Google

Google Security Engineering Technical Report
March 4, 2024

Secure by Design: Google's Perspective on Memory Safety

Alex Rebert
arebert@google.com

Christoph Kern
xtof@google.com

Executive Summary

2022 marked the 50th anniversary of memory safety vulnerabilities, first reported by Anderson [2]. Half a century later, we are still dealing with memory safety bugs despite substantial investments to improve memory unsafe languages.

Like others', Google's internal vulnerability data and research show that memory safety bugs are widespread and one of the leading causes of vulnerabilities in memory-unsafe codebases. Those vulnerabilities endanger end users, our industry, and the broader society.

At Google, we have decades of experience addressing, at scale, large classes of vulnerabilities that were once similarly prevalent as memory safety issues. Based on this experience we expect that high assurance memory safety can only be achieved via a Secure-by-Design approach centered around comprehensive adoption of languages with rigorous memory safety guarantees. As a consequence, we are considering a gradual transition towards memory-safe languages.

Over the past decades, Google has developed and accumulated hundreds of millions of lines of C++ code that is in active use and under active, ongoing development. This very large existing codebase results in significant challenges for a transition to memory safety:

- On one hand, we see no realistic path for an evolution of C++ into a language with rigorous memory safety guarantees that include temporal safety.
- At the same time, a large-scale rewrite of existing C++ code into a different, memory-safe language appears very difficult and will likely remain impractical.

This means that we will likely be operating a very substantial C++ codebase for quite some time. We thus consider it important to **complement a transition to memory safe languages for new code** and particularly at-risk components **with safety improvements for existing C++ code**, to the extent practicable. We believe that substantial improvements can be achieved through an incremental transition to a partially-memory-safe C++ language subset, augmented with hardware security features when available.

Defining Memory Safety Bugs

Memory safety bugs arise when a program allows statements to execute that read or write memory, when the program is in a state where the memory access constitutes undefined behavior. When such a statement is reachable in a program state under adversarial control (e.g., processing untrusted inputs), the bug often represents an exploitable vulnerability (in the worst case, permitting arbitrary code execution).

Defining Rigorous Memory Safety

In this context, we consider a language **rigorously memory-safe** if it:

- Defaults to a well-delineated safe subset, and
- Ensures that arbitrary code written in the safe subset is prevented from causing a spatial, temporal, type, or initialization safety violation [1]

  This can be established through any combination of compile-time restrictions and runtime protections pro-

[1] Under the assumption that all unsafe code that is part of the program is sound.

vided the runtime mechanisms guarantee that safety violation cannot occur.

With very few, well-defined exceptions, all code should be writable in the well-delineated safe subset.

In new development, potentially unsafe code should only occur in components/modules that explicitly opt into use of unsafe constructs outside of the safe language subset, and expose a safe abstraction that is expert-reviewed for soundness. Unsafe constructs should only be used when necessary, e.g. for critical performance reasons or in code that interacts with low-level components.

When working with existing code in a non-memory-safe language, unsafe code should be restricted to uses including:

- Code written in a safe language that makes calls into a library implemented by a legacy codebase written in an unsafe language.
- Code additions/modifications to existing unsafe legacy code bases, where code is too deeply intermingled to make development in a safe language practical.

Impact of Memory Safety Vulnerabilities

Memory safety bugs are responsible for the majority (~70%) of severe vulnerabilities in large C/C++ code bases. Below are the percentage of vulnerabilities due to memory unsafety:

- **Chrome**: 70% of high/critical vulnerabilities [6]
- **Android**: 70% of high/critical vulnerabilities[2] [8]
- **Google servers**: 16-29% of vulnerabilities[3]
- **Project Zero**: 68% of in-the-wild zero days [11]
- **Microsoft**: 70% of vulnerabilities with CVEs [17]

Memory safety errors continue to appear at the top of "most dangerous bugs" lists such as CWE Top 25 and CWE Top 10 of Known Exploited Vulnerabilities. Google's internal vulnerability research repeatedly demonstrates that lack of memory safety weakens important security boundaries.

[2]The fraction of memory safety vulnerabilities has gone down over the last few years thanks to memory safety improvements.

[3]The range reflects uncertainty around automated severity assessment of memory safety issues found by our automation, e.g. by fuzzing. Also note that this is across all workloads, including those written in memory-safe languages such as Go and Java/Kotlin.

Understanding Memory Safety Bugs

Classes of Memory Safety Bugs

It can be helpful to distinguish a number of subclasses of memory safety bugs that differ in their possible solutions and the impact on performance and developer experience thereof:

- **Spatial Safety** bugs (e.g. "buffer overflow", "out of bounds access") occur when a memory access refers to memory outside of the accessed object's allocated region.
- **Temporal Safety** bugs arise when a memory access to an object occurs outside of the object's lifetime. An example is when a function returns a pointer to a value in its stack frame ("use-after-return"), or due to a pointer to heap-allocated memory that has since been freed, and possibly re-allocated for a different object ("use-after-free").

  It is common in concurrent programs for these bugs to occur due to improper thread synchronization, but when the initial safety violation is outside of the lifetime of the object, we classify it as a temporal safety violation.
- **Type Safety** bugs arise when a value of a given type is read from memory that does not contain a member of this type. An example of this is when memory is read after an invalid pointer cast.
- **Initialization Safety** bugs arise when memory is read before being initialized. This can lead to information disclosures and type/temporal safety bugs.
- **Data-Race Safety** bugs arise from unsynchronized reads and writes by different threads, which may access an object in an inconsistent state. It is possible for other forms of safety bugs to also arise from improper or missing synchronization, however we do not classify these as data-race safety bugs and they are handled above. Only when the reads and writes are otherwise correct except for being unsynchronized are they considered data-race safety bugs.

  Once a data-race safety violation has occurred, subsequent execution may cause further safety bugs. We classify these as data-race safety bugs as the *initial* violation is strictly a data-race issue without any other bugs evident.

The classification used here roughly aligns with Apple's memory safety taxonomy [4].

In unsafe languages such as C/C++, it is the programmer's responsibility to ensure the safety preconditions are met to avoid accessing invalid memory. For instance, for spatial safety, when accessing elements of an array via index (e.g., `a[i] = x`), it is the programmer's responsibility to ensure the safety precondition that the index is within the bounds of validly-allocated memory.

