*Google Security Engineering Technical Report*[1]
*November 7, 2025*

# Safe Coding

## Rigorous Modular Reasoning about Software Safety (Extended Version)

Christoph Kern
xtof@google.com

Many dangerous and persistent software vulnerabilities, including memory-safety violations and code injection, stem from a common root cause: developers unintentionally violating implicit safety preconditions when using common programming constructs. In large, complex systems, these preconditions often rely on nonlocal, whole-program invariants that are difficult for any single developer to reason about correctly and consistently. Traditional approaches such as developer education and reactive bug detection have proven insufficient to reduce these vulnerabilities to an acceptable level. Fundamentally, development environments make it too easy for well-intentioned developers to introduce subtle yet potentially catastrophic coding errors.

This article introduces *Safe Coding*, a collection of software design patterns and practices that cost-effectively provides a high degree of assurance against entire classes of such vulnerabilities. The core idea is to shift responsibility for safety from the individual developer to the programming language, libraries, and frameworks. Safe Coding achieves this by identifying *risky operations*—those with complex safety preconditions—and systematically eliminating their direct use in application code. Instead, risky operations must be encapsulated within *safe abstractions:* modules whose public APIs are safe to use by design and whose implementations take full responsibility for satisfying all internal safety preconditions. These design patterns have been successfully applied at Google, nearly eliminating classes of software vulnerabilities such as XSS (cross-site scripting) and SQL injection. Safe abstractions are also a core design principle in the Rust developer community, critical for achieving high-assurance memory safety despite the necessary presence of unsafe Rust in performance-critical and low-level systems code.

Safe Coding embodies a modular, compositional approach to building and reasoning about the safety of large, complex systems. Difficult and subtle reasoning about the safety of abstractions is localized to their implementations; the safety of risky operations within an abstraction must rely solely on assumptions supported by the abstraction's APIs and type signatures. Conversely, the composition of safe abstractions with *safe code* (i.e., code free of risky operations, which constitutes the vast majority of a program) is automatically verified by the implementation language's type checker.

While not a formal method itself, Safe Coding is grounded in principles and techniques from rigorous, formal software verification. It pragmatically adapts concepts such as function contracts and modular proofs for practical large-scale use by lifting safety preconditions into type invariants of custom data types within the chosen implementation language.

This article explores these technical and formal underpinnings, demonstrating how they enable cost-effective yet rigorous reasoning about software safety in very-large-scale industrial software development.

[1]This report is the extended version of an article in the Sep/Oct, 2025 issue of ACM Queue [17].

As a guide to readers of both versions, content unique to the extended version is marked with blue borders.

## Software Specifications, Correctness, and Safety

In a formal sense, a program or component is *correct*, relative to a specification, when:

1. It implements all behaviors required by the specification (for example, an API service responds to requests with a specified answer or an appropriate error).
2. Every possible behavior of the program or component is permitted by the specification.

The most rigorous approach to demonstrating program correctness relies on capturing both its specification and implementation in a formal, mathematical framework such as a program logic and constructing a mathematical proof that the implementation satisfies the specification [12].

For industrial-scale software, however, developing a comprehensive formal specification—let alone formally proving an implementation's correctness against the specification—is difficult and costly. Despite ongoing improvements in theory and tooling, formal methods still tend to be applied primarily to safety-, security- and reliability-critical components, if at all [10, 13].

Specifications for end-to-end, large-scale industrial software systems such as application servers and their web and mobile clients are usually presented in informal prose and, for example, stipulate a set of CUJs (critical user journeys) that describe key expected behaviors of the software. Such informal specifications are usually incomplete, and they neither comprehensively describe all required behaviors nor precisely delineate permitted from undesired behaviors.

## Software Failures

With respect to informal, partial specifications, software failures in practice fall into two broad categories:

1. The software's behavior directly conflicts with its (partial) specification—for example, it doesn't produce the specified result when exercised according to a CUJ.

2. The software exhibits behaviors that, while not necessarily contradicting an explicit specification, are nonetheless undesired. Such *erroneous behavior* often arises from unusual, unexpected, or malicious inputs that exercise execution paths the system's designers did not anticipate. Since informal specifications rarely delineate the full scope of permitted behavior, the line between correct and erroneous behavior is often implicit and underspecified.

Many practically relevant kinds of erroneous behaviors arise from the improper use of foundational components the software is based on, such as the programming language, library APIs, and software frameworks and platforms. Examples of such classes of erroneous behavior include:

- Undefined behavior (see sidebar, "Undefined Behavior"), including memory-safety violations, in unsafe languages such as C and C++ and in unsafe language fragments such as unsafe Rust.
- Runtime errors raised by certain APIs and operations when presented with invalid arguments—for example, division by zero or a bounds-checked, out-of-bounds array access.
- When untrusted, external inputs are passed to an API that parses or interprets strings according to a syntactic and semantic structure, allowing the attacker to control interpretation of a string in an unintended fashion; this often leads to so-called injection vulnerabilities such as XSS, SQL injection, shell injection, and path traversal.
- Deadlocks in multithreaded software architectures.

Some of these erroneous behaviors can lead to catastrophic software failures: For example, memory-safety violations and code injection can allow an attacker to execute arbitrary code with the privileges of the vulnerable software, in turn permitting them to completely evade the software's intended security policy. In many cases this represents a *much more severe impact* than many failures of the first category, such as not supporting a CUJ in certain edge cases. Indeed, the majority of the CWE Top 25 Most Dangerous Software Weaknesses [7] and Top 10 Known-Exploited Vulnerabilities [6] have their root causes in these common types of erroneous behaviors.

### Undefined Behavior

A particularly troublesome class of erroneous behavior lies in so-called *undefined behavior:* For various reasons (including performance—avoiding the need for runtime mechanisms to trap errors), the standards defining languages such as C and C++ declare that executing certain erroneous operations constitutes *undefined behavior*. Undefined behaviors in the ISO C standard include [14]:

- Division by zero
- Conversion to or from an integer type producing a value outside the range that can be represented
- An array subscript being out of range
- An object being referred to outside of its lifetime
- The execution of a program containing a data race
- And many more (Annex J.2 of the ISO C standard lists more than 200 items)

The language standard imposes no requirements whatsoever on a program that encounters undefined behavior—the program is permitted to continue executing and do absolutely anything [24]. Even worse, "allowed to do anything" applies not just *after* the erroneous operation—it can "time travel":[a]

1. The *entire* program execution is considered meaningless.
2. The compiler is allowed to make optimizations based on the assumption that no execution of the program will encounter undefined behavior.

Consider the following program as an example:

```
#include<limits.h>
#define LEN 42
int a[LEN] = {1,2,3};

int f(int base, int off) {
  if (base < 0 || off < 0) return -1;
  int idx = base + off;
```

```
    if (idx < 0) return -2;
    if (idx >= LEN) return -3;
    return a[idx];
}

int main() {
    return f(INT_MAX,2);
}
```

This program accesses an element of array `a` whose index is determined as the sum of a `base` index and an offset. The code appears to carefully check that both `base` and `off` are non-negative and then checks *again* that their sum `idx` is non-negative and in bounds of the array.

Per the C standard, however, signed integer overflow constitutes undefined behavior. The compiler is allowed to assume that the expression `base + off` does not overflow and can conclude from both operands being non-negative that `idx` is non-negative as well. An optimizing compiler is therefore permitted to eliminate the check whether `idx` is non-negative, which is redundant under this assumption.

