Practical Security in Production

HARDENING THE C++ STANDARD LIBRARY AT MASSIVE SCALE

LOUIS DIONNE,
ALEX REBERT,
MAX SHAVRICK, AND
KONSTANTIN VARLAMOV

Over the past few years there has been a lot of talk about memory-safety vulnerabilities, and rightly so—attackers continue to take advantage of them to achieve their objectives. Aside from security, memory unsafety can be the cause of reliability issues and is notoriously expensive to debug. Considering the billions of lines of C++ code in production today, we need to do what we can to make C++ measurably safer over the next few years with as low of an adoption barrier as possible.

In 2019, Alex Gaynor, a security expert and one of the leading voices in memory safety, wrote a piece titled "Modern C++ Won't Save Us," where he gave examples of foundational types such as `std::optional` that were unsafe in the idiomatic use cases. What happens when these unsafe types are used beyond their contract? Well, you guessed it: undefined behavior. The `std::optional` type isn't the only one to behave like this. If you look at how this compares with modern languages, you can see that C++ is the outlier.

So, what's to be done? Possibly one of the best places to start today is by improving our standard libraries. They provide the baseline "vocabulary types" for developers—

and if they're not safe, it will be tough to build safety around them. The `std::optional` type is only one of many vocabulary types in the C++ Standard Library that aren't safe by default today. Given the current state, it seems mostly clear that the first step should be hardening our standard library, and in our case, this was LLVM's libc++.

THE LIMITS OF DEBUG-ONLY MODES

The idea of a *debug mode* with extra checks is not new—every major implementation of the C++ Standard Library has had one. Historically, however, they suffered from several shortcomings, notably including ABI (application binary interface) compatibility issues; but perhaps the most important shortcoming was reflected directly in its name: **debug** mode was seen as a bug-finding tool to be used in a testing environment.

There seemed to be a common sentiment reflected in other implementations of the time as well: that as long as the code is well-tested, checks in release builds were unnecessary and would introduce an unacceptable performance hit. This led to feature bloat (since performance was not a priority) and made "infeasible in release mode" a self-fulfilling prophecy.

With experience, we can confidently say that this test-only approach is not sufficient to prevent bugs in the real world. Projects with extensive test suites that make use of a multitude of bug-finding tools still suffer from costly bugs and vulnerabilities when deployed to production. Even extensive fuzzing can't provide a complete guarantee, as high-profile vulnerabilities like the one in libwebp have demonstrated that bugs can lurk even in heavily fuzzed

code. Real-world use exposes corner cases that no test suite, no matter how exhaustive, can feasibly cover—not to mention attackers with a special aptitude for finding or triggering these exact sorts of cases. Though well-intentioned, a debug-only approach ends up being of limited use during development and of no use during deployment, and, as a result, sees little adoption.

Real-world use exposes corner cases that no test suite, no matter how exhaustive, can feasibly cover— not to mention attackers with a special aptitude for finding or triggering these exact sorts of cases.

THE CASE FOR PRODUCTION HARDENING

The alternative, therefore, is to enable hardening universally in production. While testing is vital, it cannot replicate the exact conditions, subtle timings, or adversarial pressures of a live environment. Many latent bugs manifest only under production traffic or adversarial inputs. To provide safety guarantees, *checks must be active where the code actually runs.*

This stance is often met with immediate skepticism based on two reasonable and deeply ingrained fears: (1) destabilizing services with crashes, and (2) unacceptable performance overhead. For hardening to be viable, both must be addressed.

First, let's address stability. A crash from a detected memory-safety bug is not a new failure. It is the early, safe, and high-fidelity detection of a failure that was already present and silently undermining the system. The alternative to a "loud crash" is not a healthy system; it is a silently corrupted one that will fail later in a more complex, damaging, and less understandable way.

Adopting this *fail-stop* policy—terminating the process immediately upon detecting an unrecoverable memory violation—has been shown to be superior: It is more secure,

makes bugs far easier to detect and fix, and ultimately leads to more reliable systems.

