



International Journal of Computer Science Engineering and Research

Journal homepage: <https://trpub.org/journal.php?id=EihcudPEIP>

TRPub

IJC SER

International Journal of
Computer Science
Engineering and
Research

Vol. 1 Issue 1 2026

Review Article

Open Access

Memory Safety in Systems Programming Languages: A Survey of Migration Strategies From Legacy Codebases

Nadia Sundararajan^{1*}, Huong Muthukumar², Dongwoo Lefebvre³

¹Department of Computer Science, Congo School of Computer Science, Georgia

²Department of Computer Science, Sharavati Institute for Advanced Computer Science, Philippines

³Department of Computer Science, Ebro Academy of Computer Science, Czech Republic

Article Info

Volume 1, Issue 1, 2026

doi: [10.62715/ijcser.2026.v1.i1.pp7-12](https://doi.org/10.62715/ijcser.2026.v1.i1.pp7-12)

Received: 27 February 2026

Accepted: 11 May 2026

Published: 01 July 2026

Abstract

Buffer overflows, use-after-free errors and related faults have been the leading source of exploitable defects in software written in C and C++ for decades, and government agencies now ask suppliers to plan a move to memory-safe languages. Most of the code concerned cannot simply be rewritten. It is large, depended upon and often poorly tested, so the practical question is how to migrate. This narrative review surveys the strategies described in the published literature and groups them into four families: hardening existing code with sanitizers and exploit mitigations, retrofitting safety through C dialects and pointer annotations, enforcing bounds in hardware with capability architectures, and replacing components incrementally with code in a safe language, chiefly Rust, either by hand or with automated translation. For each family the review describes the mechanism, the guarantee it gives, the effort it demands of an existing codebase and the evidence on how it fares in use. It then considers how the strategies combine, what the studies of unsafe Rust and of developer experience imply for migration planning, and where the literature is thin, notably on the cost of migration, the safety of language boundaries and the treatment of code that will never be rewritten.

Keywords: Memory safety, Rust, C and C++, Legacy code migration, Software security, Capability hardware

© 2026 Nadia Sundararajan et al. This is an open access article under the CC BY license (<https://creativecommons.org/licenses/by/4.0/>), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

1. Introduction

C and C++ give the programmer direct control over memory and ask the programmer, in return, to get every pointer dereference and every deallocation right. The languages do not check. A write past the end of an array or a read through a pointer to freed storage is undefined behaviour, and attackers have turned such errors into code execution since the earliest network worms. The systematisation by Szekeres et al. (2013) describes the result as a long contest in which each new defence has been met by a new attack technique, while the underlying errors continued to be written.

What has changed in recent years is the position of institutions. The United States National Security Agency advised organisations to prefer memory-safe languages where possible (NSA, 2022). A group of

* Corresponding author: Nadia Sundararajan, Department of Computer Science, Congo School of Computer Science, Georgia. E-mail: nadia.sundararajan@cngo.edu

© 2026 Nadia Sundararajan et al. This is an open access article distributed under the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.

national cybersecurity agencies followed with a call for software manufacturers to publish roadmaps for eliminating this class of vulnerability (CISA et al., 2023), and the White House Office of the National Cyber Director made the same argument to a technical audience (ONCD, 2024). These documents rest on reports from large vendors that memory safety errors account for a large majority of their severe vulnerabilities.

The advice is simple to state and hard to follow, because the operating systems, browsers, databases, network stacks and embedded firmware concerned represent decades of work. This review is about the middle ground between doing nothing and rewriting everything: the strategies by which a legacy codebase can be moved, in whole or in part, towards memory safety.

2. Scope and Approach

This is a narrative review of the published literature. It draws on peer-reviewed papers from the programming languages, security and software engineering communities, on the principal government guidance, and on standard books. Sources were chosen for their influence on the field and for the range of strategies they represent, and no claim is made to exhaustive coverage. The review is confined to migration from C and C++, considers Rust as the main target language because the research literature does, and leaves aside garbage-collected languages, which suit application code better than the systems code at issue here.

Two terms recur. Spatial safety means that every access through a pointer stays within the object the pointer was derived from. Temporal safety means that no access occurs after the object has been freed. A strategy may provide one, both, or only a reduction in the exploitability of violations. Table 1 sets out the strategies used to organise what follows, with the rewrite family divided into its manual and automated routes.