Indeed, gcc 14.2 with option `-O1` does exactly that: The check does not appear in the resulting x86–64 assembly[b] (Clang produces similar code). In fact, the optimizing compiler inlines and constant folds `f` *after* removing the "redundant" check, resulting in assembly code for the `main` function that consists of a "hard-wired" out-of-bounds access of the array:

```
main:
    movabs  eax, DWORD PTR [a-8589934588]
    ret
```

[a] llvm.org/docs/UndefinedBehavior.html#time-travel

[b] See this example in Compiler Explorer: godbolt.org/z/4hqnj6TEW .

### Software Safety

A program is deemed *safe*—regarding a set of erroneous behaviors (e.g., division by zero, out-of-bounds access, undefined behaviors, or execution of untrusted inputs)—if no possible execution exhibits such behaviors. To be safe, a program must avoid these behaviors even when interacting with (and exposed to arbitrary inputs from) a malicious external environment.

Note that in the PL (programming languages) research community, safety is usually defined as the absence of any *untrapped* errors [4]. The definition of safety used in this article is more general—and relative to a set of erroneous behaviors relevant to the application domain; this allows us, for example, to consider erroneous behavior related to higher-level APIs (such as a SQL query API prone to code injection) and to include (uncaught) trapped errors in application domains where they are unacceptable (such as safety- and reliability-critical systems).

While the absence of erroneous behaviors is often not explicitly mentioned in informal software specifications and requirements documents, it is nevertheless a critical aspect of correctness: If a program can encounter undefined behavior and execute arbitrary, attacker-controlled code, it's hard to argue the program is correct in any meaningful sense. Furthermore, failures of the second category often form the root cause of the first: for example, undefined behavior encountered during a CUJ that results in a system crash and data corruption.

### Demonstrating Safety and Correctness

Since absence of erroneous behavior is such an important aspect of correctness, it is necessary to achieve a high degree of confidence that software is indeed safe and will not exhibit the types of erroneous behavior relevant in the application's domain, even when placed in an adversarial environment.

*Testing* can be quite effective at building confidence that software works as desired in expected, explicitly considered usage scenarios. For example, an integration test for a software system would confirm that it satisfies an informally specified CUJ such as "when a user completes the signup flow, a corresponding account record is created." It would also verify appropriate handling of expected error conditions, for example when an account record with the chosen user ID already exists.

In contrast, to rule out *erroneous behavior* with a high degree of confidence, one has to show that *all possible executions* avoid the undesired behavior, even for the most obscure edge cases and when exposed to adversarial inputs. Even with techniques such as coverage-guided fuzzing, testing is inherently limited in its ability to provide high confidence in statements quantified over all possible executions of a nontrivial program—as E.W. Dijkstra observed more than 50 years ago [8], "…testing can [show] the presence of bugs but never […] their absence."

Due to inherent limitations on the achievable confidence with approaches such as testing and fuzzing based on *observing behavior* over a *sample of concrete executions*, we need to instead consider approaches based on *reasoning about all possible behaviors* of the software in the *abstract*.

## Safe Coding

Safe Coding is a collection of software design practices and patterns that allow for *cost-effectively* achieving a high degree of confidence that a large, industrial-scale program will not exhibit relevant classes of erroneous behaviors, even when exposed to an adversarial environment.

Safe Coding offers a pragmatic approach to achieving high-assurance software safety, drawing inspiration from modular formal verification techniques. While structurally analogous to formal methods that rely on function contracts, Safe Coding adapts these concepts for large-scale industrial development. Instead of complex formal specifications, safety contracts are expressed through the type system and enforced by the language's standard type checker. Rigorous (though typically informal) expert-led reasoning about correctness remains necessary but is localized to the self-contained implementations of *safe abstractions*—which are usually a small and stable portion of the overall program. This design avoids the challenges of traditional static analysis, which often struggles to achieve sufficient soundness and precision for whole-program analysis of large complex codebases [19, 18]. By deliberately structuring code for easy analysis, the problem of verifying safety is largely reduced to scalable type checking. This positions Safe Coding as a semi-formal method, and our experience at Google over the past decade has shown it provides a favorable cost-assurance tradeoff.

## Reasoning about Safety

The root cause of many classes of erroneous behavior are operations or APIs that can be used safely only if certain conditions on the program state hold true at the moment of execution. An operation's *safety precondition* is defined as a predicate on program state that must hold immediately before its execution to ensure it does not exhibit erroneous behavior.

Consider a few common examples:

- Array access: Accessing `a[i]` has the safety precondition that `i` must be within the array's valid bounds.
- Pointer dereference: Dereferencing `*p` requires that `p` points to a valid, allocated memory location.
- SQL query execution: An API such as `db.Query(sql)` assumes `sql` is a safely constructed query, not one tainted by untrusted input.

Violating these preconditions can lead to some of the most severe software vulnerabilities (here, memory-safety violations and SQL injection). It is typically the *programmer's responsibility* to ensure that these preconditions are met. An operation with such a programmer-visible safety precondition is referred to as a *risky operation*. A block of code containing one or more risky operations is called *risky code*.[2]

In large and complex software systems, verifying that a risky operation's safety precondition is always met can be exceptionally difficult. The state that determines the precondition's validity might be established by code that is syntactically distant, perhaps in an entirely different module. For example,

[2]We use the term "risky code" instead of the more common "unsafe code" because it only *potentially* exhibits erroneous behavior, but is in fact safe if the programmer correctly ensures its safety preconditions.

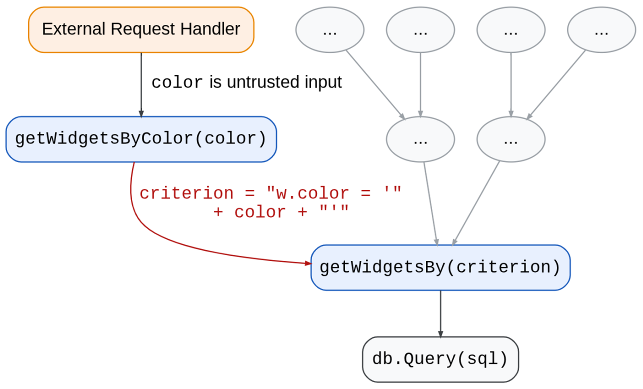

Figure 1: Transitively risky operations

the validity of a pointer passed to a function may depend on complex object lifecycle logic managed elsewhere in the application.

Consequently, reasoning about safety often degrades into a nonlocal, *whole-program* affair. Expecting developers to reason correctly about intricate, systemwide invariants for every risky operation is unrealistic and error prone. This reliance on brittle, manual, whole-program reasoning is a systemic root cause for the continued prevalence of many common classes of vulnerabilities. The key objective of the Safe Coding approach is to restructure programs to eliminate this root cause.

## Safe Abstractions

Since every use of a risky operation represents a potential hazard, it's clear that Safe Coding should aim to **minimize the use of risky operations** throughout a program's codebase.

Real-world programs, however, cannot be written without risky operations altogether: One obviously can't write a program that relies on a SQL database without calling SQL query APIs somewhere.

This creates a problem: Suppose we call `db.Query(sql)` within a function `getWidgetsBy(criterion)` whose implementation concatenates its argument `criterion` (intended to be a predicate in SQL syntax that specifies what kinds of widgets to return) into `sql`. In turn, `getWidgetsBy` is called many times throughout the application's codebase (see figure 1). In this scenario, the call to `db.Query` can exhibit erroneous behavior (resulting in a SQL injection vulnerability) if the parameter to `getWidgetsBy` incorporates untrusted program inputs without appropriate validation. That is, `getWidgetsBy` also has an implicit safety precondition and therefore is *itself* a risky operation. Since it's being used widely throughout the codebase, we're back to whole-program reasoning about safety.

