Technical Report

UCAM-CL-TR-996
ISSN 1476-2986

Number 996

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

It is time to standardize principles and practices for software memory safety (extended version)

Robert N. M. Watson, John Baldwin, Tony Chen, David Chisnall, Jessica Clarke, Brooks Davis, Nathaniel Wesley Filardo, Brett Gutstein, Graeme Jenkinson, Christoph Kern, Ben Laurie, Alfredo Mazzinghi, Simon W. Moore, Peter G. Neumann, Hamed Okhravi, Alex Rebert, Alex Richardson, Peter Sewell, Laurence Tratt, Murali Vijayaraghavan, Hugo Vincent, Konrad Witaszczyk

February 2025

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

This material is based in part upon work supported by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under contract FA8750- 24-C-B047 ("DEC"), and in part upon work supported by the Under Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Department of Defense, Under Secretary of Defense for Research and Engineering, or the U.S. Government.

This work was supported in part by Innovate UK projects 105694 and 10027440, by EPSRC grants EP/V000292/1 ("CHaOS") and EP/V000373/1 ("CapableVMs"), by UKRI (ERC-AdG-2022 funding guarantee) grant EP/Y035976/1 ("SAFER"), and by ERC-AdG-2017 grant 789108 ("ELVER"). Additional support was received from Arm, Google, and Microsoft.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986
DOI *https://doi.org/10.48456/tr-996*

Abstract

This is the extended version of the paper, *It is time to standardize principles and practices for software memory safety*, which appeared in the February 2025 issue of Communications of the ACM.

In this report, we explore memory-safety standardization, which we argue is an essential step to promoting universal strong memory safety in government and industry, and, in turn, to ensure access to more secure software for all. Over the last two decades, a set of four research technologies for strong memory safety – memory-safe systems languages, hardware and software memory protection, formal approaches, and software compartmentalization – have reached sufficient maturity to see early deployment in security-critical use cases. However, there remains no shared, technology-neutral terminology or framework with which to specify memory-safety requirements.

This is needed to enable reliable specification, design, implementation, auditing, and procurement of strongly memory-safe systems. Failure to speak in a common language makes it difficult to understand the possibilities or communicate accurately with one another, limiting perceived benefits and hence actual demand. The lack of such a framework also acts as an impediment to potential future policy interventions, and, in turn, as an impediment to stating requirements to address observed market failures preventing adoption of these technologies. Standardization would also play a critical role in improving industrial best practice, another key aspect of adoption.

We begin with an overview of the many techniques – from hardware to software to formal theories – that have been developed and redefined over several decades, and how each plays a part in moving us towards strong memory safety. We explore how these technologies can be differentiated, considering both differences in functional protection and strength. We discuss how adoption barriers and potential market failures have limited adoption, and how the standardization gap limits potential interventions. We propose potential approaches to standardization – likely a task not limited to any one institution or standards body – and conclude with an illustrative universal memory-safety adoption timeline proposing a realistic path to universal adoption given suitable incentivization.

It is time to standardize principles and practices for software memory safety (extended version)

Robert N. M. Watson[1 2], John Baldwin[7], Tony Chen[5], David Chisnall[6], Jessica Clarke[1], Brooks Davis[3], Nathaniel Wesley Filardo[5 6], Brett Gutstein[1], Graeme Jenkinson[2], Christoph Kern[4], Ben Laurie[1 2 4], Alfredo Mazzinghi[2], Simon W. Moore[1 2], Peter G. Neumann[3], Hamed Okhravi[10], Alex Rebert[4], Alex Richardson[4], Peter Sewell[1], Laurence Tratt[8], Muralidaran Vijayaraghavan[4], Hugo Vincent[9], and Konrad Witaszczyk[1]

[1] University of Cambridge [2] Capabilities Limited [3] SRI International
[4] Google, Inc [5] Microsoft, Inc [6] SCI Semiconductor
[7] Ararat River Consulting [8] King's College London [9] Arm Limited
[10] MIT Lincoln Laboratory

Table of Contents

Introduction **7**
Background 8
Industrial best practices and market failure 12
Cost vs. assurance tradeoffs 14
Enabling business processes and market interventions 18
The memory-safety standardization gap **19**
Audiences for memory-safety standardization 20
Goals for memory-safety standardization 21
Potential structures for one or more standards or documents 23
Adoption narratives and timelines **24**
Potential events and interventions 25
Candidate timeline 26
Conclusion **28**
Acknowledgements **29**

Introduction

For many decades, endemic memory-safety vulnerabilities in software Trusted Computing Bases (TCBs) have enabled the spread of malware and devastating targeted attacks on critical infrastructure, national-security targets, companies, and individuals around the world. Over the last two years, the information-technology industry has seen increasing calls for the adoption of strong memory-safety technologies, framed as part of a broader initiative for *Secure by Design*, from government[1] [2] [3] [4], academia[5], and within the industry itself[6] [7]. These calls are grounded in extensive evidence that memory-safety vulnerabilities have persistently made up the majority of critical security vulnerabilities over multiple decades, and have affected all mainstream software ecosystems and products – and also the growing awareness that these problems are almost entirely avoidable by using recent advances in strong and scalable memory-safety technology.

In this report, we explore *memory-safety standardization*, which we argue is an essential step to promoting *universal strong memory safety* in government and industry, and, in turn, to ensure access to more secure software for all. Over the last two decades, a set of four research technologies for *strong memory safety* – memory-safe systems languages, hardware and software memory protection, formal approaches, and software compartmentalization – have reached sufficient maturity to see early deployment in security-critical use cases. However, there remains no shared, technology-neutral terminology or framework with which to **specify memory-safety requirements**. This is needed to enable reliable specification, design, implementation, auditing, and procurement of strongly memory-safe systems. Failure to speak in a common language makes it difficult to understand the possibilities or communicate accurately with one another, limiting perceived benefits and hence actual demand. The lack of such a framework also acts as an impediment to potential future policy interventions, and, in turn, as an impediment to stating requirements to address observed market failures preventing adoption of these technologies. Standardization would also play a critical role in improving industrial best practice, another key aspect of adoption.

[1] The White House, *Back to the Building Blocks: A Path Towards Measurable Security*, February 2024, https://bidenwhitehouse.archives.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf.
[2] CISA, NSA, FBI, ASD's ACSC, CCCS, NCSC-UK, NCSC-NZ, and CERT-NZ, *The Case for Memory Safe Roadmaps Why Both C-Suite Executives and Technical Experts Need to Take Memory Safe Coding Seriously*, December 2023, https://www.cisa.gov/sites/default/files/2023-12/The-Case-for-Memory-Safe-Roadmaps-508c.pdf.
[3] NSA, *Software Memory Safety*, Cybersecurity Information Sheet, April 2023, https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF.
[4] Department for Business, Energy & Industrial Strategy, *Confronting cyber threats to businesses and personal data*, October 2019, https://www.gov.uk/government/news/confronting-cyber-threats-to-businesses-and-personal-data.
[5] H. Okhravi, *Memory Safety*, IEEE Security & Privacy, Volume 22, Number 4, July-August 2024, https://ieeexplore.ieee.org/document/10621922.
[6] Alex Rebert and Christoph Kern, *Secure by Design: Google's Perspective on Memory Safety*, March 2024, https://storage.googleapis.com/gweb-research2023-media/pubtools/7665.pdf.
[7] Satya Nadella, *Prioritizing security above all else*, May 2024, https://blogs.microsoft.com/blog/2024/05/03/prioritizing-security-above-all-else/.

We begin with an overview of the many techniques – from hardware to software to formal theories – that have been developed and redefined over several decades, and how each plays a part in moving us towards strong memory safety. We explore how these technologies can be differentiated, considering both differences in functional protection and strength. We discuss how adoption barriers and potential market failures have limited adoption, and how the standardization gap limits potential interventions. We propose potential approaches to standardization – likely a task not limited to any one institution or standards body – and conclude with an illustrative *universal memory-safety adoption timeline* proposing a realistic path to universal adoption given suitable incentivization.