Family	Mechanism	Main limitation	Representative sources
Harden in place	Compiler instrumentation and runtime checks on unchanged source	Detects or impedes errors; does not remove them	Serebryany et al., 2012; Abadi et al., 2005
Safe C dialects	Checked pointer types and annotations added to C	Annotation effort; safety only in converted regions	Necula et al., 2002; Elliott et al., 2018
Capability hardware	Pointers carry bounds and permissions enforced by the processor	Needs new hardware; temporal safety is added separately	Woodruff et al., 2014
Incremental rewrite	Components replaced by Rust behind a C interface	Boundary code is unsafe; skills and build integration	Anderson et al., 2016; Jung et al., 2021
Automated translation	Tools convert C to Rust, then lift raw pointers to safe types	Output is largely unsafe Rust at first	Emre et al., 2021

3. Hardening Code in Place

The cheapest strategy leaves the source alone and changes how it is compiled and run. Sanitizers instrument the program to detect violations as they happen. AddressSanitizer surrounds objects with poisoned red zones, tracks the state of memory in a shadow map and delays the reuse of freed blocks, which lets it report overflows and many use-after-free errors with a slowdown that is tolerable in testing (Serebryany et al., 2012). Its guarantee is probabilistic and its cost is generally too high for production. The survey by Song et al. (2019) classifies the many sanitizers that followed and stresses that they are bug-finding tools whose value depends on the tests or fuzzing inputs that drive them.

Research compilers went further and enforced complete safety on unmodified C. SoftBound attaches base and bound metadata to every pointer and checks each dereference (Nagarakatte et al., 2009). Its

companion, CETS, adds identifiers that detect access to freed objects (Nagarakatte et al., 2010). Together they showed that full spatial and temporal safety for C is achievable by compilation alone, at a running-time cost that has kept the approach out of general production use.

Exploit mitigations accept that errors will occur and aim to stop them being useful. Control-flow integrity restricts indirect jumps and calls to targets permitted by the program's control-flow graph (Abadi et al., 2005). Code-pointer integrity protects only the pointers through which control can be hijacked and so achieves lower overhead (Kuznetsov et al., 2014). Such defences are widely deployed and raise the cost of attack, but they leave data-only attacks and information leaks possible. As a migration strategy, hardening buys time. It is the only option for code that nobody is in a position to change.

4. Retrofitting Safety Within C

A second family keeps the language recognisably C while giving the compiler enough information to check it. Cyclone was a safe dialect with fat pointers, region-based memory management and restrictions on unsafe constructs (Jim et al., 2002). Porting existing code to it required manual changes, and it remained a research system, though its treatment of regions influenced the design of Rust. CCured took the opposite approach to effort: it analysed existing C, inferred which pointers were used safely, and inserted runtime checks only for the rest (Necula et al., 2002).

Checked C is the most recent representative. It extends C with pointer types whose bounds are declared by the programmer and verified by the compiler, statically where possible and dynamically otherwise, and it lets checked and unchecked code coexist in one file (Elliott et al., 2018). The design is aimed squarely at incremental migration: a function can be converted, compiled and tested while the rest of the program is untouched. The limitation shared by this family is that the guarantee holds only inside converted regions and concerns spatial safety first. Modern C++ offers a comparable path through resource-owning types and smart pointers, which reduce temporal errors by convention but are not enforced by the language.

5. Enforcing Safety in Hardware

Capability architectures move the check into the processor. In CHERI, a pointer is replaced by a capability that carries bounds and permissions and is protected by a tag that software cannot forge; every load and store is checked against it (Woodruff et al., 2014). Existing C and C++ can in large part be recompiled for such a machine, with source changes concentrated in code that treats pointers as integers. The attraction for legacy systems is plain: spatial safety arrives through the compiler and the hardware, without rewriting. The obstacles are equally plain. The hardware must exist in the products concerned, capabilities are wider than conventional pointers, and temporal safety needs further machinery such as revocation of stale capabilities. Hardware enforcement also turns a violation into a fault, which stops an attack but still crashes the program.

6. Incremental Replacement With Rust

Rust provides memory safety without a garbage collector through a type system built on ownership and borrowing: every value has one owner, references are either shared and read-only or unique and mutable, and the compiler rejects programs that could alias and mutate at once (Matsakis and Klock, 2014; Jung et al., 2021). Operations the compiler cannot verify are confined to blocks marked unsafe. The RustBelt project gave a machine-checked proof that a core model of the language is sound and that important library abstractions built on unsafe code preserve safety for their callers (Jung et al., 2018).