To address this issue, the Safe Coding paradigm stipulates that all risky code should be encapsulated in a *safe abstraction*. This is a software module that has both of the following properties:

1. The module's public APIs have *no safety preconditions* that are the programmer's responsibility to ensure.
2. The module's implementation comprehensively *ensures* that the safety preconditions of all risky operations *within the module* are always satisfied.

That is, a safe abstraction must take full responsibility for its *internal* safety preconditions, without placing a burden on developers to ensure preconditions of its *external* API—if it did, it wouldn't be a *safe* abstraction.

However, for a given risky operation and its safety precondition, it is often not straightforward to come up with a precondition-free safe abstraction. The following sections discuss helpful design patterns for safe-abstraction APIs.

## Runtime Safety Checks

A straightforward way for a safe abstraction to ensure the safety precondition of a risky operation within its implementation is to explicitly check the precondition. This approach conforms to the stipulations for a safe abstraction: The operation's safety precondition trivially holds because it's ensured by a runtime check right then and there, and the check happens *within* the abstraction's implementation and is not the responsibility of its callers.

For example, in C and C++, the subscript operator in an expression like `a[i]` is a risky operation with the safety precondition that the index `i` is within the bounds of the array `a`.

In contrast, the implementations of the subscript operator of array, vector, and slice types in memory-safe languages such as Java, Go, and Rust include a runtime bounds check that verifies that the provided index is in bounds and otherwise raises an error.

The introduction of runtime safety checks often requires a change to the implementation to add sufficient metadata and bookkeeping information to check against. For example, a C array does not carry any information about the allocated bounds of the array (an array is essentially equivalent to a simple pointer). To enable bounds checks, vector types in safe languages need to carry the allocated size of that instance. For example, a conceptual, simplified Rust vector might look like the following:

```
pub struct Vec<T> {
    buf: *const T,
    len: usize,
}
impl<T> Vec<T> {
    pub fn new(len: usize) -> Vec<T> { ... }
    pub fn index(&self, i: usize) -> &T {
        if i < self.len {
            unsafe { &*self.buf.add(i) }
        }
        else { panic!("Index out of bounds!") }
    }
}
```

The constructor `Vec::new` must ensure that the raw pointer `buf` points to allocated memory of sufficient size to hold `len` elements of type `T`. The implementation of the `index` method then checks `i < self.len` before returning a reference to the vector's `i`th element at the appropriate offset from `buf`.

Dereferencing a raw pointer at an offset is a risky operation with the safety precondition that the offset is in bounds (as well as the pointer itself being valid, to which we will return later). In Rust, risky operations are made conspicuous by requiring that they are enclosed in an `unsafe` block. Because the bounds check immediately precedes the risky code, however, it's straightforward to establish that it is in fact safe; thus the `Vec` datatype is a safe abstraction around the raw pointer arithmetic required to implement array indexing.

This safety comes at the cost of a runtime check for every access. However, this overhead is often negligible in practice [5], and can be mitigated in several ways. Compilers can often prove that a bounds check is unnecessary and elide it, for example in an indexed loop over an in-bounds range. Furthermore, safe abstractions can provide alternative APIs that avoid repeated checks. For example, using an iterator to traverse a vector avoids bounds checks altogether, and is often more idiomatic and readable.

There are several API design choices for what to do when a runtime check fails:

1. At the most basic, one can simply **abort the program**. While this precludes recovery from the error condition within the program itself, it is simple and straightforward and can be appropriate in certain applications.

2. **Raise an exception**: Many programming languages provide a control flow mechanism to divert program execution to handler code in a transitive caller of the current function, unwinding in-between call-stack frames; for example, `throw` and `try/catch` in Java and `panic` and `recover` in the Go language. This approach is useful in programs that service a stream of independent external requests, such as network and web application servers: An exception thrown during the servicing of a particular (potentially malicious) request can be caught at the server framework level, allowing the server to continue to operate and respond to independent, benign requests.

3. A third option is to **return an error to the caller**. For example, Rust's `Vec` has a `get` method that returns its result wrapped in an `Option` type, returning a `None` value if the requested index was out of bounds. With this approach, the method always returns, and it's up to the caller to handle values that represent the "no entry at this index" condition.

Which approach to choose depends on the software's application domain, deployment environment and threat model (is it a mobile app, network server, or mission-critical embedded code?), as well as the programming language's idioms:

- For example, Rust discourages the use of `panic!`[3] for errors that could occur in normal operation and are expected to be handled by callers, and Google's C++ style guide discourages use of exceptions altogether[4]. In contrast, in Java it's common practice to use exceptions for such types of errors.

- While aborting the program avoids the clutter of error-handling code for conditions that are expected to never happen in normal operation, recovery in case an error does happen after all requires a program or system restart; this could result in data loss, unacceptable delays, or denial-of-service vulnerabilities. Relying on exceptions similarly avoids the clutter of error-handling code, but requires care to ensure that the code is exception-safe and leaves the program in a consistent state after an exception is raised and handled. In Rust, the behavior of `panic!` can be configured to abort the program, unwind the stack (which can be recovered from using `catch_unwind`), or to invoke a custom panic handler.

- In mission-critical (especially embedded) systems, it can be important to ensure that abnormal termination cannot happen under any circumstances. This can be accomplished by having the compiler (or a separate static analysis tool) prove that branches leading to a potential abnormal termination (e.g. `panic!`) can never be taken.[5]

Adding a runtime check does not actually remove the operation's precondition; rather, it is downgraded from a *safety* precondition to a regular, functional precondition: Violating the precondition no longer results in erroneous behavior and a potential vulnerability; instead, the failure is handled through well-defined behavior that is deemed tolerable in the given application domain.

[3] doc.rust-lang.org/book/ch09-03-to-panic-or-not-to-panic.html
[4] google.github.io/styleguide/cppguide.html#Exceptions
[5] Various Rust crates implement this approach, including crates.io/crates/no-panic and crates.io/crates/panic-never .

## Preconditions as Type invariants

Revisiting the `index` method of the `Vec` example from the previous section points out two details worth noting:

1. The bounds check explicitly tests only the upper bound but does not check that `i` is non-negative.
2. Beyond the bounds check, the code relies on further implicit preconditions for the risky pointer dereference `&*self.buf.add(i)` to be safe: The pointer `self.buf` must point to a valid allocated memory region of sufficient size that has been initialized with a valid representation of `len` elements of type `T`.

The non-negativity of `i` is actually guaranteed by its type: `usize` is an unsigned integer type in Rust, so its values are always non-negative. Thus, an explicit check such as `i >= 0` is unnecessary.

The validity of `self.buf` at the point of dereferencing `index` is more subtle. Its validity can be inferred by reasoning about the properties of the `Vec` type as a whole. Given that the constructor `Vec::new` allocates memory for the buffer and that no other method in the type's implementation frees this memory during the object's lifetime, one can conclude that the pointer remains valid. Since `buf` is a private field, no code outside the module can access and invalidate it—for example, by freeing the memory it points to.

Properties that are guaranteed to hold true for all instances of a type throughout their lifetimes, such as the continued validity of `self.buf` and the correctness of `self.len` in relation to the allocated buffer, are known as *type invariants*. When code is using a value of a particular type, it can rely on that type's invariants holding true.

