How to Succeed at Memory Safety without Really Trying*

Úlfar Erlingsson, Google Cloud**

Abstract

The most important security benefit of software memory safety is easy to state: for C and C++ software, attackers can exploit most bugs and vulnerabilities to gain full, unfettered control of software behavior, whereas this is not true for most bugs in memory-safe software.

Fortunately, this security benefit—most bugs don't give attackers full control—can be had for unmodified C/C++ software, without per-application effort.

This doesn't require trying to establish memory safety; instead, it is sufficient to eliminate most of the combinatorial ways in which software with corrupted memory can execute. To eliminate these interleavings, there already exist practical compiler and runtime mechanisms that incur little overhead and need no special hardware or platform support.

1 Why Even Make the Attempt?

In modern computing, attackers continue to exploit memory-corruption vulnerabilities to devastating effect. Most distressingly, important stakeholders are starting to demand that we technologists do something about this [6].

The risk is especially high for server-side software, since software on client devices, like phones, have been quite successfully hardened against attacks [12]. This leaves us in a quandary: almost all of our server-side software platforms are written in C and C++, where memory-corruption is possible, and it will take a decade or more for us to rewrite those foundations to have memory safety [25].

For our job security, if nothing else, we have an obligation to examine whether there is any way to make rapid progress on reducing the risk of ruinous server-side attacks. In this, we must temper our expectations: without memory safety, attackers will still be able to change software behavior, at least somehow. But, perhaps there is some way—e.g., along the lines already used to successfully reduce risk in client software [4]—to prevent attackers from completely controlling software behavior, even without fully eliminating memory-corruption bugs in the software.

If we find success, we may get satisfaction from having made the world safer or even a better parking spot at work.

* This is a tongue-in-cheek paper written for the WACI (Wild and Crazy Ideas) Session at the ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), held in Rotterdam, The Netherlands, April 2025. It was inspired by Shepherd Mead's timeless guide [19], as a flippant companion to Arxiv:2503.21145, Erlingsson et al., *How to Secure Existing C and C++ Software without Memory Safety* [10].

** The opinions and humor are those of the author, not their employer.

2 What Would Success Look Like?

Our efforts have to significantly reduce the risk from attacks, in a meaningful, visible way; otherwise, nobody might even notice, so it hardly seems worth trying at all.

Remote Code Execution (RCE) attacks—where attackers exploit memory-corruption bugs to achieve complete control and dominion—are a very important class of potentially-devastating attacks that get high publicity and loom large in peoples' minds, since they occur so often in the movies.

Greatly reducing the risk of RCE attacks in C and C++ software, despite the presence of memory-corruption bugs, would therefore be one path to success—especially if such attacks could be almost completely prevented.

Of course, to seize the moment, we must make rapid headway towards this goal, and we don't have the time, energy, or resources—and, to be honest, probably not the talent—to develop or deploy new security technologies. So, for our purpose, we should consider only variants or combinations of low-cost, off-the shelf practical techniques.

3 Can We Appropriate Existing Work?

To avoid doing hard work, let's consider only security mechanisms that are already implemented on major platforms and have seen widespread deployment and use.

Also, to avoid wasting our time, let's focus on practical, low-overhead techniques known for preventing attacker exploits of memory-corruption bugs in C and C++ software.

Finally, to avoid being so esoteric that nobody cares, let's not look at approaches that require special hardware support.

Below are four such exploit-mitigation technologies.

Stack Integrity. Attackers delight in memory-corruption bugs that allow them to modify or control one of the function-call stacks of executing C and C++ software, since this can allow them to completely dictate behavior [26].

Two decades ago, CCured and XFI showed how each C and C++ execution stack can be isolated from the effects of memory-corruption bugs, effectively or absolutely, in an efficient manner [11, 20]. For this, compilers need only generate code as if for a segmented-memory system [15], permitting only constant updates to the stack pointer and moving all pointer-accessed variables off the stack—for efficiency, onto a dedicated, thread-local heap memory region [11].

