object's lifetime. Finally, we introduce key mitigation efforts, focusing on Nagarakatte's work on enforcing spatial safety and Farkhani's approach to addressing temporal safety, which aim to restore guarantees that are absent in C's memory model.

4.1 Spatial

4.1.1 Buffer Overflow[4]

Often cited as the most well known vulnerability in C, buffer overflows arise when a program writes beyond the bounds of an allocated memory region. Because C performs no automatic bounds checking, such out of bounds writes can overwrite adjacent data in memory, including other objects, heap management metadata, or even a function's return address on the stack. This can lead to crashes, corrupted program state, or, in severe cases, arbitrary code execution.

Arbitrary code execution occurs when an attacker is able to redirect program control flow to execute instructions of their choosing, rather than the code intended by the developer[]. In practice, this can allow an attacker to read or modify sensitive data, bypass security checks, escalate privileges, or take full control of the system on which the vulnerable program is running.

Let's illustrate this principle with an example:

```c
#include <stdio.h>
#include <string.h>

void handle_request(void) {
    char cmd[32];
    printf("Enter command: ");
    gets(cmd);                  /* unsafe: no length limit */
    printf("Running: %s\n", cmd);
}

int main(void) {
    handle_request();
    return 0;
}
```

The code shown here is insecure because `gets(cmd)` reads an unbounded amount of input into a fixed-size 32-byte stack buffer, so if a user enters more than 31 characters plus the null terminator, the extra bytes will overflow `cmd` and overwrite adjacent stack memory. On typical systems this memory includes saved frame pointers (registers that point to the start of a function's stack frame so the program can reliably access local variables and know where to return after the function finishes) and the function's return address. An attacker can exploit this by carefully crafting an input that overwrites the return address with a value of their choosing, causing the program to jump to attacker-controlled code or

to existing code sequences when `handle_request` returns. This enables control-flow hijacking, which can lead to arbitrary code execution, privilege escalation, or complete takeover of the process.

4.1.2 Buffer Over-Read[5]

This less famous sibling of the buffer overflow, differing by only a single word, buffer over-**reads** occur when a program reads past the bounds of an allocated memory region. While they do not overwrite memory, they can still be highly damaging by revealing data the program was never meant to expose.

In practice, buffer over-reads can leak sensitive information such as passwords, cryptographic keys, or internal program state. They may also disclose memory addresses, which is particularly dangerous in real world attacks. Such leaks can defeat address space layout randomization, a defense mechanism that relies on keeping memory locations unpredictable by randomizing where key memory regions are placed at runtime. Once an attacker learns these addresses, previously difficult control flow attacks become significantly easier to carry out.

An example:

```c
#include <stdio.h>
#include <string.h>

void log_user_input(const char *input) {
    char buf[16];

    /* BUG: copies up to 16 bytes but might not write a '\0' terminator */
    strncpy(buf, input, sizeof(buf));

    /* Buffer over-read: printf("%s") expects a null-terminated string,
       so it keeps reading past buf until it accidentally finds '\0'. */
    printf("User said: %s\n", buf);
}

int main(int argc, char **argv) {
    if (argc != 2) {
        puts("usage: ./a.out <text>");
        return 1;
    }
    log_user_input(argv[1]);
    return 0;
}
```

This code is insecure because `strncpy(buf, input, sizeof(buf))` copies raw bytes into a fixed-size stack buffer without guaranteeing that the copied data is properly terminated. In C, strings are expected to end with a special null byte (`'\0'`) that tells the program where the string stops. If the input is 16 bytes or longer, `strncpy` fills the entire buffer and does not append this terminator. When `printf` later prints `buf` using `%s`, it assumes the buffer contains a valid C string and continues reading memory past the end of `buf` until it eventually encounters a null byte elsewhere in memory. This results in a buffer over-read, which can leak adjacent stack data such as other local variables or pointers, potentially revealing sensitive information and enabling further exploitation.

4.2 Temporal

4.2.1 Use After Free[6]

This is when an uninitialized variable is used before being assigned a value, meaning it may contain residual data from memory or bit patterns that are invalid for its declared type.

