Systematic Literature Review: C/C++ Memory Safety Advances And Their Impact On Embedded Systems

Sean Lee
seanlee@u.boisestate.edu
Boise State University

Abstract

The issue of memory safety in C/C++ is an issue present since the birth of the languages, and unlikely to find an ideal solution soon. Many approaches to solving this problem have been proposed, but with the increase in attention to the memory safety issue as well as increasing awareness of embedded systems in industries such as critical infrastructure and IOT, the importance of a solution has become more pressing.

This report looks to provide a systematic literature review of the current area of research, targeting several research questions and their impact on embedded systems. The current research trends indicate that 32 bit architectures, which are very common in embedded systems, offer the most promising areas for continued research as very few of the current proposed solutions consider them. Additionally, hardware based solutions are the most common, potentially due to their much increased performance compared to software solutions, though only two hardware solutions have reached physical prototypes.

Introduction

C and C++ are well known general purpose programming languages that have found use in many areas of computer science. They are also well known for the difficulty and error prone nature of their memory management, which often leads to memory safety issues. Memory safety is generally classified into two categories, spatial and temporal. Spatial memory safety refers to ensuring a pointer only accesses memory in the region for which it is intended. A common example is accessing memory outside of the bounds of an array. Temporal memory safety refers to ensuring a pointer is only accessed when it is valid. A common example is use after free, when a pointer accesses a chunk of memory after that memory has been free'd and given back to the memory allocator.

Microsoft and Google have both reported that ~70% of the serious security bugs they encounter are memory safety issues [1], [2]. CISA (Cybersecurity and Infrastructure Security Agency) and the FBI are now encouraging providers of critical infrastructure software to develop a roadmap to using memory safe languages in their software products by the end of 2025 [3]. An area of additional concern is in embedded systems, devices that do not run general purpose computers, but rather microcontrollers with minimal memory and processing power. These devices are often programmed in C/C++, and are more difficult to maintain and update due to their minimal resource nature.

Rust is commonly recommended as a memory safe alternative to C/C++ due to its inherent safety provided at compile time. However, Rust is not always a feasible solution for application in embedded systems. Legacy devices still active in the field would require substantial rewrites to their codebases. Real time operating systems (RTOS), device drivers provided by chip manufacturers, file system implementations, and various other non-application specific code would all require re-writes. Lastly, embedded systems due to their nature are more resource constrained, and so any solution that increases C/C++ memory safety requires analysis of the impact to both runtime and memory usage.

This systematic literature review aims to provide a comprehensive overview of the current approaches to increasing memory safety of C/C++ languages and identify areas for further research targeting embedded systems. This report makes the following contributions:

- An overview of the approaches in the current research.
- Discussion of the types of performance measurements currently used.
- Specific trends that impact embedded systems.

Method

This project uses the Kitchenham and Charters methodology for conducting a systematic literature review to define the overall process of the review [4]. Their method breaks the review into three phases; planning, conducting, and reporting the review.

Planning the review involved development of research questions and selection criteria for source material. The research questions were developed with the aim of determining the current state of C/C++ memory safety enhancements and their application to embedded systems. Conducting the review involved selecting and analyzing appropriate research material from two academic databases based on the research criteria defined. Reporting the review involved summarizing these research artifacts, their input to the research questions, and identifying overall gaps in the current research. Each of these sections is further described below.

Planning the Review

Research Question #1

What are the current research methods to improve C/C++ memory safety?

This question aims to answer where the current focus of research is, as well as look for potential areas of opportunities that are seeing less focus.

Research Question #2

How is memory usage and code performance impacted? How well do the various approaches solve the underlying problem?

Embedded systems are often memory constrained and require real time performance. Memory safety additions that impact either of these aspects must be well understood. How well do the solutions work and how much change in source code do they require?

Research Question #3

What development toolchains and hardware are supported?

Embedded systems often require special toolchains to compile, link, and output binary files specific to the target hardware. Additionally, what target hardware, which is the embedded system device that ultimately runs the application, is supported by these proposals?

Conducting the Review & Dataset

Data for this review is drawn from ACM and IEEE research databases. Additionally, sources must meet the following criteria:

- Must be from 2019 or newer.
- Should discuss improvements to C/C++ languages, development workflows, compilers for these languages, or hardware that aims to improve the memory safety of these languages.
- Sources that focus on languages that are memory safe by design, e.g. Rust, will not be considered.