Such *stack integrity* is already implemented by the LLVM compiler, in a practical manner [18]. Its low overhead can be made negligible (by static analysis of C++ references), and its guarantees can be made absolute by preventing writes through pointers from modifying any stack memory.

With stack integrity, attackers cannot change the arguments, local variables, or return sites of functions—as long as function calls are also restricted, e.g., as described next.

Control-Flow Integrity. To gain complete RCE control, attackers must direct execution to code of their choosing.

It is simple to block attackers from executing their own code, e.g., by preventing the addition of any new code.

However, attackers may still exploit memory corruption to redirect execution within the existing software code [26]. Fortunately, in C and C++ software—once stack integrity is enforced—any such redirection can be tamed by preventing unintended use of function pointers and C++ vtables.

Control-flow integrity (CFI) ensures that each function-call site will—despite any memory corruption—only ever direct execution to the start of a compatible function. Specifically, CFI can ensure that any function called via a pointer has a type signature, or is a (virtual) member function of a class, that matches the source code for the call site [1, 28].

CFI is widely enforced by low-overhead, compiler-added checks [17, 28] that harden software against attack [21, 22, 29], sometimes with hardware support [3, 7]. Combined with stack integrity, CFI can inductively guarantee that software always executes, as intended, as a well-nested sequence of functions with uncorrupted arguments and variables [11].

Heap Data Integrity. Attackers usually must perform "*heap feng shui*" (i.e., manipulate the precise heap layout) to be able to corrupt heap memory in a targeted way [27].

There are many efficient ways to improve the integrity of the C and C++ heap, most using hardware support [4, 16].

One simple software means of protecting the heap is to partition memory into many disjoint regions, such that each static heap-object allocation site uses only one region [8].

With enough partitions, attackers lose their "feng shui" control over heap layout, and thereby near all means for targeted heap corruption—whether by temporal bugs, like use-after-free, or spatial bugs, like overflows. Also, by confining each heap-pointer access to the range of statically-accessible heap partitions, the attackers' ability to exploit memory corruption can be reduced even further [5, 8].

Attackers are impeded even by coarse heap partitioning, such as by size [13]. However, stronger *heap integrity* can be had by partitioning more finely, using static namespaces, types, etc. This is increasingly done, as it allows trading higher resource overheads for improved security [2, 23].

Pointer Integrity and Unforgeability. To be successful, RCE attacks using memory corruption must, near always, reliably retarget some pointer. Such retargeting can be made next to impossible, by making C and C++ pointers more similar to true capabilities—that is *unforgeable* and *unguessable.*

Randomly-generated secrets can be used to make pointers emulate true capabilities, especially on 64-bit hardware.

Randomizing the layout of memory with a secret is a common RCE defense, especially useful on servers, as it forces attacks to take an often difficult derandomization step [14].

Each pointer value can also be randomized, using a different secret for each type of pointer, as long as pointers are derandomized in the software code before each use.

This is effectively what is done in Apple software, which uses special ARM hardware support to also check *pointer integrity* at runtime—i.e., ensure each pointer access uses pointers of the right type [4, 24]. Apple uses this further to enforce a form of stack integrity, CFI, and heap integrity [4].

When used as an orthogonal layer of defense, in addition to the above three types of integrity, pointer randomization—even without runtime checks—is a formidable barrier to RCE attacks based on C and C++ memory corruption.

4 Why is Success a Sure Thing?

Can we be sure that the above protections, when combined, will significantly reduce the risk of RCE attacks in C and C++ software? After all, despite enforcing four types of integrity, we are not really trying to establish memory safety, which means attackers can still use memory corruption to change software behavior. And, we should not risk even a hint of failure, according to our timeless guide [19].

Yes, empirically, as per the incidence and price of RCE attacks on Apple client software using these defenses [12].