We currently exclude data-race safety from consideration under rigorous memory safety for the following reasons:

- Data-race safety is a bug class of its own, and only partially overlaps with memory safety. For example, Java does not provide data-race-safety guarantees, but data races in Java cannot cause violation of low-level heap integrity invariants (memory corruption).
- We currently do not have the same level of evidence for data-race unsafety leading to systemic security and reliability issues for software written in otherwise rigorously memory safe languages (e.g. Go).

Why are Memory Safety Bugs so Intractable?

Memory safety bugs are quite common in large C++ code bases. The intuition behind the prevalence of memory safety bugs is as follows:

First, in unsafe languages, programmers are responsible for ensuring that each statements' memory safety precondition holds just before it is executed, in any program state that could be possibly reached, potentially under the influence of adversarial inputs to the program.

Secondly, unsafe statements that *potentially* result in memory safety bugs are very common in C/C++ programs – there are many array accesses, pointer dereferences, and heap allocations.

Finally, reasoning about safety preconditions and whether the program ensures them in every possible program state is difficult, even with tool assistance. For example:

- Reasoning about the in-bounds-ness of a pointer/index involves wrapping integer arithmetic, which is quite non-intuitive to humans.
- Reasoning about the lifetime of heap objects often involves complicated and subtle whole-program invariants. Even local scoping and lifetime can be subtle and surprising.

"Many potential bugs" combined with "difficult reasoning about safety preconditions" and "humans make mistakes" results in a relatively significant number of *actual* bugs.

Attempts to mitigate the risk of memory safety vulnerabilities through developer education and reactive approaches (including static/dynamic analysis to find and fix bugs, and various exploit mitigations) have failed to lower the incidence of these bugs to a tolerable level. As a result, severe vulnerabilities continue to be caused by this class of vulnerabilities as discussed above.

Tackling Memory Safety Bugs

Tackling memory safety requires a multi-pronged approach consisting of:

- **Preventing** memory safety bugs through **Safe Coding**.
- **Mitigating** memory safety bugs by making exploitation more expensive.
- **Detecting** memory safety bugs, as early as possible in the development lifecycle.

We believe that all three are necessary for solving memory safety at Google's scale. Based on our experience, a strong emphasis on prevention through safe coding is necessary to sustainably achieve high assurance.

Preventing Memory Safety Bugs through Safe Coding

Our experience at Google shows that we can engineer away classes of problems at scale by eliminating the use of vulnerability-prone coding constructs. In this context, we consider a construct unsafe if it can potentially manifest a bug (e.g. memory corruption) unless a safety precondition is satisfied at its time of use. Unsafe constructs place the onus on the developer to ensure the precondition. Our approach, which we call "Safe Coding", treats unsafe coding constructs themselves as hazards (i.e., independently of and in addition to the vulnerability they might cause), and is centered around ensuring that developers do not encounter such hazards during regular coding practice [13].

In essence, Safe Coding calls for unsafe constructs to be disallowed by default, and their use to be replaced by safe abstractions in most code, with carefully-reviewed exceptions. In the domain of memory safety, safe abstractions may be provided using:

- **Statically- or dynamically-ensured safety invariants**, preventing the introduction of bugs. Compile-time checks and compiler-emitted or runtime-provided mechanisms guarantee that particular classes of bugs cannot occur. For instance:
  - At compile-time, lifetime analysis prevents a subset of temporal safety bugs.
  - At runtime, automated object initialization guarantees the absence of uninitialized reads.
- **Runtime error detection**, enforcing memory safety invariants by raising an error when a memory safety violation is detected instead of continuing execution with corrupted memory. The underlying bugs still exist and will need to be fixed[4], but the vulnerabilities are eliminated (modulo denial-of-service attacks[5]). For instance:

[4] Runtime error detection helps root-cause crashes by precisely pinpointing the underlying memory safety bug.

[5] Runtime errors can typically be caught and recovered from; e.g. an out-of-bounds access in Go raises a recoverable run-time panic. This allows

  - An array lookup may offer spatial safety error detection by verifying the given index is in-bounds. Checks may be elided where safety is proven statically.

  - A type cast may offer type safety error detection by checking that the casted object is an instance of the resulting type (e.g. `ClassCastException` in Java or CastGuard for C++).

In the memory safety domain, the Safe Coding approach is embodied by safe languages, which replace unsafe constructs with safe abstractions such as runtime bounds checks, garbage-collected references, or references adorned with statically-checked lifetime annotations.

Experience shows that memory safety issues are indeed rare in safe, garbage-collected languages such as Go and Java. However, garbage collection typically comes with significant runtime overhead. More recently, Rust has emerged as a language that embodies the Safe Coding approach based primarily on compile-time checked type discipline, resulting in minimal runtime overheads.

Data shows that safe coding works for memory safety, even in performance sensitive environments. For instance, Android 13 introduced 1.5M lines of Rust with zero memory safety vulnerabilities. This prevented an estimated hundreds of memory safety vulnerabilities: "As the amount of new memory unsafe code entering Android has decreased, so too has the number of memory safety vulnerabilities. [...] 2022 was the first year where memory safety vulnerabilities did not represent a majority of Android's vulnerabilities. While correlation doesn't necessarily mean causation, [...] the shift is a major departure from industry-wide trends listed above that have persisted for more than a decade".

As another example, Cloudflare reports that their Rust HTTP proxy outperforms NGINX, and has "served a few hundred trillion requests and [has] yet to crash due to our service code."

By applying a subset of preventative memory safety mechanisms to an unsafe language such as C++, we can partially prevent classes of memory safety issues. For instance:

- A buffer hardening RFC may eliminate a subset of spatial safety issues in C++.
- Similarly, a bounds safety RFC may eliminate a subset of spatial safety issues in C.
- Lifetime annotations in C++ may eliminate a subset of temporal safety issues.

servers to safely recover from runtime errors raised during processing of a request, without crashing the entire process. Runtime errors raised in server frameworks code itself may not be recoverable.