The second fear, performance, is equally critical. This is precisely where historical debug modes failed. Because these modes were not intended for release builds, performance was not a design constraint. Performance must be treated as a core design requirement, not an afterthought. As shown in the rest of this article, the combination of careful design and recent compiler optimization improvements made hardening affordable enough to be enabled at scale.

DESIGNING LIBC++ HARDENING FOR PRODUCTION

The affordability of production hardening was not an accident; it was the result of a long, deliberate evolution in design. The initial push at Apple began by exploring domain-specific classes that provided improved bounds safety, but it quickly became clear that requiring code modification to migrate to nonstandard utilities lacking a rich ecosystem made adoption an uphill battle. For example, introducing a span-like class that provides improved bounds safety is a daunting task if it does not interoperate seamlessly with the plethora of algorithms provided by the standard library.

We then noticed that many C++ Standard Library data structures already had enough information to ensure bounds safety (sometimes in limited ways) and that simply hardening those accesses was the *constriction point* that would allow improving bounds safety in a large amount of existing code.

This led to successive iterations of hardening in libc++,

culminating in the current design in LLVM 18, which is built on a set of core principles that make it practical for production use.

A safety spectrum, not a switch

Early experiences with an on/off "safe mode" (introduced in LLVM 15) were encouraging. The key difference from previous debug modes was that safe mode was intended to be used in production. This imposed significant constraints on the design, as features could no longer be enabled without serious consideration of their performance impact.

A one-size-fits-all approach is too blunt; developers need to choose the right security-versus-performance tradeoff for their environment.

As we gained experience with deployment of safe mode, new requirements surfaced. While deployment experience showed this to be a particularly good fit for some projects with adoption in Safari and Chromium, it quickly became clear that there were environments for which safe mode was too expensive. A one-size-fits-all approach is too blunt; developers need to choose the right security-versus-performance tradeoff for their environment.

The current incarnation of hardening in libc++, first released in LLVM 18, reflects this by offering four hardening modes. The two most important are intended for production:

- `Fast` mode is passionately minimalistic and enables only those hardening checks that are security-critical and can be performed with low overhead, usually in constant time; in practice, this almost exclusively means checking for out-of-bounds accesses (and thus improving spatial memory safety). It is a very lucky coincidence that some of the most valuable checks also happen to be some of the cheapest!

- `Extensive` mode enables all available library checks that can be performed with relatively low performance overhead, including those that lead to undefined behavior but aren't security-critical.

Enabling a hardening mode is a matter of passing the right compiler flags and rebuilding the application. If the application doesn't violate library preconditions, the code should not require any changes.

The idea is that almost all applications should be able to allow `fast mode`, while more security-conscious applications might opt into `extensive` mode. Additionally, there is a `none` mode (no hardening checks—that is, the status quo) and a (new, unrelated to legacy) `debug` mode; `debug` mode contains more expensive checks, although it still aims to never affect the big-O complexity of algorithms. Each subsequent mode is a superset of the previous one, both in terms of the number of checks and the performance overhead (`none` -> `fast` -> `extensive` -> `debug`).

ABI compatibility, if needed

A critical lesson from past experience is that hardening must be orthogonal to the ABI. While some attractive checks would require an ABI break (e.g., storing bounds information in iterators), tying safety to the ABI can make it unusable in many production environments. An application cannot unilaterally declare a new ABI that differs from the rest of the platform with which it must link. Platforms that can allow ABI breaks might use libc++ ABI flags that enable additional hardening checks (such as hardened iterators); when an application selects a hardening mode, it enables all checks possible in the current ABI configuration.

Partial enablement options

For real-world adoption, developers must be able to selectively opt out of hardening in the most performance-critical parts of their code. The practical choice is often between disabling checks for one percent of the code or not enabling hardening at all.