Yes, intuitively. From the attacker's viewpoint, RCE attacks are enabled by the vast number of invalid paths and interleavings that become possible in the "weird machine" of corrupted software execution [9]. With these protections, attackers can choose only new behaviors that conform to valid, well-nested sequences of calls to compatible functions in the software—with uncorrupted arguments and local variables—that operate on strictly-partitioned data objects accessed via almost-unforgeable pointer values.

Of course, attackers may still be able to achieve RCE due to factors such as misconfiguration or bugs in the protections, platform, or hardware, but this is also possible for memory-safe software. Also, for certain, particular software, the protections may simply not be strong enough; for example, in software that contains a general execution engine, an RCE attack may be possible by corrupting the memory that contains the interpreter inputs.

In all cases, the protections form a sound basis for whatever additional defenses are required, such as isolated partitioning of interpreter inputs or further runtime checks [1].

Also, it wasn't like we were shooting for the moon, or that penthouse corner office. Our success goal has only been to prevent most of the most devastating types of memory-corruption attacks, most of the time—in a practical, low-cost manner, using existing techniques that can be rapidly adopted—and get a window office on a higher floor.

We should make the attempt, looking forward to the view.

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-Flow Integrity Principles, Implementations, and Applications. *ACM Transactions on Information and System Security (TISSEC)* 13, 1 (2009), 1–40. https://doi.org/10.1145/1631030.1631034

[2] Apple Security Research. 2022. Towards the next generation of XNU memory safety: kalloc_type. https://security.apple.com/blog/towards-the-next-generation-of-xnu-memory-safety/ Accessed: 2025-03-22.

[3] Arm Limited. [n. d.]. Arm Instruction Set Reference Guide: BTI - Branch Target Identification. https://developer.arm.com/documentation/100076/0100/A64-Instruction-Set-Reference/A64-General-Instructions/BTI Accessed: 2025-03-22.

[4] Zechao Cai, Jiaxun Zhu, Wenbo Shen, Yutian Yang, Rui Chang, Yu Wang, Jinku Li, and Kui Ren. 2023. Demystifying Pointer Authentication on Apple M1. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 2833–2848. https://www.usenix.org/conference/usenixsecurity23/presentation/cai-zechao

[5] Miguel Castro, Manuel Costa, and Tim Harris. 2006. Securing Software by Enforcing Data-flow Integrity. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)* (Seattle, WA). USENIX Association. https://www.usenix.org/legacy/event/osdi06/tech/castro.pdf

[6] CISA, NSA, FBI, ASD's ACSC, CCCS, NCSC-UK, NCSC-NZ, and CERT-NZ. 2023. *The Case for Memory Safe Roadmaps.* Technical Report. Cybersecurity and Infrastructure Security Agency (CISA). https://www.cisa.gov/resources-tools/resources/case-memory-safe-roadmaps

[7] Jonathan Corbet. 2022. Indirect branch tracking for Intel CPUs. *LWN.net* (Mar 2022). https://lijuanru.com/reading/news/2022-05-11/Indirect%20branch%20tracking%20for%20Intel%20CPUs%20[LWN.net].html Accessed: 2025-03-22.

[8] Dinakar Dhurjati, Sumant Kowshik, and Vikram Adve. 2006. SAFECode: Enforcing alias analysis for weakly typed languages. In *ACM SIGPLAN conference on Programming language design and implementation (PLDI'06)*. https://doi.org/10.1145/1133981.1133999

[9] Thomas Dullien. 2017. Weird machines, exploitability, and provable unexploitability. *IEEE Transactions on Emerging Topics in Computing* 8, 2 (2017), 391–403. https://doi.org/10.1109/TETC.2017.2740168

[10] Úlfar Erlingsson. 2025. How to Secure Existing C and C++ Software without Memory Safety. arXiv:2503.21145 [cs.CR] https://arxiv.org/abs/2503.21145

[11] Úlfar Erlingsson, Mihai Budiu, Martín Abadi, and Michael Vrable. 2006. XFI: Software Guards for System Address Spaces. In *7th Symposium on Operating Systems Design and Implementation (OSDI '06)*. 47–60. https://www.usenix.org/legacy/event/osdi06/tech/full_papers/erlingsson/erlingsson.pdf

[12] Lorenzo Franceschi-Bicchierai. 2024. Price of zero-day exploits rises as companies harden products against hackers. *TechCrunch* (april 2024). https://techcrunch.com/2024/04/06/price-of-zero-day-exploits-rises-as-companies-harden-products-against-hackers/ Accessed: 2025-03-22.