Exploit Mitigations

Exploit mitigations complicate exploitation of memory safety vulnerabilities, rather than fixing the root cause of these vulnerabilities. For instance, mitigations include sandboxing of unsafe libraries, control-flow integrity, and data execution prevention.

While safe abstractions prevent memory corruption, denying exploitation primitives to attackers, exploit mitigations assume that memory can be corrupted. Exploit mitigations aim to make it difficult for attackers to escalate from some exploitation primitives to unrestricted code execution.

Attackers regularly bypass these mitigations, raising the question of their security value. To be useful, mitigations should require attackers to chain additional vulnerabilities, or invent a novel bypass technique. Over time, bypass techniques become more valuable to attackers than any single vulnerabilities. The security benefit of a well-designed mitigation lies in the fact that bypass techniques should be far rarer than vulnerabilities.

Exploit mitigations rarely come for free; they tend to incur a runtime overhead that is generally a low single-digit percentage. They provide a tradeoff between security and performance, which we can adjust based on each workloads' needs. Runtime overheads can be reduced by building mitigations directly in the silicon, as was done for pointer authentication, shadow call stack, landing pads, and protection keys. Due to their overhead and opportunity costs of hardware features, considerations around adoption of, and investment in, those techniques are nuanced.

In our experience, sandboxing is an effective mitigation for memory safety vulnerabilities and is commonly used at Google to isolate brittle libraries with a history of vulnerabilities. However, there are several challenges to the adoption of sandboxing:

- Sandboxing can incur significant overheads in latency and bandwidth, as well as costs for the required code refactoring. This sometimes necessitates reuse of sandbox instances across requests, which weakens the mitigation.
- Creating a sandbox policy that is sufficiently restrictive to be an effective mitigation can be challenging for developers, especially when sandbox polices are expressed at a low level of abstraction, such as system calls filters.
- Sandboxing can cause reliability risks, when unusual (but benign) code paths are exercised in production and trigger sandbox policy violations.

Overall, exploit mitigations are an essential tool in improving the security of a large pre-existing C++ code base, and will also benefit residual use of unsafe constructs in memory-safe languages.

Finding Memory Safety Bugs

Static analysis and fuzzing are effective tools for detecting memory safety bugs. They reduce the volume of memory safety bugs in our code base as developers fix the detected issues.

However, in our experience, bug finding alone does not achieve an acceptable level of assurance for memory-unsafe languages. As an example, the recent webp high severity 0-day (CVE-2023-4863) affected extensively fuzzed code. The vulnerability was missed despite high fuzzing coverage (97.55% in the relevant file). In practice, we miss many memory safety bugs, as demonstrated by the steady stream of memory safety vulnerabilities in well-tested memory-unsafe code.

In addition, finding bugs does not in itself improve security. The bugs must be fixed and the patches deployed. There is evidence suggesting that bug finding capabilities are outpacing bug fixing capacity. For instance, syzkaller, our kernel fuzzer, has found 5k+ bugs in the upstream Linux kernel , such that at any given time there are hundreds of open bugs (a large fraction of which are likely security-relevant), a number that is been steadily growing since 2017.

We nevertheless believe that bug finding is an essential part of tackling memory unsafety. Bug finding techniques that put less strain on bug fixing capacity are particularly valuable:

- "Shifting-left", such as fuzzing in presubmit, reduces the rate of new bugs shipped to production. Bugs found earlier in the SDLC (software development life cycle) are cheaper to fix[6], consequently increasing our bug fixing capacity.
- Bug finding techniques, like static analysis, may also suggest fixes, which can be provided through the IDE or pull requests, or applied automatically to proactively change existing code.
- Bug finding tools like sanitizers , which identify root causes and generate actionable bug reports, help developers fix issues faster, also increasing our bug fixing capacity.

Additionally, bug finding tools find bug classes beyond memory safety, which broadens the impact of investing into those tools. They can find reliability, correctness and other safety issues, for instance:

- Property-based fuzzing finds inputs violating application-level invariants, such as correctness properties encoded by developers. For instance, cryptofuzz has found 150+ bugs in crypto libraries.
- Fuzzing finds resource-usage bugs (e.g. infinite recursions), and plain crashes affecting availability. In particular, runtime error detection (e.g. bounds checking) transforms memory safety vulnerabilities into runtime errors, which remain a reliability and DoS concern.
- Advances in detecting vulnerabilities beyond memory safety are showing promise .

[6] "The average cost of finding and fixing a bug increases about 10 times with every step of the development process", [12]

Deep Dive: Safe Coding Applied to Memory Safety

Google has developed Safe Coding , a scalable approach to drastically reduce the incidence of common classes of vulnerabilities, and to achieve a high degree of assurance that vulnerabilities are absent.

Over the past decade, we have applied this approach very successfully at Google's scale, primarily to so-called Injection Vulnerabilities , including SQL injection and XSS. While at a technical level very different from memory safety bugs, there are relevant parallels:

- Like memory safety bugs, injection bugs occur when a developer uses a potentially-unsafe code construct, and fails to ensure its safety precondition.
- Whether the precondition holds depends on complex reasoning about whole-program, or whole-system, data flow invariants. For example, the potentially-unsafe construct occurs in browser-side code, but the data might arrive via several microservices and a server-side datastore. This makes it hard to reason about where data really came from, and whether necessary validation has been correctly applied somewhere along the way.
- Potentially-unsafe constructs are common in typical code bases.

As with memory safety bugs, "many 1000's of potential bugs" led to 100's of actual bugs. Reactive approaches (code review, pen testing, fuzzing) were largely unsuccessful.

To address this issue at scale and with high assurance, Google applied Safe Coding to the domain of injection vulnerabilities. This was unequivocally successful and resulted in a very significant reduction, and in some cases complete elimination of XSS vulnerabilities. For example, before 2012, web frontends like GMail often had a few dozen XSS per year; after refactoring code to conform to Safe Coding requirements, defect rates have dropped to near zero. The Google Photos web frontend (which has been developed from the start on a web application framework that comprehensively applies Safe Coding) has had zero reported XSS vulnerabilities in its entire history.

In the following sections, we discuss in more detail how the Safe Coding approach applies to memory safety, and draw parallels to its successful use in eliminating classes of vulnerabilities in the web security domain.

