Why is C the safest language?

By Eskil Steenberg Hald 2025

Representative of Sweden in iso Wg14

eskil at quelsolaar dot com @eskilsteenberg

The most trusted software in the world like OpenSSL, Apache, SQLite, Curl, CPython, FFMpeg, PHP, the GNU free software collection, most OS kernels, and most filesystems are all written in C. No other language has managed to produce anywhere near the same amount of safety and security critical software deployed the world over as C. In evolutionary terms, it is clear that security critical C projects have a much higher survival rate than security critical projects written in any other language.

Despite the overwhelming success of C as a language for developing security critical software, many security researchers claim that C is an unsafe language that should be avoided for security critical software. They take it as a given that C is unsafe, because of some of C features, most notably the lack of required bounds checking. It is a very unscientific to assume this feature would outweigh possible other benefits of using C, when clearly this feature hasn't stopped C from being the most successful language in security. When assumption says one thing but real world experience says something completely different, its time to re-evaluate ones assumptions.

If security researchers are interested in research that is based on the scientific method, they should clearly want to investigate why C has produced so many successful security critical projects, instead of dismissing C as an insecure language, against clear evidence. Security researchers should investigate various explanations for why C has been so successful, despite the reasons for why someone may think it shouldn't be. To simply dismiss all the people who have chosen C as their language, and then have gone on to produce some of the most trusted software on the planet (and other planets), as simply not knowing what they are doing, is both disrespectful and ignorant, and should not be part of scientific discord with the ultimate goal of finding the truth and making progress.

Actual research in to why C has been so successful for security critical software could have great impact on future security policy, possible language design, how we see Cs future, and generally how we think about software development. Simply ignoring C developers success, or worse trying to gaslight the world into thinking C developer have not had great contribution to the security of software, robs the world of valuable insights in to how successful software is developed.

As an experienced C developer, I have a number of hypotheses as to why C has been so successful, that warrants further research. I want to be clear, I am not a researcher, nor am I claiming to present quantitative evidence for these theories. I do however think they are among theories that should be thoroughly

investigated. As part of this list of theories, I also want to point out some possible ramifications for how we should view software development if they are shown to pan out.

Theory 1: Its about readability.

C is language with very few abstractions. This makes it very easy to read and reason about the code. The difficulty in writing code is almost always to close the gap between, what the code does, and what the code is meant to do. Therefor being able to follow every step of execution, and have each step be explicit aids greatly. C code becomes safe, because it is easy to reason about, and easy to audit. Most things are written in place, and therefor require less knowledge of a larger system in order to be understood. This lack of interconnectedness that abstractions create also makes it harder for changes in one part of the code to break other parts of the code. C is a verbose, explicit language and while this creates a less convenient programming experience, it creates trustworthy software. An interesting observations is that other "unsafe" C derived languages like C++ and Objective-C, have produced has less trusted safety critical software. This indicates that the simplicity and lack of features of C are contributing factors. Another observation is that the successful software projects that do use C, tend to use older versions of C and restrict the use of features. The recent success of Python and Lua over more complex languages like Perl also indicates that simple readable languages with fewer features are in fact more reliable.

Potential lesson: A general focus on clarity over expressiveness and cleverness in language design could lead to safer and simpler code. We should start looking at abstractions with a more critical eye in computer science. For C developers, this means avoiding complex macros, and using long and expressive naming. There are also many opportunities here to improve tooling. Compilers reason about code in order to optimize it, but rarely do they show the programmer its reasoning. It would be very valuable if code paths that are optimized away, assumptions made about values possible ranges, or memory model assumptions where presented to the user.

Theory 2: C is fun.

Many C programmers claim that C is a programming language they enjoy using. The fun factor is an underestimated factor in programming. An enjoyable programming experience leads to more engagement and long term maintenance of software project. A weak software project that is consistently updated and improved over a long period of time will eventually surpass a good software project that no one cares to improve. Something that speaks for this theory is that so many of the mentioned C projects are open-source project mainly maintained by volunteers. For a volunteer project to succeed motivation is key.

Potential lesson: From a security perspective, we need to find the right balance between safety procedures and the joy and agility of programming. If every change results in onerous re-certification processes, and pointless warnings and procedures that need to be addressed, software will not be maintained properly. We need to find ways to better triage where the risks are and spend our time accordingly and avoid making necessary changes hard to make.

For other languages, we may also consider where the fun lies. C++ is a programming language that many people greatly enjoy architecting code in. It has a rich set of features to choose from, to build intricate structures. These structures, while fun to invent, are less fun to maintain, hence most C++ programmers have never seen any C++ code they did not want to rearchitect. It is an open question how to design languages that remains fun to program in as projects grow, but it should be explored seriously.