[13] Google. 2020. *TCMalloc Design and Implementation.* https://google.github.io/tcmalloc/design.html Accessed: 2025-03-22.

[14] Google Cloud Security Team. 2023. Six facts about Address Space Layout Randomization on Windows. https://cloud.google.com/blog/topics/threat-intelligence/six-facts-about-address-space-layout-randomization-on-windows/ Accessed: 2025-03-22.

[15] IBM. 1989. *IBM OS/2 Version 1.2, Programming Overview.* Technical Report 64F3830. https://www.os2museum.com/files/docs/os212pti/64F3830_OS2_V1_2_PTI_Programming_Overview.pdf

[16] Arm Limited. 2021. *Armv8.5-A Memory Tagging Extension.* Technical Report. https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf

[17] LLVM. 2024. *Control Flow Integrity - Clang 18.0.0git documentation.* https://clang.llvm.org/docs/ControlFlowIntegrity.html Accessed: 2025-03-22.

[18] LLVM. 2024. *SafeStack - LLVM 18.0.0git documentation.* https://clang.llvm.org/docs/SafeStack.html Accessed: 2025-03-22.

[19] Shepherd Mead. 1952. *How to Succeed in Business Without Really Trying: The Dastard's Guide to Fame and Fortune.* Simon and Schuster.

[20] George C. Necula, Scott P. McPeak, S. Rahul, and Westley G. Weimer. 2002. CCured: Memory Safety in C/C++ Programs. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon, USA) *(POPL '02)*. ACM, New York, NY, USA, 106–119. https://doi.org/10.1145/503272.503283

[21] Android Open Source Project. 2025. *Control Flow Integrity.* https://source.android.com/docs/security/test/cfi Accessed: 2025-03-22.

[22] The Chromium Projects. 2015. *Control-Flow Integrity.* https://www.chromium.org/developers/testing/control-flow-integrity/ Accessed: 2025-03-22.

[23] The Chromium Projects. 2022. *PartitionAlloc.* https://www.chromium.org/developers/partitionalloc/ Accessed: 2025-03-22.

[24] Qualcomm Technologies, Inc. [n. d.]. Pointer Authentication. https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/pointer-auth-v7.pdf Accessed: 2025-03-22.

[25] Alex Rebert, Chandler Carruth, Jen Engel, and Andy Qin. 2024. Safer with Google: Advancing memory safety in C++ and beyond. *Google Security Blog* (oct 2024). https://security.googleblog.com/2024/10/safer-with-google-advancing-memory.html Accessed: 2025-03-22.

[26] Hovav Shacham. 2007. The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86). In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS '07)*. ACM, New York, NY, USA, 552–561. https://doi.org/10.1145/1315245.1315313

[27] Alexander Sotirov. 2007. Heap Feng Shui in JavaScript. https://www.blackhat.com/presentations/bh-usa-07/Sotirov/Whitepaper/bh-usa-07-sotirov-WP.pdf Accessed: 2025-03-22.

[28] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Compiler-based Control-Flow Integrity. In *23rd USENIX Security Symposium (USENIX Security 14)*. 491–504. https://www.usenix.org/system/files/conference/usenixsecurity14/sec14-paper-tice.pdf

[29] Trend Micro. 2015. In-depth Look at Control-Flow Guard Technology in Windows 10. https://www.trendmicro.com/vinfo/us/security/news/vulnerabilities-and-exploits/in-depth-look-at-control-flow-guard-technology-windows-10 Accessed: 2025-03-22.