Safe abstractions

In our experience, the key to eliminating classes of bugs is to identify programming constructs (APIs or language-native constructs) that cause these bugs, and then to eliminate the use of such constructs in common programming practice. This requires the introduction of safe constructs with equivalent functionality, which often take the form of safe abstractions around the underlying unsafe constructs.

For example, XSS are caused by the use of Web Platform APIs that are unsafe to call with partially attacker-controlled strings. To eliminate the use of these XSS-prone APIs in our code, we introduced a number of equivalent safe abstractions, designed to collectively ensure that safety preconditions hold when the underlying unsafe constructs (APIs) are invoked. This includes type-safe API wrappers, vocabulary types with safety contracts, and safe HTML templating systems.

Safe abstractions to ensure memory safety preconditions might take the form of wrapper APIs in an existing language (e.g. Smart Pointers to be used in place of raw pointers, including `MiraclePtr` which protects 50% of use-after-free issues against exploitation in Chrome's browser process), or constructs closely tied to language semantics (for example, garbage collection in Go/Java; statically-checked lifetimes in Rust).

The design of safe constructs needs to navigate a 3-way tradeoff between runtime costs (CPU, memory, binary size, etc), development-time costs (developer friction, cognitive load, build times), and expressiveness. For example, garbage collection provides a general solution for temporal safety, but can cause problematic variability in performance [7]. Rust lifetimes combined with the borrow checker ensure safety entirely at compile time (at no runtime cost) for large classes of code[7]; however require more upfront effort by the programmer to demonstrate that the code is in fact safe. This is similar to how static typing requires more upfront effort compared to dynamic typing, but prevents a large swath of type errors at compile time.

Sometimes, developers need to choose alternative idioms to avoid runtime overhead. For example, the overhead of a runtime bounds check for indexed traversal of a vector can be avoided by using a range-for loop.

To successfully reduce the incidence of bugs, a collection of safe abstractions needs to be sufficiently expressive to allow most code to be written without resorting to unsafe constructs (nor convoluted, non-idiomatic code that is technically safe, but difficult to understand and maintain).

[7] Exceptions include cyclical data structures, which can be implemented using runtime-checked interior mutability

Safe-by-default, unsafe-by-exception

In our experience, it is not sufficient to merely make safe abstractions available to developers on an optional basis (e.g. suggested by a style guide) as too many unsafe constructs, and hence too much risk of bugs, tend to remain. Rather, to achieve a high degree of assurance that a codebase is free of vulnerabilities, we have found it necessary to adopt a model where unsafe constructs are used only by exception, enforced by the compiler.

This model consists of the following key elements:

1. It is possible to decide at build time whether a program (or part of a program, e.g. a module) contains unsafe constructs.
2. A program consisting only of safe code is guaranteed to maintain safety invariants at runtime.
3. Unsafe constructs are not permitted unless explicitly allowed/opted-into, i.e. code is safe by default.

In our work on injection vulnerabilities, we achieved safety at scale by restricting access to unsafe APIs through language-level and build-time visibility, and in some cases through custom static checks.

In the context of memory safety, achieving safety at scale requires the language[8] to prohibit the use of unsafe constructs (e.g. unchecked indexing into arrays/buffers) by default. Unsafe constructs should cause a compile-time error unless a portion of code is explicitly opted into the unsafe subset as discussed in the next section. For example, Rust allows unsafe constructs only inside clearly-delineated `unsafe` blocks.

Soundness: Safely-encapsulated unsafe code

As noted above, we assume that available safe abstractions are sufficiently expressive to allow most code to be written using safe constructs only. In practice however, we expect most larger programs to require use of unsafe constructs in some cases. In addition, the safe abstractions themselves will often be wrapper APIs for underlying unsafe constructs. For example, the implementation of safe abstractions around heap memory allocation/deallocation ultimately needs to deal with raw memory, e.g. `mmap(2)`.

When developers introduce (even small amounts) of unsafe code, it is important to do so without negating the benefits of having written most of a program using only safe code.

To that end, developers should adhere to the following principle: **Uses of unsafe constructs should be encapsulated in demonstrably-safe APIs.**

[8] In this broader context, this could mean a memory-safe language, or a safe subset of an otherwise unsafe language.

That is, unsafe code should be encapsulated behind an API that is sound for any arbitrary (but well-typed) code calling this API. It should be possible to demonstrate, and review/verify, that the module exposes a safe API surface without making any assumptions about the calling code (other than its well-typedness).

For example, suppose the implementation of a type uses a potentially-unsafe construct. Then it is the *type's implementation's* responsibility to independently ensure that the unsafe construct's precondition holds when it is invoked. The implementation must not make any assumptions about the behavior of its callers (besides well-typedness), for example that its methods are called in a certain order.

In our work on injection vulnerabilities, this principle is embodied in guidelines for the use of so-called Unchecked Conversions (which represent unsafe code in our vocabulary-type discipline). In the Rust community, this property is called Soundness [14]: a module with `unsafe` blocks is sound if a program consisting of that module, combined with arbitrary well-typed safe Rust, cannot exhibit Undefined Behavior.

This principle can be difficult or impossible to adhere to in certain situations, like when a program in a safe language (Rust or Go) calls into unsafe C++ code. The unsafe library might be wrapped in a "reasonably safe" abstraction, but there is no practical way to demonstrate that the implementation is truly safe and does not have a memory safety bug.

Expert review of unsafe code

Reasoning about unsafe code is difficult and can be error-prone, especially for non-experts:

- Reasoning about whether a module containing unsafe constructs in fact exposes a safe abstraction requires domain expertise.
  - For example, in the web security domain, deciding if an unchecked conversion into the SafeHtml vocabulary type is safe requires a detailed understanding of the HTML spec, and applicable data escaping and sanitization rules.
  - Deciding whether Rust code with `unsafe` is sound requires a deep understanding of unsafe Rust semantics and the boundaries of Undefined Behavior (an area of active research).
- In our experience, developers focused on solving a problem at hand frequently do not seem to appreciate the importance of safely encapsulating unsafe code, and do not attempt to devise a safe abstraction. Expert review is needed to steer those developers towards safe encapsulation, and to help design an appropriate safe abstraction.