A shortened version of this report appeared as an article in the February 2025 issue of Communications of the ACM[8].

Background

For over two decades, memory-safety vulnerabilities have consistently made up around two thirds of critical security vulnerabilities in every major open-source and proprietary systems software TCB, including Windows,[9] Linux, Android, iOS, Chromium,[10] OpenJDK, vxWorks[11], FreeRTOS[12], and others. While many of the vulnerabilities were discovered, reported and fixed before they were potentially used to build successful attack vectors, studies show that this class of vulnerabilities is the foundation of many 0-day exploits observed in the wild[13] – and that these vulnerabilities sometimes continue to be present in unpatched (sometimes unpatchable) systems for years after they become known[14]. These problems primarily originate from an existing multi-billion line-of-code C/C++ code corpus that is difficult (probably impossible in practice) to entirely replace due to its scale. According to an industry estimate, it costs around $1 trillion dollars to rewrite one billion lines of code[15]. Of particular importance within this are the language runtimes of many type-safe and/or memory-safe programming languages, such as Java, JavaScript, and Python, which are often implemented in (or depend heavily on) C and C++. This can especially

[8] Robert N. M. Watson, John Baldwin, Tony Chen, David Chisnall, Jessica Clarke, Brooks Davis, Nathaniel Wesley Filardo, Brett Gutstein, Graeme Jenkinson, Ben Laurie, Alfredo Mazzinghi, Simon W. Moore, Peter G. Neumann, Hamed Okhravi, Alex Rebert, Alex Richardson, Peter Sewell, Laurence Tratt, Muralidaran Vijayaraghavan, Hugo Vincent, and Konrad Witaszczyk, *It is time to standardize principles and practices for software memory safety*, Communications of the ACM, Volume 68, Number 2, February 2025.

[9] David Weston, *Windows 11: The journey to security by default*, BlueHat IL, March 2023, slide 38: https://github.com/dwizzzle/Presentations/blob/master/David%20Weston%20-%20Windows%2011%20Security%20by-default%20-%20Bluehat%20IL%202023.pdf.

[10] Google, *Memory safety*, The Chromium Projects' documentation, Originally published 2020. https://www.chromium.org/Home/chromium-security/memory-safety/.

[11] Armis, *URGENT/11 Affects Additional RTOSs - Highlights the Risks on Medical Devices*, Originally published 2020, https://www.armis.com/research/urgent-11/.

[12] Zimperium, *FreeRTOS TCP/IP Stack Vulnerabilities – The Details*, December 2018, https://www.zimperium.com/blog/freertos-tcpip-stack-vulnerabilities-details/.

[13] Google, *The More You Know, The More You Know You Don't Know*, Project Zero, April 2021, https://googleprojectzero.blogspot.com/2022/04/the-more-you-know-more-you-know-you.html.

[14] CISA, *PRC State-Sponsored Actors Compromise and Maintain Persistent Access to U.S. Critical Infrastructure*, February 2024, https://www.cisa.gov/news-events/cybersecurity-advisories/aa24-038a.

[15] D. Wallach, TRACTOR Proposers Day Presentation (slide 26), August 2024, https://www.darpa.mil/research/programs/translating-all-c-to-rust.

present a problem for language runtimes that are routinely exposed to malicious code, such as JavaScript interpreters embedded in web browsers, where memory safety of the language itself does not translate to freedom from exploitable memory-safety vulnerabilities.

Memory-safety vulnerabilities are particularly important because, when combined with network communications or other malignant data, they can enable an attacker to escalate (via a multi-step exploit chain) to arbitrary code execution, operating outside the confines of the programming language[16]. These vulnerabilities have proven impossible to completely prevent with conventional engineering, and are especially dangerous because a single error (perhaps one line in a multi-million line-of-code system) is sufficient for an attacker to achieve total control of a vulnerable system.

Defensive techniques have not stood still – a series of incremental (and reactive) mitigation techniques have (in the short term) complicated work for attackers – but in the longer term these simply contributed to an evolving arms race with attack techniques that are able to bypass them[17]. Despite countless hours of manual source-code auditing, and significant investments in static analysis tooling and fuzzing, the rate of memory-safety vulnerabilities has remained roughly constant for over two decades.

Mitigation and sanitization techniques frequently fail in the longer term because they are **incomplete** (e.g., PAC[18] or CFI[19], which defend against only a narrow range of attack techniques, or a limited set of vulnerability types identifiable with specific static analysis tools) and/or because they are **probabilistic** (e.g., because they utilize secrets or keys that can be leaked or guessed, such as ASLR[20] or MTE[21]). These increasingly widely deployed techniques, which reflect *current industry best practice* in software TCBs, include:

[16] Haroon Meer, *Memory Corruption Attacks: The Almost Complete History*, BlackHat, June 2010.
[17] László Szekeres, Mathias Payer,Tao Wei, and Dawn Song, *SoK: Eternal War in Memory*, 2013 IEEE Symposium on Security and Privacy, Berkeley, CA, USA, May 2013.
[18] Arm, *Armv8.1-M PACBTI Extensions*, March 2024, https://developer.arm.com/documentation/109576/0100/Pointer-Authentication-Code/Introduction-to-PAC.
[19] Burow, Nathan, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Matthis Payer, *Control-Flow Integrity: Precision, security, and performance*, ACM Computing Surveys (CSUR) Volume 50, Issue 1, April 2017.
[20] Brad Spengler, *PaX: The Guaranteed End of Arbitrary Code Execution*, retrieved 4 February 2025, https://grsecurity.net/PaX-presentation.pdf.
[21] Arm, *Introduction to the Memory Tagging Extension (MTE)*, April 2024, https://developer.arm.com/documentation/108035/0100/Introduction-to-the-Memory-Tagging-Extension.

Table 1: Current industry best practice.

Category	Description	Examples
Development-time techniques (static and dynamic)	Static checking and automated dynamic bug finding	Coverity and Fortify; fuzzing combined with dynamic techniques such as Valgrind[22], ASAN[23], MSAN[24], and UBSAN[25] sanitizers; subsets of otherwise unsafe languages that can reduce exposure to memory-safety issues, such as MISRA C/C++[26]
Run-time techniques	Systems that handle violations of memory safety at run time, coercing them into fail stops, masking their effects, or limiting their exploitability	Software-only techniques such as stack canaries, ASLR and CFI, and also hardware-enabled techniques such as PAC, MTE, W^X (a.k.a. DEP), and architectural "safe stacks"[27]

Fortunately, the last decade has seen the maturation of practically deployable research technologies that have a realistic chance of breaking that arms race in favor of the defending side, introducing **strong memory safety** that **non-probabilistically prevents** a broad set of memory-safety vulnerabilities and attack techniques in critical software TCBs. Broadly, these technologies, now seeing *early industrial adoption*, fall into four categories:

Table 2: Strong memory-safety techniques.

Category	Description	Examples
Memory-safe and type-safe languages	Fully memory-safe and/or type-safe languages; statically checkable safe subsets of unsafe languages	Rust, Python, Swift, Java, C#, SPARK, and OCaml – excluding code in their unsafe TCBs (e.g., Unsafe Rust); memory-safe C++ subsets[28] [29]

[22] The Valgrind Developers, Valgrind, Retrieved 4 February 2025, https://valgrind.org/.
[23] The Clang Team, Clang 21.0.0 git documentation: AddressSanitizer, retrieved 4 February 2025, https://clang.llvm.org/docs/AddressSanitizer.html.
[24] The Clang Team, Clang 21.0.0 git documentation: MemorySanitizer, retrieved 4 February 2025, https://clang.llvm.org/docs/MemorySanitizer.html.
[25] The Clang Team, Clang 21.0.0 git documentation: UndefinedBehaviorSanitizer, retrieved 4 February 2025, https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html.
[26] MISRA, MISRA, Retrieved 4 February 2025, https://misra.org.uk/.
[27] Intel, *Control-flow Enforcement Technology Preview*, Document 334525-002, Intel, June 2017.
[28] LLVM Project, *C++ Safe Buffers*, retrieved December 2024, https://clang.llvm.org/docs/SafeBuffers.html.
[29] Sean Baxter and Christian Mazakas, *Safe C++*, September 2024, https://safecpp.org/draft.html.

