TLP:CLEAR

Exploring Memory Safety in Critical Open Source Projects

Publication: June 26, 2024

Cybersecurity and Infrastructure Security Agency (CISA)
Federal Bureau of Investigation (FBI)
Australian Signals Directorate's (ASD's) Australian Cyber Security Centre (ACSC)
Canadian Centre for Cyber Security (CCCS)

Table of Contents

Executive Summary

In December 2023, the Cybersecurity and Infrastructure Security Agency (CISA), the National Security Agency (NSA), the Federal Bureau of Investigation (FBI), and international cybersecurity authorities from Australia, Canada, New Zealand, and the United Kingdom, published The Case for Memory Safe Roadmaps. This joint publication notes that memory safety vulnerabilities are among the most prevalent classes of software vulnerability and generate substantial costs for both software manufacturers and consumers related to patching, incident response, and other efforts. It further recommends software manufacturers create memory safe roadmaps, including plans to address memory safety in external dependencies, which commonly include open source software (OSS).

This joint report, authored by CISA, FBI, the Australian Signals Directorate's Australian Cyber Security Center (ASD's ACSC), and the Canadian Centre for Cybersecurity (CCCS), provides a starting point for these roadmaps by investigating the scale of memory safety risk in selected OSS. To understand the state of memory unsafety in OSScode, we explored a set of critical open source projects to determine the extent to which they are written in memory-unsafe languages.[1] Memory-unsafe languages require developers to properly manage memory use and allocation. Mistakes, which inevitably occur, can result in memory-safety vulnerabilities such as buffer overflows and use after free. Successful exploitation of these types of vulnerabilities can allow adversaries to take control of software, systems, and data. Memory-safe languages shift the abstraction layer and responsibility for writing memory-safe code from the developer to the compiler or interpreter, vastly reducing opportunities to introduce memory-safety vulnerabilities.

Upon analyzing a list of 172 projects derived from the Open Source Security Foundation (OpenSSF) Securing Critical Projects Working Group's List of Critical Projects,[2] we observe that:

- 52% of the projects contain code written in a memory-unsafe language.
- 55% of the total lines of code (LoC) for all projects were written in a memory-unsafe language.
- The largest projects are disproportionately written in memory-unsafe languages. Of the ten largest projects by total LoC, each has a proportion of memory unsafe LoC above 26%. The median proportion using memory-unsafe languages across the ten projects is 62.5% and four of the ten project proportions exceed 94%.
- Dependency analysis of three projects written in memory-safe languages demonstrated that each one depended on other components written in memory-unsafe languages.

[1] Memory-unsafe languages used in this analysis are defined in Table 1.
[2] https://github.com/ossf/wg-securing-critical-projects/tree/main/Initiatives/Identifying-Critical-Projects.

Hence, we determine that most critical open source projects analyzed, even those written in memory-safe languages, potentially contain memory safety vulnerabilities. This can be caused by direct use of memory-unsafe languages or external dependency on projects that use memory-unsafe languages. Additionally, low-level functional requirements to disable memory safety may create opportunities for memory safety vulnerabilities in code written in otherwise memory-safe languages. These limitations highlight the need for continued diligent use of memory safe programming languages, secure coding practices, and security testing.

We encourage additional efforts to understand the scope of memory-unsafety risks in OSS and continued discussion of the best approaches to managing and reducing this risk. These discussions should not only consider the risk reduction of a given approach, but also resource constraints, performance requirements, and direct and indirect costs of implementation. We note that while memory safety vulnerabilities are the most prevalent class of vulnerabilities, there is important work to be done to reduce other systemic classes of vulnerabilities.[3] Please read further recommendations to address memory safety in the joint document, The Case for Memory Safe Roadmaps and CISA's Technical Advisory Council of CISA's Cybersecurity Advisory Committee report on memory safety.

Background

Recent Efforts

The cybersecurity community renewed its focus on memory safety in recent years. Consumer Reports released an October 2022 report noting that "roughly 60 to 70 percent of browser and kernel vulnerabilities—and security bugs found in C/C++ code bases—are due to memory unsafety."[4] The report also cited an interesting discussion of the security costs and benefits of using memory-safe languages instead of memory-unsafe ones.[5]

In November 2022, the NSA released guidance for software developers and operators on protecting against memory safety issues.[6] The 2023 National Cybersecurity Strategy (March 2023) and corresponding implementation plan (July 2023) both discuss investing in memory safety and collaborating with the open source community.[7] The National Cybersecurity Strategy Implementation Plan Initiative 4.1.2, titled "Promote Open Source Software Security and the Adoption of Memory Safe Programming Languages," directs the establishment of "an Open Source Software Security Initiative (OS3I) to champion the

[3] CVE-2022-44228 is an example of an impactful vulnerability that is outside the class of memory safety vulnerabilities: https://www.cisa.gov/news-events/news/apache-log4j-vulnerability-guidance.
[4] Grauer, Yael. "Future of Memory Safety," *Consumer Reports,* January 2023. https://advocacy.consumerreports.org/wp-content/uploads/2023/01/Memory-Safety-Convening-Report.pdf.
[5] Gaynor, Alex. "What science can tell us about C and C++'s security." 27 May 2020. https://alexgaynor.net/2020/may/27/science-on-memory-unsafety-and-security/.
[6] National Security Agency, "Software Memory Safety Cybersecurity Information Sheet" https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF.
[7] "National Cybersecurity Strategy." March 2023. https://www.whitehouse.gov/wp-content/uploads/2023/03/National-Cybersecurity-Strategy-2023.pdf.

