Comprehensive Memory Safety Validation: An Alternative Approach to Memory Safety

Kaiming Huang | The Pennsylvania State University
Mathias Payer | EPFL
Zhiyun Qian | University of California, Riverside
Jack Sampson and Gang Tan | The Pennsylvania State University
Trent Jaeger | University of California, Riverside

Comprehensive memory safety validation identifies the memory objects whose accesses provably comply with all classes of memory safety, protecting them from memory errors elsewhere at low overhead. We assess the breadth and depth of comprehensive memory safety validation.

Memory safety is paramount for software security. In the early days of computer security research, the Anderson Report underscored the risks of unchecked memory access, providing a foundational understanding of memory errors. Subsequent events, such as Code Red, the Morris Worm, and Slammer, showcased the real-world impact of memory exploits. Google and Microsoft recently shed light on the persistent nature of memory vulnerabilities, emphasizing the need for continuing memory safety enhancement.

The vulnerability landscape now spans from ransomware to high-profile vulnerabilities, including Spectre, Meltdown, DirtyPipe, and DirtyCred, to memory errors generated by large-language-model-based artificial intelligence code generators, highlighting the enduring relevance of memory safety.

The efforts of the past twenty years have produced many defenses against memory errors that target one or more of the classes of memory safety: spatial, type, and temporal safety. However, building defenses against memory errors poses significant challenges as any effective defense must navigate the intricate balance of three properties, which we call the 3-C principle. The ideal memory error defense should 1. offer enforcement of *all classes* of memory safety and 2. ensure *coverage* for all memory objects. Importantly, any defense should 3. achieve these objectives at a reasonable *cost*, minimizing the impact of performance and memory overheads. Despite a wide variety of defenses, e.g., based on runtime checks, memory-safe programming languages, and hardware-assisted enforcement, the search for a perfect solution, a defense that protects against all classes of memory errors and covers all memory objects for a reasonable cost, remains elusive.

The common goal among the proposed research defenses is that they tend to pursue complete coverage of all memory objects. To achieve this goal, these defenses often present challenging tradeoffs between protected error classes and cost; most defenses only enforce one or a subset of the classes of memory safety. Even so, effective defenses often retain high performance and memory overheads, preventing their adoption in practice. As a result, the defenses applied in practice do not cover all angles from which memory errors may be exploited, leaving all memory objects at risk.

Digital Object Identifier 10.1109/MSEC.2024.3379947

CCured[1] offers an alternative approach for enforcing memory safety by triaging memory accesses through *memory safety validation*. Memory safety validation aims to determine whether all memory accesses to a memory object (e.g., through pointers that may reference a memory object) comply with memory safety. The CCured method validates whether a pointer cannot violate spatial and type safety to elide runtime checks. The safe stack approach[2] takes this idea further by isolating stack objects found to comply with spatial safety on a separate stack. Researchers have also proposed isolating objects on separate heaps to enforce temporal memory safety.[3] Both the stack and heap isolation techniques enforce protection against memory errors for low overhead. However, none of these techniques provide memory safety validation for all classes of memory safety nor ensure protection from all classes of memory errors for even one object.

We examine the potential for applying comprehensive memory safety validation for all classes of memory errors to provide a foundation of memory safety for programs. The idea is that the objects whose memory accesses can be proven to comply with all classes of memory safety can be isolated from memory errors in other accesses to protect a large fraction of objects (but not all objects). After motivating the potential for memory safety validation, we introduce the DataGuard[4] and Uriah[5] systems, which perform comprehensive memory safety validation for the stack and heap regions, respectively. We then perform a study of Ubuntu Linux packages to evaluate the fraction of stack and heap objects whose memory accesses can be proven to satisfy memory safety for all classes, finding that over 77% of heap objects and over 85% of stack objects can be protected at low overhead via isolation. We then perform a longitudinal study over software versions covering the last 10 years, finding that the number of objects whose memory safety can be validated comprehensively is increasing, hinting that leveraging memory safety validation may become even more beneficial in the future. Finally, we examine other possible benefits of applying comprehensive memory safety validation, from reducing effort in bug finding, even for nonmemory bugs, to improving defenses.

The Memory Safety Problem

Unsafe programming languages, such as C and C++, distinguish memory objects from memory references (i.e., pointers), allowing pointers to reference any object. While these languages remove the overhead of code that maintains memory safety and provide programmers with tremendous flexibility for accessing and managing memory use, this separation often leads to programming errors, generally called *memory errors*, where pointers may be used to access memory locations and/or interpret memory incorrectly. These errors create powerful attack vectors for adversaries, enabling them to hijack program executions for a variety of malicious purposes.

Memory errors can be categorized into three classes: spatial errors, type errors, and temporal errors. A memory access causes a *spatial error* when the pointer used refers to memory outside the referent object's (i.e., the object to which a pointer is assigned) allocated memory region. Spatial errors are the most well-known memory errors, such as buffer overflows and overreads. A memory access causes a *type error* when the pointer used refers to a memory location using a different data type than that associated with the object or field at that location. A common type error allows a pointer object to be referenced as a data object or vice versa. Finally, a memory access causes a *temporal error* when the pointer may be used before an object is assigned (i.e., use-before-initialization) or after it is deallocated (i.e., use-after-free).