Formal methods	Mathematically rigorous formal verification of memory safety, and broader safety / correctness properties for memory-safety TCBs themselves	Machine-checked formal proofs (e.g., using Coq[30], Isabelle[31], or Lean[32]), of systems such as CompCert[33] or seL4[34]; formal verification of unsafe language fragments (e.g., RustBelt[35]); tools for verification of C/C++ source code (e.g., CBMC[36], CN[37], Frama-C[38], or VeriFast[39])
Hardware memory protection	Systems that deterministically detect violations of memory safety at run time, coercing them into fail stops, masking their effects, or preventing their exploitation	CHERI C/C++ memory safety[40]

[30] The Coq Development Team, *The Coq Reference Manual*, retrieved January 2025, https://coq.inria.fr/distrib/current/refman/.

[31] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson, *Isabelle/HOL: a proof assistant for higher-order logic*, Springer-Verlag, 2002.

[32] Leonardo De Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer, *The Lean theorem prover (system description)*, in Automated Deduction-CADE-25: 25th International Conference on Automated Deduction, Springer International Publishing, August 2015.

[33] Xavier Leroy, *Formal verification of a realistic compiler*, Communications of the ACM Volume 52, Number 7, July 2009, https://doi.org/10.1145/1538788.1538814.

[34] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser, *Comprehensive formal verification of an OS microkernel*. ACM Transactions on Computer. Systems, Volume 32, Number 1, Article 2, February 2014. https://doi.org/10.1145/2560537.

[35] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer, *RustBelt: securing the foundations of the Rust programming language*, Proceedings of the ACM on Programming Languages, Volume 2 (POPL), Article 66, January 2018. https://doi.org/10.1145/3158154.

[36] Edmund Clarke, Daniel Kroening, and Flavio Lerda. 2004. *A tool for checking ANSI-C programs*. In Tools and Algorithms for the Construction and Analysis of Systems: 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29-April 2, 2004, Proceedings 10, Springer, 2004.

[37] Christopher Pulte, Dhruv C. Makwana, Thomas Sewell, Kayvan Memarian, Peter Sewell, and Neel Krishnaswami, *CN: Verifying systems C code with separation-logic refinement types*, Proceedings of the ACM on Programming Languages, Volume 7, Issue POPL, https://dl.acm.org/doi/10.1145/3571194.

[38] Patrick Baudin, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles, and Nicky Williams. 2021. *The dogged pursuit of bug-free C programs: the Frama-C software analysis platform*. Communications of the ACM, Volume 64, Number 8, August 2021. https://doi.org/10.1145/3470569.

[39] Pieter Philippaerts, Jan Tobias Mühlberg, Willem Penninckx, Jan Smans, Bart Jacobs, and Frank Piessens. 2014. *Software verification with VeriFast: Industrial case studies*. Science of Computer Programming, Volume 82, March 2014. https://doi.org/10.1016/j.scico.2013.01.006.

[40] Robert N.M. Watson, David Chisnall, Jessica Clarke, Brooks Davis, Nathaniel Wesley Filardo, Ben Laurie, Simon W. Moore, Peter G. Neumann, Alexander Richardson, Peter Sewell, Konrad Witaszczyk, and Jonathan Woodruff. *CHERI: Hardware-Enabled C/C++ Memory Protection at Scale*. IEEE Security & Privacy, Volume 22, Number 4, July-August 2024.

Software fault isolation and software compartmentalization	Systems that allow continued operation through privilege minimization despite effective exploitation of memory unsafety, limiting further rights and attack surfaces exposed to attackers. These systems are frequently built on the above techniques, and add the further ability to constrain attacks that have already achieved arbitrary code execution.	Deterministic sandboxing using processes or virtual machines as found in iOS and Android, software-only techniques such as eBPF[41] and WASM[42], OS compartmentalization such as HAKC[43], and hardware-enabled techniques such as CHERI compartmentalization

This work has not happened in isolation: concepts such as hardware memory protection and type-safe programming languages have existed almost since the inception of computer systems. However, these current technologies are incrementally adoptable within current hardware or systems software stacks, and their growing maturity comes alongside an increasingly critical need for memory safety.

Industrial best practices and market failure

Universally deployed strong memory safety enabled by new memory-safety protection technologies presents a remarkable opportunity. However, our excitement is tempered by the understanding that it will require a substantial change in approach by an industry that may see little economic incentive to change the status quo – and, in fact, the real risk of market disadvantage in doing so.

Today, common industrial practices consist of the widespread use of memory-unsafe languages and coding practices in even our most sensitive computing environments, albeit with some adoption of incomplete or probabilistic memory-safety mitigations such as those described in Table 1. Changes such as the widespread deployment of CHERI hardware, the Rust language, or formal verification are challenging in several ways. They would involve immediate (perceived or real) deployment or development costs, due to disruption of existing software ecosystems, and also (and more importantly) require vendors to potentially divert

[41] eBPF.io authors, *eBPF*, retrieved 4 February 2025, https://webassembly.github.io/spec/core/.
[42] Andreas Rossberg, *WebAssembly Specification (Release 2.0 draft 2025-01-28)*, retrieved 4 February 2025, https://ebpf.io/.
[43] Derrick McKee, Yianni Giannaris, Carolina Ortega, Howard Shrobe, Mathias Payer, Hamed Okhravi, and Nathan Burow, *Preventing Kernel Hacks with HAKCs*, Network and Distributed System Security symposium (NDSS), April 2022.

engineering resources from other areas of development – a high **opportunity cost**[44]. Some vendors find these costs difficult to justify when the immediate benefits of strong memory safety are not clearly expressed through market signals or when customer demand appears focused on other features.

This lack of incentive to address fundamental security flaws has fostered a large and profitable after-market security industry. This situation is reminiscent of the automotive industry's reliance on after-market kits necessary to fix flawed car designs before adequate safety regulations were in place[45]. In that era, just as in the software industry today, there was little economic motivation for manufacturers to proactively address safety issues. Instead, a secondary market emerged to patch the problems, often ineffectively. Similarly, in the software world, we see a proliferation of security add-ons and services that attempt to mitigate the risks of memory-unsafe code, rather than eliminating the root cause. These after-market solutions, while sometimes necessary, add complexity, increase costs, and expose us to additional significant safety risks themselves.[46] While this sector demonstrates that there is money to be made in addressing security vulnerabilities, it primarily focuses on reactive, after-the-fact solutions, rather than incentivizing proactive, secure-by-design development that would prevent these vulnerabilities from arising in the first place.

We suggest that the slow adoption of strong memory safety in spite of its clear security benefits may reflect a potential **market failure**[47]: society, as a whole, pays an extremely high cost for memory-safety vulnerabilities[48], as well as taking on a very high risk as these vulnerabilities are present in essentially all critical infrastructure, national security applications, and systems protecting financial and privacy-sensitive data. The history of catastrophic failure associated with these vulnerabilities can be traced at least as far back as the Morris Worm in 1988[49], and many recent examples include ransomware spread[50] or widespread denial of service[51] originating from memory-safety issues. Furthermore, the existence of a thriving after-market security sector does not negate this market failure; rather,

---

[44] National Academies of Sciences, *Engineering, and Medicine. Workshop on Secure Building Blocks for Trustworthy Systems*, panel discussion with Robert Watson and Richard Grisenthwaite, Seattle, Washington, USA, 31 July 2024, https://www.nationalacademies.org/event/43213_07-2024_workshop-on-secure-building-blocks-for-trustworthy-systems.

[45] Ralph Nader, *Unsafe at any speed: The designed-in dangers of the American automobile*, Grossman Publishers, New York, 1965.