In the web security domain, we found it necessary to mandate expert review of unsafe constructs in many cases, like for new uses of Unchecked Conversions . Without mandatory review we observed a large number of unnecessary/unsound uses of unsafe constructs, which diluted our ability to reason about safety at scale. Mandatory review requirements need to carefully consider the impact on developers and the bandwidth of the review team, and are likely only appropriate if they are sufficiently rare .

Whole-Program Safety and Compositional Reasoning

Ultimately, our goal is to ensure an adequate safety posture for an entire binary.

Binaries typically include a large number of direct and transitive library dependencies. These are typically maintained by many different teams within Google, or even externally in the case of third party code. Yet, a memory safety bug in any of the dependencies can potentially result in a security vulnerability of the dependent binary.

A safe language, combined with a development discipline to ensure that unsafe code is encapsulated in sound, safe abstractions, can enable us to scalably reason about the safety of large programs:

- Components written solely in the language's safe subset are by construction sound and free of safety violations.
- Components that do contain unsafe constructs expose safe abstractions to the rest of the program. For these components, expert review provides solid assurance of their soundness, and that they will not cause safety violations when combined with arbitrary other components.

When all transitive dependencies fall into one of these two categories, we have solid assurance that the entire program is free of safety violations. Importantly, we do not need to reason about how each component interacts with every other component in the program; rather we can arrive at this conclusion solely based on reasoning about each component in isolation.

To maintain and ensure assertions about whole program safety over time, especially for security-critical binaries, we need mechanisms to ensure constraints on the "soundness level" of all transitive dependencies of a binary (i.e., whether they consist of safe code only or have been expert reviewed for soundness).

In practice, some transitive dependencies will have a lower level of assurance for their soundness. For example, a third-party OSS dependency might use unsafe constructs, but is not structured to expose cleanly-delineated safe abstractions that are effectively reviewable for soundness. Or, a dependency

might consist of an FFI wrapper into legacy code written entirely in an unsafe language, making it effectively impossible to review for soundness to a high degree of assurance.

Security-critical binaries may want to express constraints such as "all transitive dependencies are either free of unsafe constructs or are expert-reviewed for soundness, with the following specific exceptions", where exceptions might be subject to additional scrutiny (e.g. extensive fuzz coverage). This allows the owners of a critical binary to maintain a well-understood and acceptable level of residual unsafety risk.

Memory Safety Guarantees and Tradeoffs

Applying Safe Coding principles to memory safety of a programming language and its surrounding ecosystem (libraries, program analysis tooling) involves tradeoffs, primarily between costs incurred at development time (e.g., cognitive load placed on developers) and at deployment and run time.

This section provides an overview of possible approaches to sub-classes of memory safety, and their associated tradeoffs.

Spatial Safety

Spatial safety is relatively straightforward to incorporate into a language and library ecosystem. The compiler and container types such as strings and vectors need to ensure that all accesses are checked to be in-bounds. Checks can be elided if proven to be unnecessary based on static analysis or type invariants. Typically, this means that type implementations need metadata (size/length) to check against.

Approaches include:

- Bounds checks incorporated into APIs (e.g. `std::vector::operator[]` with safety assertions).
- Compiler-inserted bounds checks, potentially aided by annotations .
- Hardware-support such as bounds-checked CHERI capabilities.

Safe languages such as Rust, Go, Java, etc, and their standard libraries, impose bounds checks for all indexed access. They are only elided if they can be proven redundant.

It seems plausible, but has not been demonstrated for large-scale codebases like Google's monorepo or Linux kernel, that an unsafe language such as C or C++ can be subsetted to achieve spatial safety.

Bounds checks incur a small, but unavoidable run-time overhead. It is up to the developer to structure code such that bounds checks can be elided where they would otherwise accumulate to a significant overhead.

Type and Initialization Safety

Making a language type and initialization safe may include:

- Disallowing type-unsafe code constructs such as (untagged) unions and `reinterpret_cast`.
- Compiler instrumentation that initializes values on stack (unless the compiler can prove that the value will not be read before a later explicit write).
- Container type implementations that ensure that (accessible) elements are initialized.[9]

In statically-typed languages, type safety can be primarily ensured at compile time, without runtime overhead. However, there is some potential for runtime overhead in certain scenarios, for example:

- Unions must include a discriminator at runtime, and be represented as a type-safe higher-level construct (e.g. sum types). In some cases, the resulting memory overhead can be optimized away , e.g. `Option<NonZeroUsize>` in Rust.
- There may be superfluous initializations of values that are never read, but in a way that the compiler cannot prove. In cases where the overhead is significant (e.g. default initialization of large vectors), it is the responsibility of the programmer to structure code such that superfluous initializations can be avoided, for example through use of `reserve` and `push` or optional types.

Temporal Safety

Temporal safety is fundamentally a much harder problem than spatial safety: For spatial safety, it is possible to relatively cheaply instrument a program such that the safety precondition can be checked via an inexpensive runtime check (bounds check). In common cases it is straightforward to structure code such that bounds checks can be elided (e.g. using iterators).

In contrast, there is no straightforward way to establish the safety precondition for temporal safety of heap-allocated objects.

Pointers and allocations they point to, which in turn can themselves contain pointers, induce a directed (possibly cyclic) graph. The graph induced by the sequence of allocations and deallocations of an arbitrary program can get arbitrarily complex. It is in the general case impossible to infer properties of this graph based on static analysis of program code.

[9] Zeroing memory may not be sufficient because not all types may have a valid zero value.

When an allocation is freed, all that is at hand is the graph node corresponding to this allocation. There is no a-priori efficient (constant-time) way to determine whether there is still another inbound edge (i.e. another, still-reachable pointer into this allocation). Deallocating an allocation to which there are still inbound pointers implicitly invalidates those pointers (turns them into "dangling" pointers). A future dereferencing of such an invalid pointer would result in undefined behavior and a "use after free" bug.

Since the graph is directed, there is also no efficient (constant-time, or even linear in the number of in-bound pointers) way to find all still-reachable pointers into the about-to-be-deleted allocation. If available, this could be used to explicitly invalidate/null those pointers, or to defer deallocation until all inbound pointers are deleted from the graph.