adoption of memory safe programming languages and open source software security."[8] Initiative 4.2.1 in the same document, titled "Accelerate Maturity, Adoption, and Security of Memory Safe Programming Languages," also calls for investment in memory-safe programming languages and seeks the involvement of the newly-established OS3I.[9] In February 2024, the Office of the National Cyber Director released a report calling on technology manufacturers to migrate to memory safe programming languages.[10]

In August 2023, CISA and the Office of the National Cyber Director, in coordination with the OS3I, the National Science Foundation, Defense Advanced Research Projects Agency, and the Office of Management and Budget, published a request for information on OSS security seeking input on "areas of long-term focus and prioritization efforts."[11]

In a 2023 speech, CISA's Director remarked on the need to address memory-unsafety, via "migrating to memory safe code from legacy code bases," and by addressing memory-unsafety in open source projects and university coursework.[12] In December 2023, CISA jointly published The Case for Memory Safe Roadmaps. In the same month, the Technical Advisory Council of CISA's Cybersecurity Advisory Committee issued recommendations for CISA action in the area of memory safety.[13]

The OSS community produced a wealth of commentary and research relating to memory safety. Of particular interest is the work of the OpenSSF Best Practices Working Group.[14] The mission of this working group's Memory Safety Special Interest Group is to "understand and reduce memory safety vulnerabilities in OSS."[15] Another OpenSSF working group, Securing Critical Projects Working Group, produced the list of software from which we derived our dataset.[16]

Other efforts to promote memory safety across the open source ecosystem include Prossimo, an Internet Security Research Group project whose "first goal is to move the Internet's security-sensitive software infrastructure to memory safe code."[17] Another is Google's OSS-Fuzz, a tool for debugging open source projects that evolved to detect problems in both memory-safe and memory-unsafe languages.[18]

---

[8] Initiative 4.1.2, "National Cybersecurity Strategy Implementation Plan," July 2023. https://www.whitehouse.gov/wp-content/uploads/2023/07/National-Cybersecurity-Strategy-Implementation-Plan-WH.gov_.pdf p. 36.
[9] Initiative 4.2.1, "National Cybersecurity Strategy Implementation Plan," July 2023. https://www.whitehouse.gov/wp-content/uploads/2023/07/National-Cybersecurity-Strategy-Implementation-Plan-WH.gov_.pdf p. 39.
[10] Office of the National Cyber Director, "Back to the Building Blocks: A Path Toward Secure and Measurable Software," February 2024. https://www.whitehouse.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf.
[11] Office of the National Cyber Director, https://www.federalregister.gov/documents/2023/08/10/2023-17239/request-for-information-on-open-source-software-security-areas-of-long-term-focus-and-prioritization.
[12] Easterly, "CISA Director Easterly Remarks at Carnegie Mellon University." https://www.cisa.gov/cisa-director-easterly-remarks-carnegie-mellon-university.
[13] CISA Cybersecurity Advisory Committee Technical Advisory Council, "Report to the CISA Director," December 2023. https://www.cisa.gov/sites/default/files/2023-12/CSAC_TAC_Recommendations-Memory-Safety_Final_20231205_508.pdf.
[14] https://openssf.org/; https://best.openssf.org/.
[15] https://github.com/ossf/Memory-Safety.
[16] https://github.com/ossf/wg-securing-critical-projects/tree/main/Initiatives/Identifying-Critical-Projects.
[17] Prossimo, "About Prossimo," https://www.memorysafety.org/about/.
[18] "OSS-Fuzz" https://google.github.io/oss-fuzz/.

Report Terminology

This report refers to code, programming languages, and projects as "memory-safe" or "memory-unsafe" (or simply, "unsafe"). This terminology is based on our designation of certain programming languages in Table 1 as memory unsafe for the purposes of this analysis. A memory-safe language handles memory management on behalf of the developer. In theory, a program written in a memory safe language will never encounter a memory allocation or memory use error, two error types that the report collectively describes as "memory-safety" errors.[19] Our language designations simplify the complexities of determining memory safety to make it possible to draw rough conclusions from millions of lines of code (MLoC). We acknowledge that more granular analysis would identify numerous instances of memory unsafety within code written in languages other than those designated to be generally "memory-unsafe," (e.g., Rust code contained within an "Unsafe" block). We make no judgments about code quality, other security aspects, or other kinds of safety such as physical safety of material property.

Methodology and Results

This section describes the methods (how) and the results (what) of our analysis. Interpretation (why) and recommendations follow in the Discussion section.