Researchers have explored a multitude of methods to eliminate memory errors and/or prevent their exploitation. On one hand, researchers have long advocated switching to safe programming languages, even safe variants of the C language. However, C/C++, which serves as the pivotal language for system-level programming (e.g., OS kernel and embedded systems) and performance-critical applications (e.g., web browsers and server programs), remain popular among programmers and cannot be completely substituted given the large legacy codebase. On the other hand, a variety of defenses have been proposed to provide safety to programs written in unsafe languages. While some defenses with lower cost, such as stack canaries, have been adopted, they do not offer complete protection. Consequently, memory errors remain in programs, and adversaries can still exploit them. More complete defenses that prevent exploitation of memory errors (for at least one class) have been found to be too expensive. An open question is whether a new approach is needed.

A Case for Memory Safety Validation

The CCured system proposed the idea of memory safety validation to avoid unnecessary runtime checks to prevent memory errors.[1] CCured highlights the fact that spatial errors are not possible for pointers that are never used in pointer arithmetic operations, and type errors are not possible for pointers that are never used in type cast operations. With these insights, CCured found that approximately 90% of all pointers in C/C++ programs are never used in either pointer arithmetic or type casts. Thus, only approximately 10% of pointers require runtime checks to prevent either of these classes of memory errors.

There are two conjectures that one can draw from these results. On one hand, since only 10% of pointers require runtime checks to enforce spatial and type safety, defenses only need to prevent unsafe memory accesses on this fraction of pointers to enforce memory safety with full *coverage* of all unsafe operations. Unfortunately, the experience of many researchers has shown that the number of operations using this fraction of unsafe pointers is still sufficiently large enough to create a substantial runtime and/or memory overhead. In addition, CCured did not assess pointers for possible violations of temporal safety, so additional runtime checks will be necessary to enforce memory safety for all three memory error classes. As a result, all memory objects remain prone to exploitation due to memory errors in current programs.

An alternative conjecture is that many objects have only memory-safe accesses, so these *safe objects* should be protected to maintain their safety. The idea is to apply *comprehensive* memory safety validation for each object, whereby a memory object is classified as *safe* only when every pointer that may reference (i.e., alias; the term *alias* refers to the pointers that reference [i.e., point to] the same memory object [i.e., memory location]) the object can be proven to comply with memory safety for all three classes of memory errors. If so, these objects do not require any memory safety defenses for their aliases and can be protected from unsafe memory accesses to other, *unsafe* objects, which can be accomplished without runtime checks [e.g., via information hiding, such as address space layout randomization (ASLR)]. Researchers have proposed techniques to isolate objects from memory errors using separate stacks[2] and typed heaps,[6] in which objects that have been shown to be safe from some classes of memory errors are isolated from accesses to objects that may be prone to such memory errors. While isolation can be a simple and efficient defense, these prior techniques did not consider memory safety comprehensively, i.e., for all three classes of memory safety, potentially exposing supposedly isolated objects to memory errors.

While extensive research has explored the first conjecture extensively to try to protect all memory objects from memory errors, little effort has examined the potential of identifying and protecting safe objects using *comprehensive memory safety validation*. In the rest of this article, we examine methods for comprehensive memory safety validation, their impact on enforcing memory safety in programs, and their potential impact in enhancing software security more broadly.

Methods for Comprehensive Memory Safety Validation

We have developed memory safety validation methods for the stack, called *DataGuard*,[4] and heap, called *Uriah*.[5] Memory safety validation aims to prove that every alias of an object must only be used in operations that satisfy spatial, type, and temporal safety. If we cannot prove all classes of memory for even one alias, then the object is classified as unsafe. The memory safety validation has to be conservative to ensure that any object classified as safe must not be potentially unsafe. We require such a conservative analysis to avoid placing any object in a safe region (stack or heap) that may possibly be unsafe as any unsafe operation could compromise the entire safe region.

By isolating the safe and unsafe memory regions, DataGuard and Uriah protect safe objects from all classes of memory errors and enable developers to inspect which objects may be unsafe. Isolation for both DataGuard and Uriah relies on ASLR. While ASLR is prone to compromise via information leaks, DataGuard and Uriah both ensure that no pointers (aliases) to safe objects are located in the unsafe memory regions. That is, any otherwise safe object that is referenced by an object classified as unsafe becomes unsafe. This guarantees that potential information leaks resulting from memory errors in an unsafe region will not expose any addresses in the safe region.

The protection of the unsafe memory region is beyond the scope of this work. We note that any existing defenses can be utilized in the unsafe region (e.g., ASan[7] and FuZZan[8]). It is one of our goals to remove unnecessary runtime checks on proven safe memory objects. Moreover, we discuss how memory safety validation may improve testing for exploitable memory errors and hardware-assisted enforcement of memory safety later in the article.

Figure 1 shows the high-level approach for comprehensive memory safety validation; there are several critical differences for validating stack versus heap memory, described below. We first describe stack memory safety validation via DataGuard, then describe the additions and changes necessary to perform heap memory safety validation using Uriah. First, CCured showed that many pointers satisfy spatial and type safety because they never perform any operations that could cause a violation, so DataGuard extends this analysis with a simple escape analysis for temporal safety to find objects whose aliases all satisfy the three classes of memory errors trivially (i.e., have no aliases that are used in any operation that could violate memory safety). We found that 72% of stack objects satisfy memory safety comprehensively using this analysis.[4] Second, for objects that may be aliased by pointers through which potentially unsafe memory operations may be performed (e.g., pointer arithmetic), we find they can still be validated if we can determine concrete safety constraints for spatial, type, and temporal safety. Objects lacking concrete constraints (e.g., the concrete size of a buffer for spatial safety) are classified as unsafe. Third, DataGuard applies static analyses for each class of memory safety to validate compliance with each object's constraints for all of its aliases. An additional 16% of stack objects are validated statically.[4] Fourth, for objects found unsafe via static analysis, we apply a targeted symbolic execution that follows the execution paths (i.e., def-use chains) of each pointer found unsafe to validate whether they actually comply with the object's memory safety constraints (i.e., were a false positive in static analysis). Approximately 4% of stack objects are validated via this method,[4] leading to a total of over 91% of stack objects being safe. The set of safe objects are then isolated using the safe stack technique[2] to protect them from unsafe memory accesses, incurring an average overhead of 4.3% for SPEC CPU 2006 benchmarks. (The original SafeStack mechanism presents 0.1% overhead in Kuznetsov et al.[2] SafeStack in Clang is different from the academic prototype. Our evaluation shows 11.3% of runtime overhead for Clang's SafeStack for SPEC CPU 2006 Benchmark.)