[46] Sean Peisert, Bruce Schneier, Hamed Okhravi, Fabio Massacci, Terry Benzel, Mohammad Mannan, Jelena Mirkovic, Atul Prakash, and James Bret Micahel, *Perspectives on the SolarWinds Incident*, IEEE Security & Privacy, Volume 19, Number 2, March-April 2021.

[47] Emmanuel Kopp, Lincoln Kaffenberger, and Nigel Jenkinson, *Cyber risk, market failures, and financial stability*, International Monetary Fund, August 2017.

[48] Andy Greenberg, *Ransomware Payments Hit a Record $1.1 Billion in 2023*, Wired Magazine, February 2024, https://www.wired.com/story/ransomware-payments-2023-breaks-record/.

[49] U.S. v. Morris, 928 F.2d 504 (2d Cir. 1991). https://scholar.google.com/scholar_case?case=551386241451639668.

[50] NHS Digital, *WannaCry Ransomware Using SMB Vulnerability*, Originally published May 2017, https://digital.nhs.uk/cyber-alerts/2017/cc-1411.

[51] CISA, *Widespread IT Outage Due to CrowdStrike Update*, Originally published July 2024, https://www.cisa.gov/news-events/alerts/2024/07/19/widespread-it-outage-due-crowdstrike-update.

it is a symptom of it. The need for these add-on solutions highlights the underlying issue: the failure of the market to incentivize the production of secure software from the outset.

This analysis is consistent with many other past economic analyses of security, in which **negative security impact is an externality**[52] uncaptured by production costs, sales of products, or (beyond the short term) market cap. The potential cost savings of avoiding deployment of strong memory safety are immediate, tangible and concentrated, while the costs arising from failures are often delayed or dispersed. The market thus provides little immediate pressure on vendors to prioritize strong memory safety[53], and the financial impacts of failing to do so tend to be externalized from vendors[54], through mechanisms like after-market solutions, disaster recovery, and national security implications, with billions of dollars in damage arising from even a small number of high-profile incidents and data breaches.

This disconnect between the cost of insecurity and the responsibility for mitigating that risk, compounded by two-sided incomplete information, can result in under-investment in robust security measures. The knowing, continued use of memory-unsafe technologies with serious consequences for both individuals and society may be enabled, in part, by a **lack of direct feedback from the market, lack of liability for the impact of product defects, and the challenges all market participants face in accurately assessing the risks, costs of adoption, and benefits of memory safety**.

Cost vs. assurance tradeoffs

A significant part of this "incomplete information" problem comes from the lack of a standardized framework for understanding and evaluating memory safety. As discussed in the previous section, the market currently offers a range of solutions, from weaker, probabilistic mitigations to strong, deterministic protections. These solutions vary significantly in their development costs, runtime overheads, and the level of assurance they provide. Without a common framework to describe these solutions, it is difficult for vendors to make informed decisions about which approach is best suited for their specific needs and constraints. Similarly, customers struggle to express their safety requirements in a way that vendors can understand and reliably fulfill.

The considerations involved are multi-dimensional, and intersect with constraints relevant to the specific use case. For example, memory-safe, garbage-collected languages such as Java, Kotlin, Go or Scala, and interpreted languages such as Python, Ruby or JavaScript, are popular with developers and are already widely used for the development of software that

[52] Ross Anderson and Tyler Moore, *The Economics of Information Security*, October 2006, https://www.science.org/doi/abs/10.1126/science.1130992.
[53] CISA, *Secure by Demand Guide: How Software Customers Can Drive a Secure Technology Ecosystem*, Originally published August 2024. https://www.cisa.gov/resources-tools/resources/secure-demand-guide.
[54]The Register, *CISA boss: Makers of insecure software must stop enabling today's cyber villains*, Originally published September 2024., https://www.theregister.com/2024/09/20/cisa_software_cybercrime_villains/.

can tolerate their performance overhead.[55] Conversely, until Rust emerged as a viable alternative over the past decade, developers of performance-critical and low-level systems software had no memory-safe languages at their disposal. Furthermore, developers with large, existing codebases in unsafe languages such as C and C++ are faced with the (likely prohibitive in most scenarios) cost of translation into a safe language, and therefore must navigate the nuance of an incremental transition towards memory safety.

This lack of a common framework exacerbates the market failures outlined earlier. For instance, a vendor might choose to implement a weaker, less costly mitigation because they are unable to accurately assess the added benefits of a stronger, more expensive solution. Conversely, a customer might be willing to pay a premium for strong memory safety but lack the means to communicate this preference effectively or verify that a vendor's product meets their needs. This ambiguity hinders both supply and demand for strong memory safety.

To address this issue, we need to understand the subtle tradeoffs between the costs (both development- and run-time) of various approaches to memory safety, and the achievable level of assurance. This understanding will pave the way for a standardized framework that can guide decision-making and facilitate communication between stakeholders, including vendors, consumers, and policy makers. While a comprehensive discussion of all these tradeoffs is beyond the scope of this paper, we provide in the following some illustrative examples to highlight the complexities involved:

- Formal methods can achieve very high degrees of assurance of security and functional correctness properties (which necessarily implies memory safety and absence of undefined behavior), but at the same time can incur very substantial development cost[56] [57]. In addition, formal verifiability as a design goal can impose constraints on the overall design[58].
- The overhead and architectural constraints of providing strong temporal safety through runtime mechanisms such as garbage collection, reference counting, quarantining, or sweeping revocation can be non-trivial and make them unsuitable for certain classes of software.
- Memory-safe systems languages such as Rust can provide strong memory safety with negligible or small run-time overhead and at much lower development cost compared to full formal verification. This however comes at somewhat reduced levels of assurance due to the in-practice unavoidable use of unsafe Rust in some components (including the standard library), validation of which at present relies on human code

[55] JetBrains, *State of Developer Ecosystem Report 2024*, retrieved January 2025, https://www.jetbrains.com/lp/devecosystem-2024/#swtype_by_lang.
[56] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser, *Comprehensive formal verification of an OS microkernel*, ACM Transactions on Computer Systems (TOCS), Volume *32,* Number 1, February 2014, https://dl.acm.org/doi/abs/10.1145/2560537.
[57] Daniel Matichuk, Toby Murray, June Andronick, Ross Jeffery, Gerwin Klein, and Mark Staples, *Empirical Study Towards a Leading Indicator for Cost of Formal Software Verification*, IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE), May 2015.
[58] Toby Murray and P.C. van Oorschot, 2018, September. *BP: Formal proofs, the fine print and side effects*. In 2018 IEEE Cybersecurity Development (SecDev), IEEE, September 2018, https://ieeexplore.ieee.org/abstract/document/8543381.

review and informal reasoning. However, there is promising research applying formal methods to the verification of Rust modules that present safe abstractions around unsafe Rust[59] [60], making separate, modular formal verification of unsafe-Rust modules a possibility.

- Hardware-based memory-safety mechanisms such as CHERI provide strong memory safety for code written in unsafe languages while typically requiring no or minimal changes to existing code, hence incurring minimal development-time costs. However, their use involves tradeoff considerations around run-time overheads of temporal safety mechanisms, memory bandwidth due to out-of-band metadata, tradeoffs with reliability when ECC bits are used to hold metadata, memory overhead of wide pointers, the opportunity cost of die area, the market bootstrapping problem of achieving deployment in platforms, and so on.

Weaker memory safety protections, such as runtime exploit mitigations, provide less strong assurance, since they typically do not remove the underlying software defect, but rather focus on blocking its exploitation. They are typically easier to adopt due to their lower overhead and non-disruption of existing source code and software ecosystems, even though they can still incur non-trivial run-time overhead in some cases.