Our analysis is based on the OpenSSF Securing Critical Projects Working Group's Securing Critical Projects List.[20] This list is developed using a combination of quantitative and expert qualitative inputs, including the OpenSSF Criticality Score.[21] The list acknowledges various potential sources of bias, including that the Criticality Score, "…prefers projects that are extremely active on specific forges. Such projects are likely to be important (at least to the participants). However, this is not a perfect measure; some projects will score low here and yet be very critical."[22]

While we identified GitHub repositories for all projects in the list, GitHub is only one code hosting provider and does not represent all open source repositories.[23] Despite these potential sources of bias, we consider the Securing Critical Projects List to be a reasonable selection of open source projects to analyze.

We modified the Securing Critical Projects List as necessary to identify specific repositories for analysis. In some cases, these modifications excluded projects that were not clearly defined at the repository level (e.g., Alpine Linux), and in other cases, selected specific repositories from multiple repositories that could potentially represent a given project.

---

[19] Yang H, O'Hearn P. A semantic basis for local reasoning. International Conference on Foundations of Software Science and Computation Structures 2002 Mar 15 (pp. 402-416). Berlin, Heidelberg: Springer Berlin Heidelberg.
[20] https://github.com/ossf/wg-securing-critical-projects/tree/main/Initiatives/Identifying-Critical-Projects.
[21] https://openssf.org/blog/2023/07/28/understanding-and-applying-the-openssf-criticality-score-in-open-source-projects/.
[22] https://github.com/ossf/wg-securing-critical-projects#meetings-times.
[23] Trujillo, Milo Z., Laurent Hébert-Dufresne, James Bagrow, "The penumbra of open source: projects outside of centralized platforms are longer maintained, more academic and more collaborative." Cornell University. 22 May 2022. https://doi.org/10.48550/arXiv.2106.15611.

Through this process, we confirmed and identified public source code repositories for 172 projects on the Securing Critical Projects List, most of them hosted on GitHub. We then analyzed these repositories using the "cloc" tool, which identifies files that contain source code and counts LoC within each such file that correspond to a given programming language.[24] We did not evaluate the accuracy of "cloc." It is possible that "cloc" incorrectly classified some files or LoC.

Table 1: Memory-Unsafe and Non-Executable Languages and File Types

Language and File Type	Languages and File Types
Memory-unsafe	Assembly, C, C++, C/C++ Header, Cython, D
Non-executable	CSV, diff, HTML, INI, JavaScript Object Notation (JSON), Markdown, reStructuredText, Text, Web Services Description, XHTML, XML, XSD, XSLT, YAML

In the following results (including the full project list in Table 6), "Unsafe LoC" represents the sum of LoC written in the set of languages that we defined as memory-unsafe in Table 1 and "Total LoC" represents the sum of LoC written in any language, excluding blank lines, comments, and non-executable languages as defined in Table 1. The categories in Table 1 are based on our analysis of languages detectable by "cloc" and are not necessarily comprehensive or complete sets of all memory-unsafe or non-executable languages or file types.[25]

We observed that source code repositories often include test code that is usually only used in development and not by the fielded software. For example, the "packaging" project has six lines of memory-unsafe C code in a single file: `./tests/hello-world.c`. It would make sense to exclude development test code that is not likely to be exposed to attacks, however, broadly excluding repository directories named /test and /tests may be too coarse of an approach. Thus, our analysis did not attempt to identify or exclude test code and therefore includes some LoC that are not likely to be exposed to attacks.

The "By Project" heading in Table 2 identifies projects containing any amount of memory-unsafe code, and the "By LoC" heading identifies memory-unsafe LoC in the aggregate, independent of project. Roughly half of the projects contain some code written in a memory-unsafe language and slightly more than half of the total executable LoC in our analysis were written in a memory-unsafe language.

[24] https://github.com/AlDanial/cloc.
[25] https://github.com/AlDanial/cloc?tab=readme-ov-file#recognized-languages.

Table 2: Proportion of Memory Unsafety

	By Project	By LoC		
Only memory-safe language	82	48%	141,167,614	45%
Any memory-unsafe language	90	52%	173,588,291	55%
Total	172		314,755,905	

Table 3 and Figures 1 and 2 describe the interquartile ranges for total and memory-unsafe LoC by project. Note that the ranges are calculated independently for each column (e.g., the project with the maximum Total LoC is not necessarily the same as the project with the maximum Unsafe ratio). The largest projects (Chromium, the Linux kernel, gecko-dev, kvm, and linux-yocto-contrib) contain 25 million or more LoC, much of which is written in memory-unsafe languages. Most projects are smaller than one million LoC. These characteristics create the skewed distribution and compressed interquartile ranges visible in Figure 1. Figure 2 limits the maximum vertical axis value to 3 MLoC to zoom in on the interquartile ranges.