Validating memory safety for the heap in Uriah follows the same high-level flow but has significant differences in each step as detailed in Figure 1. First, there is no general, static memory safety validation method for temporal safety, so only validation for spatial and type safety are performed. Objects that satisfy spatial and type safety are allocated in a manner that enforces a form of temporal memory safety called *temporal type safety*.[3] Second, Uriah validates type safety for compound types, which was not validated by DataGuard, by generating and checking constraints on safe upcasts. Third, Uriah's static analysis for heap data accounts for dynamic resizing of heap objects during reallocation, which was not necessary for stack objects. Fourth, Uriah applies symbolic execution to prune infeasible paths until only feasible safe paths remain, in addition to validating compliance in execution paths as in DataGuard. Uriah can validate that over 70% of allocation sites produce only objects whose memory accesses (i.e., via all their aliases) satisfy spatial and type memory safety, accounting for over 70% of allocated objects at those sites for the programs tested.[5] Uriah employs a per-type allocator that enforces temporal type safety using strict type-based reuse at runtime, preventing both use-before-initialization (e.g., zeroing memory before the first use) and use-after-free (e.g., enforcing temporal type safety) with 2.9% overhead.

Memory Safety Validation Results

We compute safe objects from Linux packages to assess the applicability and effectiveness of DataGuard and Uriah over across a diverse set of software. The evaluation is performed on Ubuntu 20.04 with Linux kernel 5.8.0-44-generic and LLVM 10.0, using the published versions of DataGuard and Uriah, including their PDG and SVF analysis capabilities. (DataGuard is available open source at https://github.com/Lightninghkm/DataGuard. The link to the Uriah source is: https://github.com/Lightninghkm/Uriah. The Uriah article is still under submission process, we will open-source Uriah to the provided link upon acceptance.) We source preinstalled packages directly from the official Ubuntu repositories. For generating LLVM bitcode, we opt for the uClibc library and employ the wllvm tool to compile Linux packages. Packages incompatible with this toolchain are excluded from further analysis.

Memory Safety in Ubuntu Packages

First, we examine DataGuard and Uriah's adaptability across diverse Linux packages. We assess whether

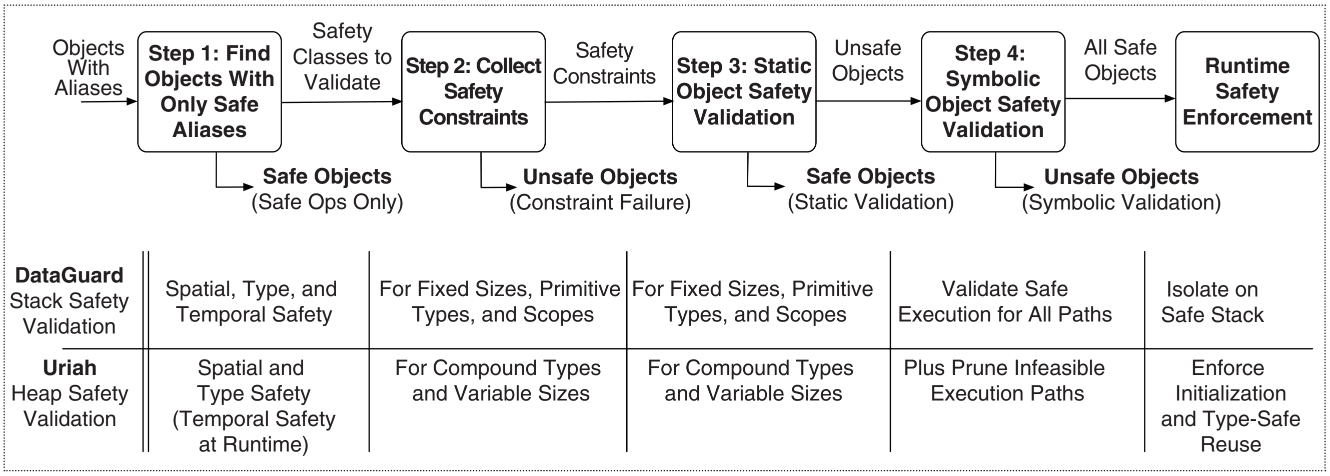
Objects With Aliases
Step 1: Find Objects With Only Safe Aliases
Safety Classes to Validate
Step 2: Collect Safety Constraints
Safety Constraints
Step 3: Static Object Safety Validation
Unsafe Objects
Step 4: Symbolic Object Safety Validation
All Safe Objects
Runtime Safety Enforcement
Safe Objects (Safe Ops Only)
Unsafe Objects (Constraint Failure)
Safe Objects (Static Validation)
Unsafe Objects (Symbolic Validation)
DataGuard Stack Safety Validation
Spatial, Type, and Temporal Safety
For Fixed Sizes, Primitive Types, and Scopes
For Fixed Sizes, Primitive Types, and Scopes
Validate Safe Execution for All Paths
Isolate on Safe Stack
Uriah Heap Safety Validation
Spatial and Type Safety (Temporal Safety at Runtime)
For Compound Types and Variable Sizes
For Compound Types and Variable Sizes
Plus Prune Infeasible Execution Paths
Enforce Initialization and Type-Safe Reuse