Table 1 shows the database, search query, and number of results for each of the databases used.

Once the first set of sources were collected they were then screened based on the developed selection criteria. 23 sources were filtered as duplicates and 22 sources were filtered for relevance. After initial screening, each source's abstract and introduction sections were further reviewed for relevance. 14 sources were removed as part of this secondary screening. Most of the sources screened for relevance were either targeting memory safe languages (Rust), or were only considering memory safety as a secondary concern. Figure 1 shows the selection process.

Through this screening process, 22 sources were selected for full analysis as part of this literature review.

Database: Query	**Number of results**
ACM: Abstract:("memory safety" AND "embedded systems")	9 results
ACM: Abstract:("memory safety" AND ("C/C++" OR "C++/C"))	28 results
ACM: Abstract:("memory safety" AND ("C/C++" OR "C++/C") AND hardware)	12 results
IEEE: ("All Metadata":"memory safety") AND ("All Metadata": "embedded systems")	9 results
IEEE: ("All Metadata":"memory safety" AND ("All Metadata":”C/C++” OR "All Metadata:": "C++/C"))	21 results

Table 1: Research queries and number of results used for the survey.

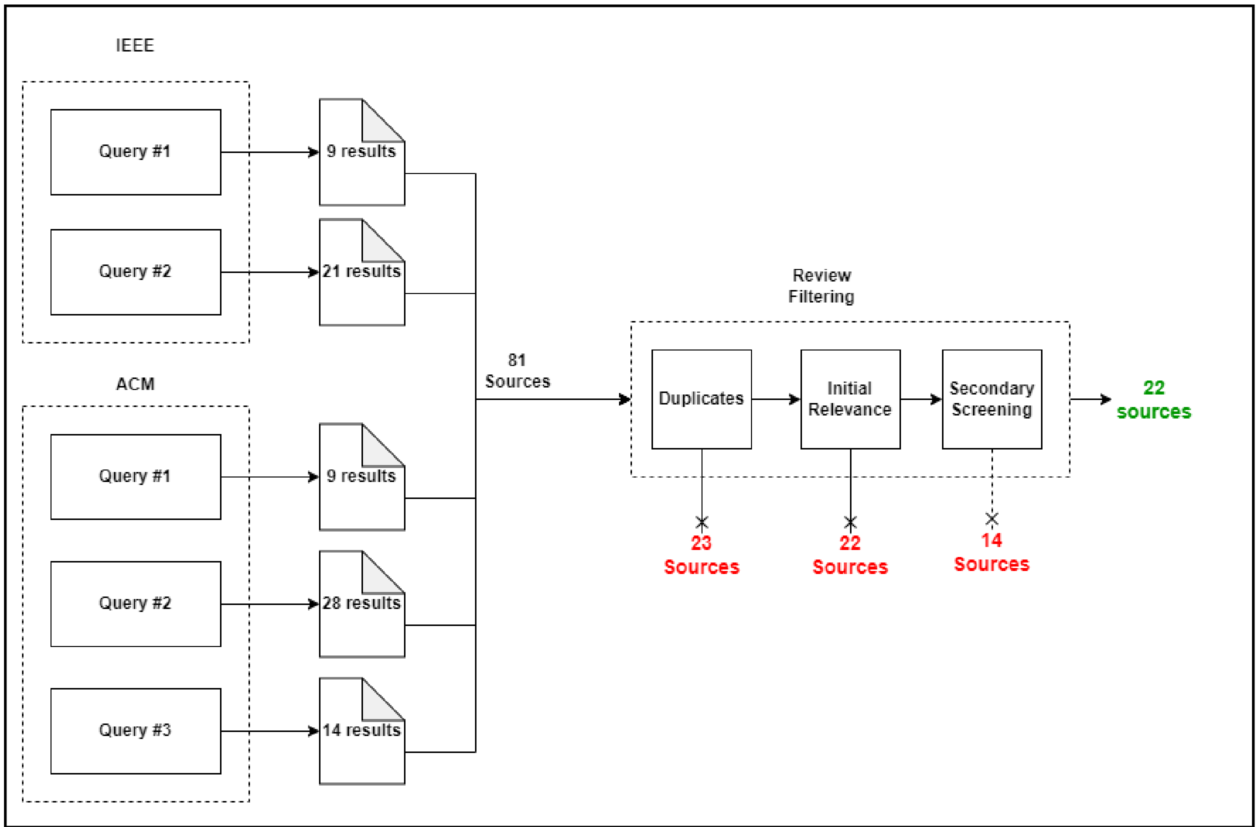

Figure 1: Filtering of research sources for final review

Reporting the Review and Results

Research Question 1

The 22 research sources surveyed can be grouped into 4 different categories as shown in Table 2.