Table 3: Interquartile Ranges

	Total LoC	Unsafe LoC	Unsafe Ratio
Minimum	234	0	0%
Q1	82,498	0	0%
Q2 (median)	310,495	25	0%
Q3	1,012,546	136,124	53%
Maximum	34,677,183	24,721,815	99%

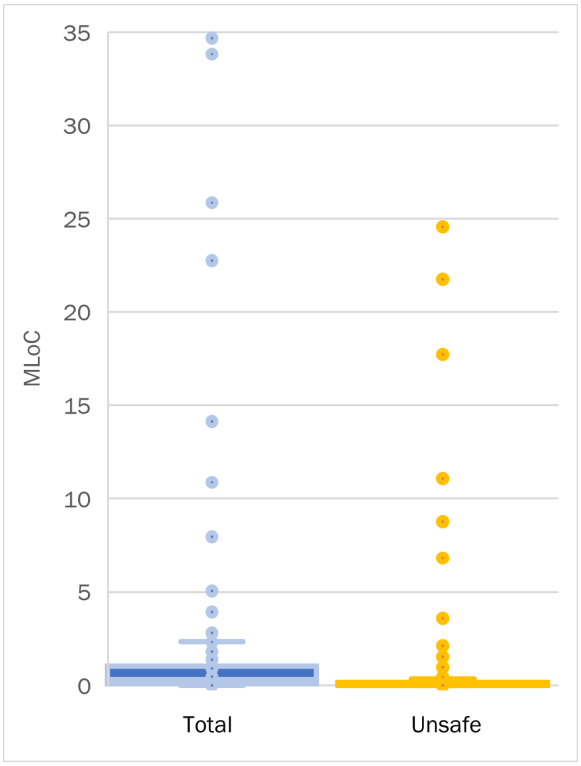

Figure 1: Interquartile Ranges

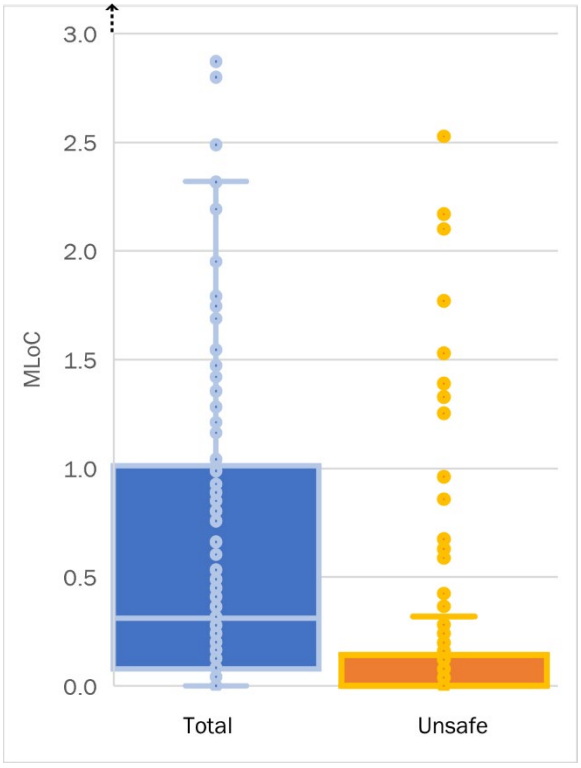

Figure 2: Interquartile Ranges, Y Axis Limited to 3 MLoC

Figures 3 and 4 compare project size (in MLoC) to the proportion of memory-unsafe code used in each project. Figure 4 limits the maximum vertical axis value to 3 MLoC to zoom in on smaller projects. Note that the locations of project name labels in Figures 3 and 4 are approximate.