Figure 1. The memory safety validation approach: applied to the stack (DataGuard) and heap (Uriah).

these approaches can be applied to Ubuntu packages automatically. Out of the 1,623 packages in the Ubuntu distribution, DataGuard and Uriah successfully processed 1,245, representing 76.7% of the distribution. This translates to analyzing roughly 266 million source lines of code (SLOC), which constitute 77.8% of the total 342 million SLOC. However, 378 packages remain unanalyzable due to compatibility issues, such as conflicts with the LLVM version used by DataGuard and Uriah. Uriah is able to analyze and harden all 202 Linux packages that make use of heap allocations, with its original tool chain.

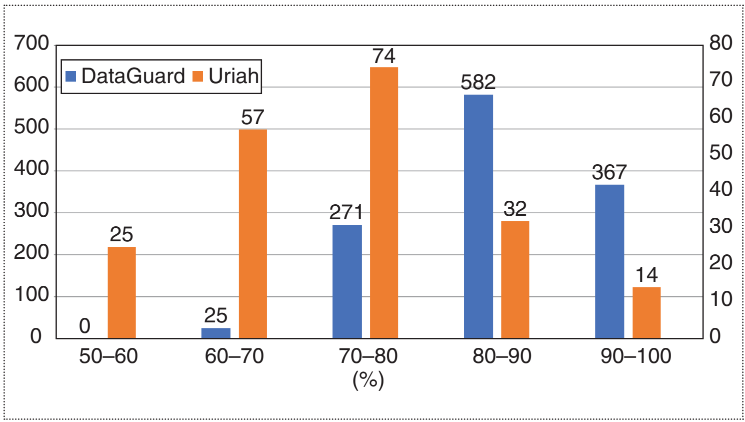

Figure 2. The distribution of packages w.r.t. the fraction of safe stack objects and safe heap allocations. The x-axis represents the interval of the fraction of safe stack objects or safe heap allocations that are protected by DataGuard or Uriah; 0%–50% is omitted since both DataGuard and Uriah offer at least 50% protection among all packages. The y-axis represents the number of packages that fall into the corresponding fraction interval of safe stack objects (blue) and safe heap allocations (orange).

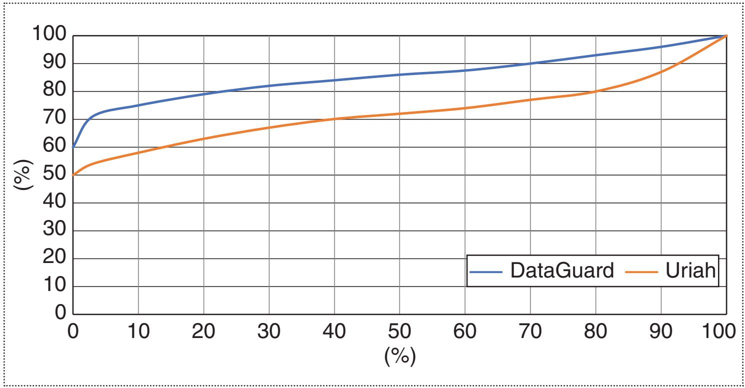

Figure 3. The cumulative distribution of the fraction of protected safe stack objects and safe heap allocations for all analyzed Linux packages. The x-axis represents the percentage of analyzed Linux packages. The y-axis represents the percentage of safe stack objects and safe heap allocations found by DataGuard and Uriah. The figure can be understood as "(1 − x-axis)% of analyzed packages have *at least* y-axis% of safe stack objects or safe heap allocations." The slope of the line correlates to the order of packages; we followed the sequence in Ubuntu repositories.

Second, we investigate DataGuard and Uriah's potential to automatically protect stack and heap objects against memory errors. We compute the safe objects within a Linux distribution. Among all the packages analyzed, DataGuard validates that all accesses to 12,484,971 out of 14,627,355 (85.35%) stack objects are free from all three classes of memory errors. These objects can all be protected by stack isolation. Uriah validates that all accesses to objects produced in 425,317 out of 545,560 heap allocation sites (77.96%) satisfy spatial and type safety. These objects are protected from attacks on temporal memory errors and memory accesses from unsafe objects using the Uriah runtime allocation scheme. We note that this is a slightly greater fraction of the protected heap objects than in the Uriah article.[5] One reason is that these Linux packages are the most recent versions, so heap use tends to be safer than for older SPEC CPU 2006 programs. Also, some of the SPEC benchmarks that were evaluated originally in Uriah have a limited number of heap allocations, and a large fraction are unsafe, which biases the results.

Third, we assess the security impact by analyzing the fraction of protected stack and heap objects validated. Figure 2 shows the distribution of the fractions of safe stack objects (i.e., allocation sites for the heap) across the Ubuntu packages. As we can see, using the memory safety validation of DataGuard and Uriah, a majority of the Ubuntu packages have more than 70% of stack and heap objects protected from all classes of memory safety errors. The cumulative distribution (Figure 3) also shows a similar finding. Specifically, DataGuard protects more than 70% of the stack objects for 97% of the packages (i.e., at 3% in Figure 3) and more than 80% of the stack objects for 75% (i.e., at 25%) of the packages. Uriah protects more than 60% of the heap allocation sites for 90% of the packages (i.e., at 10% in Figure 3) and more than 70% of the heap allocation sites for 60% (i.e., at 40%) of the packages.