Hardware and compiler approaches include proposed changes to underlying hardware mechanisms such as instruction set architecture or coprocessors and require accompanying compiler support. Compiler and runtime solutions propose changes to compilers and runtimes (such as standard libraries). Software only approaches propose changes that can be accomplished purely through software instrumentation such as shadow memory and memory allocators. Other approaches include all sources that do not fit one of the above methods.

Approach	**Number, Percentage of Results**	**Sources**
Hardware and compiler	11, 50%	[6, 7, 13, 14, 15, 18, 19, 21, 24, 27, 29]
Compiler and runtime	5, 23%	[10, 11, 16, 17, 28]
Software only	3, 13.5%	[20, 22, 26]
Other	3, 13.5%	[8, 23, 25]

Table 2: Research approach breakdown by type

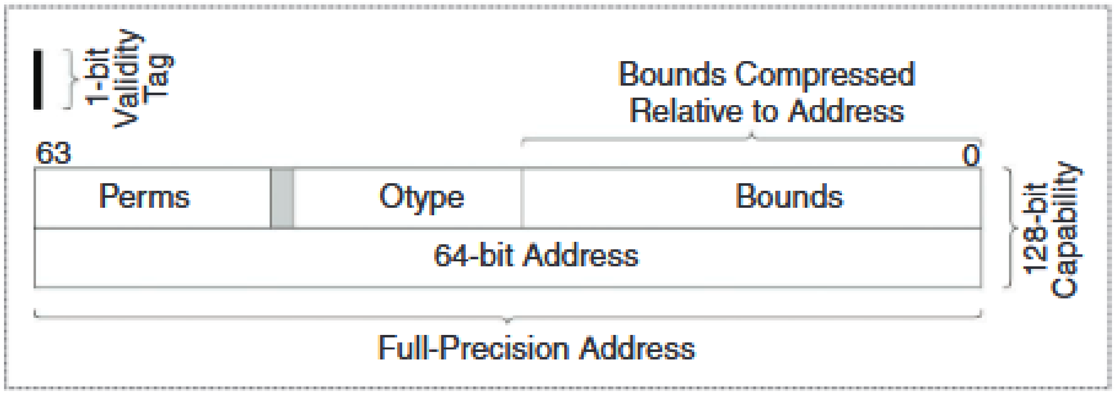

Figure 2: Cheri architecture pointer details showing pointer value and bounding information

One of the most interesting findings based on the above results is the prevalence of hardware based approaches. The hardware based approaches are a mix of custom FPGA implementations with two notable instances of devices seeing initial vendor hardware support. The Cheri project has partnered with Arm and Microsoft to develop the Morello and CherIOT devices respectively and a small set of development boards has been produced and shipped for further research [5]. The trend towards hardware research is potentially a side effect of software only implementations having large runtime and memory impacts.

Regardless of software, compiler, or hardware approaches, the majority of research falls into one of two categories:

1. Metadata stored within a pointer
2. Metadata stored outside of a pointer

The pointer metadata approach refers to using the unused bits of a pointer to store additional information. For example, in current 64 bit architectures the upper bits are generally not used for actually addressing memory. 2^{64} provides 18.4 exabytes of addressable memory which is far more than any current or likely future system could make use of. For this reason, the upper bits are not actually used in 64 bit architectures, so additional metadata can be stored there. Some research proposals, namely Cheri, increase the native size of the pointer as well, for example from 64 to 128 bits. Figure 2 shows the native pointer layout for the Cheri architecture [6]. Due to the increase in native pointer size, the full 64 bit address to be used remains unmodified. In the upper 64 bits information regarding the pointer's bounds, validity, and permission bits are stored. This additional data is used by a custom hardware coprocessor implementation to provide spatial and temporal safety. For example, when a memory address is accessed in Cheri, a hardware coprocessor first checks the bounding information stored in the enlarged 128 bit pointer to ensure the 64 bit address is contained in those bounds (spatial memory safety) and that the memory is still valid (temporal memory safety).