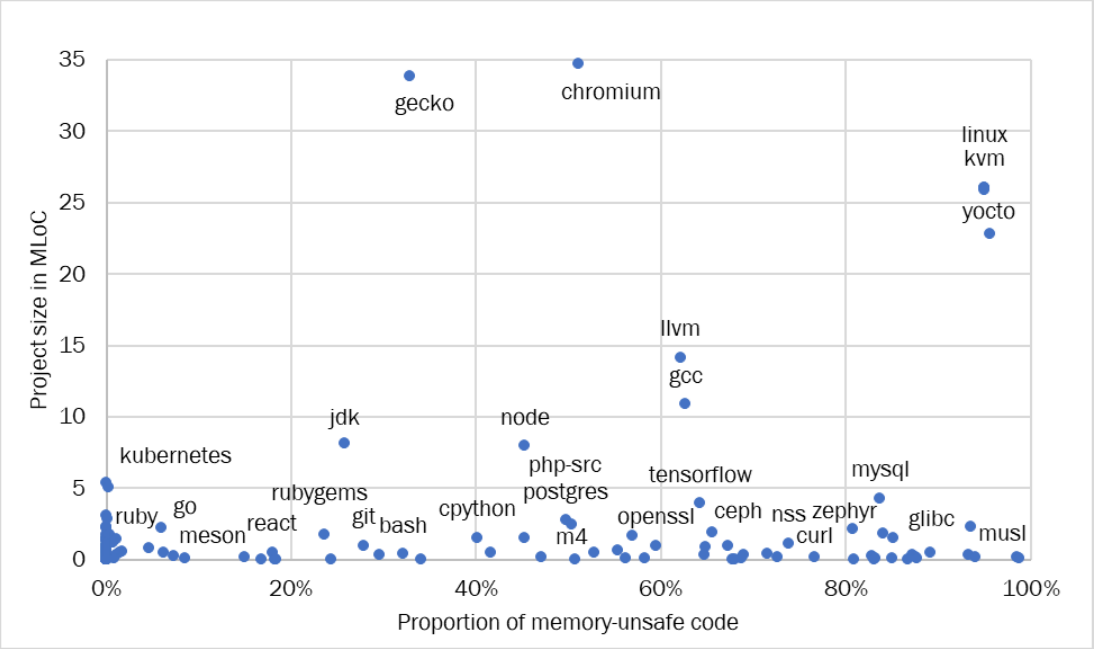

Figure 3: Comparison of Project Size to Memory-Unsafety

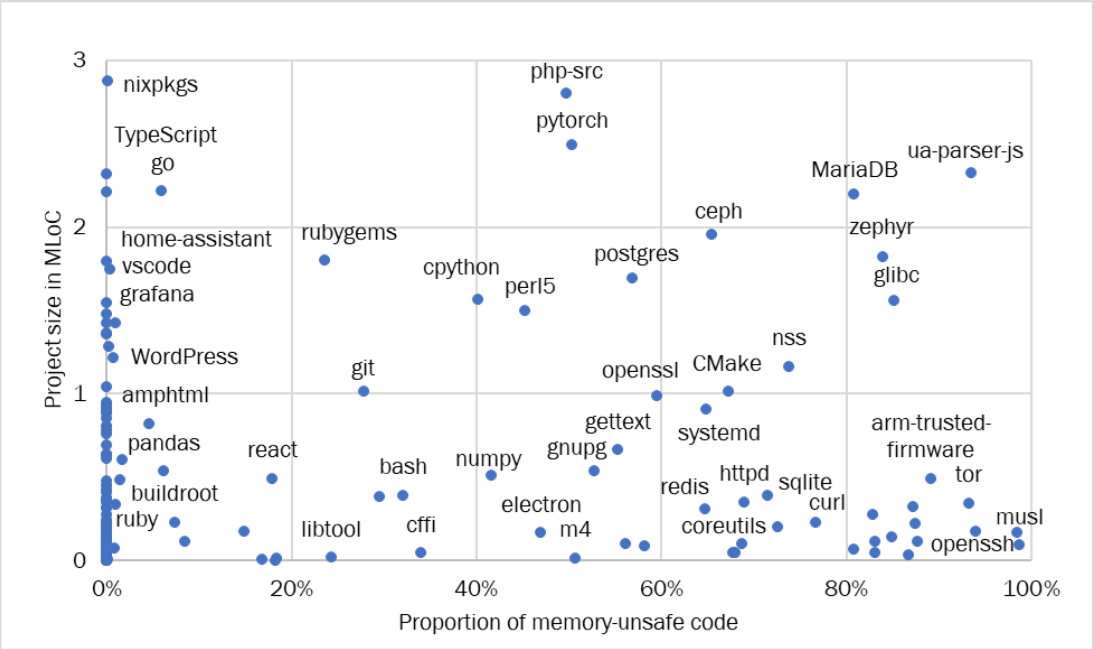

Figure 4: Comparison of Smaller Project Sizes to Memory-Unsafety, Y Axis Limited to 3 MLoC

Most of the largest projects (including the Linux kernel) predominantly use memory-unsafe languages. Some other large projects (Chromium and Gecko web browser frameworks) use memory-unsafe languages for roughly half of their code. This is expected since operating systems and web browsers perform functions, such as providing multiple code execution environments and managing memory directly, that have traditionally been difficult to implement in fully memory-safe ways.

Table 4 shows the count and distribution of projects based on the percentage of code in each project that is written in a memory-unsafe language.

Table 4: Proportional Memory-Unsafe Language Use

Percentage of memory-unsafe language use	Number of projects	Proportion of all projects	Proportion of memory-unsafe projects
0%	82	48%	
> 0%, ≤ 25%	31	18%	34%
> 25%, ≤ 50%	12	7%	13%
> 50%, ≤ 75%	23	13%	26%
> 75%, ≤ 95%	20	12%	22%
> 95%, ≤ 100%	4	2%	5%

Dependencies

The initial LoC analysis did not consider dependencies because dependency analysis at scale is complex. Different projects and languages have different, and sometimes multiple, ways to specify dependencies. The computation time required to identify and analyze dependencies increases substantially compared to analyzing the project alone. Due to the effort required to perform comprehensive dependency analysis on all the projects, we selected three: Ansible, Distribution, and Home Assistant.

Of the three projects, only Distribution directly contained any memory-unsafe code. Distribution contained approximately ten thousand lines of code (10 KLoC), making up 1.72% of the total LoC. This code is part of a Go module dependency that is included as part of the Distribution repository for cryptography and low-level interactions with operating systems.[26] The inclusion of dependencies bundled with a project source repository further complicates the comparison of "project only" to "project with all dependencies."

[26] https://pkg.go.dev/golang.org/x/sys, https://github.com/distribution/distribution/tree/main/vendor/golang.org/x/sys/unix.