Thus, an important requirement for hardening in libc++ is that it can be turned on and off on a per-TU (translation unit) basis. This is achieved using Itanium ABI tags—the hardened mode that is in effect in a given TU is encoded in the tag that is attached to all library functions and affects their mangled names. Thus, a call to a vector's subscript operator would resolve to two distinct functions if one TU calls it under `fast mode` and another under `none` mode (same for all other modes) so the ODR (One Definition Rule) is not violated.

Efficient and customizable failures

By default, when a check fails, libc++ terminates the program with a trap instruction, which is the most secure and lowest-overhead option. This behavior can be completely overridden by the vendor of a given platform when building the library by providing a custom header with the desired implementation of the termination handler. This is different from the weak definition approach used in safe mode. While more flexible, the previous approach resulted in binary "bloat" since the linker in the general case cannot inline the function call. In practice, most applications don't need to override the termination handler and, in line with the general C++ principle, should not pay for what they don't use.

DEPLOYING HARDENING AT SCALE

While a flexible design is essential, its true value is proven only by deploying it across a large and performance-critical codebase. At Google, this meant rolling out libc++ hardening across hundreds of millions of lines of C++ code, providing valuable practical insights that go beyond theoretical benefits. While hardening has also been adopted in various open-source codebases (e.g., Google Chrome, Apple's WebKit) and in a variety of other security-critical projects at Apple, the best documented case study is that of Google's adoption of the feature across its server-side production systems, to be discussed next.

Phase 1: Enabling hardening in tests

The journey to production began more than a year before the final rollout with a large-scale cleanup effort to enable hardening in pre-production builds. The first attempt to enable the checks in our unit and integration tests broke more than 1,000 tests.

Fixing this required a concerted effort, including crowdsourcing fixes from interested engineers using their 20 percent time, resulting in hundreds of patches across Google's monorepo. This was essential to establish a "green" baseline, ensuring new code submitted with hardening violations would fail in CI (continuous integration), preventing backsliding.

Once the tests passed, the hardened runtime was enabled in pre-production environments (canary, staging). This allowed developers time to learn about the hardening, fix newly surfaced issues, and build confidence.

This process also drove improvements to libc++

hardening itself. For example, the original two percent binary size increase was a blocker in certain environments, so a non-verbose mode was added with a much smaller footprint, reducing the binary size overhead to less than 0.5 percent.

This phase demonstrated the sheer volume of latent issues and reinforced the necessity of the project, setting the stage for the move to production.

Phase 2: Data-driven consensus building

With a clean test baseline, the next hurdle was proving that production hardening was viable for a fleetwide deployment. This required performance measurement, production pilots, and consensus building.

The primary concern was performance. To address this, key services were benchmarked to understand libc++ hardening's performance characteristics. This is where we identified that profile-guided optimization allowed us to keep hardening overhead low.

Armed with early performance data, we ran pilots with early adopters, including large, high-traffic services. Those pilots provided real-world evidence that systems remained stable and that the crashes were manageable and highly valuable for debugging. It also provided real-world performance data to estimate the cost of a fleetwide rollout, relying on Google's continuous profiling infrastructure.[1]

These success stories and production data were key to building broad consensus for a default-on rollout. We made the case to Google's engineering leadership based on four arguments:

- Demonstrated affordability of hardening
- Clear security benefits
- Immediate impact on debuggability
- Long-term improvements to reliability

Ultimately, securing buy-in across a large engineering organization was the most time-consuming phase of the project, a reflection not on the technology, but on the diligence required for a change at this scale.

Ultimately, securing buy-in across a large engineering organization was the most time-consuming phase of the project, a reflection not on the technology, but on the diligence required for a change at this scale.

Phase 3: Production rollout

By this point, more than 100 pilots were running in production, ranging from security-critical services to high-performance parts of the Search backend. The final phase was the full production rollout, activating hardening by default across the fleet. This began the most critical stage of the project: uncovering and fixing the latent bugs that manifest only under the unique pressures of a live production environment.