Theory 3: Control matters

C gives the programmer a very high degree of control. As I am fond of saying: "In the beginning all you want is results, in the end all you want is control". Because C offers little in terms of pre-made facilities and a minimal standard library, you have to do most things yourself. This means that if you complete a project in C, you have a much more complete understanding of your software, than if it were written in a higher level language. This deep understanding is critical for identifying potential security vulnerabilities. Linus Torvalds once observed that when he writes C, he can in his mind see the assembly instructions being executed.

Theory 4: C is somewhat uniform

Compared to a language such as C++ where many different styles and paradigms exist, C is relatively simple. Because so many other languages like C++, Java and others borrow syntax from C, many non-C programmers are able to read and understand C to some degree. This lingua franca of computing aids in getting many more eyes on critical code and therefore makes C code more secure. Unlike languages like C++ there is less divergence in the options of how the language should be programmed, simply because there are fewer ways to do things. This also makes it easier for open source projects to attract more developer who can contribute.

Potential lesson: We should put higher weight on writing plain code, and avoid clever tricks or new language features. We can create new guidelines and tools that evaluate code according to these guidelines. MISRA, and CERT are such guidelines focusing on safety and security, but additional guidelines that focus on simplicity, readability, and portability could be created.

Theory 5: Security doesn't matter as much you think

It is possible that C is successful despite its apparent security shortcomings, simply because users have different priorities. The main priority that could override security is Cs performance. Performance isn't just about how long something takes to execute it also translates in to battery life, power consumption, hardware scale, cooling costs, CO2 emissions, and directly to cost of operation. For a large scale cloud operations a few percentage points of performance degradation, translates in to hundreds of millions in added capex for hardware, power and cooling. Data centres alone are estimated to use 3-4% of all generated power by the end of the decade, and added to this is all the power consumed by other devices running C based software. Even a small degradation of efficiency at this scale results in large costs and CO2 emissions.

Switching from trusted legacy C software, to new comparably untested software written in an other language, with substantially higher running cost, with the promise of better future security is a substantial leap of faith.

Further evidence for this theory is that, C does not have to be memory unsafe. The C standard clearly states that any implementation is free to define a safe and documented behaviour for anything left undefined by the standard. Such implementations exist such as UBSan and Valgrind. It is entirely possible for anyone who prioritizes security to run any C software in these implementations. Yet, few people do (outside of debugging), this indicates that the majority of users have other priorities.

If things like performance and memory usage are highly valued properties of C software, perhaps this popularity affords C projects more time to mature and therefor address other issues like security issues.

Potential lesson: Security professionals need to learn to accept that they live in a world where other considerations often take precedence over security. An example of this is various speculative execution mitigations that have been needlessly forced on most users. These mitigations have probably cost untold billions. These costs are rarely taken in to consideration when security is being discussed, and while many security changes have negligible impact on a specific system, in the aggregate, and when staked on to of layers of security protections, on a global scale have very large impacts.

Theory 6: C Is old

C enjoys some advantages of being old, in that many projects have had time to mature and people have had time build up skills and knowledge about the language. Similarly C has wide range of tools and implementations that are mature and offers a lot of advantages. Most safety critical software have had years to mature and gain trust. While this factor can explain some of Cs success, many languages like C++ or Java are now old enough that if they would have

offered significant advantage over C, they should have supplanted C by now, this clearly hasn't happened.

Potential lesson: Computer science needs to stop equating new with better. You can not prove a negative, and therefor there is no way to prove that software is free from flaws. In fact, the best way we have of evaluating if something is good, is its longevity. This goes for languages, and code bases a like.

Theory 7: C programmers are different.

It is possible that C programmers are so used to dealing with things like memory management that, it becomes a natural part of how they think, and therefore presents far less far fewer issues than an outsider may suspect.

For a pedestrian who is not used to cars, living in a city where 2 tone metal boxes race down the streets at 50kph may seem like an extremely dangerous environment where inhabitants are contently under immense stress to avoid being hit by cars. For anyone who have coexisted with cars in a city for a long time, its a known risk, but one that one that represent an insignificant portion of lives worries. To developers coming from languages where memory is handled for you, the task can feel daunting, but to long time C developers it is a natural part of development that take up very little brainpower to maintain. On the scale of things C experienced programmers worry about memory management ranks low as there are other much more challenging tasks that needs to be completed.