For the `Vec<T>` type, the key invariants are:

1. `buf` points to a valid, allocated memory region.
2. This memory region is large enough to hold `len` elements of type `T`.
3. Each of the `len` elements in this memory region is initialized to a valid value of type `T`.

Confidence in these invariants is established by carefully inspecting all methods of `Vec`:

- The type's constructor `Vec::new` (not shown in the snippet) must establish the invariants when an instance is created: It is responsible for allocating and initializing sufficient memory and for initializing `buf` and `len` accordingly.

- All of the type's methods must maintain the invariants: Given the assumption that the invariants hold before a method is called, they must also hold after the method completes. In this case, none of `Vec`'s methods frees memory while the `Vec` instance is live (deallocation only occurs when the instance is destructed in an implementation of the `Drop` trait for `Vec`, omitted from the listing for brevity).

- Type encapsulation ensures that the invariants are not disturbed by code elsewhere in the program: `buf` is a private field, meaning no code outside the `Vec` module can access `buf` directly, for example, to free the memory it points to or change `buf` to an invalid pointer.

When `Vec::index` is invoked, Rust's type system ensures that `self` is a valid reference to an instance of `Vec`. Because of this, the `index` method can rely on `Vec`'s established type invariants. These invariants, in conjunction with the explicit `i < self.len` check, support the conclusion—without any additional runtime checks—that the safety preconditions of the pointer dereference operation `&*self.buf.add(i)` are always satisfied.

The example illustrates a **powerful design pattern for ensuring safety preconditions** of risky operations, which is applicable even when these preconditions are difficult, expensive, or infeasible to verify with a runtime check. The key elements of this approach are:

1. **Elevating safety preconditions into type invariants**, by designing safe abstractions that leverage types whose invariants capture the safety preconditions of relevant risky operations. Often this involves introducing custom *vocabulary types* or *wrapper types* specifically designed to carry the invariant.
2. Designing these types' APIs (constructors and methods, and related functionality such as builder APIs) and their implementations to **rigorously establish and maintain their invariants**. The implementations should be encapsulated so that client code cannot disturb the type invariant.
3. **Encapsulating risky operations within safe abstractions whose APIs *require* these invariant-carrying types**. The safe abstraction is designed such that the safety preconditions of any risky operations within its implementation are implied by the type invariants of its methods' receiver and argument types, in conjunction with runtime checks if applicable.

### Example: Temporal Memory Safety

A key challenge in languages such as C/C++ is ensuring temporal memory safety—that is, preventing access to memory that has already been deallocated (a "use-after-free" error). Raw pointers lack the metadata needed to perform a runtime check to determine if the memory they point to is still valid. The core difficulty is that pointers are not exclusive; code elsewhere in a program can hold another pointer to the same memory and free it, invalidating the original pointer without its knowledge. This makes reasoning about pointer validity a complex, whole-program analysis problem.

In the `Vec` example, this general problem for the internal `buf` pointer is addressed by establishing a strong type invariant: `buf` is effectively a unique, private pointer to memory whose lifetime is managed solely by the `Vec` instance itself. The rules of safe Rust ensure exclusive mutability of each instance and prevent code outside the module from prematurely freeing this memory.

For more general scenarios involving shared ownership of heap-allocated data, this pattern of creating safe abstractions with underlying type invariants is also key. There are several approaches with the common goal of ensuring that live pointers and references always point to a valid memory allocation:

- Reference counting is implemented by *managed pointer* types such as Rust's `Rc<T>`, which associate a counter with each managed allocation. Their implementations increment and decrement this count every time an instance of the managed pointer is copied or destroyed, respectively—thus maintaining the invariant that the reference count always equals the number of live copies of the managed pointer. The underlying memory is freed only when the last copy of the managed pointer is destructed, thus ensuring the invariant that live instances of the managed pointer type always refer to valid, allocated memory.
- In garbage-collected languages such as Go, Java, Python, and many others, the language's runtime takes full responsibility for managing heap-allocated memory and ensures that memory is freed only through a garbage-collection algorithm that identifies allocations that can no longer be reached via any live reference. This, in turn, ensures the global invariant that all live references in the program point to valid, allocated memory.
- Rust's type system incorporates lifetimes, and its compiler ensures statically (without incurring runtime overhead) that the lifetime of a referenced location always exceeds the lifetime of its references.

### Example: Code-Injection Safety

Code-injection vulnerabilities arise when an API that consumes strings that will be interpreted as code is passed values derived (at least in part) from untrustworthy inputs. If these inputs are not appropriately validated, encoded, or otherwise neutralized, an attacker can potentially control the meaning of these strings, adversarially influencing their interpretation and execution. This broad class of vulnerabilities includes well-known examples such as XSS, SQL injection, shell command injection, and LDAP (Lightweight Directory Access Protocol) injection. SQL injection is used here as a representative example of such a vulnerability that consistently ranks high in lists such as the CWE Top 25 [7].

APIs for interacting with SQL databases typically accept a string-typed value that is then parsed and evaluated as a SQL statement. For example, the Go standard library's `database/sql` package provides a `Query` method:

```
func (db *DB) Query(
    query string, args ...any) (*Rows, error)
```

This API carries an implicit safety precondition: The `query` string must represent a SQL statement that has been safely constructed, meaning it should originate from trustworthy (or appropriately neutralized) inputs or query fragments.

This flavor of safety precondition presents two key challenges:

1. This property depends not just on the *value* of the string, but also on its *provenance*—how it was constructed. A standard string type, being merely a sequence of characters, carries no metadata about its origin or how it was assembled. This means there is no straightforward way to check this precondition at runtime.

2. The phrase "has been safely constructed" is vague. To reason about safety rigorously, a more precise and enforceable definition is needed.

A widely recommended best practice for preventing SQL injection is to avoid direct concatenation of untrusted strings into a query. Instead, queries should be parameterized using placeholders, with actual values supplied via a mechanism known as parameter binding.

This practice corresponds to a stronger, more restrictive, but also more readily verifiable formulation of the safety precondition: The query must be a concatenation solely of developer-controlled strings, such as string literals embedded in the program's source code.

This precondition implies that no part of the query consists of external (and possibly untrustworthy) program inputs. It also ensures that code follows best practice: Since external inputs cannot be concatenated into a query, external parameters must be supplied via parameter binding. This, in turn, supports the high-confidence assertion that call sites of the SQL query API that adhere to this narrower safety precondition are not vulnerable to SQL injection attacks.

Consider the following code snippets. The first satisfies this stronger safety precondition and consequently adheres to the "parameter binding" best practice:

```
q1 := "SELECT y FROM table"
q1 += " WHERE x = ?"
rows, err := db.Query(q1, inputX)
```

In contrast, the following code violates the precondition and is vulnerable to injection if `inputX` is an attacker-controlled input:

```
q2 := "SELECT y FROM table"
q2 += " WHERE x = " + inputX
rows, err := db.Query(q2)
```

To design a safe abstraction for the SQL query API, there needs to be a way to programmatically distinguish between these two cases. With an API that consumes simple strings, this is difficult: When the combined string `q2` is passed to the `Query` API, there is no record in the representation of `q2` that part of the query originated from the untrusted string `inputX`.

This can be solved by introducing a simple wrapper type for strings, called `TrustedSqlString`, and elevating the required safety precondition as this type's invariant.