Another approach is to store information related to the pointer in a location other than the pointer itself. One implementation of this approach is in memory shading. **Figure 3** shows this approach as implemented by HWASanIO [7].

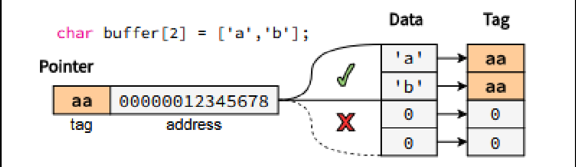

Figure 3: HWASanIO pointer showing memory shading

During allocation, a random tag is assigned to the area of memory being allocated. This tag is different from the tag associated with the memory directly before or after this new memory region. This tag is then added to the upper bits of a given pointer as well as “painted” across a corresponding region of shadow memory. When code dereferences a given pointer its tag is first extracted and checked against the tag stored in shadow memory for the target region of memory. If the two do not match, the memory access is not allowed because it has violated memory safety for the address in question. This approach is simple and provides spatial safety. However, it requires additional software instructions, additional shadow memory for tracking the tags, and brings into question the granularity for a given tag value. HWASan, from which HWASanIO is derived, implemented a 16 to 1 tag mapping [7]. That is to say, no fewer than 16 bytes of application data used a given tag . This limits the granularity that can be provided for intra-object safety.

Of the other approaches reviewed, one stands out. The paper by Yadav and Wilson discussed an approach to identify memory safety issues in code by training machine learning algorithms to detect memory safety violations [8]. Their research attempted to find a way to generate larger sets of data to assist in training these models, as the current data sets are rather small and manually developed. Their approach involved taking a known good software artifact, in their case Gentoo Linux, and instrumenting its source code with memory safety violations. In this way they developed a larger dataset with known good and bad code samples. An interesting aspect of their approach is the assumption that their input source code is free of memory safety violations to begin with. They argue the source code is large, widely used, and sufficiently mature to assume this. This area of research is quite interesting as it would not require hardware or software changes (except for bug fixes) for existing products, potentially minimizing impact to existing code bases.

Research Question 2

All sources except for 3 provided some amount of performance analysis, either runtime performance, memory usage impact, or a combination of both though performance impacts varied significantly across the research. Hardware approaches such as Cheri saw runtime overhead increases as small as 1.82% [6]. Softbound+CETS, a compiler based approach, saw performance overheads as high as 139% [22]. An additional finding regarding performance research is that many of the papers included some amount of analysis of the correctness of their solution via testing against known memory unsafe code. One common method was the Juliet test suite [9]. The Juliette test suite is a set of C/C++ test cases targeting 118 different common weakness enumerations (CWE). Of the sources surveyed, only 27% used the Juliet test suite for verifying correctness.

Only two of the sources used real applications as part of their performance testing as opposed to just benchmarking software [10,11]. Performance of a real application is a very subjective term, as one application's performance expectations may be very different from others. However, it is very interesting to see this emphasis as embedded systems often have real time requirements and lack hardware features that can hide performance problems from applications (caching, substantial memory, etc). The Mpchecker research used a suite of software programs including bison, ed, grep, gzip, redis, and average, showing anywhere from 2.1% - 68.31% overheard [10]. The "Look Before You Access" researchers used wasm, a javascript interpreter, a lua_basic interpreter, and psa crypto running in the RIOT operating system as their real world applications. This last set is particularly interesting as Lua, wasm, and especially psa crypto are often found in embedded systems. The researchers did not provide specific runtime overheads for this use case, rather they analyzed sizes of dynamic memory allocations as their research was focused on preventing memory safety for dynamic memory allocations.

Another area of impact is legacy code. That is to say, how much do pre-existing projects need to change in order to implement a given proposal. Any proposal that requires a compiler change at a minimum requires recompilation of source code and potentially integration of a new toolchain. Depending on the application of the final product this may incur minimal or significant testing and validation efforts. Variants that require custom hardware implementations will require board changes as well as compiler changes, which for embedded systems likely necessitates additional code change due to hardware peripherals changing independently. Ignoring this impact and focusing only on impacts to pre-existing code, variants such as Cheri require only minimal changes to code that directly access raw memory (malloc, memset, etc). The Cheri project reported only changing 1.4% of the entire FreeBSD kernel when porting it to their hardware, a fairly minimal change for such a large code set [6]. Follow on work to port the Linux kernel to Cheri architecture found the source code changes were constrained to low level memory access (malloc, memset, etc), interrupt handlers, assembly code, and some standard library optimized code [18]. Many approaches did not require custom hardware, but rather made use of memory features in newer chipsets such as those found in Armv8 architecture. These proposals would in theory only require recompilation of existing code. Compiler only approaches such as Mpchecker that relied on compiler changes require only recompilation, so long as the build system for the project in question already uses LLVM.