We evaluated multiple source code dependency analysis tools and primarily used It-Depends, ScanCode Toolkit, CMake, and—for visualization–Graphviz.[27,28,29,30] These tools examine source code repositories, sometimes relying on included dependency information, and sometimes analyzing source code directly. For example, It-Depends looks for requirements.txt or setup.py but does not analyze import statements.[31] To understand and confirm the tool results we performed manual analysis, primarily using mechanisms inherent to the languages (such as `pip show` and `requirements.txt` for Python and `go list -m all` for go). We also limited the analysis to dependencies specified by the project in some way. We did not include development or build requirements, operating system libraries, or application programming interfaces (APIs).

The tools typically output lists of dependencies, their relationships, and sometimes the type of dependency. Output is usually graph information in a variety of formats such as JSON, comma-separated values, or Graphviz DOT.[32] For each dependency, we downloaded the source code, performed the "cloc" tool analysis to measure use of unsafe languages, and compared the LoC counts for each project by itself to the project with all dependencies included, as shown in Table 5.

Table 5: Selected Projects and Their Dependencies

Project (language)	Project only			Project and dependencies			
	Total KLoC	Unsafe KLoC	Ratio	Dependency count	Total KLoC	Unsafe KLoC	Ratio
Ansible (python)	171	0	0%	8	693	19	2.7%
Distribution (go)	600	10	1.72 %	283	18,001	227	1.3%
Home Assistant (python)	1,792	0	0%	44	2,379	44	1.9%

Superficially, the projects alone contain very little memory-unsafe code, but each has dependencies that do contain memory-unsafe code. Cursory analysis shows that dependencies using unsafe languages commonly provide features such as interfaces to memory-unsafe languages (typically C), graphics, cryptography, and compression.

[27] https://github.com/trailofbits/it-depends.
[28] https://github.com/nexB/scancode-toolkit.
[29] https://cmake.org/.
[30] https://graphviz.org/.
[31] https://github.com/trailofbits/it-depends/blob/master/it_depends/pip.py.
[32] https://graphviz.org/doc/info/lang.html.

For the purpose of our initial dependency analysis, the tools worked reasonably well for languages that explicitly support dependencies (for example, `requirements.txt` for Python, `pom.xml` for Java Maven, `package.json` for `Node.js` npm, `go.sum` for go). Some complexity remains, however, because even languages with explicit dependency specifications may have multiple ways to create dependencies. In Python, for example, import statements may create dependencies on modules that are not included in the standard Python library or a specific Python distribution.[33] Both `requirements.txt` and `install_requires` (setuptools) are methods to specify dependencies needed to install a Python project. All three methods may be necessary to analyze dependencies.

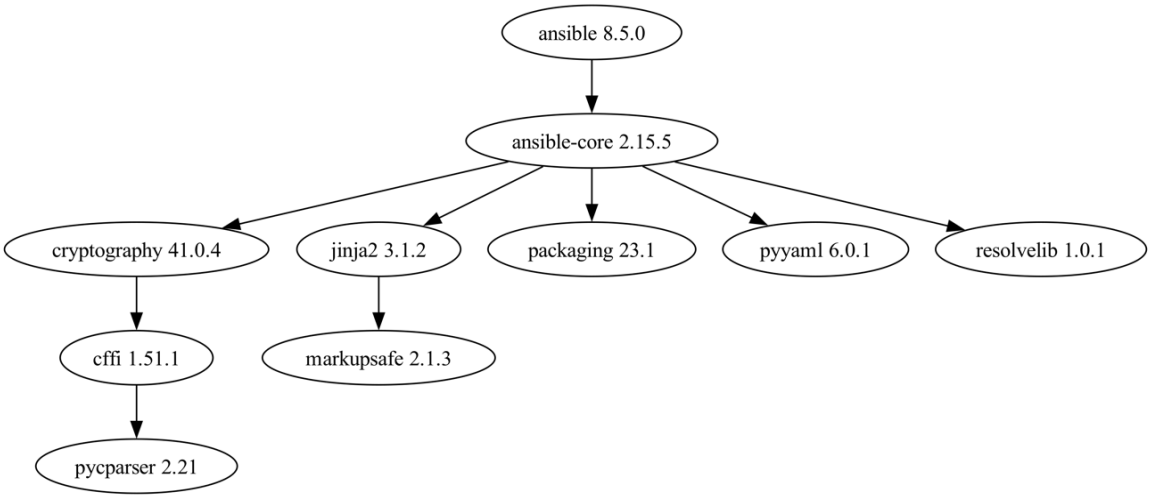

Figure 5: Ansible Dependencies

Graphs are a common way to visualize dependencies, subject to practical limitations on the size of the graph. Ansible has eight effective dependencies (the lowest number of the projects analyzed). Its dependencies are displayed in Figure 5.

The graph in Figure 5 is limited to explicitly declared Python package dependencies, effectively the results of `pip show`. A cursory examination of the Python cryptography package reveals additional static dependencies on a number of Rust crates and OpenSSL (written primarily in C).[34] We did not attempt to analyze the extent of memory unsafety introduced by the Rust or OpenSSL dependencies. Nonetheless, this example further illustrates the challenges of dependency analysis, specifically that multiple methods and abstractions are necessary to fully resolve dependencies and that otherwise memory-safe code can include memory-unsafe dependencies that use different programming languages.