This invariant is upheld by carefully designing the constructors, factory functions, and builder APIs that are exclusively permitted to produce instances of `TrustedSqlString`. A key primitive is to restrict the inputs to these builders: For example, the `Append` method of a builder API for `TrustedSqlString` would be constrained to accept only compile-time-constant string expressions. This can be enforced in Go by using a module-private type alias for `string` or in languages such as Java through a simple custom static analysis check.[6] For a deeper discussion, see Chapter 12 of *Building Secure and Reliable Systems* [1].

With this wrapper type in place, it is straightforward to create a safe abstraction around the underlying database APIs. This abstraction simply wraps the original API but modifies its signatures to require queries to be of type `TrustedSqlString` instead of a plain `string`:

```
type SafeDB struct {
    db *sql.DB
}

func (sdb *SafeDB) Query(
    query TrustedSqlString, args ...any) (
    *sql.Rows, error) {
    return sdb.db.Query(query.String(), args)
}
```

Changing the injection sink API to require a `TrustedSqlString` lifts the implicit safety precondition into an explicit type contract. Any code that successfully

[6]For example, the `CompileTimeConstant` check in the Error Prone framework, errorprone.info/bugpattern/CompileTimeConstant.

compiles against this safe API is, by construction, not vulnerable to SQL injection, because the type system itself ensures that the query string was constructed safely. A similar approach can be used to prevent other classes of injection vulnerabilities such as XSS [15, 27].

## Putting it all Together: Modular, Compositional Reasoning

We now discuss in more detail how the key elements of the Safe Coding approach work together to ensure, at a high degree of confidence, that for all risky operations in a program their safety preconditions are met throughout all possible executions, even when subjected to adversarial inputs.

**Tool-Verified Safe Code**: A block or module constitutes *safe code* if it contains no risky operations. By definition, it cannot directly cause the specific erroneous behaviors associated with those operations.

As we will see in the following section, it is important to ensure that the vast majority of a program consists of safe code. This requires automated checks, integrated into the developer toolchain, that can distinguish safe from risky code and disallow risky operations by default.

In statically-typed languages, detecting calls to risky functions and methods (e.g., `java.sql.Statement.execute`) can be expressed as a straightforward predicate on the type-decorated AST, and is easily implemented in many static analysis frameworks.[7]

In Rust, the distinction between a safe and an unsafe (in our terminology, risky) language fragment is part of the language specification itself [25]. Safe Rust (which is the default) disallows use of risky language primitives such as dereferencing raw pointers, as well as calls to functions decorated with the `unsafe` keyword (which explicitly marks them as risky, i.e. requiring preconditions that are the caller's responsibility to ensure). Risky operations are only allowed in unsafe Rust, which must be demarcated with `unsafe` blocks.

**Modular Reasoning about Safe Abstractions**: As discussed earlier, a safe abstraction must take full responsibility for ensuring the safety preconditions of any risky operations within the abstraction, without making assumptions about *its callers* ensuring any preconditions of the abstraction's APIs—that's what makes the abstraction a safe one. However, it's impractical for an abstraction to do so in the context of a program that might arbitrarily misbehave. Rather, safe abstractions are permitted to make the following assumptions about the rest of the program:

[7] See, for example, the "restricted API" checker included in the Error Prone framework, errorprone.info/bugpattern/RestrictedApi .

1. The safe abstraction's API is used from safe, well-typed code (obviously, the API must be designed so it is *permitted* under the constraints that define safe code). In particular, usage of the safe abstraction is subject to the programming language's encapsulation rules; i.e., calling code cannot invoke private methods within the abstraction, nor access private fields of the abstraction's types.

2. When reasoning about safety preconditions within the safe abstraction, one may assume the invariants of types appearing in the abstraction's API signatures; i.e. calling code is expected to not disturb those invariants. Of course, these types' APIs and their implementations must be designed so as to uphold their type invariants, under the assumption they too are used in any arbitrary, but safe and well-typed code. Effectively, these types are part of the *Trusted Codebase* (TCB) that correctness of the safe abstraction depends on.

3. When a safe abstraction is called from code that is *not* itself safe (for example, the implementation of a different safe abstraction), that code nevertheless respects the safe abstraction's internals, and does not, for example, use a risky operation to disturb a type invariant.

How rigorously these assumptions can be upheld depends on the design of the underlying programming language, and in particular the strength of its encapsulation rules (some limitations are discussed later in the article).

Revisiting the invariants of the `Vec<T>` type, it is worth noting that ill-behaved unsafe Rust code could, for example, use raw pointers to access `Vec`'s internal fields and invalidate the invariant. That is, it is in fact reliant on the assumption that `Vec` is used only from safe Rust, or from unsafe Rust that refrains from using its "unsafe powers" to disturb `Vec`'s invariants.

**Automated Compositional Reasoning**: *If* a program is composed entirely from (a) safe code, and (b) verified, safe abstractions around risky code, *then the entire program is safe* and will not exhibit any of the classes of erroneous behavior under consideration. This whole-program property follows from the following observations:

1. Every risky operation in the entire program is encapsulated within a safe abstraction that ensures the operation's safety preconditions, and is able to do so under the assumption that the abstraction is used by well-typed, safe code.

2. Safe code cannot disturb type invariants that safe abstractions rely on. This in turn relies on the use of language-level encapsulation mechanisms to prevent code outside the module from mutating an abstraction's internal state.

3. It is expected and assumed that risky code (which, again, can only occur inside a safe abstraction), also respects the internals and invariants of *other* safe abstractions.

Reasoning about safe abstractions relies on invariants of types occurring in their API signatures, and the assumption that runtime values indeed have their declared types. Furthermore, safe abstractions rely on language-native encapsulation mechanisms to ensure that their internal invariants are not disturbed by outside code. Both properties are checked automatically by the programming language's compiler (or in some cases, runtime).

This means that the compositional reasoning pertaining to the safe portion of a program (typically, the vast majority of the code) is *fully automated through language-native tooling*.

As discussed in more detail below, in real-world development environments there are some limitations to the rigor of achievable assurance.

## Safe Coding in Developer Platforms and Ecosystems

The principle of compositional reasoning is powerful, but its validity in a real-world setting hinges on a critical assumption: that *all* uses of risky operations are indeed encapsulated in safe abstractions. Upholding this property across a large codebase maintained by many developers requires expert curation and disciplined practice, supported by robust tooling. Safe Coding works most effectively when it is integrated into developer platforms and incorporated into the processes and workflows of the entire developer ecosystem [16].

A primary challenge is ensuring that the boundary between safe and risky code is strictly maintained. It is all too easy for a developer to accidentally introduce a risky operation in a module that is not a formally reviewed safe abstraction. When this happens, the module's APIs may themselves become implicitly risky without being explicitly recognized as such. This allows unsafety to "bleed out" and undermines the foundation of high-confidence compositional reasoning about desired whole-program safety properties.

To prevent this, it is instrumental for the development environment to enforce that code is *safe by default*. This can be achieved through mandatory *conformance checks*, integrated into developer workflows, that disallow risky operations by default. The most effective approach is to make this part of the language and compiler itself, as Rust does by disallowing risky operations outside `unsafe` blocks, and via the `#![forbid(unsafe_code)]` attribute, which disallows unsafe Rust throughout an entire library. Alternatively, use of risky APIs and operations can be blocked through lightweight static analysis checks [2]. It is helpful to deploy conformance checks as part of compilation, or mandatory pre-submit checks. This treats conformance violations like any other build failure, making the policy clear and actionable for developers.