Example:

```c
#include <stdio.h>
#include <stdlib.h>

void logError(const char *msg, const char *detail) {
    printf("ERROR: %s | detail: %s\n", msg, detail);
}

int main(void) {
    int abrt = 0;

```

```
    char *ptr = (char *)malloc(32);
    if (!ptr) return 1;

    /* imagine ptr is filled with some useful text */
    snprintf(ptr, 32, "txn_id=4815");

    /* something goes wrong */
    abrt = 1;
    free(ptr);                  /* ptr is now dangling: memory is released */

    /* BUG: use-after-free */
    if (abrt) {
        logError("operation aborted before commit", ptr);
        /* ptr is used after it was freed */
    }

    return 0;
}
```

This code has a use-after-free because `ptr` is freed but later reused as if it still points to valid memory. After `free(ptr)`, that region is no longer owned by the program, so the allocator is allowed to reuse it for future allocations at any time; `ptr` becomes a dangling pointer that still holds an address, but the data at that address is now invalid and may change unpredictably. When `logError(..., ptr)` treats `ptr` as a string, it may read garbage, crash, or leak unrelated data, and in more serious cases an attacker can influence what gets placed into that freed region (for example by causing another allocation to reuse it) so that later reads or writes through `ptr` operate on attacker-controlled bytes. That can turn an "accidental bug" into memory corruption, information disclosure, or even control-flow exploitation depending on what the program does with the dangling pointer.

4.2.2 Double Free[7]

Calling `free()` twice on the same pointer can corrupt the heap's internal metadata. This may crash the program or cause later `malloc()` calls to return the same address, enabling overlapping allocations. If an attacker controls writes to this memory, the resulting write-what-where condition can be exploited for buffer overflows and arbitrary code execution. Example:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define BUFSIZE1 512
#define BUFSIZE2 ((BUFSIZE1/2) - 8)

int main(int argc, char **argv) {
    char *buf1R1;
    char *buf2R1;
    char *buf1R2;

    buf1R1 = (char *)malloc(BUFSIZE2);
    buf2R1 = (char *)malloc(BUFSIZE2);

    free(buf1R1);
    free(buf2R1);                   /* buf2R1 is freed here */

    buf1R2 = (char *)malloc(BUFSIZE1);
    strncpy(buf1R2, argv[1], BUFSIZE1 - 1);
    buf1R2[BUFSIZE1 - 1] = '\0';

    free(buf2R1);                   /* BUG: double free of the same pointer */
    free(buf1R2);