Partly this is a result of experience, but its also that, people who seek out C, and chose to program in C are wired in a way that fits well with Cs design. C programmers do tend to be a different bread of programmers, who want to get close to the metal, and value control over convenience.

Potential lesson: Perhaps we should consider C programmers separate, and recognize that C is a language that requires a different mindset to master than another languages. While many programmers tend to be able to switch languages often, maybe we should be more careful about who is assigned to program C. Instead of teaching C as yet another language, perhaps it should be treated more as a specialty. Advertising for C/C++ programmers isn't very helpful when the two languages require very different skills and mindsets.

Theory 8: C developers are self selecting good programmers.

Perhaps the language C has the advantage that it simply attracts good programmers. It is even conceivable that the perceived challenge of writing C code contributes to its popularity among seasoned programmers. This would mean that C projects are in part successful, because C programmers tend to be good

programmers, and that these projects would see similar success with the same people using a different language. Linus torvalds famously pronounced that he wont let C++ in to the Linux Kernel, for the simple reason that he doesn't want to work with people who like C++.

Potential lesson: If many of the best programmers choose C, then maybe C isn't that bad.

Theory 9: Its about tooling

Most languages only have one or very few implementations, while C enjoys a wide range of implementations with very different aims. C also has many different debuggers, linters, fuzzes and static analysis tools that aid in debugging. The range, quality and maturity of C tools greatly aids in development or robust software.

Potential lesson: Many other languages would do very well to focus more on tooling then syntax. IMO having a good debugger is by far the most important tool for a programmer, yet many programmers don't use debuggers, and many languages have very few debugging tools. While C has a lot of tools, there are still great improvements that can be made.

Theory 10: C fails fast and hard.

C is famous for its unforgiving nature. Because so many C bugs cause segfaults or other showstopping issues, much fewer bugs survive debugging. Modern C tools often make C even more unforgiving by detecting things like reads of uninitialized variables. This is a much stricter requirement than a language that may automatically initialize all values, and this forces developers to explicitly state their intention, as compilers do not assume that the user wants a default initialization. Many languages in the name of convenience, lets the programmer get away with things that do create hard to find bugs. For example java script lets you access object members without first declaring them. You can easily write object->member = 42; and then by mistake try to access the value x = object->Member; Java script will then assign x a default value, instead of issuing an error. Yes its convenient to not have to declare members in advance, but saving a few seconds of typing and then loosing hours to debugging or worse shipping broken software is much worse.

Potential lesson: Policies that force programmers to engage with issues and ambiguities, instead of issuing warnings, or worse make assumptions about the users intentions are important to improve security. Most C compilers can be configured to fail compilation when it encounters selected warnings. This could be made much stricter, where more code is rejected, even though it technically is standard compliant. Implicit type conversions is one such feature that should

not have been made part of C, but can easily be detected and remedied with tools.

Theory 11: The issues are known.

As an old and well known language the issues in C are mostly well known. The fact that the language is smaller than many other languages, means that compilers and other infrastructure is less likely to encounter code that trigger an unexplored corner case. Most of the security issues are well known and easy to audit for. Because C is so wide spread there is also a large amount of people who are able to review and read C code.

Theory 12: Survivorship bias

Given that C code has been so successful, any issue in this widely deployed C code gets and outsized impact. A security vulnerability in OpenSSL has far greater ramifications, than another SSL implementation that don't have nearly as many users, no matter what language it was implemented in. Other languages like javascript, and SQL code, that have well known exploits have a much more diffuse attack surface. There are numerous websites that have a wide range of exploits, and may have lots of vulnerabilities, yet the finding of an exploit would not raise the headlines that an exploit in a widely used C system like the Linux kernel. Given the size and scope of projects like the Linux kernel and the number of people who have eyes of the project, its surprising how seldom serious exploits are found.

Potential lesson: Its worth appreciating how seldom major security issues appear in major security critical software's written in C.

Theory 13: The roads not taken.

Any person who finds bugs in someone else's code, will have a bias against the design decisions that made the bugs possible or likely. This is another form of survivorship bias. However what they do not see are the issues that the design prevented. All engineering is inherently about trade-offs. Any decision will make some issues more likely and some other issues less likely. It is possible that while the design trade-offs in C creates some classes of bugs that reoccur, it on balance prevents much more issues, then it creates.

A good engineer, in any field, knows that she has to weigh the potential risks and benefits of any decision. Trying to cover for all risks, no matter how miniscule is to not properly allocate time, effort and resources on the problems that are likely to cause problems. Security researches have along track record of raising security issues that have a extremely low probability of being exploited.