Assessing Memory Safety Over Time

We perform a longitudinal study using DataGuard and Uriah to assess how memory safety has evolved in programs over time. We showcase results for Nginx (versions 1.4.0–1.25.0), Httpd (2.2.24–2.4.57), and Firefox (21–115), spanning ten years (January 2013 to May 2023) of released versions for each.

For stack memory protection, we deployed DataGuard on the three programs. Figure 4 shows that memory safety, in terms of the fraction of stack objects validated to be free from all classes of memory errors, has been trending upward over the past ten years for all three programs. We observe a few brief reductions in the fraction of safe stack objects, such as the year 2016 for Firefox and the year 2018 for Nginx and Httpd. We note that major updates of the corresponding programs (e.g., new functionalities, modules, or interfaces) occurred during these years, which may have introduced new, unsafe objects. For heap memory, we deployed Uriah on the three programs as well. Similarly, Figure 5 shows that memory safety in terms of the fraction of heap allocation sites has trended upward the past ten years for all three programs. Drops are also observed for safe heap allocations (e.g., 2016 for Httpd and 2017 for Nginx) due to new version releases.

Generally speaking, the surge in safe memory objects on both stack and heap can be attributed to a confluence of several factors: evolving coding standards, developers' awareness of memory safety, and powerful vulnerability detection tools. This synergy strengthens the memory safety of software by progressively removing unsafe memory operations for more memory objects. Moreover, programmers can leverage the static memory safety validation from DataGuard and Uriah to get feedback on the remaining potential unsafety in their code (i.e., unsafe memory operations/objects). This empowers the developers to fix their code, making it resilient against memory errors.

Applying Memory Safety Validation to Further Improve Security

In addition to enabling the protection of safe memory objects, memory safety validation may improve software security in other important ways. Below, we examine the potential for improving security in ways that range from improved bug detection, even for nonmemory bugs, to more effective prevention of the remaining memory errors.

Improving Dynamic Memory Safety Testing

One key goal is to test software for memory errors for the remaining unsafe objects and their aliases. However, in testing unsafe operations, the enforcement of memory safety must be *effective* but not necessarily complete. An approximation of memory safety suffices to detect flaws. In testing, there is no adversary that carefully tailors exploits to bypass the checks. The lack of an active adversary allows the memory safety checks for detecting bugs to use a weaker safety property. Instead of ensuring the *validity of pointers*, i.e., all pointer uses are memory safe with respect to the referenced objects, memory safety testing only ensures that memory accesses target *valid objects*. This is a subtle but important change of perspective. On one hand, a program has strictly fewer live objects than live pointers, so fewer metadata need to be managed. On the other hand, the metadata lookup can be organized in a much more efficient way, reducing the key cost of checks. Additionally, keeping metadata per object increases compatibility with software as the underlying memory location already varies for each execution due to, e.g., ASLR.

Address sanitizer[7] introduced the idea of memory safety checks to test that memory accesses target valid objects, called *sanitizers* in general. Memory sanitization cleverly pads memory objects on both sides with so-called red zones. When accessing memory, it checks if the pointer references a red zone and triggers an exception. Strict memory safety requires tracking each pointer to ensure it remains tied to its original object

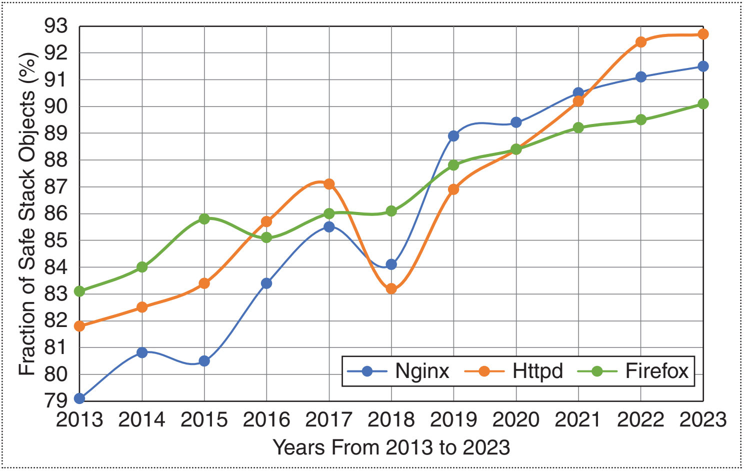

Figure 4. The fraction of safe stack objects by DataGuard over the past 10 years.

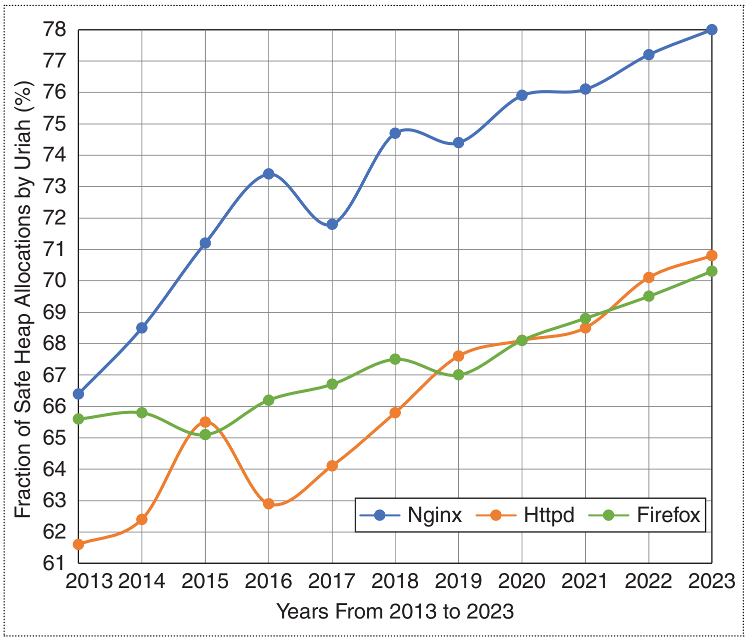

Figure 5. The fraction of safe heap allocations by Uriah over the past 10 years.