    return 0;
}
```

This code has a double-free because `buf2R1` is released with `free(buf2R1)` and then later released again with another `free(buf2R1)` even though it was never reassigned to a new allocation. After the first `free`, `buf2R1` becomes an invalid (dangling) pointer: it still contains an address, but the allocator now considers that chunk available for reuse. Calling

`free` a second time can corrupt the allocator's internal bookkeeping (it may try to "put back" a chunk that is already in its free lists), which can cause crashes or, in worse cases, be exploited by an attacker to manipulate heap metadata and eventually gain control over what memory gets returned by future `malloc` calls, leading to memory corruption and potentially arbitrary code execution.

4.3 Mitigation Attempts

4.3.1 SoftBound

Nagarakatte et al. aim to solve spatial memory safety violations in C through their proposal called SoftBound. The authors argue that prior approaches either provide incomplete coverage, break compatibility with existing C code, or impose prohibitive performance costs. SoftBound's goal is to enforce complete spatial memory safety while preserving C's memory layout, pointer representation, and programming idioms.

SoftBound achieves this by associating every pointer with explicit base and bound metadata, which describe the valid memory region the pointer may access. Instead of embedding this metadata directly into pointers, SoftBound stores it in a disjoint metadata structure indexed by the pointer's storage location. On every pointer dereference, SoftBound checks that the access lies within the stored bounds, preventing spatial violations at runtime.

A crucial strength of SoftBound is that it handles arbitrary pointer arithmetic and casts, including pointers to sub-objects, without sacrificing completeness. Pointer arithmetic is allowed freely, but safety is enforced at dereference time, aligning closely with C's semantics. The system also supports separate compilation and legacy code, making it far more practical than earlier safe-C approaches. The authors further provide a formal semantics and proof that SoftBound-enforced programs cannot perform illegal spatial memory accesses.

Despite its rigor, SoftBound has clear limitations. It enforces only spatial safety and does nothing to prevent temporal errors. The runtime overhead, while reasonable, is still substantial for pointer-heavy code, and the additional metadata management increases complexity.

4.3.2 PTAuth

This paper by Farkhani et al. aims to solve temporal memory safety violations in C programs. The authors observe that many existing temporal safety mechanisms are either incomplete, incur high overhead, or critically depend on spatial memory safety to protect their metadata, making them vulnerable if spatial bugs are present. PTAuth's goal is to detect all heap-based temporal violations without relying on spatial memory safety at all, addressing a major weakness in prior work.

The core idea is points-to authentication: each heap object is assigned a fresh random identifier at allocation time, and each pointer to that object carries a cryptographic authentication code that binds the pointer to that specific object instance. On every pointer dereference or deallocation, PTAuth verifies that the pointer still refers to a live object with the expected identity. If the object has been freed, reallocated, or never validly owned by the pointer, the authentication fails and the program aborts.

A key design choice is storing pointer metadata inside the pointer itself, using unused bits, and storing object metadata inline with the object. This eliminates shadow memory and avoids the need to separately protect metadata.

The paper's main strengths are its clear separation of temporal safety from spatial safety, its minimal metadata overhead, and its robustness against metadata tampering. However, its limitations are significant: it is restricted to heap objects, does not support multi-threaded programs in its current form, and relies on ARM-specific hardware features, limiting portability. More broadly, PTAuth highlights how difficult it is to retrofit temporal safety into C without strong assumptions, reinforcing the argument that lifetime safety is fundamentally at odds with C's low-level semantics.

5 Comparison to Other Programming Languages

This section compares C with C++, Rust, and Zig across several dimensions that are directly relevant to systems programming. The comparison focuses on memory safety, performance, syntax and maintainability, tooling maturity, and real world adoption. These criteria reflect both language design goals and the practical constraints encountered when writing low level software.

The languages chosen for comparison all target similar problem domains to C. C++ extends C with higher level abstractions while preserving low level control. Rust aims to provide memory safety without a garbage collector through static ownership and borrowing rules. Zig emphasizes explicitness and simplicity while retaining manual memory management. Comparing C against these languages therefore allows a focused discussion of alternative approaches to safety and performance within the same systems programming space, rather than across fundamentally different execution models.

Languages with garbage collectors such as Java or C# are intentionally excluded. While they offer strong safety guarantees and high developer productivity, their managed runtimes introduce non deterministic memory reclamation (memory management behavior where freed memory may be reclaimed and reused at an unspecified time, rather than immediately or predictably), additional runtime overhead, and limited control over memory layout. These characteristics make them less suitable for operating systems, kernels, embedded systems, and performance critical infrastructure, which are the primary contexts in which C is traditionally used.