Conversely, there are encouraging signs that commercial software developers have been able to successfully navigate these tradeoffs and have found it cost effective to adopt memory-safe, and more generally, secure-by-design, development practices. It appears that in some cases, not only is the opportunity cost of switching to a safe development environment relatively small, but benefits beyond safety and security add favourably to the overall cost-benefit equation:

- Over the past 6 years, Android has been gradually transitioning away from memory unsafe languages for development of new code, with Android 13 being the first release whose majority of new code is developed in a memory safe language (including Rust, Java and Kotlin)[61]. This has coincided with a significant drop in the fraction of reported memory safety vulnerabilities in Android (76% to 24%)[62]. Beyond security and reliability, there are indications that Rust adoption has had substantial benefits to developer experience and productivity in the Android team[63].

[59] Nima Rahimi Foroushaani and Bart Jacobs, *VeriFast for Rust: Towards Sound journeys through Unsafe areas riding VeriFast*, In Fourth Rust Verification Workshop, co-located with ETAPS 2024, April 2024, https://lirias.kuleuven.be/retrieve/778974.
[60] Andrea Lattuada, Travis Hance, Chanhee Cho , Matthias Brun, Isitha Subasinghe, Yi Zhou, Jon Howell, Bryan Parno, and Chris Hawblitzel, *Verus: Verifying rust programs using linear ghost types*. Proceedings of the ACM on Programming Languages, Volume 7, Issue OOPSLA1, April 2023, https://dl.acm.org/doi/abs/10.1145/3586037.
[61] Jeffrey Vander Stoep, *Memory Safe Languages in Android 13*, December 2022, https://security.googleblog.com/2022/12/memory-safe-languages-in-android-13.html.
[62] Jeff Vander Stoep and Alex Rebert, *Eliminating Memory Safety Vulnerabilities at the Source*, September 2024, https://security.googleblog.com/2024/09/eliminating-memory-safety-vulnerabilities-Android.html.
[63] Lars Bergstrom, *Beyond Safety and Speed: How Rust Fuels Team Productivity*, Rust Nation UK, March 2024, https://youtu.be/QrrH2lcl9ew?si=rBiOwWtfuhfsaCjg&t=323.

- Memory-safe languages relying on garbage-collection (including Java, C#, Go and many others) have been widely used for decades to develop client and server-side application components.
- Engineers at Cloudflare report that their HTTP proxy, implemented in Rust, not only achieved significantly improved security and reliability, but also realized substantial performance and efficiency benefits over their previous NGINX-based solution[64].
- The early use of Rust within the Linux kernel, despite some tensions, further demonstrates the growing acceptance of Rust even in performance-critical and historically C-dominated environments.
- Google reports on enabling bounds checks in the C++ standard library[65] throughout user-facing application and infrastructure workloads, providing strong spatial safety for libc++ data structures at modest, sub-1% run-time overhead[66].
- Similarly, Apple's creation and promotion of Swift within its ecosystem demonstrates a major industry player's commitment to memory-safe languages. Notably, Swift 6 expanded its safety guarantees to prevent data races at compile time.
- Arm's development of the Morello processor and platform[67] has enabled large-scale demonstrations of memory-safe C/C++ in open-source software. Major software packages like FreeBSD[68], nginx[69], and KDE[70] have been successfully adapted to CHERI, showcasing its potential in real-world scenarios.
- Building on this momentum, Microsoft's CHERIoT[71] microcontroller platform is experiencing early adoption across multiple vendors, with CHERI-enabled products expected to ship as early as 2026, fostering an open-source ecosystem around hardware-enforced memory safety.
- In other domains with stubborn classes of vulnerabilities, in particular Cross-site-script and SQL injection vulnerabilities (ranked 2nd and 3rd in the list of

---

[64] Yuchen Wu and Andrew Hauck, *How we built Pingora, the proxy that connects Cloudflare to the Internet*, September 2022, https://blog.cloudflare.com/how-we-built-pingora-the-proxy-that-connects-cloudflare-to-the-internet/.
[65] LLVM Project, *libc++ documentation: Hardening Modes*, retrieved 30 January 2025, https://libcxx.llvm.org/Hardening.html.
[66] Alex Rebert, Max Shavrick and Kinuko Yasuda. *Retrofitting spatial safety to hundreds of millions of lines of C++*, November 2024, https://security.googleblog.com/2024/11/retrofitting-spatial-safety-to-hundreds.html.
[67] Richard Grisenthwaite, Graeme Barnes, Robert N. M. Watson, Simon W. Moore, Peter Sewell, and Jonathan Woodruff, *The Arm Morello Evaluation Platform — Validating CHERI-based security in a high-performance system*, IEEE Micro, Volume 43, Issue 3, May-June 2023.
[68] Brooks Davis, Robert N.M. Watson, Alexander Richardson, Peter G. Neumann, Simon W. Moore, John Bladwin, David Chisnall, Jessica Clarke, Nathaniel Wesley Filardo, Khilan Gudka, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, J. Edward Maste, Alfredo Mazzinghi, Edward Tomasz Napierala, Robert M. Norton, Micahel Roe, Peter Sewell, Stacey Son, and Jonathan Woodruff, *CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-time Environment*, Architectural Support for Programming Languages and Operating Systems (ASPLOS), April 2019.
[69] Graeme Jenkinson, Alfredo Mazzinghi, and Robert N.M. Watson, *CHERI-based memory protection and compartmentalisation for web services on Morello*, Capabilities Limited Technical Report, April 2024, https://www.capabilitieslimited.co.uk/_files/ugd/893621_985a92a599bf41208e4c5710abcf3a68.pdf.
[70] Robert N.M. Watson, Ben Laurie, and Alex Richardson, *Assessing the Viability of an Open-Source CHERI Desktop Software Ecosystem*, Capabilities Limited Technical Report, September 2021, https://www.capabilitieslimited.co.uk/_files/ugd/f4d681_e0f23245dace466297f20a0dbd22d371.pdf.
[71] Saar Amar, David Chisnall, Tony Chen, Nathaniel Wesley Filardo, Ben Laurie, Kunyan Liu, Robert Norton, Simon W. Moore, Yucong Tao, Robert N.M. Watson, and Hongyan Xia, *CHERIoT: Complete Memory Safety for Embedded Devices*, IEEE MICRO, November 2023.

Stubborn Weaknesses in the CWE Top 25[72]), approaches based on safe-by-design APIs and platform features have achieved near-zero residual reported-defect rates across 100+ systems, at marginal amortized cost[73]. While in a different domain, there are substantial parallels between the approach used to prevent code injection vulnerabilities and key concepts in memory-safe languages, suggesting that some aspects of the rollout experience might transfer[74].

- Commercial software developers have found it beneficial to apply formal methods to critical system components, even outside of safety-critical applications. This includes verification of memory safety and (partial) functional correctness of critical systems components and foundational software libraries[75] [76]. Use of formal methods has enabled performance optimizations that might have been deemed too risky otherwise[77].

Enabling business processes and market interventions

A detailed analysis of this apparent market failure, and potential interventions affecting incentives, is beyond the scope of this paper. However, we observe that a common requirement for many conceivable interventions (and a gap in current thinking) – for example, in regulating consumer electronics or in informing government procurement – is the ability to **concisely express strong memory-safety requirements or guarantees in a technology-neutral manner**. This becomes obvious when trying to imagine how one might:

- Improve industrial best practice to utilize strong memory-safety solutions in all areas
- Enable concise acquisition requirements that incorporate memory safety
- Enable reliable and meaningful procurement of strongly memory-safe systems
- Inform product liability legislation and insurance
- Enable review and audit of systems for strong memory safety
- Enable test and evaluation (T&E) for memory safety
- Enable Common Criteria Certification Requirements to include lab-certifiable memory safety requirements
- Enable subsidies, tax incentives, or other mechanisms to encourage the rapid adoption of strong memory safety