and that the object remains valid. Keeping metadata for each object loses the relationship between pointer and object. Instead, it only allows detection if a pointer points outside of all objects. The underlying hypothesis is that pointers will only go out of bounds near the target object. This hypothesis is true for testing where developers aim at finding bugs but fails if an adversary carefully modifies the pointer to go out of bounds into an adjacent object (and not into a red zone). Memory sanitization is therefore useful for testing but insufficient for hardening.

Dynamic testing engines, commonly fuzz testing today, apply memory sanitizers as an oracle that tells them when a bug is triggered. For fuzzing, performance is key. Any cycle spent on sanitizer checks is a cycle not spent running another test, so reducing the overhead is key (e.g., the ASan extension ASan–[9] and FuZZan[8]). When analyzing the performance of the address sanitizer, the key overheads fall into the following categories: startup cost, managing metadata, and individual checks. The startup cost is paid once per execution to initialize the metadata memory structure, which entails allocating a large area of memory. Managing metadata incurs a cost for each allocated object to store the initial management information. During execution, each check incurs a cost.

By increasing the number of objects that can be proven to satisfy classes of memory safety via memory safety validation, we can reduce overheads proportionally. DataGuard and Uriah validate memory safety for each class independently, enabling a reduction of metadata management and checks for those classes found safe. For the safe classes, metadata may be removed entirely. For the per-check cost, optimizations can simplify the checks or fully remove them. In addition to removing checks for more objects that can be proven safe for all memory error classes, we can leverage memory safety validation to remove checks for individual classes proven safe. We will explore other optimizations, such as merging checks for related aliases and for operations dominated by other checks, extending approaches proposed recently.[9] These optimizations can improve the performance of dynamic testing without reducing the precision.

Improving Hardware Enforcement of Memory Safety

Memory safety is a software correctness property, but hardware can offer attractive benefits in either reducing the overhead of enforcing safety properties, e.g., bounds checking or cryptographic authentication acceleration, or by providing new features, such as *capabilities*, that expose safety properties at the instruction set architecture (ISA) level. However, while an increasing number of features have been deployed in recent years, such as Intel MPK and cryptographic pointer authentication support, these features are almost universally *opt in* with respect to effective use; hardware is ill equipped to understand the *intended* memory safety semantics of an arbitrary binary. Thus, static analyses and/or explicit programmer use of intrinsics remain a fundamentally necessary component in utilizing the breadth of hardware features intended to enhance memory safety.

While existing hardware support is employed to enhance memory safety, ubiquitous adoption, by software, of any particular hardware mechanism as the basis to support memory safety remains elusive. Among the limiters of adoption are that many commercial hardware memory safety mechanisms are platform specific and reduce code portability. This, in turn, slows the automation of toolchains looking to map the memory accesses of unsafe languages into the specific constraints needed to utilize a heterogeneous offering of protection, enforcement, and violation detection techniques. Further, acutely finite bounds in many hardware mechanisms have either made them hard to apply (e.g., how to map more than 16 domains into MPK's 4-bit space) or susceptible to attacking the hardware security mechanism directly (e.g., the PACMAN attack on Apple M1 pointer authentication).

The focus, as exemplified by DataGuard[4] and Uriah,[5] on identifying inherently safe objects and protecting them from unsafe interactions aligns well with the strengths and limitations of hardware support for memory safety. Provably safe objects not only allow elision of explicit checks, they also elide tracking of associated metadata, which aligns well with the finite resources hardware can devote to tracking it. By collapsing the most common (and, as seen in Figures 4 and 5, increasingly common) case of safe objects into a single safety class, approaches with small domain counts (e.g., 4 bits for MPK) become more practical to employ, and the working set of metadata for approaches like capabilities becomes more scalable. Similarly, focusing on the protection of safe objects aligns with hardware design and verification complexity: protecting known-safe objects from being accessed by any pointer not similarly proven safe has well-defined semantics and limited state and can be encoded as a static property of a memory access instruction, whereas covering all possible origins and implications of unsafe accesses requires substantial generality.

One of the more promising efforts in the evolving space of hardware support for memory safety is the CHERI[10] project that seeks to provide a *common* set of capability-based features that can be added as extensions to *any* RISC style architecture. The Morello platform represents an application of CHERI to the ARM ISA, showing its practical viability to industry standard designs. While replacing pointer-based access with hardware-native capabilities addresses the protected memory error classes and coverage aspects of the 3-C principle, prior capability-based architectures have often run aground on cost: capabilities are fundamentally larger than pointers, involve more computation, and require managing (even with hardware support, such as capability caching) larger amounts of metadata. Along similar lines to the DataGuard and Uriah approaches, the Morello toolchain heavily leverages static analysis to mitigate the number of distinct capabilities required: all provably safe stack accesses share the *same* capability ranging over the entirety of the safe stack, vastly reducing runtime overheads. Whether it be through capabilities (as with CHERI), memory protection domains, or other novel mechanisms to enforce isolation of safe objects from the unsafe, current trends indicate that making the common case of protecting provably safe objects cheap and easy, more so than any particularly clever means of curtailing a given class of unsafe accesses, will be a foundational component of any hardware memory safety support that achieves widespread adoption.