Consequently, whenever a pointer is dereferenced, there is no efficient way to determine whether this operation constitutes undefined behavior because the pointer destination has already been freed.

There broadly are three ways to achieve rigorous temporal safety guarantees:

1. Ensure through compile-time checking that a pointer/reference cannot outlive the allocation it points to. For example, Rust implements this approach through the borrow checker and the exclusivity rule. This mode supports temporal safety of both heap and stack objects.
2. With runtime support, ensure that allocations are only deallocated when there are no valid pointers to it remaining.
3. With runtime support, ensure that pointers become invalid when the allocation they point to is deallocated, and raise a fault if such an invalid pointer is later dereferenced.

Several variations of 2 and 3 have been devised and they incur a non-trivial amount of runtime cost. Both reference counting and garbage collection provide the desired safety but can be expensive. Quarantining of deallocations is a strong mitigation, but does not fully guarantee safety and nevertheless carries an overhead. Memory tagging relies on specialized hardware and only provides probabilistic mitigation (unless combined with MarkUs [3, 1]).

In all cases, for temporal safety, there is no cheap (let alone free) lunch. Either developers structure and annotate code such that a compile-time checker (e.g, Rust borrow checker) can statically prove temporal safety, or we pay the runtime overhead to achieve safety or even partially mitigate these bugs.

Unfortunately, temporal safety issues remain a large fraction of memory safety issues, as indicated by a variety of reports:

- **Chrome**: 51% of high/critical memory safety vulnerabilities [6]
- **Android**: 20% of high/critical memory safety CVEs in 2022
- **Project Zero**: 33% of in-the-wild memory safety exploits [11]
- **Microsoft**: 32% of memory safety CVEs [5]
- **GWP-ASan**: finds 4x more UAFs than OOBs across multiple ecosystems [19]

Runtime Techniques and Tradeoffs

A wide range of runtime instrumentation techniques have been explored to address temporal safety, but they all come with challenging tradeoffs. They have to take into account concurrency when used in multi-threaded programs, and in many cases only mitigate these bugs without providing guaranteed safety.

- Reference counting, either to provide the correct lifetime or to detect and prevent incorrect lifetimes. Variations of this technique include `std:shared_ptr`, Rust's `Rc/Arc`, automatic reference counting in Swift or Objective-C, and Chrome's experiment with `DanglingPointerDetector`. Enforced exclusivity may be used with reference counting to reduce its overhead, but not eliminate it.
- Garbage-collected heaps. Enforced exclusivity may also be combined with GC to reduce overhead.
- Quarantining of deallocations, based on reference counting and allocation poisoning, as proposed by Chrome's `BackupRefPtr`, or combined with traversal and invalidation of pointers to quarantined deallocations, as proposed by MarkUs [1]. These approaches avoid interfering with destructor timing, but may provide only a partial mitigation rather than true temporal safety in some cases. They could be seen as variations of reference counting and garbage collection that do not interfere with destructor timing while preventing reallocation behind dangling pointers, but trade that off by introducing poison values (and resulting undefined behavior) into the runtime if accessed after being freed.
- Memory tagging labels pointers and allocated memory regions with one of a small set of tags (colors). When memory is deallocated and reallocated, it is re-colored according to a defined strategy. This implicitly invalidates remaining pointers which would still have the "old" color. In practice, the set of tags/colors is small (e.g. 16 in the case of ARM MTE [18]). Thus in most cases it provides probabilistic mitigation rather than true safety, as there is a non-trivial chance (e.g., 6.25%) that dangling pointers are not marked as invalid because they were randomly re-colored with the same color. MTE also carries significant run-time overhead. Memory tagging also speeds up MarkUs [1] and *Scan [3] approaches, providing strong temporal safety.

Production Language Safety Overview

This section provides a brief overview of the memory safety properties of current and near-future production languages at Google, and some languages that might play a role in a more distant future.

JVM languages (Java, Kotlin)

In Java and Kotlin, memory-unsafe code is clearly delineated and confined to use of the Java Native Interface (JNI). JDK standard libraries rely on a large number of native methods to invoke low-level system primitives and to use native libraries e.g. for image parsing. The latter have been affected by memory safety vulnerabilities (e.g. CESA-2006-004 , Sun Alert 1020226.1).

Java is a type-safe language. The JVM ensures spatial safety through runtime bounds checks and temporal safety based on a garbage-collected heap.

Java does not extend Safe Coding principles to concurrency: a well-typed program can have data races. However the JVM ensures that data races cannot violate memory safety. For example a data race can result in violation of higher-level invariants and exceptions being thrown, but cannot result in memory corruption.

Go

In Go, memory-unsafe code is clearly delineated and confined to code using package `unsafe` (with the exception of memory unsafety arising from data races, see below).

Go is a type-safe language. The Go compiler ensures that all values are initialized by default with their type's zero value , ensures spatial safety via run-time bounds checks, and temporal safety via a garbage-collected heap. Except via package `unsafe`, there is no facility to unsafely create pointers.

Go does not extend Safe Coding principles to concurrency: A well-typed Go program can have data races. Furthermore, data races can lead to violation of memory safety invariants[10].

Rust

In Rust, memory-unsafe code is clearly delineated and confined to `unsafe` blocks . Rust is a type-safe language. Safe Rust enforces that all values are initialized, and ensures spatial safety by adding bounds checks where necessary. Dereferencing a raw pointer is not allowed in safe Rust.

Rust is the only mature, production-ready language that provides temporal safety without run-time mechanisms such as garbage collection or universally-applied refcounting, for large classes of code. Rust provides temporal safety through compile-time checks on the lifetimes of variables and references.

The constraints imposed by the borrow checker preclude the implementation of certain structures, in particular those involving cyclic reference graphs. The Rust standard library includes APIs that allow such structures to be implemented safely, but with runtime overhead (based on reference counting).

In addition to memory safety, Rust's safe subset also guarantees data-race safety ("Fearless Concurrency "). Incidentally, data-race safety allows Rust to safely avoid unnecessary overhead when using runtime temporal safety mechanisms: Both `Rc` and `Arc` implement reference-counted pointers. However, `Rc`'s type precludes it from being shared across threads, so `Rc` can safely use a cheaper, non-atomic counter.