Research Question 3

Two trends emerge from the current research. The first is that solutions that require compiler changes overwhelmingly require LLVM support. LLVM is a compiler and toolchain project that provides front and backend tools for developing compilers [12]. This emphasis is likely due to its intentional design goal of being flexible. GCC's copyleft GPL license may also impact this decision, however that is not clear based on the current research.

The second trend is that every source with the exception of two requires or performs their research on a 64 bit architecture. Many of the sources specifically state using a more general purpose computer during their research. This has two interesting implications. The first is that performance measures may be skewed based on results from general purpose computing devices. Hardware features such as multiple cores, hardware threads, multiple levels of caches, and gigabytes of RAM that are present in these systems are not common in embedded systems. Secondly, many embedded systems are 32 bit architectures such as the STM32F4 and Nordic nRF54 series devices. These are certainly not the only embedded systems microcontrollers in use, but they are extremely popular and widely known, and therefore provide a good reference point. The only sources to specifically address a 32 bit solution were CherIOT and Efficient Pointer Integrity (EPI). CherIOT is part of the larger Cheri project [13]. EPI is an approach that supports 32 bit native pointer sizes by making use of tagging data that is inline with normal application data [14]. Both of these efforts require hardware changes. As mentioned previously, only 2 hardware based proposals have moved past a prototype FPGA implementation stage, one of which being CherIOT, the other being Arm Morello which also uses the Cheri architecture [5].

This focus on 64 bit architectures poses both a challenge and an opportunity. 32 bit embedded architectures may struggle with some of the performance implications of software based solutions (eg. address sanitization). Given that the current state of hardware research for 32 bit architectures has largely not moved past custom FPGA implementations, it's unlikely industry will implement these solutions as is today. This opens an area of further research, analysing non hardware based solutions in a 32 bit environment or further researching validity of 32 bit hardware based systems as they move towards initial production stages.

Boise State University Program Learning Outcomes

Courses provided in the Boise State University Computer Science Masters Program that directly impacted this project include Programming Language Translation (CS550), Operating Systems (CS552) and Advanced Operating Systems (CS554). Operating systems courses are directly applicable to this research due to being minimal resource, close to the hardware environments generally written in C, all traits shared by embedded systems. A large area of current research focuses on compiler driven solutions, requiring topics discussed in CS550.

This project satisfies the program learning outcomes of the masters program, especially the ability to engage in self directed learning required to research the current technical landscape, develop requirements for the SLR, conduct large amounts of research, and deliver written and oral presentations [30].

Conclusion

C/C++ are well known and widely used programming languages. Their memory safety flaws inherent to the language itself are unlikely to change soon, as is the large amount of legacy code written in these languages. Embedded systems are often written in C/C++ and are less resourced and more difficult to update due to their constrained nature. Any solution to the C/C++ memory safety issue in embedded systems should balance functionality with impact to existing code and systems to ensure success.

This systematic literature review attempts to provide a thorough survey of the current state of research in this area. Research questions that evaluate the overall approaches, performance impacts, current hardware support, and impacts to legacy code have been formulated and the selected research evaluated against.

The research shows that hardware/compiler based approaches are very common and that LLVM is the de facto choice for compiler research in this area. Despite a wide variety of research proposals, the two most common strategies for handling memory safety are either adding additional information to a pointer itself, or storing additional information about a pointer in other areas of memory. A notable area of opportunity is in 32 bit architectures, which are extremely common in embedded systems but only present in two sources reviewed in this paper. This presents an opportunity to focus research in this area. Two areas to pursue are analyzing performance impacts of software solutions in 32 bit architectures, and continuing analysis and viability of upcoming hardware solutions such as CherIOT.

References

[1] MSRC Team, “A proactive approach to more secure code” msrc.microsoft.com. https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/ (accessed 26 Jan 2025)

[2] The Chromium Project, “Memory Safety” www.chromium.org. https://www.chromium.org/Home/chromium-security/memory-safety/ (accessed 26 Jan 2025)

[3] Cybersecurity and Infrastructure Security Agency, “Product Security Bad Practices” https://www.cisa.gov/resources-tools/resources/product-security-bad-practices (accessed 25 Jan 2025).