Improving the Use of Privilege Separation

Privilege separation in software refers to separating a software application into multiple modules, each with its own set of privileges and each loaded into separate *protection domains*. Privilege separation provides a coarse-grained notion of memory safety in the sense that a protection domain's code can only access its own domain's memory and is prevented from accessing any other domains' memory directly. A buffer overflow within a domain remains possible, but such a buffer overflow cannot read or modify memory in another domain. This notion of cross-domain memory safety can be enforced using either software-based fault isolation or one of many hardware-based mechanisms.

Privilege separating programs in unsafe languages such as C/C++ is especially beneficial to security because they are prone to memory errors. For example, OpenSSH was refactored to create unprivileged monitor domains for handling user connections that were prone to memory errors and to isolate one privileged server domain.[11] Such efforts, however, relied on labor intensive manual efforts, but recent work has shown that many tasks in privilege separation can be automated,[12] even for kernel,[13] which reduces threats due to memory safety for privileged domains.

While automated privilege separation prevents the exploitation of memory errors among domains, memory safety validation may address some limitations. First, even after privilege separation, the data conveyed from unprivileged domains may cause memory errors in privileged domains. For example, an unprivileged domain may provide a value used to compute a memory reference in the privileged domain (e.g., an offset), which may cause a memory error when used in the privileged domain. One possible solution is to apply memory safety validation to determine the constraints on inputs from unprivileged domains to provably prevent memory errors. We will have to explore the fraction of cases for which such constraints can be derived for the privileged domain code. Second, memory safety validation could be used to reduce the size of unprivileged domains or the frequency of domain crossings among domains. For example, memory safety validation may enable us to find that a significant fraction of the unprivileged domain's code is memory safe, perhaps enabling optimizations that improve the performance of privilege-separated systems. We will have to explore how knowledge of memory safety in unprivileged domains can improve security–performance tradeoffs in privilege separation.

Improving Detection of Nonmemory Bugs

Programmers also want to ensure that the secrecy and integrity of their program data are protected for all possible executions. The lack of memory safety in C and C++ has precluded the development of automated analyses to validate secrecy and integrity requirements for programs in these languages. However, memory safety validation may make validation of secrecy and integrity in C/C++ programs practical in some cases.

Researchers have long known how to validate the secrecy and integrity of program data in memory-safe languages. Denning formalized the notion of secure information flow in programs in 1975.[14] There are two types of information flows in programs: 1. *explicit flows*, where a direct assignment (e.g., a = b) implies an information flow (i.e., from b to a), and 2. *implicit flows*, where an assignment that is indirectly dependent on a conditional [e.g., if (c) then a = b] implies an information flow from the conditional expression to another assigned value (i.e., from c to a). All explicit and implicit flows must comply with a security policy for a program to have only secure information flows. Later, Myers defined language extensions to validate that a program has only secure information flows for memory-safe languages[15] (i.e., building an extension of the Java programming language). This approach has since been applied to many programs to validate the secrecy and integrity of sensitive data.

However, programs in unsafe languages, such as C and C++, may have memory errors, which create data flows that are not explicitly defined by a program. For example, a buffer overflow may allow a statement to read or write memory objects other than the objects referenced in the statement (e.g., by accessing memory outside the memory region of the referenced object due to a spatial memory error). This is what occurred in the Heartbleed vulnerability. As a result, information flow analyses for C/C++ programs, when applied, assume memory safety. While the use of information flow analysis assuming memory safety may identify bugs that can occur, even when a memory error is not exploited, such analyses cannot validate that the secrecy and integrity of program data will be protected in all executions (e.g., validate that the secrecy of keys cannot be violated by an error like Heartbleed).

Memory safety validation may enable programmers to validate the secrecy and integrity of some program data in C/C++ programs. The key insight is that the ability to protect the memory safety of a large fraction of program objects means that information flow validation may leverage these objects to validate secure information flows. For example, consider a secret value "key" that is not supposed to be leaked to a public sink p. Currently, we cannot validate the secrecy of "key" in a C/C++ program, because there may be a memory error that enables access to another variable to access "key," as in the Heartbleed bug. With memory safety validation, if we can validate the memory safety of "key" and all the variables dependent on information flows from "key" in the program, then these variables can be isolated from memory errors, preventing access to "key" outside of the program's information flows. If this situation were true for the keys in Heartbleed, then they could have been protected from the bug, even if a bug in an unsafe memory access remained latent.

There are questions to assess before we can determine whether memory safety validation may enable information flow analysis. One question is what fraction of sensitive data are memory safe and only flow to other memory-safe objects. We have previously assessed the fraction of objects used in conditionals that is memory safe (i.e., to assess implicit flows), finding that 91.3% of such objects can be validated to satisfy memory safety. While this is not 100%, programmers may be able to apply defenses to preserve memory safety in a targeted way to enable the validation of secrecy and integrity for critical sensitive data. Again considering Heartbleed, if some object that is dependent on an information flow from a key may be accessed in a way that may violate one or more classes of memory safety, memory safety validation can identify which classes cannot be proven to satisfy memory safety, enabling defenses to be targeted to prevent these violations, enabling validation of information flows.