Carbon

Carbon is an experimental successor language to C++ with the explicit design goal to facilitate large-scale migration from existing C++ codebases. As of 2023, details of Carbon's safety strategy are still in flux[11]. Carbon 0.2 plans to introduce a safe subset that provides rigorous memory safety guarantees. However, it will need to retain an effective migration strategy for existing unsafe C++ code. Handling mixtures of unsafe and safe Carbon code will need similar guard rails as with mixtures of C++ and a safe language like Rust.

While we expect newly-written Carbon to be in its memory-safe subset, Carbon that originated from a migration from existing C++ will likely rely on unsafe Carbon constructs. We expect an automated, large-scale subsequent migration from unsafe to safe Carbon to be difficult and often impractical. Mitigation of memory safety risk in the remaining unsafe code will be based on hardening via build modes (similar to our handling of legacy C++ code). The hardened build mode will enable run-time mechanisms that attempt to prevent the exploitation of memory safety bugs.

A Safer C++

Given the large volume of pre-existing C++, we recognize that a transition to memory-safe languages might take decades, during which we will be developing and deploying code consisting of a mix of safe and unsafe languages. Consequently, we believe it is necessary to improve the safety of C++ (or its successor language if applicable).

While defining a rigorously memory safe C++ subset that is sufficiently ergonomic and maintainable remains an open research question, it might in principle be possible to define a

[10] `https://research.swtch.com/gorace`

[11] The Safety section of the Carbon Language design doc appears under "unfinished tales ".

subset of C++ that provides reasonably strong memory safety guarantees. C++ safety efforts should take an iterative and data-oriented approach to defining a safer C++ subset: identifying the top security and reliability risks, and deploying guarantees and mitigations with the highest impact and ROI.

A stepping stone for an incremental transition

A safer C++ subset would provide a stepping stone towards a transition to memory-safe languages. For example, enforcing definite initialization or disallowing pointer arithmetic in a C++ codebase will simplify an eventual migration to Rust or safe Carbon. Similarly, adding lifetimes to C++ will improve interoperability with Rust. Consequently, in addition to targeting top risks, C++ safety investments should prioritize the improvements that will also accelerate and simplify an incremental adoption of memory-safe languages.

In particular, safe, performant and ergonomic interoperability is a key ingredient for an incremental transition to memory safety. Both Android and Apple are following a transition strategy centered around interoperability, with Rust [10, 20] and Swift [16, 15] respectively.

For this, we need improved interoperability tooling, and improved support of mixed-language code bases in existing build tooling.[12] In particular, the existing production-quality interoperability tooling for C++/Rust assumes a narrow API surface. This has been sufficient for some ecosystems, like Android, but other ecosystems have additional requirements. Higher fidelity interoperability enables incremental adoption in additional ecosystems, as done for Swift already, and explored for Rust in Crubit. For Rust, there remains open questions, like how to guarantee that C++ code does not violate Rust code's exclusivity rule, which would create new forms of undefined behaviors.

By replacing components one-by-one, security improvements are delivered continuously instead of all at once at the end of a long rewrite. Note that a full rewrite may eventually be achieved with this incremental strategy, but without the risks typically associated with complete rewrites of large systems. Indeed, during that time, the system remains a single code base, continuously tested and shippable.

MTE

Memory Tagging [18] is a CPU feature, available in ARM v8.5a, that allows memory regions and pointers to be tagged with one of 16 tags. When enabled, dereferencing a pointer with a mis-matching tag raises a fault.

[12] Google has recently announced a $1M grant to support interop improvements.

Multiple security features can be built on top of MTE, for instance:

- **Use-after-free and out-of-bounds detection**. When memory is deallocated (or reallocated), it is randomly re-tagged. This implicitly invalidates remaining pointers, which would still have the "old" tag. In practice, the set of tags is small (16). Thus it provides probabilistic mitigation rather than true safety, as there is a non-trivial chance (6.25%) that dangling pointers are not marked as invalid (because they were randomly re-tagged with the same tag).
  - Similarly, this can also detect out-of-bounds bugs probabilistically.
  - This can deterministically detect inter-allocation linear overflows, assuming the allocator ensures that consecutive allocations never share the same tag.
  - It may be possible to build a deterministic heap use-after-free prevention on top of MTE using an additional GC-like scan like MarkUs.
- ***Sampled* use-after-free and out-of-bounds detection**. The same as above, but only on a fraction of allocations to reduce runtime overhead sufficiently for broad deployment.

  With sampled MTE, exploits are expected to succeed after a few attempts; attacks won't be stopped. However, failed attempts generate noise (i.e. MTE crashes) we can inspect.

Using those two techniques, MTE can result in:

- Bugs being found sooner in the SDLC. Unsampled MTE should be cheap enough to deploy in presubmit and canaries.
- More bugs being detected in production. Sampled MTE permits 3 orders of magnitude higher sampling rate compared to GWP-ASan at the same cost.
- Actionable crash reports. Synchronous MTE reports where the bug happened, instead of crashing due to hard-to-root-cause secondary effects of a bug. In addition, sampled MTE can be combined with heap instrumentation to provide bug reports with similar fidelity to GWP-ASan
- Improved reliability and security as those bugs get fixed.
- A decrease in exploits' ROI for attackers. Attackers either need to find additional vulnerabilities to deterministically bypass MTE, or risk detection.
  - Defender's reaction speed will depend on their ability to distinguish exploitation attempts from other MTE violations. Exploitation attempts may be able to hide in the noise of MTE violations happening organically.

- Even without the ability to distinguish exploitation attempts from organic MTE violations, MTE should reduce the exploitation window, i.e. how often and how long an attacker can reuse a given exploit. The faster MTE violations are fixed, the shorter the exploitation window will be, which decreases the ROI of exploits.
- This highlights the importance of fixing MTE violations promptly to achieve MTE's security potential. To do so without overwhelming developers, MTE should be combined with proactive work to reduce the volume of bugs.

Unsampled MTE may also be deployed as an exploit mitigation, deterministically protecting against 10%-15% of memory safety bugs (assuming no GC-like scan). However, due to non-trivial memory and runtime overhead, we expect production deployments to primarily be in small-footprint, but security-critical workloads.