Hardened services were progressively rolled out to production, and, as expected, the safety checks began to fire. Our hypothesis was that these new deterministic crashes would not be creating new instability but, rather, mostly displacing more dangerous and opaque memory-corruption errors. Live monitoring during the rollout confirmed this theory: As the new assertion failures appeared, the baseline of segmentation fault crashes began to recede.

We staffed a centralized response to rapidly diagnose and fix underlying issues, sending hundreds of patches across our monorepo. This process purged a significant volume of latent bugs. In some rare cases, we temporarily

opted out specific workloads from hardening while we worked on a fix.

Performance

The most significant concern—performance—proved largely unfounded in practice. Across Google's server-side C++ codebase, the average production performance overhead of enabling libc++ hardening was measured at a remarkably low 0.3 percent.

This affordability wasn't accidental. As Chandler Carruth, Distinguished Engineer and overall C++ language lead at Google, detailed, several factors likely converged:

- *Efficient check implementation.* The hardening checks in libc++ were carefully implemented to be lightweight.
- *Compiler optimizations.* Modern compilers such as Clang/LLVM became adept at optimizing checks, eliminating redundant ones within loops or proven code paths.
- *Cross-pollination.* LLVM's optimization capabilities for these kinds of checks have significantly improved over the years, partly driven by the needs of memory-safe languages such as Swift and Rust, which rely heavily on runtime checks and use LLVM as a compiler backend. C++ benefited indirectly from this broader ecosystem investment.
- *PGO (profile-guided optimization).* This was critical. High-quality PGO data allows the compiler to identify hot paths and often move checks out of the hot paths, minimizing impact on latency-sensitive code.

We anticipated that some critical code paths would be too sensitive for any overhead. To address this, we

provided two distinct escape hatches: a mechanism to opt an entire service out of hardening, and a fine-grained API to bypass checks for a specific line of code. The final tally after the rollout was remarkable. Across hundreds of millions of lines of C++ at Google, only five services opted out entirely because of reliability or performance concerns. Work is ongoing to eliminate the need for these few remaining exceptions, with the goal of reaching universal adoption.

Even more telling, the fine-grained API for unsafe access was used in just seven distinct places, all of which were surgical changes made by the security team to reclaim performance in code that was correct but difficult for the compiler to analyze. This widespread adoption stands as the strongest possible testament to the practicality of the hardening checks in real-world production environments.

While a production performance overhead of 0.3 percent is relatively small, at Google's scale, it represents a substantial absolute cost in terms of computing resources and energy. This was a deliberate, strategic investment in improving security and reliability.

The payoff: quantifiable impact

- ***Bug detection.*** More than 1,000 bugs were found and fixed during the rollout, including several security vulnerabilities and bugs that had lurked in the codebase longer than a decade. Hardening is projected to prevent 1,000 to 2,000 new bugs annually at the current development velocity.
- ***Reliability.*** The baseline segmentation fault rate

across the production fleet dropped by approximately 30 percent after hardening was enabled universally, indicating a significant improvement in overall stability. There was an initial uptick in crashes due to hardening checks failures, but this matched the expected hypothesis mentioned earlier; those failures would replace many segmentation faults.

- *Security.* Hardening demonstrably disrupted active internal offensive exercises and would have prevented others, proving its real-world effectiveness against exploitation attempts.
- *Debuggability.* Many subtle memory corruptions that are notoriously hard to debug were transformed into immediate, easily identifiable crashes at the point of error, saving significant developer time.

THE PATH FORWARD

The ultimate goal is to make these safety guarantees portable and available to all C++ developers. There is a quickly growing recognition in the C++ community that the status quo is undesirable, creating momentum for real change.

One way to push memory safety forward is to put a notion of a hardened Standard Library into the C++ Standard itself, so that developers across the board can get portable security guarantees. The initial proposal from Apple,[2] based on the implementation of hardening in libc++, has been recently voted into the upcoming C++26 Standard; the successful deployment experience of the hardened libc++ at Google and Apple has been crucial in getting the paper adopted.