Because Rust can call and be called through the C application binary interface, a legacy program can be migrated one component at a time. The Servo project demonstrated the approach on a browser engine and reported how ownership caught errors that are routine in comparable C++ code (Anderson et al., 2016). Tock showed that an operating system kernel for microcontrollers could use the type system to isolate drivers at no runtime cost (Levy et al., 2017). The Linux kernel has since accepted Rust as a language for new drivers, which brings the approach to one of the largest C codebases in existence.

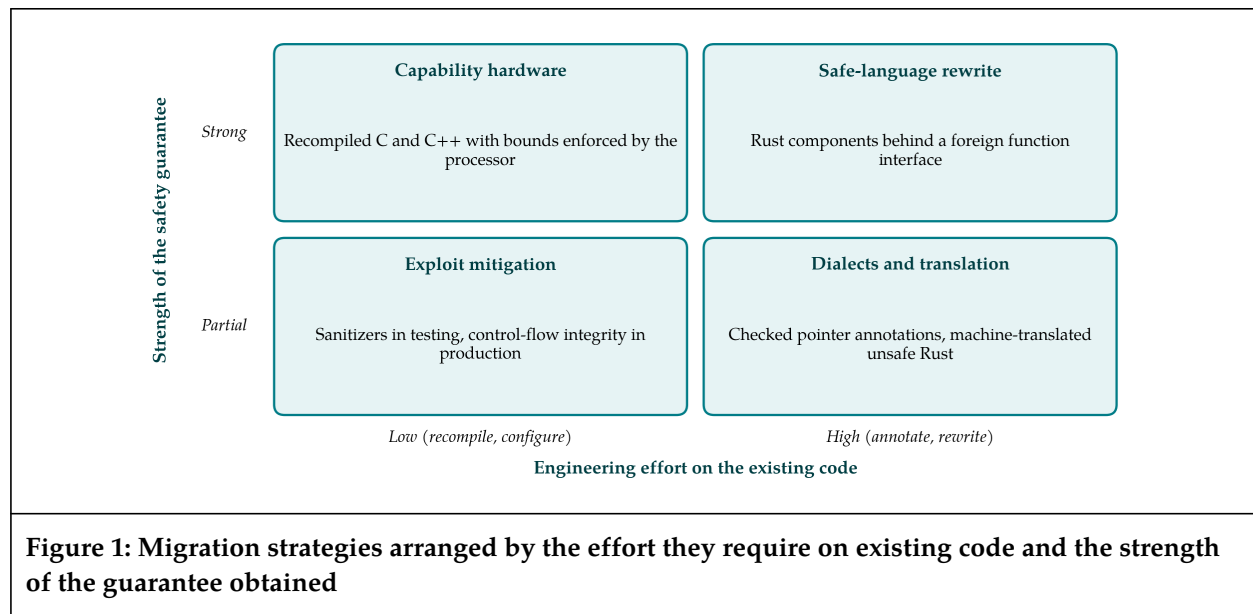
The empirical literature tempers the picture in three ways. First, unsafe code is common where Rust meets C. [Astrauskas et al. \(2020\)](#) found that unsafe blocks in published libraries are mostly small and are used largely to call foreign or other unsafe functions, and [Evans et al. \(2020\)](#) showed that many libraries which contain no unsafe code themselves depend on others that do. Second, the bugs that remain cluster there: in the study of real-world Rust defects by [Qin et al. \(2020\)](#), the memory safety bugs examined all involved unsafe code. Third, adoption has human costs. Developers interviewed and surveyed by [Fulton et al. \(2021\)](#) valued the confidence Rust gave them and reported a steep learning curve, slow compilation and friction in interoperating with existing code.

7. Automated Translation

Hand rewriting is slow, which makes mechanical translation attractive. Current tools translate C syntax into Rust that preserves the behaviour of the original by using raw pointers and unsafe blocks throughout. The output compiles with the Rust toolchain but is no safer than its source. [Emre et al. \(2021\)](#) examined why, and used the compiler's own borrow-checking errors to rewrite some raw pointers into safe references. They found that only a modest fraction could be lifted this way, because C programs depend on aliasing patterns and pointer arithmetic that the ownership discipline forbids. Translation is therefore best understood as a first step that changes the language and the build, after which the hard work of restructuring data ownership remains to be done by people or by better tools.