When a conformance check fails due to an attempted use of a risky operation, the error message should point developers to corresponding safe abstractions as an alternative. This provides developer guidance more effectively than point-in-time mandatory secure-coding training: Guidance is provided in context when needed and—because of the underlying automated conformance check—can't be forgotten.

A second, related challenge is that designing and implementing abstractions that are truly safe is difficult and often requires deep domain expertise. Expert attention to risky operations and the safety of their surrounding abstractions can be ensured by layering **toolchain-enforced expert reviews** on top of conformance checks that forbid risky operations: Conformance checks are augmented with an exception mechanism governed by a central allowlist that is managed by the appropriate team of experts. Before risky operations can be used in a code module (which then should be a safe abstraction), that module must be added to the allowlist, which gives the domain experts the opportunity to review the code *before* it is committed to the source repository. When reviewing abstractions for their safety, it is important to consider the entire module, not just the code immediately surrounding a risky operation (such as an `unsafe` block in Rust): Code anywhere in the module can be responsible for upholding invariants that are necessary to ensure the safety preconditions of risky operations. The allowlist mechanism can be implemented through custom tooling integrated into CI/CD (continuous integration/continuous delivery) workflows; in smaller repositories, a simple `grep`-based presubmit check can suffice.

Conformance checks and mandatory expert reviews intentionally place friction on the introduction of risky code into a codebase. For this to be sustainable, both in terms of impact on developer velocity and with respect to available expert bandwidth, it is crucial that developers almost never need to write custom, application-specific code involving risky operations. This means that the developer ecosystem and its platforms, frameworks, and standard libraries must provide a **comprehensive, expert-curated set of safe abstractions** that are sufficiently expressive and ergonomic to support the development of almost all application code typically encountered in a development organization. Some of these abstractions tend to be common and shareable across broad classes of applications (e.g., web or mobile apps); others might be more specifically tuned to the needs of a particular product development organization.

## Formal Foundations

The Safe Coding approach is not a comprehensive formal method, but it is deeply inspired by principles from formal software verification, particularly the idea of modular and compositional reasoning. Understanding this connection helps to illustrate the rigorous underpinnings of Safe Coding and explains its effectiveness in practice.

### Modular Verification and Function Contracts

Formal verification aims to prove a program's correctness with mathematical rigor. This is achieved by giving the program a formal semantics, which precisely describes how its execution changes the program's state. Properties of the program are expressed as logical predicates over this state. A common way to specify a program fragment's behavior is the Hoare Triple, written

$$\{P\} \quad S \quad \{Q\}$$

which asserts: *if* predicate $P$ (the precondition) holds before executing statement $S$, *then* predicate $Q$ (the postcondition) will hold afterward [12]. Proof rules for the programming language's constructs allow such assertions to be systematically reduced to proof obligations in the underlying logic, which can then be discharged using automated or interactive theorem provers.

Reasoning about an entire program at once is intractable for all but the smallest programs. To scale verification, the process must be modular. A common approach is to specify a formal contract for each function, consisting of pre- and post-conditions. The proof of a function's implementation can then proceed in isolation, assuming its own preconditions hold on entry and ensuring its postconditions hold on exit. When verifying a function's *caller*, the callee's contract is used to check correctness without needing to re-examine its implementation. This modular approach is the foundation of many verification tools such as Dafny [20], Frama-C [3], and VeriFast [22].

When a function contains a risky operation, its safety precondition becomes a proof obligation. For example, consider a function `f` that uses a risky operation `r` with safety precondition $S(x, y)$. The `assert` statement in the following pseudocode represents the *verification condition* that a formal verification tool would generate and require to be proven.

```
fun f(x: X, y: Y)
    requires P(x), Q(y) {
    s_1
    // ... statements preceding risky operation
    s_n
    // risky operation with safety precondition
    assert S(x, y)
    r(x,y)
    // ...
}
```

As a concrete example, `r` might be the signed integer addition `x + y` in C/C++, where signed arithmetic overflow constitutes undefined behavior and hence is subject to the safety precondition `x + y <= MAX_INT`.

To prove `f` correct, we assume its preconditions $P(x)$ and $Q(y)$ and must show that $S(x, y)$ holds at the point where `r` is called, that is, after executing $s_1;\ldots;s_n$ starting in a state that satisfies the preconditions. Formally, we need to prove the Hoare triple

$$\{P(x) \wedge Q(y)\} \quad s_1;\ldots;s_n \quad \{S(x, y)\}$$

for all $x$ and $y$. Importantly, this proof is entirely local, and only reasons about the implementation of `f` and its contract.

Similarly, to prove correctness of a function with a postcondition, we have to show that its return value satisfies the postcondition under all circumstances (in this case, by initializing a new value of type `X` so that it satisfies predicate $P$):

```
fun g() -> (res: X)
    ensures P(res) {
    // ...
    res := new X(/* ... */)
}
```

Callers of `f` are responsible for ensuring that its preconditions are satisfied at the point of call.

```
fun use(v: Y) {
    if (! Q(y)) { return }
    let u := g()
    f(u, v)
}
```

For verification purposes, the call to `g` is replaced by an assumption of its postcondition, and the call to `f` is replaced by an assertion of its preconditions:

```
fun use(v: Y) {
    if (! Q(v)) { return }
    assume P(u) // g's postcondition

    assert P(u) // f's preconditions, applied
                // to actual arguments
    assert Q(v)
}
```

In this example, the verification of `use` succeeds because `Q(v)` is established by an explicit runtime check, and `P(u)` is established by `g`'s postcondition.

## From Function Contracts to Type Contracts

Since `f` has preconditions necessary to ensure safety of the risky operation in its implementation, we need to explicitly reason about whether they hold at every use of `f`; this often fans out into reasoning about preconditions of transitive callers.

Relying on informal human reasoning to ensure pre- and postconditions (for example, captured as structured comments) for an entire industrial-size program is in our experience brittle and unreliable, even when individual contracts are not particularly complex. We *can* achieve high confidence in such reasoning by formalizing contracts as annotations verified by an automated prover or proof checker. However, this requires that the entire program is processed by a formal-methods tool, and that the entire development team is familiar with its use.

As discussed above, we can avoid the need to reason about safety preconditions throughout the entire program by applying Safe Coding design patterns and encapsulating risky operations in safe abstractions: Instead of annotating functions with explicit pre- and postconditions, we lift these properties into ***type invariants***. Risky operations are then wrapped in safe abstractions whose APIs express their contracts through these specialized types.

To refactor the example, a wrapper type `W` is introduced whose invariant is the predicate $P(x)$, and this type is used in the type signatures of `f` and `g`:

```
struct W {
    x:  X  // type invariant P(x)
}

fun f(w: W, y: Y) {
    // P(w.x) holds based on W's type invariant
    if (! Q(y)) { return }
    s_1; ... ; s_n

    // risky operation with safety
    // precondition S(x, y)
    r(w.x,y)
    // ...
}

fun g() -> (res: W) {
    x := new X
    // ... ensure P(x) holds
    return new W(x)
}
```

In this revised design, the module containing `f`, `g` and `W` forms a safe abstraction: The function `g` acts as a factory for `W`, responsible for establishing its type invariant (formally, $\forall_{w:W} P(w.x)$). The function `f` can now rely on this invariant for any argument of type `W`, thanks to encapsulation (the `x` field is private). The precondition $Q(y)$ is checked internally by `f`. The proof that these together ensure the safety precondition $S(x,y)$ is still required; as before it is fully localized within the implementation of the safe abstraction.