The paper essentially standardizes a subset of libc++'s `fast mode` under the name of a *hardened implementation* that a program may choose, covering spatial memory safety in some of the most widely used standard classes, such as contiguous containers (whether a hardened or non-hardened implementation is the default is ultimately the choice of the vendor; some security-oriented platforms might choose to default to the hardened implementation). The paper is based on an observation that in fact all the hardening checks are already stated, almost always explicitly, in the Standard in the form of preconditions; it's just that violating a precondition used to result in the dreaded undefined behavior. Changing these cases of undefined behavior into useful well-defined behavior is, from the textual point of view, quite straightforward, making the proposal a lot less disruptive than might be expected.

Follow-up papers are intended to cover any missing checks that satisfy `fast mode` criteria (security-critical and low performance overhead), and it is expected that new additions to the standard library will use hardened preconditions as appropriate to avoid OOB (out of bounds) in the hardened implementation. Modes beyond `fast mode` are not currently considered for standardization; at least for the time being, the Standard should contain only checks that almost any program can afford. There are also no plans to propose any checks that would change the ABI.

Notably, the Standard leverages another C++26 feature, Contracts, which provides an extensive framework for specifying program invariants and handling violations. That gives developers significant flexibility in how they handle

a failing hardening check. Contracts were designed with consideration for Library Hardening as a use case, ensuring that libc++ assertion failures can be modeled directly by the Contracts evaluation semantics (specifically, the trapping mechanism used in libc++ hardening is precisely the quick-enforce evaluation semantic).

The path for other organizations to adopt hardening is now significantly clearer and less daunting.

CONCLUSION

The challenge of improving the memory safety of the vast landscape of existing C++ code demands pragmatic solutions. Standard library hardening represents a powerful and practical approach, directly addressing common sources of spatial safety vulnerabilities within the foundational components used by nearly all C++ developers.

Our collective experience at Apple and Google demonstrates that significant safety gains are achievable with surprisingly minimal performance overhead in production environments. This is made possible by a combination of careful library design, modern compiler technology, and profile-guided optimization.

Rolling this out initially at a massive scale was a large undertaking. However, much of the foundational work, in both the toolchain and in uncovering issues, has now been completed. The path for other organizations to adopt hardening is now significantly clearer and less daunting.

While not a panacea and not without tradeoffs, hardening eliminates entire classes of bugs and provides a substantial return on investment for security and reliability. *As such, we highly recommend that any organization using C++ enable hardening in their standard*

library today. Whether this means enabling hardening in LLVM's libc++ or requesting a comparable safety feature from other standard library implementations, it is a critical and affordable step forward in building a more secure and reliable C++ ecosystem.

References

1. Ren, G., Tune, E., Moseley, T., Shi, Y., Rus, S., Hundt, R. 2010. IEEE Explorer 30(4), 65-79; https://ieeexplore.ieee.org/abstract/document/5551002.
2. Varlamov, K., Dionne, L. 2025. Standard library hardening; https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2025/p3471r4.html.

Louis Dionne *works at Apple in Languages and Runtimes. He is the lead maintainer of libc++, has contributed to various security initiatives in recent years, and is an active member of the C++ Standards Committee.*

Alex Rebert *is an engineer at Google, where he focuses on memory safety. Previously, he co-founded ForAllSecure and led the creation of Mayhem, the autonomous system that won the 2016 DARPA Cyber Grand Challenge. He was recognized by* MIT Technology Review *as one of the 35 Innovators Under 35 and by* Forbes's *30 Under 30.*

Max Shavrick *is a security engineer at Google and one of the technical leads focusing on addressing memory unsafety. Before that, he worked on Windows and Azure security, finding and fixing remote code execution vulnerabilities. He was also previously president and captain of RPISEC, a*

student-run cybersecurity club and CTF team at Rensselaer Polytechnic Institute.

Konstantin Varlamov *works at Apple in Languages and Runtimes and is one of the maintainers of libc++ and a member of the C++ Standards Committee. For the past few years, his primary focus has been on the development and standardization of libc++ hardening.*

CONTENTS