8. Choosing and Combining Strategies

The families differ along two dimensions that matter to a project planning migration: how much must be done to the existing code, and how strong a guarantee results. Figure 1 places them on those axes. No quadrant dominates. Mitigations are cheap and partial. A rewrite is costly and strong. Capability hardware offers strength at low source cost but depends on the platform, and dialects and translation sit in between.



In practice the strategies are layered. A plausible plan hardens the whole codebase at once, writes new components in a safe language, and selects existing components for rewriting by exposure: parsers of untrusted input, protocol handlers and code with a history of vulnerabilities come first. The government guidance recommends this kind of prioritised roadmap in place of wholesale replacement ([CISA et al., 2023](#)). Whatever the target, the advice of the legacy code literature applies. Behaviour should be pinned down with characterisation tests before a component is replaced, and the seams at which a component can be detached have to be found or created first ([Feathers, 2004](#)).

9. Gaps and Future Work

Several questions that a migration planner would ask are poorly answered in the literature. The first is cost. Published accounts describe what was migrated and what was learned, but controlled evidence on the effort of rewriting, annotating or translating a given volume of C is scarce, as is evidence on defect rates in migrated components over time. The second is the boundary. A mixed-language program has an interface at which Rust's assumptions about aliasing and lifetimes must be upheld by C code that knows nothing of them, and tools for checking that interface are immature compared with tools for either language alone.

The third gap concerns the residue. Every realistic plan leaves a body of C and C++ in service for many years, and research on making that residue safer, through better sanitizers in production, capability hardware or checked dialects, receives less attention than the target language. Finally, most studies of Rust in use examine open-source libraries. Evidence from safety-critical, embedded and regulated settings, where certification and toolchain qualification govern language choice, would widen the basis for the policy now being made.

10. Conclusion

Memory safety can be approached from an existing C or C++ codebase by four routes: harden what exists, annotate it into a checked dialect, run it on hardware that enforces bounds, or replace it piece by piece with code in a safe language. The literature shows each to be feasible and none to be sufficient alone. Rust offers the strongest guarantee for new and rewritten code, with the caveat that safety must be argued, not assumed, wherever unsafe blocks and foreign interfaces appear. The older strategies keep their place because most legacy code will be migrated late or never. Sound planning treats migration as a portfolio ordered by risk, and the research most needed is on its costs and on the interfaces between safe and unsafe worlds.

Conflict of interest

To be declared by the author(s) before publication.

Funding

To be declared by the author(s) before publication.

References

- Abadi, M., Budiu, M., Erlingsson, Ú. and Ligatti, J. (2005). [Control-flow integrity](#). *Proceedings of the 12th ACM Conference on Computer and Communications Security*, 340-353.
<https://doi.org/10.1145/1102120.1102165>
- Anderson, B., Bergstrom, L., Goregaokar, M., Matthews, J., McAllister, K., Moffitt, J. and Sapin, S. (2016). [Engineering the Servo web browser engine using Rust](#). *Proceedings of the 38th International Conference on Software Engineering Companion*, 81-89.
- Astrauskas, V., Matheja, C., Poli, F., Müller, P. and Summers, A.J. (2020). [How do programmers use unsafe Rust?](#) *Proceedings of the ACM on Programming Languages*, 4(OOPSLA): Article 136.
<https://doi.org/10.1145/3428204>
- CISA, NSA, FBI and international partner agencies (2023). [The Case for Memory Safe Roadmaps](#). Cybersecurity and Infrastructure Security Agency, Washington, DC.
- Elliott, A.S., Ruef, A., Hicks, M. and Tarditi, D. (2018). [Checked C: Making C safe by extension](#). *Proceedings of the 2018 IEEE Cybersecurity Development Conference (SecDev)*, 53-60.
- Emre, M., Schroeder, R., Dewey, K. and Hardekopf, B. (2021). [Translating C to safer Rust](#). *Proceedings of the ACM on Programming Languages*, 5(OOPSLA): Article 121. <https://doi.org/10.1145/3485498>
- Evans, A.N., Campbell, B. and Soffa, M.L. (2020). [Is Rust used safely by software developers?](#) *Proceedings*