[4] B. Kitchenham, “Procedures for Performing Systematic Reviews,” 2004. Accessed: 27 Jan 2025. Available: https://www.inf.ufsc.br/~aldo.vw/kitchenham.pdf

[5] R. Grisenthwaite, G. Barnes, R. N. M. Watson, S. W. Moore, P. Sewell and J. Woodruff, "The Arm Morello Evaluation Platform—Validating CHERI-Based Security in a High-Performance System," in IEEE Micro, vol. 43, no. 3, pp. 50-57, May-June 2023, doi: 10.1109/MM.2023.3264676.

[6] R. N. M. Watson et al., "CHERI: Hardware-Enabled C/C++ Memory Protection at Scale," in IEEE Security & Privacy, vol. 22, no. 4, pp. 50-61, July-Aug. 2024, doi: 10.1109/MSEC.2024.3396701.

[7] Konrad Hohentanner, Florian Kasten, and Lukas Auer. 2023. HWASanIO: Detecting C/C++ Intra-object Overflows with Memory Shading. In Proceedings of the 12th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis (SOAP 2023). Association for Computing Machinery, New York, NY, USA, 27–33. https://doi.org/10.1145/3589250.3596139

[8] A. S. Yadav and J. N. Wilson, "BOSS: A Dataset to Train ML-Based Systems to Repair Programs with Out-of-Bounds Write Flaws," 2024 IEEE/ACM International Workshop on Automated Program Repair (APR), Lisbon, Portugal, 2024, pp. 26-33, doi: 10.1145/3643788.3648013.

[9] National Institute of Standards and Technology. 2017. Juliet C/C++ 1.3. Retrieved April 7, 2025 from https://samate.nist.gov/SARD/test-suites/112

[10] W. Qiang, W. Li, H. Jin and J. Surbiryala, "Mpchecker: Use-After-Free Vulnerabilities Protection Based on Multi-Level Pointers," in IEEE Access, vol. 7, pp. 45961-45977, 2019, doi: 10.1109/ACCESS.2019.2908022.

[11] Jeonghwan Kang, Jaeyeol Park, Jiwon Seo, and Donghyun Kwon. 2024. Look Before You Access: Efficient Heap Memory Safety for Embedded Systems on ARMv8-M. In Proceedings of the 61st ACM/IEEE Design Automation Conference (DAC '24). Association for Computing Machinery, New York, NY, USA, Article 215, 1–6. https://doi.org/10.1145/3649329.3655949

[12] LLVM. The LLVM Compiler Infrastructure. Retrieved 7 April 2025 from https://llvm.org/

[13] S. Amar et al., "CHERIoT: Complete Memory Safety for Embedded Devices," 2023 56th IEEE/ACM International Symposium on Microarchitecture (MICRO), Toronto, ON, Canada, 2023, pp. 641-653.

[14] M. T. I. Ziad, M. A. Arroyo, E. Manzhosov, V. P. Kemerlis and S. Sethumadhavan, "EPI: Efficient Pointer Integrity For Securing Embedded Systems," 2021 International Symposium on Secure and Private Execution Environment Design (SEED), Washington, DC, USA, 2021, pp. 163-175, doi: 10.1109/SEED51797.2021.00028.

[15] Cyril Bresch, David Hély, Roman Lysecky, Stéphanie Chollet, and Ioannis Parissis. 2020. TrustFlow-X: A Practical Framework for Fine-grained Control-flow Integrity in Critical Systems. ACM Trans. Embed. Comput. Syst. 19, 5, Article 36 (September 2020), 26 pages. https://doi.org/10.1145/3398327

[16] Emanuel Q. Vintila, Philipp Zieris, and Julian Horsch. 2021. MESH: A Memory-Efficient Safe Heap for C/C++. In Proceedings of the 16th International Conference on Availability, Reliability and Security (ARES '21). Association for Computing Machinery, New York, NY, USA, Article 16, 1–10. https://doi.org/10.1145/3465481.3465760

[17] Andreas Hager-Clukas and Konrad Hohentanner. 2024. DMTI: Accelerating Memory Error Detection in Precompiled C/C++ Binaries with ARM Memory Tagging Extension. In Proceedings of the 19th ACM