Importantly however, the functions `f` and `g` no longer have explicit pre- or postconditions in the form of logical predicates; their **contracts are expressed entirely by their parameter and return types**. This means that for a consumer of this abstraction, verification effectively reduces to standard type checking:

```
fun use(v: Y) {
    let w := g()
    f(w, v)
}
```

In a (hypothetical) programming language that combines a formally specified, sound type system with formally-verified reasoning about contracts and type invariants, applying Safe Coding design patterns centered around safe abstractions could formally prove absence of erroneous behavior for industrial-size programs.

Importantly, it could do so while relying *solely on type checking* for the vast majority of the program—only the implementations of safe abstractions require reasoning about pre- and postconditions, and hence the need to involve theorem provers and proof checkers.

This is a significant practical advantage: Development of code outside of safe abstractions (i.e., the vast majority of code) would not involve advanced formal methods tooling, and developers would only interact with the type system and its checker—a formal-methods tool that developers are familiar with (and generally don't think of as "formal methods" at all, despite its formal foundations).

## Practicalities and Soundness Gaps

In practice, we are constrained by the properties of the programming languages and tooling available in our developer ecosystems. These tend to have type systems that are not entirely sound, and typically don't come with formal verification tooling tightly integrated into their design. Furthermore, even if tooling were available, formally verifying the safety of all abstractions in a large program may be beyond the capacity of domain and verification experts in the organization.

Thus, the assurance provided by applications of Safe Coding in the real world will fall somewhat short of a fully-fledged formal method. The goal is to select a favorable point in the cost-versus-assurance spectrum and achieve a notion of "*adequate soundiness*" [19].

First and foremost, correctness of safe abstractions is often not formally proven, but rather relies on careful, informal reasoning. In many cases, this reasoning is straightforward and convincing. For example, the correctness of the `TrustedSqlString` abstraction rests on a simple inductive argument about string concatenation. When building safe abstractions around common data structures like balanced trees, we can rely on well-established correctness proofs of the underlying algorithms to reason about the safety of a concrete implementation that uses risky operations on raw pointers.

In our experience, this pragmatic approach tends to be effective for two reasons:

1. Abstractions are deliberately designed to be self-contained, so that their safety can be reasoned about *in isolation*, based solely on assumptions that follow from their APIs' type signatures. This often allows even informal (but rigorous) reasoning (akin to "paper and pencil proofs") to achieve a high degree of confidence.

2. The most prevalent defects historically do not stem from subtle bugs in well-vetted code in the implementations of abstractions. Instead, they arise from application code whose developer simply made a subtle mistake when directly using a lower-level risky operation. Safe Coding addresses this by far most prevalent root cause of defects by altogether eliminating the use of risky operations from most of the codebase.

In some scenarios, reasoning about the safety of abstractions with sufficient confidence may indeed call for full formal verification of their implementations. Theory and tooling to support such efforts is emerging, e.g. in the context of an initiative to verify safety of unsafe code in the Rust Standard Library [26].

A second crucial area where practical application deviates from formal ideals is in the enforcement of type invariants. The compositional reasoning at the heart of Safe Coding is built on a foundational assumption: that type invariants in fact do hold for every a value of a given type. These invariants are essential for discharging the proof obligations for risky operations within safe abstractions. However, the degree to which real-world programming languages can guarantee the integrity of these invariants varies, introducing another potential soundness gap.

Some production languages' type systems are inherently unsound, and do not guarantee that the value of a variable or reference has the declared type at all. This type-system unsoundness is sometimes deliberate, for example to enable optional static typing ("gradual typing") in TypeScript. Unsoundness also arises inherently from features of unsafe languages, such as void pointers and C-style casts.

A more subtle soundness gap arises from the discrepancy between the theoretical soundness of a language's type system and the practical "soundness of encapsulation" it provides for developer-defined data-types. Safe abstractions critically rely on encapsulation to protect their internal state and maintain their type invariants. However, many languages provide mechanisms that can circumvent these protections. For instance, even a statically-typed language with a sound type system like Java offers reflection APIs that permit application code to access and modify an abstraction's private fields, potentially disturbing its invariants. In dynamic languages, encapsulation can be weaker still, often primarily relying on naming conventions. On the other hand, some languages offer stronger guarantees; in Go, for example, unexported (private) fields of structs cannot be modified via reflection. Rust does not provide a native reflection mechanism at all (and modifying private fields of another module's types through low-level pointer and memory manipulation necessarily requires code annotated as `unsafe`).

In practice, the risk posed by these gaps depends on how common and idiomatic it is to use such features to break abstractions, and whether their use can be adequately curbed through coding style guidelines and automated code conformance or lint checks.

Another challenge lies in the potential discrepancy between a practically expressible type invariant and the more nuanced property required to truly ensure safety. For example, our `TrustedSqlString` abstraction enforces the type invariant that its values are constructed from string literals present in the program. At first sight, this seems to imply that its values are trustworthy, as each fragment of the SQL query originates from the program's own source code, which is considered trusted in typical threat models for server-side applications.

However, a determined programmer could construct a function that takes an arbitrary, untrusted string and builds a `TrustedSqlString` from it, character by character, by looking up each character in a map of literals. While this code would successfully compile against the safe API, it effectively "launders" untrusted strings into values of `TrustedSqlString`, and thus reintroduces the potential for SQL injection vulnerabilities the abstraction was designed to prevent.

In our experience at Google, this is not a significant issue in practice; such code is "sufficiently obviously inappropriate" that trusted developers are very unlikely to write it. Nevertheless, this example highlights an important aspect of the threat model: Safe Coding is designed to help well-intentioned developers avoid unintentional errors. It is not intended to provide assurance in environments where code authors themselves must be assumed to be untrusted or actively malicious, such as in a hosted cloud runtime.

Finally, it is worth noting that even fully-fledged formal methods do not provide absolute assurance. Formal proofs are only as strong as the assumptions they are built on. In practice, these assumptions can be flawed, leading to gaps between the formalized model and the real-world system it represents [21, 9].

## Safe Coding in Practice

The principles of Safe Coding have been applied successfully in large-scale, real-world software development environments, demonstrating both their effectiveness in drastically reducing entire classes of defects and their cost effectiveness over time.

At Google, the application of Safe Coding principles has led to the **near elimination of several classes of security vulnerabilities** that consistently rank among the most dangerous software weaknesses [11]. For example, by integrating Safe Coding practices into its core web application frameworks, Google has virtually eradicated XSS from hundreds of user-facing web applications. These frameworks use mechanisms such as strictly auto-escaping template systems [15] and the enforcement of Trusted Types API [27], which ensure that untrusted data cannot be inadvertently interpreted as code in a browser context.

The results have been striking: Over the past several years, hundreds of complex web applications built on these hardened frameworks have averaged less than one XSS vulnerability report per year in total. Products such as Google Photos, developed from the outset on these secure-by-design frameworks, have had no XSS vulnerabilities reported in their entire lifetime. Similarly, SQL injection vulnerabilities have been systematically addressed by redesigning database query APIs to rely on the `TrustedSqlString` type invariant. The application of Safe Coding to database APIs has resulted in no reported SQL injection vulnerabilities for more than a decade across the hundreds of applications using these secure database interfaces.

A cornerstone of Safe Coding in practice is the adoption of memory-safe languages such as Rust, Java, and Go. While memory-safety violations constitute the majority of severe security defects in code written in unsafe languages such as C/C++, they are exceedingly rare in software predominantly written in memory-safe languages.