[72] MITRE, *Stubborn Weaknesses in the CWE Top 25, September 2023,* https://cwe.mitre.org/top25/archive/2023/2023_stubborn_weaknesses.html.
[73] Christoph Kern, *Developer Ecosystems for Software Safety*, Communications of the ACM, May 2024, Volume 67, Number 6, https://dl.acm.org/doi/full/10.1145/3651621.
[74] Alex Rebert and Christoph Kern, *Secure by Design: Google's Perspective on Memory Safety*, March 2024, https://storage.googleapis.com/gweb-research2023-media/pubtools/7665.pdf.
[75] Byron Cook, Khazem Khazem, Daniel Kroening, Serdar Tasiran, Michael Tautschnig, and Mark R. Tuttle, *Model checking boot code from AWS data centers*, Formal Methods in System Design, Volume 57, July 2021.
[76] Nathan Chong, Byron Cook, Jonathan Eidelman, Konstantinos Kallas, Kareem Khazem, Felipe R. Monteiro, Daniel Schwartz‐Narbonne, Serdar Tasiran, Michael Tautschnig, and Mark R. Tuttle, *Code‐level model checking in the software development workflow at Amazon web services*, Software: Practice and Experience, Volume 51, Number 4, April 2021.
[77] Joel Kuepper, Andres Erbsen, Jason Gross, Owen Conoly, Chuyue Sun, Samuel Tian, David Wu, Adam Chlipala, Chitchanok Chuengsatiansup, Daniel Genkin, Markus Wagner, and Yuval Yarom, *CryptOpt: Verified compilation with randomized program search for cryptographic primitives*, Proceedings of the ACM on Programming Languages, June 2023.

- Support policy interventions to mandate the use of security best practices including strong memory safety in specific classes of products or use cases
- Define safe harbor provisions in a potential software liability regime

We see a set of closely linked problems that must be resolved in order to lift industrial best practices, enable business changes (such as expressing strong memory-safety requirements during procurement), or support potential market interventions (such as regulation of critical infrastructure technologies to ensure use of strong memory safety):

- Develop an **intellectual framework** that allows these diverse technologies and approaches to be consistently described, with their benefits and costs documented in common language that can be used in reasoning about potential use cases
- Develop and document **improvements to current industrial practices**, based on these technologies, able to support the development and composition of strongly memory-safe systems in a manner acceptable to industry
- Enable the clear **enunciation of technology-neutral memory-safety requirements** facilitated by these technologies, and of improved practices for the purposes of acquisition, compliance, regulation, composition, and so on.

The memory-safety standardization gap

When designing, implementing, test and evaluating, certifying, and procuring systems able to resist attacks on memory-safety vulnerabilities, it is easy to imagine a broad range of desirable policies enabled by new memory-safety technologies and accompanying improved industrial best practices; for example:

- A smartphone's general-purpose OS and all of its network-facing applications must be implemented with at least non-deterministic data and control-flow pointer protections within five years, and strong memory safety within fifteen years. Mobile device management (MDM) systems must support enterprises administratively prohibiting installation of memory-unsafe applications.
- All data-center TCBs responsible for isolating hosted government systems from each other, and from other customers, must be implemented with strong memory safety within fifteen years.
- All smartphone TCBs that handle, store, and process biometric and other privacy-sensitive data, hold keys required to authenticate OS updates or use encrypted storage, implement NFC payments, hold financial data, or implement low-level wireless support must implement strong memory safety within five years.
- All cloud-hosted, customer-facing systems of a particular instance type selected by the customer must implement strongly memory-safe kernels and network services within fifteen years. Customer-provided legacy applications not implementing strong memory safety can be installed and used, but must be deployed in environments that adequately mitigate the impact of potential compromise, for example through isolation mechanisms including sandboxes, virtual machines and network isolation.

- All networking infrastructure (such as wireless access points) or cyber-physical systems where software interacts with the physical environment (such as automobiles or certain IoT devices such as smart locks, security cameras, smart thermostats, etc.) shipped after 2034 must be implemented with strong memory safety.
- All Common Criteria-certified Smart Cards, Secure Elements and Trusted Platform Modules should satisfy stringent memory safety requirements on the firmware to enforce that the Common Criteria Requirements (such as keys not extractable) of the certification cannot be bypassed due to a memory safety vulnerability.
- Smart phones and IoT devices using machine-learning models on sensitive personal data, such as inputs from cameras, microphones, GPS and other sensors as well as stored data preserved in order to answer questions such as “where are my glasses?” must, by 2040, be strongly isolated using compartmentalization in order to preserve the privacy and integrity of the data, particularly from the large number of other applications typically running on the same device.

Today, however, there is no consistent and widely adopted means to signal these types of general requirements for memory safety, nor even specific choices (such as a requirement for deterministic memory safety). This gap significantly impedes adoption even of current mitigation technologies; there appear to be no widely adopted means even of requesting or validating the use of commonly available features such as stack canaries, hardware pointer protections, address-space layout randomization, and so on in vendor-neutral forms, let alone stronger memory-safety techniques in a technology-neutral way. We propose to fill this gap through the production of standards that enable practical and reliable engineering and procurement, as well as eventual compliance requirements.

Audiences for memory-safety standardization

Two closely related goals of standards are to (a) allow the clear and practical communication of requirements between consumers of systems and those providing or implementing them, and (b) similarly allow those providing or implementing systems to describe conformance of systems to consumers. Important audiences for this work would include:

- Those specifying requirements for acquisition (e.g., US DoD, US GSA, UK MoD, and UK NCSC).
- Memory-safety system designers and implementers (e.g., the authors of Rust or OCaml, or those adapting operating systems to support CHERI).
- Application software designers and developers (e.g., the authors of Firefox or Chrome).
- Industrial bodies specifying approaches and technologies to be used within specific sectors (e.g., AutoSAR for automotive systems).
- Government and/or regulatory bodies seeking to incentivize rapid adoption of strong memory safety, or limit the use of memory-unsafe systems through laws, liability, tax incentives, or other mechanisms

- End system designers, implementers, and integrators (e.g., designers of a smartphone product).
- Test and evaluation (T&E), certification, and accreditation bodies (e.g., Common Criteria testing laboratories, DOT&E, external security auditors, system integrators, and administrators).
- Those educating future designers, engineers, and others (e.g., those teaching computer science in universities, or [re-]training staff within companies).

Goals for memory-safety standardization

We argue that **standardizing memory safety** is an essential step to widespread adoption of strong memory-safety technologies. Currently, those technologies are seeing early use in selected critical use cases in government and industry – especially in roots of trust and prototypes of more secure IoT or cloud infrastructure. Examples include the use of Rust in an increasing number of "from-scratch" software components, and Microsoft's CHERIoT-Ibex processor seeing early deployment across multiple key industry players. However, getting these technologies to mainstream adoption will require a clear articulation of the benefits, appropriate engineering, and standards that support effective interaction and business models between the producers and consumers of systems. We believe that there are multiple gaps, which this work would aim to fill through the development of both a technology-neutral framework for memory safety, and technology-specific mappings of that framework alongside guidance for their use:

- Develop broad, cross-sector technical consensus on a **practical systemization of strong memory-safety properties and a clear intellectual framework** in which to explain their strengths and weaknesses, appropriate use cases, and so on. This would include classifying sets of technologies based on properties such as coverage of attacks, contributions to abstract memory-safety goals (such as spatial or temporal safety), being probabilistic/secrets-based or deterministic, support for compartmentalization, the potential need for total software rewrites or ABI changes, dependencies on new underlying hardware, potential costs in use and deployment, compartmentalization scalability, memory-safety granularity, and so on. It would also explore the tension between design principles underlying memory-protection technologies (e.g., definitions and implementations of topics such as "spatial safety", "temporal safety", etc.) versus a vulnerability-oriented perspective (in which memory safety is defined in terms of known forms of memory unsafety). The current lack of a working consensus and standardized vocabulary prevents systems designers, engineers, security evaluators, and consumers from agreeing on the basic properties of systems being built and procured.
- Define **best practices for the use of specific memory-safety technologies**, with respect to this framework, such as when and to what extent dependence on unsafe Rust code is suitable within larger Rust software systems, guidelines on structuring such dependencies to support compositional reasoning about safety, the uses of CHERI C and C++ that maximize safety, how to validate whether the implemented

hardware-software stack correctly makes use of the memory-safety features, etc. Pragmatic documentation of best practices will be essential to the consistent and safe deployment of these technologies, as well as enabling management and minimization of risks associated with necessary memory-safety TCBs.