5.1 Overview of the Languages Under Comparison

5.1.1 C++[8]

C++ extends C with higher-level abstractions such as classes, and templates while preserving low-level control.

5.1.2 Rust[9]

Rust is a modern systems programming language that emphasizes performance, reliability, and practical low-level programming through strong compile-time guarantees.

5.1.3 Zig[10]

Zig is a low-level systems language that emphasizes explicitness and simplicity, retaining manual memory management while providing safer defaults, compile-time execution, and improved tooling compared to C.

5.2 Metrics

5.2.1 Memory Safety

- **C**: The C language provides no automatic memory safety; developers manually manage memory using pointers, malloc/free , etc. There are no bounds checks or built-in protections against memory errors, resulting in them being commonplace in C code. In fact, industry studies have found that about 70% of security vulnerabilities at Microsoft and Google are due to memory safety issues in C/C++ code [12]. C relies (largely) on the programmer to prevent such errors, making memory management error-prone.
- **C++**: C++ inherits C's manual memory management by default and is also not memory-safe by design . However, modern C++ offers safer constructs to mitigate memory bugs. Features like smart pointers (`std::unique_ptr` , `std::shared_ptr`) help automate memory cleanup and prevent some leaks or double-frees. The C++ standard library provides containers (`std::vector` , etc.) that can do optional bounds checking (e.g., `vector::at()` throws on out-of-range). Despite these improvements, C++ still mainly retains the issues that C has had.
- **Rust**: Rust was designed from the ground up with memory safety as a primary goal. It uses a strict ownership and borrowing model enforced at compile time by the borrow checker. In safe Rust code, the compiler guarantees no dangling pointers and no buffer overflows (array accesses are bounds- checked at runtime). Notably, Rust achieves memory safety without needing a garbage collector. Instead, it performs static analysis to ensure references are valid and memory is freed when it's no longer accessible (deterministic destruction). All these checks occur at compile time, so they incur "zero-cost" at runtime. The result is that if your code compiles in Rust (without using `unsafe`), whole classes of memory errors are eliminated.
- **Zig**: Zig takes a middle ground approach. Like C, Zig uses manual memory management (no automatic garbage collection) , but it introduces several safety mechanisms to reduce mistakes. For example, Zig has a built-in slice type (pointer + length) and performs bounds- checking on array accesses by default in debug mode . It uses an optional type instead of raw null pointers, so you cannot dereference a potentially null pointer without first checking it (similar to Rust's `Option`). Zig's type system tracks pointer alignment and forbids misaligned pointer casts without explicit override. Zig also provides a debug allocator in its standard library which can catch use-after-free and double-free errors at runtime by poisoning freed memory and detecting leaks. These features remove many "footguns" present in C, making Zig code safer by default in testing.
- Winner: **Rust**

5.2.2 Performance

At a high level, all these languages deliver similar performance because they compile to native machine code and give programmers direct control over memory layout, data representation, and execution. None of them require garbage collection, and all can express low-level constructs such as pointer arithmetic, stack allocation, and explicit heap management, allowing the generated code to map closely to the underlying hardware.

- Winner: **None**

5.2.3 Syntax and Maintainability

- **C**: The syntax of C is simple and minimalistic. It's a small language by modern standards, which can be an advantage for clarity because a C programmer doesn't have to juggle templates, classes, or generics when reading code. However, C lacks built-in abstractions for many things, so large C codebases tend to rely on manual conventions (for example, manually managing strings). This can make maintenance challenging as projects grow.
- **C++**: C++ adds a great deal of complexity on top of C in exchange for more powerful abstraction mechanisms. On the one hand, C++ supports object-oriented and generic programming, which lets developers create modular, reusable, and high-level constructs that can improve maintainability. For example, classes and templates allow writing code that is generic and type-safe, avoiding repetition. On the other hand, C++ is famously complex: the language has accumulated many features (multiple inheritance, template metaprogramming, etc.), and there are often many ways to do something. This richness can result in a steep learning curve and readability issues if abused.
- **Rust**: Rust's syntax and semantics were designed with clarity and correctness in mind, though some newcomers find it initially intimidating. Rust is very explicit about potential errors and lifetimes. For example, Rust has no null references and no exceptions. This leads to very transparent error handling in code: when reading a Rust function, you can usually see all the places that might fail and how failures are dealt with. This predictability improves maintainability (fewer hidden failure paths), but it does mean Rust code is often more verbose than equivalent code in C++ (which might ignore an error or throw exceptions implicitly). Once you're used to it, Rust's strictness tends to produce code that is robust and refactoring-friendly. The syntax itself is modern and expressive: it has closures, pattern matching, generics, etc., akin to a blend of functional and imperative styles.