- of the 42nd International Conference on Software Engineering, 246-257.
- Feathers, M.C. (2004). *Working Effectively with Legacy Code*. Prentice Hall, Upper Saddle River, NJ.
- Fulton, K.R., Chan, A., Votipka, D., Hicks, M. and Mazurek, M.L. (2021). Benefits and drawbacks of adopting a secure programming language: Rust as a case study. *Proceedings of the Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021)*, 597-616.
- Jim, T., Morrisett, G., Grossman, D., Hicks, M., Cheney, J. and Wang, Y. (2002). Cyclone: A safe dialect of C. *Proceedings of the 2002 USENIX Annual Technical Conference*, 275-288.
- Jung, R., Jourdan, J.-H., Krebbers, R. and Dreyer, D. (2018). RustBelt: Securing the foundations of the Rust programming language. *Proceedings of the ACM on Programming Languages*, 2(POPL): Article 66. <https://doi.org/10.1145/3158154>
- Jung, R., Jourdan, J.-H., Krebbers, R. and Dreyer, D. (2021). Safe systems programming in Rust. *Communications of the ACM*, 64(4): 144-152. <https://doi.org/10.1145/3418295>
- Kuznetsov, V., Szekeres, L., Payer, M., Candea, G., Sekar, R. and Song, D. (2014). Code-pointer integrity. *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, 147-163.
- Levy, A., Campbell, B., Ghena, B., Giffin, D.B., Pannuto, P., Dutta, P. and Levis, P. (2017). Multiprogramming a 64kB computer safely and efficiently. *Proceedings of the 26th Symposium on Operating Systems Principles*, 234-251. <https://doi.org/10.1145/3132747.3132786>
- Matsakis, N.D. and Klock, F.S. (2014). The Rust language. *ACM SIGAda Ada Letters*, 34(3): 103-104. <https://doi.org/10.1145/2692956.2663188>
- Nagarakatte, S., Zhao, J., Martin, M.M.K. and Zdancewic, S. (2009). SoftBound: Highly compatible and complete spatial memory safety for C. *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 245-258. <https://doi.org/10.1145/1542476.1542504>
- Nagarakatte, S., Zhao, J., Martin, M.M.K. and Zdancewic, S. (2010). CETS: Compiler enforced temporal safety for C. *Proceedings of the 2010 International Symposium on Memory Management*, 31-40.
- Necula, G.C., McPeak, S. and Weimer, W. (2002). CCured: Type-safe retrofitting of legacy code. *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 128-139. <https://doi.org/10.1145/503272.503286>
- NSA (2022). *Software Memory Safety. Cybersecurity Information Sheet*. National Security Agency, Fort Meade, MD.
- ONCD (2024). *Back to the Building Blocks: A Path Toward Secure and Measurable Software*. Office of the National Cyber Director, The White House, Washington, DC.
- Qin, B., Chen, Y., Yu, Z., Song, L. and Zhang, Y. (2020). Understanding memory and thread safety practices and issues in real-world Rust programs. *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 763-779. <https://doi.org/10.1145/3385412.3386036>
- Serebryany, K., Bruening, D., Potapenko, A. and Vyukov, D. (2012). AddressSanitizer: A fast address sanity checker. *Proceedings of the 2012 USENIX Annual Technical Conference*, 309-318.
- Song, D., Lettner, J., Rajasekaran, P., Na, Y., Volckaert, S., Larsen, P. and Franz, M. (2019). SoK: Sanitizing for security. *Proceedings of the 2019 IEEE Symposium on Security and Privacy*, 1275-1295.
- Szekeres, L., Payer, M., Wei, T. and Song, D. (2013). SoK: Eternal war in memory. *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, 48-62. <https://doi.org/10.1109/SP.2013.13>
- Woodruff, J., Watson, R.N.M., Chisnall, D., Moore, S.W., Anderson, J., Davis, B., Laurie, B., Neumann, P.G., Norton, R. and Roe, M. (2014). The CHERI capability model: Revisiting RISC in an age of risk. *Proceedings of the 41st Annual International Symposium on Computer Architecture*, 457-468.

Cite this article as: Sundararajan N., Muthukumar H. and Lefebvre D. (2026). Memory Safety in Systems Programming Languages: A Survey of Migration Strategies From Legacy Codebases. *International Journal of Computer Science Engineering and Research*. 1(1), 7-12. doi: 10.62715/ijcser.2026.v1.i1.pp7-12