Asia Conference on Computer and Communications Security (ASIA CCS '24). Association for Computing Machinery, New York, NY, USA, 1173–1185. https://doi.org/10.1145/3634737.3637655

[18] Kui Wang, Dmitry Kasatkin, Vincent Ahlrichs, Lukas Auer, Konrad Hohentanner, Julian Horsch, and Jan-Erik Ekberg. 2024. Cherifying Linux: A Practical View on using CHERI. In Proceedings of the 17th European Workshop on Systems Security (EuroSec '24). Association for Computing Machinery, New York, NY, USA, 15–21. https://doi.org/10.1145/3642974.3652282

[19] Michael B. Sullivan, Mohamed Tarek Ibn Ziad, Aamer Jaleel, and Stephen W. Keckler. 2023. Implicit Memory Tagging: No-Overhead Memory Safety Using Alias-Free Tagged ECC. In Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA '23). Association for Computing Machinery, New York, NY, USA, Article 67, 1–13. https://doi.org/10.1145/3579371.3589102

[20] Yiyu Zhang, Tianyi Liu, Zewen Sun, Zhe Chen, Xuandong Li, and Zhiqiang Zuo. 2023. Catamaran: Low-Overhead Memory Safety Enforcement via Parallel Acceleration. In Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023). Association for Computing Machinery, New York, NY, USA, 816–828. https://doi.org/10.1145/3597926.3598098

[21] Konrad Hohentanner, Philipp Zieris, and Julian Horsch. 2023. CryptSan: Leveraging ARM Pointer Authentication for Memory Safety in C/C++. In Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing (SAC '23). Association for Computing Machinery, New York, NY, USA, 1530–1539. https://doi.org/10.1145/3555776.3577635

[22] Benjamin Orthen, Oliver Braunsdorf, Philipp Zieris, and Julian Horsch. 2024. SoftBound+CETS Revisited: More Than a Decade Later. In Proceedings of the 17th European Workshop on Systems Security (EuroSec '24). Association for Computing Machinery, New York, NY, USA, 22–28. https://doi.org/10.1145/3642974.3652285

[23] G. Roascio, G. Serra and V. Eftekhari Moghadam, "Em-RIPE: Runtime Intrusion Prevention Evaluator for ARM Microcontroller Systems," 2022 International Conference on Electrical, Computer, Communications and Mechatronics Engineering (ICECCME), Maldives, Maldives, 2022, pp. 1-6, doi: 10.1109/ICECCME55909.2022.9988527.

[24] T. Nyman et al., "HardScope: Hardening Embedded Systems Against Data-Oriented Attacks," 2019 56th ACM/IEEE Design Automation Conference (DAC), Las Vegas, NV, USA, 2019, pp. 1-6.

[25] B. Novković, "A Taxonomy of Defenses against Memory Corruption Attacks," 2021 44th International Convention on Information, Communication and Electronic Technology (MIPRO), Opatija, Croatia, 2021, pp. 1196-1201, doi: 10.23919/MIPRO52101.2021.9596951.

[26] N. Wesley Filardo et al., "Cornucopia: Temporal Safety for CHERI Heaps," 2020 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 2020, pp. 608-625, doi: 10.1109/SP40000.2020.00098.

[27] Y. Kim, A. Kar, J. Lee, J. Lee and H. Kim, "Hardware-Assisted Code-Pointer Tagging for Forward-Edge Control-Flow Integrity," in IEEE Computer Architecture Letters, vol. 22, no. 2, pp. 117-120, July-Dec. 2023, doi: 10.1109/LCA.2023.3306326.

[28] X. Wang, B. Zhang, C. Tang and L. Zhang, "Highly Comprehensive and Efficient Memory Safety Enforcement with Pointer Tagging," 2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W), Brisbane, Australia, 2024, pp. 74-81, doi: 10.1109/DSN-W60302.2024.00026.

[29] R. Boivie, G. Saileshwar, T. Chen, B. Segal and A. Buyuktosunoglu, "Hardware Support for Low-Cost Memory Safety," 2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks - Supplemental Volume (DSN-S), Taipei, Taiwan, 2021, pp. 57-60, doi: 10.1109/DSN-S52858.2021.00032.

[30] Boise State University Computer Science Department, “Department of Computer Science PLOs” https://www.boisestate.edu/ie-assessment/resources/plos-college/coen/computer-science/ (accessed 24 Jan 2025)