The experience of the Android team provides a powerful case study. Following a strategic shift around 2019 to prioritize memory-safe languages for new development, Android has seen a dramatic and disproportionate reduction in memory-safety vulnerabilities. The percentage of Android's total vulnerabilities attributed to memory-safety issues fell from 76 percent in 2019 to just 24 percent in 2024. This decline occurred even while the bulk of the existing codebase remained in memory-unsafe C/C++, illustrating that it can be highly effective to focus preventive measures on areas of new and active development [23].

While there is an upfront investment in developing a mature ecosystem of safe abstractions and the tooling to ensure their consistent use, this approach is highly cost effective in the long run. The cost of building and maintaining a central set of safe libraries and frameworks is amortized across hundreds of applications.

At Google, a small team of security experts maintains the Safe Coding libraries and conformance checks that support thousands of application developers. This creates a powerful force multiplier, freeing application teams from the need to become security experts and allowing them to focus on product features. It also shifts from a reactive model with ongoing, per-project costs borne by security and product teams (testing, security reviews, incident response, remediation) to a proactive investment in prevention, providing continuous assurance that entire classes of vulnerabilities will not be introduced in the first place [16].

## Acknowledgements

The ideas presented in this article have been developed and refined over more than a decade by members of the Information Security Engineering team at Google. The framing of Safe Coding around safe abstractions has been influenced by the work and philosophy of the Rust community. I am grateful for the insightful feedback from reviewers of earlier drafts of this article.

## References

[1] H. Adkins, B. Beyer, P. Blankinship, P. Lewandowski, A. Oprea, and A. Stubblefield. *Building secure and reliable systems: Best practices for designing, implementing, and maintaining systems*. O'Reilly Media, 2020. Available at https://sre.google/books/building-secure-reliable-systems/.

[2] E. Aftandilian, R. Sauciuc, S. Priya, and S. Krishnan. Building useful program analysis tools using an extensible Java compiler. In *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, pages 14–23, 2012. doi: 10.1109/SCAM.2012.28.

[3] P. Baudin, F. Bobot, D. Bühler, L. Correnson, F. Kirchner, N. Kosmatov, A. Maroneze, V. Perrelle, V. Prevosto, J. Signoles, and N. Williams. The dogged pursuit of bug-free C programs: the Frama-C software analysis platform. *Commun. ACM*, 64(8):56–68, July 2021. URL https://doi.org/10.1145/3470569.

[4] L. Cardelli. Type systems. In A. B. Tucker, editor, *Computer Science Handbook*, chapter 97. Chapman & Hall/CRC, 2nd edition, 2004. URL https://dl.acm.org/doi/book/10.5555/1027496. Available at http://lucacardelli.name/Papers/TypeSystems.pdf.

[5] C. Carruth. Story-time: C++, bounds checking, performance, and compilers, 2024. URL https://chandlerc.blog/posts/2024/11/story-time-bounds-checking/.

[6] CWE. Top 10 known-exploited vulnerabilities, 2024. URL https://cwe.mitre.org/top25/archive/2024/2024_kev_list.html.

[7] CWE. Top 25 most dangerous software weaknesses, 2024. URL https://cwe.mitre.org/top25/.

[8] E. W. Dijkstra. On the reliability of programs. In K. R. Apt and T. Hoare, editors, *Edsger Wybe Dijkstra: His Life, Work, and Legacy*. ACM Books,

2022. URL https://doi.org/10.1145/3544585.3544608. Original text dated 1970.

[9] P. Fonseca, K. Zhang, X. Wang, and A. Krishnamurthy. An empirical study on the correctness of formally verified distributed systems. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 328–343, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349383. URL https://doi.org/10.1145/3064176.3064183.

[10] H. Garavel, M. H. ter Beek, and J. van de Pol. The 2020 expert survey on Formal Methods. In M. H. ter Beek and D. Ničković, editors, *Formal Methods for Industrial Critical Systems*, volume 12327 of *Lecture Notes in Computer Science*. Springer, Cham, 2020. URL https://doi.org/10.1007/978-3-030-58298-2_1.

[11] Google. An overview of Google's commitment to Secure by Design, 2024. URL https://publicpolicy.google/resources/google_commitment_secure_by_design_overview.pdf.

[12] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, Oct. 1969. URL https://doi.org/10.1145/363235.363259.

[13] L. Huang, S. Ebersold, A. Kogtenkov, B. Meyer, and Y. Liu. Lessons from formally verified deployed software systems (extended version). 2023. URL https://doi.org/10.48550/arXiv.2301.02206.

[14] International Organization for Standardization. ISO/IEC 9899:2024: Information technology — Programming languages — C, 2024.

[15] C. Kern. Securing the tangled Web. *Commun. ACM*, 57(9):38–47, Sept. 2014. URL https://doi.org/10.1145/2643134.

[16] C. Kern. Developer ecosystems for software safety. *Commun. ACM*, 67 (6):52–60, May 2024. URL https://doi.org/10.1145/3651621.

[17] C. Kern. Safe Coding: Rigorous modular reasoning about software safety. *ACM Queue*, 23(5), 2025. URL https://doi.org/10.1145/3773098.

[18] S. Lipp, S. Banescu, and A. Pretschner. An empirical study on the effectiveness of static C code analyzers for vulnerability detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2022)*, pages 544–555, New York, NY, USA, 2022. Association for Computing Machinery. URL https://doi.org/10.1145/3533767.3534380.

[19] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis. In defense of Soundiness: A manifesto. *Commun. ACM*, 58(2):44–46, Feb. 2015. URL https://doi.acm.org/10.1145/2644805.

[20] K. R. M. Leino. Accessible Software Verification with Dafny . *IEEE Software*, 34(06):94–97, Nov. 2017. URL https://doi.ieeecomputersociety.org/10.1109/MS.2017.4121212.

[21] T. Murray and P. van Oorschot. Bp: Formal proofs, the fine print and side effects. In *2018 IEEE Cybersecurity Development (SecDev)*, pages 1–10, 2018. URL https://doi.org/10.1109/SecDev.2018.00009.

[22] P. Philippaerts, J. T. Mühlberg, W. Penninckx, J. Smans, B. Jacobs, and F. Piessens. Software verification with VeriFast: Industrial case studies. *Science of Computer Programming*, 82:77–97, 2014. URL https://doi.org/10.1016/j.scico.2013.01.006. Special Issue on Automated Verification of Critical Systems (AVoCS'11).

[23] A. Rebert and J. V. Stoep. A practical guide to transitioning to MSLs. *ACM Queue*, 23(5), 2025. URL https://doi.org/10.1145/3773096.

[24] J. Regehr. A guide to undefined behavior in C and C++, 2010. URL https://blog.regehr.org/archives/213.

[25] Rust Authors. The Rustonomicon: Meet safe and unsafe. URL https://doc.rust-lang.org/nomicon/meet-safe-and-unsafe.html.

[26] Rust Foundation Team. Rust Foundation collaborates with AWS initiative to verify Rust standard libraries, 2024. URL https://rustfoundation.org/media/rust-foundation-collaborates-with-aws-initiative-to-verify-rust-standard-libraries/.

[27] P. Wang, B. Á. Guðmundsson, and K. Kotowicz. Adopting Trusted Types in production web frameworks to prevent DOM-based cross-site scripting: A case study. In *2021 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 60–73. IEEE, 2021. URL https://doi.org/10.1109/EuroSPW54576.2021.00013.

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