- **Zig**: Zig deliberately prioritizes a simple, readable syntax and a small feature set, closely aligned with C's ethos but with modern refinements. Its syntax is familiar to C developers: imperative style, `fn` for functions, `struct` for data, no inheritance, no function or operator overloading, and no implicit behavior. A core design rule is no hidden control flow: operations that can fail or allocate memory must be explicit, with error handling enforced via returned error values and `try`/`catch`. This makes Zig code highly predictable to read and trace, since expressions and function calls cannot silently change meaning or behavior.
- Winner: **Zig**

5.2.4 Real-World Use Cases

- **C**: C remains king in operating systems and low-level embedded software. The canonical example is the UNIX/Linux kernel and many parts of Windows, which are primarily C. Device drivers, hardware interface firmware, and BIOS/UEFI firmware are often written in C due to its direct hardware control and minimal runtime. In the embedded world (microcontrollers, IoT devices, medical devices), C is extremely common because it is lightweight and predictable, running on tiny CPUs with no OS . Many networking and telecommunications systems (routers, telecom switches) use C internally. Basically, if maximum portability and decades of reliability are needed, C is a top choice. The downside is that new projects in these spaces are cautiously moving to safer alternatives, but C's legacy and efficiency ensure it will be used and maintained in critical systems for a long time.
- **C++**: C++ is prevalent in scenarios where performance is critical and software complexity is high enough that C would be too low-level. A prime example is game development: almost all major game engines (Unreal Engine, Unity's core, CryEngine) and many AAA game codebases are in C++ . The ability to use object-oriented design and templates helps manage the complexity of game logic, while still getting near-C performance for math-heavy code. In financial services, trading platforms and quantitative models often use C++ to get speed for calculations while structuring the code with classes. High- performance databases and storage systems: MySQL and MongoDB (as noted) use C++ for its blend of performance and higher-level constructs (and indeed things like Bloomberg's high-performance financial data store are C++). Essentially, C++ is used anywhere you need to build a large, complex system that directly runs on the machine without a VM. Its compatibility with C (and ability to gradually migrate C projects) also means a lot of legacy C codebases (like certain OS components or older libraries) have portions rewritten or extended in C++ for better maintainability.

- **Rust**: Rust, being newer, has seen enthusiastic uptake in areas that value safety alongside performance. A notable and growing use case is systems-level components that were traditionally C/C++ but are security-critical. For example, Rust is being used for operating system components: the Linux kernel's inclusion of Rust drivers , exploratory OS projects like Google's Fuchsia using Rust, and Microsoft experimenting with Rust for low-level Windows components. The momentum is strong and likely to continue as more success stories (e.g., a key part of Android is now Rust-based to eliminate certain vulnerabilities) come out.
- **Zig**: Zig's adoption is nascent, but it tends to attract projects that could be done in C but whose developers want a better experience. Systems and tools are a common theme. High-performance financial or database systems like TigerBeetle choosing Zig is a notable example; they needed absolute control over memory (using custom allocators, avoiding fragmentation) and predictable performance, and Zig gave them that with less friction than C and without the complexities of C++ or Rust's borrow checker. In summary, current real-world Zig use cases are somewhat specialized: developer tools, some performance- critical services, and systems-level programming by those willing to try a new language.
- Winner: **C/C++**

Summary of Results:

Metric \ Language	C	C++	Rust	Zig	Winner
Memory Safety	Poor	Average	Excellent	Good	**Rust**
Performance	Good	Good	Good	Good	**None**
Syntax and Maintainability	Average	Good	Good	Excellent	**Zig**
Real-World Use Cases	Excellent	Excellent	Average	Poor	**C/C++**

6 Personal Experience with C

My experience with C has been shaped largely through systems-level coursework, particularly a programming assignment (CS241). This project exposed both the power of C and the kinds of failures that arise from its low-level control and lack of built-in safety guarantees.

A recurring issue was obviously manual memory management. In one instance, I introduced a memory leak along a rarely taken failure path in a hash set implementation: the function returned early without freeing allocated memory. The bug was difficult to diagnose precisely because it occurred infrequently and did not cause immediate crashes. This illustrates a common problem in C: correctness depends on the programmer exhaustively reasoning about all control-flow paths, including error handling, with no language support enforcing cleanup.

I also encountered undefined behavior caused by incorrect annotations and assumptions. I declared a function with the `noreturn` attribute but later returned from it. This mismatch led to highly non-intuitive behavior, manifesting as an apparent infinite loop. The compiler assumed the function would never return and optimized accordingly, demonstrating how C's permissiveness allows the programmer to violate assumptions that the compiler then treats as axiomatic.