- Consider the **implications of composing multiple technologies**, which will frequently be present in complete computer systems or products – for example, a C-language OS kernel and C++ language run-time (protected weakly by current mitigation techniques or more strongly with CHERI C/C++ in the future), and an application stack written in a type-safe and memory-safe language. Note also that some technologies implementing partial memory safety may have competing/conflicting side effects on other technologies composed in the larger system[78].

To be successful, we believe that a memory-safety standardization framework must:

- **Incorporate existing weaker protection technologies** into this systemization, enabling their specification while also making clear that they are points on a longer-term – and escalating – roadmap for memory safety. It is essential to recognize current industry leaders' efforts in creating and deploying weaker but more accessible technologies within industry. They have been at the forefront of reducing the harms from memory unsafety, and we will need them to lead in adopting stronger protections. Those early explorations and deployments have taught us valuable lessons. In particular, it has become clear from the continued high-rate of memory-safety vulnerabilities that they have proven insufficient: We need the industry to transition to strong protections.
- **Focus on enabling approaches that are technology and vendor neutral**, which will avoid hampering future procurement processes that require independent competing proposals. For example, a clear request for "strong memory safety" in a requirements statement might be satisfied by either Rust or CHERI C/C++ in a responding proposal.
- **Make clear the boundaries between industrial best practice and ongoing research** to: (a) prevent premature engagement with still immature aspects of memory-safety technologies, (b) reassure implementers that likely extensions to current strong memory-safety technologies will be incrementally adoptable, and (c) lay out a long-term roadmap for future memory-safety technology improvements.
- **Establish tiered safety assurance levels** to guide technology selection based on requirements and constraints, acknowledging their varying costs.
- **Provide distinct guidance for new systems and existing codebases**, recognizing that different strategies may be necessary depending on the context.

Overall, the aim would be to create a foundation for the improvement of industry practices in adopting memory safety, taking into account their complexities, which include:

[78] Samuel Mergendahl, Nathan Burow, and Hamed Okhravi, *Cross-Language Attacks*, *Network and Distributed System Security (NDSS)*, April 2022.

- Currently inconsistent definitions of memory safety;
- Necessary limitations to the various technologies imposed by constraints of their use cases – e.g., limitations of Rust static analysis and lack of support for compiler distrust, CHERI temporal memory safety for stacks and focus on type safety rather than memory safety, and the need for sizeable TCBs for both; and
- The reality is that complex real-world systems are composed of many parts, which may individually use one or more forms of memory safety, or not use memory safety at all – for example, memory-safe applications (e.g., written in Rust) running on a memory-unsafe OS or firmware (e.g., running Linux compiled for a non-CHEIR ISA).

Potential structures for one or more standards or documents

It would be premature to try to fix the best structure for the results of this effort. It seems likely that it could include some combination of standards, engineering best practices, and/or technical reports written for a specific audience (e.g., application software designers). However, we expect that they should, in some form address the following:

- Define, in a technology- and vendor-neutral form, a **standard terminology** and an **intellectual framework** for discussing and specifying memory-safety principles and impacts.
- Define **engineering practices in a technology- and vendor-neutral form**, considering topics such as TCB minimization, interoperability with legacy memory-unsafe components to be deprecated or adapted to memory safety in the future, management of weaknesses or omissions in memory-safety technologies, identification of potential performance and/or power efficiency changes, composition of multiple parts (e.g., in a software stack or across an SoC) utilizing different memory-safety technologies, and documentation practices aimed to support review and assessment.
- Define, per-technology, **engineering best practices specific to each technology** (e.g., for use of CHERI, Rust, etc.)
- Define a methodology for reasoning about the **composition of multiple forms of memory safety** within a single system.
- Provide guidance on memory-safety **T&E**, **review, and assessment practice**.

Recognizing that different deployment scenarios, and resulting tradeoffs, require different levels of assurance, it may be helpful to incorporate hierarchies of assurance levels into memory safety standards, similar to leveled standards in other security domains such as supply chain integrity[79]. Leveled standards can help frame discussions around cost vs assurance tradeoffs, and also provide a structure for incremental and intermediate adoption targets.

[79] The Linux Foundation, *Safeguarding artifact integrity across any software supply chain: SLSA Security Levels*, retrieved 30 January 2025, https://slsa.dev/spec/v1.0/levels.

Adoption narratives and timelines

A key function of this work will be to enable longer-term adoption narratives for strongly memory-safe systems. Of particular interest to us are two classes of widely used systems:

- **Industrial best-practice systems** utilize rigorous and engaged practices employed for commodity software at well-funded companies such as Microsoft, Apple, and Google in developing platforms for application writers. Today, these vendors are aggressively adopting memory-safety mitigation technologies such as ASLR and hardware-enabled cryptographic pointer protections. It is not, however, clear that this is generally reflective of industry practice outside of these companies, where we do see widespread use of ASLR, but only limited adoption of techniques such as PAC.
- **Security- and privacy-critical systems** reflect engineering used specifically for essential TCBs in mission-critical systems such as those that are used in critical and national infrastructure, defenses, and aerospace. Today, vendors of such systems are already engaging with systematic deployment of strong memory-safety technologies such as Rust and CHERI.

The timeline for memory-safety deployment is necessarily a long one, with security-critical applications leading in adoption – a key question being how to guide and enable the adoption of stronger technologies in commercial off-the-shelf (COTS) products.

One sample narrative we have been exploring is a gradual escalation of expectation that both rewards current and near-term engagement with memory safety, and makes clear that there will be ratcheting up of requirements with suitable lead time to allow the more significant engineering lifts required to achieve those goals.

We also differentiate new systems from legacy ones – it is easiest to deploy these technologies in the design of a fresh system, especially when new software ecosystems may be created, than it is to deploy them into existing ones. A clear challenge with this narrative is that entirely new systems are only built infrequently, and even where they could be written with a memory-safe language from scratch, they will be created within a large pre-existing memory-unsafe ecosystem that would also need to be migrated or have suitable interfaces created. To facilitate a gradual transition, new components built with memory-safety technologies must be able to interoperate with existing unsafe legacy components.

Finally, we are concerned with policies enabled by standardization and improvements in industry practices, such as documentable requirements in acquisition, regulation, insurance, liability determination, and so on. These become reasonable to enforce only as widespread availability and adoption is achieved, and also depend on clear expectation setting and messaging. However, there is also clearly a form of cyclic dependency here: Technologies will become more available in response to well-signposted roadmaps from regulators, insurers, and critical consumer sectors.

Potential events and interventions

We have developed a timeline based on an "organic" adoption of strong memory-safety technologies that follows current trends and allows significant time periods for consensus building across government and industry. This takes into account an increasing sense of urgency but continuing difficulties in transition due to significant non-technical adoption barriers (such as market failures). However, there are both external events and potential interventions that could accelerate (or stall) such a timeline. These include:

New research: An important consideration is the potential impact of new research on accessible timelines. For technologies such as Rust, reliable and scalable automated techniques to migrate existing source code bases into a memory-safe representation or execution environment might be transformative for adoption[80]. Similarly, significant improvements in automation for formal methods might improve their adoptability outside of a narrow set of higher-assurance use cases. Either of those, and any number of other potential research contributions in languages, hardware, and formal methods could significantly accelerate and broaden adoption.

Education: Another key barrier to the adoption of memory-safety technologies lies in ensuring that there is an educated workforce that is familiar and comfortable with those technologies, which might be achieved through substantive new interventions in higher education internationally. For example, the provision of memory-safe CHERI hardware, template teaching material, and grants would enable much more widespread and hands-on teaching of undergraduate computer-science students, who could then enter industry or the defense industrial base with direct expertise necessary to support adoption.