[33] A custom analysis of Python dependencies: https://www.hudsonrivertrading.com/hrtbeat/dependency-graph-python-codebase/.
[34] https://cryptography.io/en/latest/faq/#why-does-cryptography-require-rust and https://cryptography.io/en/latest/openssl.

Discussion

We observed that many critical open source projects are partially written in memory-unsafe languages and limited dependency analysis indicates that projects inherit code written in memory-unsafe languages through dependencies.

Where performance and resource constraints are critical factors, we have seen, and expect the continued use of, memory-unsafe languages. Examples include operating system kernels and drivers, cryptography, and networking, particularly in embedded applications. It may, however, be an effective security investment to transition these types of projects to memory safe languages, and new projects should also consider using memory safe languages. Recent advancements allow memory safe programming languages, such as Rust, to parallel the performance of memory-unsafe languages.[35]

Even when using memory-safe languages, developers may disable memory-safety features. In their response to the Request for Information on Open Source Software Security, Amazon Web Services (AWS) included a recommendation to "write new code in memory safe languages like Rust, but note that not all code written in nominally memory-safe languages is actually memory-safe." AWS cites two studies that support a comment that "…it's easy and fairly common (and often reasonable) for developers to disable the compiler features that make Rust memory safe."[36]

Our analysis explored the challenges of assessing memory safety at scale, especially with limited resources for dependency analysis. Complete dependency analysis is difficult, possibly intractable, for at least two major reasons. First, languages often have multiple mechanisms to specify or create dependencies. In some cases, and more commonly among memory-safe languages, there are explicit mechanisms to define dependencies for a project. As noted in a previous example, Python dependencies can be created using import statements and specified using `requirements.txt` and `install_requires/setup.py` (setuptools). The second, related issue, is the computational cost of comprehensively and recursively identifying dependencies to some limit. This report only examined dependencies for a few selected projects and limited the recursion depth to what was readily available using language-specific dependency mechanisms.

Obstacles to achieving full memory safety are fundamental to how computers work. Source code must be compiled or interpreted to execute as native code on a given operating system and hardware platform. Somewhere underneath every programming language stack and dependency graph, memory-unsafe code is written and executed. Despite these fundamental issues, using memory-safe languages clearly reduces memory-safety vulnerabilities. Memory-safe languages shift the abstraction layer and responsibility for

[35] The Computer Language Benchmarks Game https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html.
[36] Comment from Amazon Web Services https://www.regulations.gov/comment/ONCD-2023-0002-0082.

writing memory-safe code from the developer to the compiler or interpreter, decreasing reliance on fallible humans.

This analysis is primarily limited by data selection issues associated with both the choice of projects and the lack of systemic dependency resolution. Improved project selection and dependency resolution would offer increased accuracy and greater insight into the use of memory-unsafe languages but would also require significant tooling and analysis effort. Analyzing closed-source or proprietary software would require different techniques and additional effort but would grant insight into otherwise opaque software. A more extensive dataset, especially one designed to include important projects that are not hosted on centralized platforms, may provide a more representative sample of OSS. Additional analysis would be required to estimate the costs of rewriting existing projects and the future benefits associated with reducing the number of memory-safety vulnerabilities.

We encourage others to build on this analysis to further expand our collective understanding of memory-unsafety risk in OSS, evaluate approaches—such as targeted rewrites of critical components in memory-safe languages—to reducing this risk, and to continue efforts to drive risk-reducing action by software manufacturers. For those considering further investment in memory safe programming practices, we recommend two references: The Case for Memory Safe Roadmaps and the December 2023 report on memory safety by the Technical Advisory Council of CISA's Cybersecurity Advisory Committee.[37]

Disclaimer

CISA, FBI, ASD's ACSC, and CCCS do not endorse any commercial entity, product, company, or service, including any entities, products, or services mentioned within this document. Any reference to specific commercial entities, products, processes, or services by service mark, trademark, manufacturer, or otherwise, does not constitute or imply endorsement, recommendation, or favoring by CISA, FBI, ASD's ACSC, or CCCS.

[37] CISA Cybersecurity Advisory Committee Technical Advisory Council, "Report to the CISA Director," December 2023. https://www.cisa.gov/sites/default/files/2023-12/CSAC_TAC_Recommendations-Memory-Safety_Final_20231205_508.pdf.

Annex

Table 6 contains the list of projects and basic KLoC statistics.

Table 6: Summary Project List and Data Set

Project	Total KLoC	Unsafe KLoC	Ratio
chromium	34,677	17,718	51%
gecko-dev	33,821	11,084	33%
kvm	26,024	24,722	95%
linux	26,023	24,721	95%
linux-yocto-contrib	25,860	24,570	95%
linux-yocto	22,765	21,751	96%
llvm-project	14,133	8,775	62%
gcc	10,874	6,814	63%
jdk	8,172	2,101	26%
node	7,963	3,602	45%
bitwarden-clients	5,393	0	0%
kubernetes	5,039	9	0%
mysql-server	4,289	3,589	84%
tensorflow	3,932	2,527	64%
DefinitelyTyped	3,141	0	0%
nixpkgs	2,872	5	0%
php-src	2,798	1,391	50%
pytorch	2,489	1,254	50%
ua-parser-js	2,320	2,169	94%
TypeScript	2,316	0	0%
go	2,213	133	6%
magento2	2,206	0	0%
MariaDB_server	2,192	1,771	81%
ceph	1,951	1,278	66%