Despite these problems, C also demonstrated clear advantages over higher-level languages such as Python (which I also have some experience with). The performance benefits were tangible; the minimal runtime overhead made C well-suited for low-level networking and OS code. Debugging at the machine level, while difficult, provided precise visibility into program behavior, which is often obscured in managed languages.

7 Broader Implications in the field of POPL

Working through memory safety in C shifted my understanding of Programming Languages from "syntax and features" to formal guarantees and semantic trade-offs. Amorim et al.'s characterization was especially useful because it frames memory safety as a property about how programs can and cannot be influenced by memory outside an object's intended bounds or lifetime, making the distinction between spatial and temporal safety feel principled rather than ad hoc. It also clarified why C is difficult to reason about modularly: once undefined behavior and memory corruption are possible, local reasoning about a component is fragile because unrelated parts of the program can affect each other through accidental memory interference. Overall, this essay made it clear that PoPL research is often about deciding which invariants should be enforced by the language semantics (as in Rust's type system) versus retrofitted by instrumentation or runtime checks (as in C mitigations), and what each choice costs in performance, compatibility, and developer freedom.

8 Source Evaluation

8.1 Main Papers

8.1.1 Amorim

Amorim et al.'s The Meaning of Memory Safety is a theory-driven paper published by researchers from the University of Pennsylvania and Inria, both well-established institutions in programming languages research. The work is grounded in formal semantics and mechanized proofs using Coq, which significantly strengthens its credibility by reducing reliance on informal argumentation. Rather than proposing a system, the paper provides a rigorous conceptual framework for memory safety and evaluates existing mechanisms against it, making it methodologically sound and non-promotional.

8.1.2 Nagarakatte

Nagarakatte et al.'s SoftBound: Highly Compatible and Complete Spatial Memory Safety for C was published at PLDI 2009[11], one of the top-tier conferences in programming languages and systems research. The authors are affiliated with the University of Pennsylvania, and the paper combines formal reasoning with extensive empirical evaluation on real C programs. Its claims are supported by a clear threat model, detailed design rationale, and measured performance results, including comparisons against existing tools such as Valgrind and Mudflap. The combination of formal justification, reproducible implementation details, and realistic benchmarks makes this a highly reliable source on spatial memory safety for C.

8.1.3 Farkhani

Farkhani et al.'s PTAuth: Temporal Memory Safety via Robust Points-to Authentication appears in the Proceedings of the 30th USENIX Security Symposium (2021)[12], a leading venue for applied systems security research. The paper is authored by researchers from Northeastern University and presents a concrete system with a well-defined threat model, implementation on ARM architectures, and evaluation using both synthetic test suites and industry-standard benchmarks. Its reliance on hardware features such as ARM Pointer Authentication is explicitly discussed, and limitations are clearly acknowledged. The work's experimental rigor, transparent assumptions, and publication in a highly selective security conference establish it as a credible and trustworthy source on temporal memory safety

8.2 Documentation

8.2.1 C++

C++ documentation, especially the ISO C++ standard and cppreference, is generally reliable due to strong community scrutiny and alignment with the standard. However, the language's complexity and long evolution mean that documentation can be fragmented, version-dependent, and difficult to reason about, increasing the risk of outdated or context-specific misunderstandings.

8.2.2 Rust

Rust's official documentation is highly trusted because it is centrally maintained, versioned, and tightly integrated with the compiler and tooling. The Rust Book and standard library docs emphasize correctness and safety guarantees, though the language's relative youth means some areas evolve quickly, which can invalidate older explanations.

8.2.3 Zig

Zig's documentation is generally reliable for core concepts because it is maintained by the language designers and closely tracks implementation. However, Zig is still evolving rapidly, and some features or design decisions may change, making older documentation or third-party explanations less dependable.

8.3 Other Sources

8.3.1 MITRE

MITRE is a highly trustworthy source due to its institutional authority, rigorous curation, and widespread adoption of standards like CWE. Its content is factual and conservative, but often high-level and descriptive, offering limited technical depth or implementation guidance.

8.3.2 Other

Blogs and general websites were seldom used as core sources for this essay. This is because they vary widely in trustworthiness. They are useful for intuition, examples, and practical insights, but are often opinionated, simplified, or outdated, and should be treated as supplementary rather than authoritative sources. However, using a combination of MITRE for the concrete facts, and blogs for details on implementation and general understanding proved to be quite effective.

9 References

- [1]The Meaning of Memory Safety, Amorim Et Al., 2018
- [2]SoftBound: Highly Compatible and Complete Spatial Memory Safety for C,Nagarakatte Et Al., 2009
- [3] PTAuth: Temporal Memory Safety via Robust Points-to Authentication,Farkhani Et Al., 2021
- [4] MITRE CWE-121: Stack-based Buffer Overflow
- [5] MITRE CWE-126: Buffer Over-read
- [6] MITRE CWE-416: Use After Free
- [7] MITRE CWE-415: Double Free
- [8] cppreference
- [9] Rust Documentation
- [10] Zig Documentation
- [11] PLDI
- [12] USENIX