Market changes: In addition, there is the potential for substantial changes in market conditions, including market interventions, to shorten adoption timelines. These might include changes such as:

- Industry, academia, and government **self-organize to preemptively improve industrial practices** through industry organizations, contributions to shared open-source hardware and software TCBs, and educational efforts. This might happen as a result of nation-state governments, the automotive sector, and/or the finance industry – who are particularly exposed or suffer from greater threats – making a concerted effort to utilize their influence to enable more rapid memory-safety adoption.
- A **major cyber event** significantly impacts sectors or markets, triggering more rapid adoption of memory-safety technologies, especially in affected or exposed sectors.
- A **coordinated regulatory effort** is made, internationally, to improve the rate of adoption of memory safety in national security, critical infrastructure, automotive, healthcare, or other sectors by virtue of mandating engineering standards, changing liability regarding software defects and their impacts, and so on.

[80] DARPA, *Translating All C to Rust (TRACTOR)*, retrieved 19 October 2024, https://www.darpa.mil/program/translating-all-c-to-rust.

- A coordinated effort is made by the international insurance industry and underwriters, perhaps motivated by anticipation of potential **liability changes**, to require improved engineering practices for manufacturers and software developers to maintain professional indemnity coverage.
- A **success story** in one region or one sector creates a sense of urgency in other regions/sectors to adopt memory safety.

New barriers: And, of course, there is the potential for events that lengthen adoption timelines, such as:

- Ultimately **unsuccessful attempts to prematurely mandate adoption** of memory-safety technologies that are unready for, or inappropriate to, use cases or sectors, discrediting the cause of strong memory-safety adoption.
- Memory-safety technologies that prove **too incomplete or vulnerable** to have a long-term impact on the exploitable vulnerability rate lead to a disaffection with strong memory safety following poor deployment experiences.
- A belief that **deferring investment in strong memory safety** is a preferred strategy due to the potential for future research to further reduce adoption cost or performance overheads, which could, in effect, increase the window of exposure by decades, or even defer adoption indefinitely. While techniques such as automated C-to-Rust conversion or using generative machine learning to correct memory-safety bugs have inspired enormous interest, it is currently entirely unclear when (or even if) these research threads, or others like them, will come to fruition.
- Large volumes of **LLM-generated unsafe code** accelerate at a faster rate than what technology today can scalably secure.

One of the greatest risks to adoption will be that of discrediting individual technologies, or in fact the entire approach, due to attempts to push the memory-safety agenda too early or in directions unacceptable to industry.

Candidate timeline

Establishing potential timelines for adoption is challenging given the potential for enabling interventions of research combined with historically strong industrial reluctance to adopting disruptive technologies with less clear translation into concrete consumer demand. The following candidate timeline has been developed based on what we see as realistic timelines given the state of the technology, combined with evolving thinking on potential interventions including the growing appetite for regulation of technologies that have strong impacts on personal data privacy, especially around machine learning, as well as in growing interest in software liability, which might help motivate improvements in industrial practice:

Period	Sector	Narrative
2018-2027 (current period)	Industry best practices	Industry leaders widely deploy **probabilistic protection** techniques such as ASLR and PAC in well engineered, non-critical applications and devices.
	Security-critical applications	Newly designed critical devices and software systems from industry leaders and national security system acquisition are adopting **deterministic memory-safety** technologies such as Rust and CHERI, and selected use of **formal methods**.
2028-2037 (coming decade)	Industry best practices	Over the course of this decade, newly designed non-critical devices and software systems will increasingly ship with partial or complete **deterministic memory-safety**. The use of branding and certification schemes to clearly signpost less-safe systems as damaging to security and privacy; organizations increasingly require policy exemptions for use of memory-unsafe systems in more security-sensitive environments. Legacy systems and applications continue to use probabilistic protection where it is economically infeasible to transition, but at potentially growing cost due to a **shift in industry best practice** leaving vendors open to product liability claims or regulatory problems. Component deprecation and support stoppage may act as force functions to retire legacy components. Toward the tail end of this period, it becomes reasonable for **insurers to incentivise the use of memory safety**, both with respect to software development (professional indemnity insurance) and software procurement and deployment (cybersecurity insurance), as well as for regulators setting standards for next generations of devices to require the use of strong memory safety
	Security-critical applications	**Near universal adoption of deterministic memory safety** in newly deployed systems is achieved in this decade, with significant regulatory, acquisition requirement, insurance efforts to ensure memory safety in new systems, and to de-certify non-memory-safe systems.

2038-2047 (longer term)	Industry best practices	**Near universal adoption of deterministic memory safety** in newly deployed systems. Small pools of remaining non-memory-safety in long-lived products such as deeply embedded, non-network-connected devices; long lived legacy software stacks that must run only in highly protected environments that impose limitations on their casual use. Successful completion of the long-term project to ground memory safety in formally verified designs and implementations increase confidence in strong memory-safety technologies, and in particular that their TCBs are vulnerability-free.
	Security-critical applications	**Elimination of non-memory safety** outside of very small pools of long-lived, fielded devices, but with significant effort made to totally eliminate them as well. No new security-critical systems without memory safety are created during this decade.

Conclusion

We believe that contemporary language-based, hardware-based, and formal techniques for achieving strong memory safety are now of sufficient maturity to allow a path to be planned towards *universal memory safety*, the adoption of strong memory-safety techniques throughout all forms of computer systems. The timeline for such an adoption path is long – likely multiple decades – requiring the deployment of a combination of new hardware, software, and formal techniques serving different adoption paths and catering to differing tolerances for disruption. However, to achieve these goals, industry requires a clear definition of memory safety, accompanied by improvements in engineering practice.

Memory-safety standardization will therefore play an essential role in allowing requirements to be framed in design and procurement, engineering of systems to be tailored to those requirements, and suitable implementation to be auditable. Today, attempts to request memory safety in acquisition, regulation, or liability contexts would be hampered by a lack of a clear set of definitions and practice. Filling this gap requires building industrial consensus on technical approaches, but also a collaborative effort with government and academia to bring such effort to fruition. Despite the need for research to further improve aspects of these technologies, and especially to understand their composition, it is urgent that an effort to appropriately define memory safety begin as quickly as possible based on current technologies and understandings, to feed not just into research, but also improvements in training and delivery.

Acknowledgements

We gratefully acknowledge Graeme Barnes (Arm), Ron Black (Codasip), Mike Eftimakis (Codasip), Andy Frame (VyperCore), John Goodacre (UKRI), Richard Grisenthwaite (Arm), Alice Hutchings (University of Cambridge), William Martin (NSA), Ed Nutting (VyperCore), Anjana Rajan (ONCD), Jonathan Ring (ONCD), Carl Shaw (Codasip), Howie Shrobe (DARPA), Domagoj Stolfa (University of Cambridge), Dan Wallach (DARPA), and Paul Waller (NCSC) for their thoughtful comments and detailed conversations with us about these ideas. Finally, we acknowledge – and remember – Professor Ross Anderson (University of Cambridge), who developed key ideas in the application of economic principles to computer security, and whose advice helped inspire this work.

Distribution Statement A: Approved for public release. Distribution is unlimited. This material is based in part upon work supported by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL) under contract FA8750-24-C-B047 ("DEC"), and in part upon work supported by the Under Secretary of Defense for Research and Engineering under Air Force Contract No. FA8702-15-D-0001. Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Department of Defense, Under Secretary of Defense for Research and Engineering, or the U.S. Government.

This work was supported in part by Innovate UK projects 105694 and 10027440, by EPSRC grants EP/V000292/1 ("CHaOS") and EP/V000373/1 ("CapableVMs"), by UKRI (ERC-AdG-2022 funding guarantee) grant EP/Y035976/1 ("SAFER"), and by ERC-AdG-2017 grant 789108 ("ELVER"). Additional support was received from Arm, Google, and Microsoft.