Memory safety validation emerges as a game-changer in the memory vulnerability war. In this article, we evaluated the state-of-the-art breakthrough defenses that leverage memory safety validation: DataGuard and Uriah. Results show that they deliver coverage for 77% of heap and 85% of stack objects while enforcing all memory safety classes: spatial, type, and temporal. Given the current situation where the number of safe objects increases over time, memory safety validation will become more important. The future holds even brighter possibilities: improving bug detection using fuzzing and information flows, layering with other defensive techniques like privilege separation, and harnessing hardware for dramatic performance boosts. ■

Acknowledgment

Kaiming Huang and Trent Jaeger are the corresponding authors.

References

1. G. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "CCured: Type-safe retrofitting of legacy software," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 3, pp. 477–526, 2005, doi: 10.1145/1065887.1065892.
2. V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *Proc. USENIX Symp. Oper. Syst. Des. Implementation (OSDI)*, 2014, pp. 147–163, doi: 10.1145/3129743.3129748.
3. E. van der Kouwe, T. Kroes, C. Ouwehand, H. Bos, and C. Giuffrida, "Type-after-type: Practical and complete type-safe memory reuse," in *Proc. 34th Annu. Comput. Secur. Appl. Conf. (ACSAC)*, New York, NY, USA: Association for Computing Machinery, 2018, pp. 17–27, doi: 10.1145/3274694.3274705.
4. K. Huang et al., "The taming of the stack: Isolating stack data from memory errors," in *Proc. Netw. Distrib. Syst. Secur. Symp. (NDSS)*, 2022, pp. 1–17, doi: 10.14722/ndss.2022.23060.
5. K. Huang, M. Payer, Z. Qian, J. Sampson, G. Tan, and T. Jaeger, "Top of the heap: Efficient memory error protection for many heap objects," 2023, *arXiv:2310.06397*.
6. D. Dhurjati, S. Kowshik, and V. S. Adve, "SAFECode: Enforcing alias analysis for weakly typed languages," in *Proc. ACM Conf. Program. Lang. Des. Implementation (PLDI)*, 2006, pp. 144–157, doi: 10.1145/1133981.1133999.
7. K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *Proc. USENIX Annu. Tech. Conf. (ATC)*, 2012, pp. 309–318.
8. Y. Jeon, W. Han, N. Burow, and M. Payer, "FuZZan: Efficient sanitizer metadata design for fuzzing," in *Proc. USENIX Annu. Tech. Conf.*, 2020, pp. 249–263.

9. Y. Zhang, C. Pang, G. Portokalidis, N. Triandopoulos, and J. Xu, "Debloating address sanitizer," in *Proc. USENIX Secur. Symp.*, 2022, pp. 4345–4363.
10. P. Neumann and R. Watson, "Capabilities revisited: A holistic approach to bottom-to-top assurance of trustworthy systems," in *Proc. 4th Layered Assurance Workshop*, 2010.
11. N. Provos, M. Friedl, and P. Honeyman, "Preventing privilege escalation," in *Proc. USENIX Secur. Symp.*, 2003, pp. 231–242.
12. S. Liu, G. Tan, and T. Jaeger, "PtrSplit: Supporting general pointers in automatic program partitioning," in *Proc. 24th ACM Conf. Comput. Commun. Secur. (CCS)*, 2017, pp. 2359–2371, doi: 10.1145/3133956.3134066.
13. Y. Huang et al., "KSplit: Automating device driver isolation," in *Proc. 16th USENIX Symp. Oper. Syst. Des. Implementation (OSDI)*, 2022, pp. 613–631.
14. D. E. Denning, "A lattice model of secure information flow," *Commun. ACM*, vol. 19, no. 5, pp. 236–243, 1976, doi: 10.1145/360051.360056.
15. A. C. Myers, "JFlow: Practical mostly-static information-flow control," in *Proc. Symp. Princ. Program. Lang.*, Jan. 1999, pp. 228–241, doi: 10.1145/292540.292561.

Kaiming Huang is a Ph.D. candidate at The Pennsylvania State University, State College, PA 16802 USA. His research interests include software and system security, static and dynamic program analysis for memory safety, reverse engineering, automatic misbehavior detection, and exploit generation. Huang received an M.S. in computer science and Engineering from The Pennsylvania State University. Contact him at kzh529@psu.edu.

Mathias Payer is an associate professor at EPFL, 1015 Lausanne, Switzerland. His research interests include automated vulnerability discovery, compartmentalization of complex code, and developing strong mitigations against exploitation. Payer received a Ph.D. in computer science from ETH Zurich. He is a Senior Member of IEEE. Contact him at mathias.payer@epfl.ch.

Zhiyun Qian is a professor at the University of California, Riverside, Riverside, CA 92521 USA. His research interests include vulnerability discovery, exploit generation, and applied program analysis. Qian received a Ph.D. in computer science and engineering from the University of Michigan. He is a Senior Member of IEEE. Contact him at zhiyunq@cs.ucr.edu.

Jack Sampson is an associate professor at The Pennsylvania State University, State College, PA 16802 USA. His research interests include hardware–software interfaces for semantic-rich memory access, near-data computation, and energy-efficient computer architectures. Sampson received a Ph.D. in computer science (computer engineering) from the University of California, San Diego. He is a Member of IEEE. Contact him at jms1257@psu.edu.

Gang Tan is a professor at The Pennsylvania State University, State College, PA 16802 USA. His research interests include software security, formal methods, and programming languages. Tan received a Ph.D. in computer science from Princeton University. He is a Senior Member of IEEE and a member of ACM. Contact him at gtan@psu.edu.

Trent Jaeger is a professor at the University of California, Riverside, Riverside, CA 92521 USA. His research interests include software and operating systems security. Jaeger received a Ph.D. in computer science and engineering from the University of Michigan. He is a Senior Member of IEEE. Contact him at trentj@ucr.edu.