Despite its limitations, we believe MTE is a promising path to decrease the volume of temporal safety bugs in large existing C++ code bases. There are currently no alternatives for C++ temporal safety that can be realistically deployed at scale.

CHERI

CHERI [21] is an intriguing research project that has the potential to provide rigorous memory safety guarantees for legacy C++ code (and perhaps Carbon in hardened mode), with minimal porting effort. CHERI temporal safety guarantees rely on quarantining of deallocated memory [9], and sweeping revocation, and it remains an open question whether the runtime overhead will be acceptable for production workloads.

Beyond memory safety, CHERI capabilities also enable additional interesting security mitigations, such as fine-grained sandboxing.

Conclusion

After 50 years, memory safety bugs remain some of the most stubborn and most dangerous software weaknesses. As one of the leading causes of vulnerabilities, they continue to result in significant security risk . It has become increasingly clear that memory safety is a necessary property of safe software. Consequently, we expect the industry to accelerate the ongoing shift towards memory safety in the coming decade. We are encouraged by the progress already made at Google, and at other large software manufacturers.

We believe that a Secure-by-Design approach is required for high assurance memory safety, which requires adoption of languages with rigorous memory safety guarantees. Given the long timeline involved in a transition to memory-safety languages, it is also necessary to improve the safety of existing C and C++ code bases to the extent possible, through the elimination of vulnerability classes.

Acknowledgments

We would like to thank our colleagues Chandler Carruth, Kostya Serebryany, Kinuko Yasuda, Jon McCune, Manuel Klimek and Mark Brand for their helpful comments and contributions to this paper.

References

[1] S. Ainsworth and T. M. Jones. MarkUs: Drop-in use-after-free prevention for low-level languages. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 578–591. IEEE, 2020.

[2] J. P. Anderson. Computer Security Technology Planning Study. Technical Report ESD-TR-73-51, U.S. Air Force Electronic Systems Division, 10 1972. URL https://seclab.cs.ucdavis.edu/projects/history/papers/ande72.pdf.

[3] M. L. Anton Bikineev and H. Payer. Retrofitting Temporal Memory Safety on C++. https://security.googleblog.com/2022/05/retrofitting-temporal-memory-safety-on-c.html, 2022. Accessed: 2023-12-06.

[4] Apple. Towards the next generation of XNU memory safety: kalloc_type. https://security.apple.com/blog/towards-the-next-generation-of-xnu-memory-safety/, 2022. Accessed: 2023-12-06.

[5] J. Bialek, K. Johnson, M. Miller, and T. Chen. Security analysis of memory tagging. https://raw.githubusercontent.com/microsoft/MSRC-Security-Research/master/papers/2020/Security%20analysis%20of%20memory%20tagging.pdf, 2020. Accessed: 2023-12-06.

[6] Chromium Security. Memory safety. https://www.chromium.org/home/chromium-security/memory-safety/. Accessed: 2023-12-06.

[7] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56:74–80, 2013. URL http://cacm.acm.org/magazines/2013/2/160173-the-tail-at-scale/fulltext.

[8] K. Deus, J. Galenson, B. Lau, I. Lozano, and A. S. . P. Team. Data driven security hardening in Android. https://security.googleblog.com/2021/01/data-driven-security-hardening-in.html, 2021. Accessed: 2023-12-06.

[9] N. Filardo, B. F. Gutstein, J. Woodruff, S. Ainsworth, L. Paul-Trifu, B. Davis, H. Xia, E. T. Napierala, A. Richardson, J. Baldwin, D. Chisnall, J. Clarke, K. Gudka, A. Joannou, A. Theodore Markettos, A. Mazzinghi, R. Norton, M. Roe, P. Sewell, S. Son, T. M. Jones, S. Moore, P. G. Neumann, and R. N. M. Watson. Cornucopia: Temporal safety for CHERI heaps. In *IEEE Symposium on Security and Privacy*. IEEE, May 2020.

[10] J. Galenson, M. Maurer, and A. Team. Rust/C++ interop in the Android platform. https://security.googleblog.com/2021/06/rust-c-interop-in-android-platform.html, 2021. Accessed: 2023-12-06.

[11] B. Hawkes and P. Zero. 0day "in the wild". https://googleprojectzero.blogspot.com/p/0day.html, 2019. Accessed: 2023-12-06.

[12] W. S. Humphrey. *Managing Technical People: Innovation, Teamwork, and the Software Process*. 1997. ISBN 9788177582710.

[13] C. Kern. Developer ecosystems for software safety. *ACM Queue*, 22(1), Feb 2024. doi: 10.1145/3648601. URL https://research.google/pubs/pub53103/.

[14] R. Levien. The Soundness Pledge. https://raphlinus.github.io/rust/2020/01/18/soundness-pledge.html, 2020. Accessed: 2023-12-06.

[15] K. Malawski. Swift as C++ successor in FoundationDB. *Strange Loop*, 2023. Accessed: 2023-12-06.

[16] J. McCall. Introducing a memory-safe successor language in large C++ code bases. *CppNow*, 2023. Accessed: 2023-12-06.

[17] MSRC. A proactive approach to more secure code. https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/, 2019. Accessed: 2023-12-06.

[18] K. Serebryany. ARM memory tagging extension and how it improves C/C++ memory safety. *Login USENIX Mag*, 44(5), 2019.

[19] K. Serebryany, C. Kennelly, M. Phillips, M. Denton, M. Elver, A. Potapenko, M. Morehouse, V. Tsyrklevich, C. Holler, J. Lettner, D. Kilzer, and L. Brandt. GWP-ASan: Sampling-based detection of memory-safety bugs in production, 2023.

[20] J. V. Stoep. Memory safe languages in Android 13. https://security.googleblog.com/2022/12/memory-safe-languages-in-android-13.html, 2022. Accessed: 2023-12-06.

[21] 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. *ACM SIGARCH Computer Architecture News*, 42(3):457–468, 2014.

Alex Rebert is a Senior Staff Software Engineer in Google's Security Foundation Team. His primary focus is on reducing memory safety risks.

Christoph Kern is a Principal Software Engineer in Google's Security Foundation Team. His primary focus is on developing scalable, principled approaches to software security.