Project	Total KLoC	Unsafe KLoC	Ratio
zephyr	1,822	1,530	84%
rubygems	1,802	425	24%
home-assistant	1,792	0	0%
Rust	1,747	6	0%
Postgres	1,689	962	57%
cpython	1,566	629	40%
glibc	1,560	1,328	85%
vscode	1,545	0	0%
perl5	1,494	676	45%
grafana	1,474	0	0%
gradle	1,421	15	1%
spark	1,421	0	0%
symfony	1,362	0	0%
solr	1,356	0	0%
flutter	1,284	4	0%
kata-containers	1,212	8	1%
nss	1,164	859	74%
WordPress	1,043	0	0%
CMake	1,014	682	67%
git	1,012	282	28%
openssl	989	588	60%
cli	949	0	0%
material-ui	928	0	0%
systemd	906	588	65%
keycloak	905	0	0%
salt	900	0	0%
drupal	889	0	0%

Project	Total KLoC	Unsafe KLoC	Ratio
PrestaShop	886	0	0%
Signal-Desktop	851	0	0%
amphtml	817	38	5%
three.js	804	0	0%
gatsby	779	0	0%
cassandra	759	0	0%
angular	691	0	0%
gettext	662	366	55%
babel	639	0	0%
PowerShell	628	0	0%
vault	609	0	0%
distribution	604	10	2%
gnupg	537	284	53%
pandas	534	33	6%
numpy	508	212	42%
react-native	492	88	18%
arm-trusted-firmware	491	438	89%
Signal-iOS	486	7	1%
guava	479	0	0%
Signal-Android	450	0	0%
bitwarden-server	425	0	0%
tomcat	410	0	0%
bash	390	125	32%
sqlite	389	278	72%
julia	384	113	30%
rabbitmq-server	378	0	0%
rails	370	0	0%

Project	Total KLoC	Unsafe KLoC	Ratio
resolvelib	365	0	0%
httpd	351	242	69%
homebrew-core	349	0	0%
tor	343	320	93%
poky	338	3	1%
grub	322	281	87%
joomla-cms	318	0	0%
puppet	314	0	0%
redis	307	199	65%
haproxy	278	230	83%
cryptography	272	0	0%
yocto-kernel-cache	241	0	0%
storybook	232	0	0%
buildroot	231	17	7%
kong	230	0	0%
curl	226	173	77%
framework	222	0	0%
busybox	220	192	88%
logging-log4j2	214	0	0%
jackson-databind	203	0	0%
mbedtls	202	147	73%
webpack	183	0	0%
zookeeper	177	26	15%
libarchive	172	162	94%
pip	172	0	0%
ansible	171	0	0%
nginx	166	163	99%

Project	Total KLoC	Unsafe KLoC	Ratio
electron	165	78	47%
ant-design-charts	158	0	0%
brew	155	0	0%
openssh-portable	142	120	85%
traefik	139	0	0%
ruby	127	0	0%
bootstrap	118	0	0%
openvpn	112	98	88%
zstd	111	92	83%
meson	111	9	9%
maven	111	0	0%
jackson-core	106	0	0%
commons-lang	105	0	0%
Signal-Server	105	0	0%
pro-components	103	0	0%
libpng	102	57	56%
coreutils	98	67	69%
async	93	0	0%
homebrew-cask	92	0	0%
musl	91	90	99%
mobile	91	0	0%
mosquitto	90	52	58%
automake	85	0	0%
httpcomponents-core	84	0	0%
httpcomponents-clien	77	0	0%
logback	77	0	0%
libsignal	72	1	1%

Project	Total KLoC	Unsafe KLoC	Ratio
reprepro	65	52	81%
ant-design	60	0	0%
commons-io	53	0	0%
ant-design-mobile	49	0	0%
memcached	48	32	68%
rekor	48	0	0%
libjpeg	44	37	83%
zlib	44	30	68%
cffi	44	15	34%
autoconf	42	0	0%
make	34	30	87%
commons-codec	32	0	0%
cosign	32	0	0%
readable-stream	32	0	0%
lodash	23	0	0%
libtool	18	4	24%
jinja	18	0	0%
fulcio	17	0	0%
u-boot	17	0	0%
slf4j	16	0	0%
sigstore-java	15	0	0%
yocto-kernel-tools	15	0	0%
m4	12	6	51%
pycparser	11	2	18%
isarray	10	0	0%
pyyaml	8	1	17%
packaging	8	0	0%

Project	Total KLoC	Unsafe KLoC	Ratio
inherits	8	0	0%
ant-design-pro	7	0	0%
community	7	0	0%
sigstore-python	6	0	0%
qs	4	0	0%
natives	3	0	0%
string_decoder	3	0	0%
markupsafe	2	0	18%
coa	1	0	0%
kind-of	1	0	0%
minimist	1	0	0%
rc	0	0	0%
safe-buffer	0	0	0%