This article has been accepted for publication in IEEE Security & Privacy. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/MSEC.2024.3363142

Full Spatial and Temporal Memory Safety for C

Santosh Nagarakatte, *Rutgers University, New Brunswick, NJ, 08854, USA*

Abstract—Lack of memory safety is the root cause of many security exploits even today. This paper describes the design decisions, implementation choices, and the trade-offs required to address the under-specification in the C standard and the de-facto C dialects used by applications to enforce full spatial and temporal safety.

Introduction

The C programming language is still the preferred choice for implementing low-level system software because it provides a thin abstraction layer on top of the hardware (*e.g.*, features such as low-level control over memory layout and explicit memory management). There are more than a billion lines of C code in various critical components of the computing stack. Any transition from C to other languages will likely take time and is challenging.

Incorrect usage of some low-level features is undefined behavior according to the C standard, which allows the compiler and the runtime to improve the performance of correct programs. Hence, the C compiler and the runtime often does not add runtime checks to determine whether the programmer uses the low-level features correctly. In the absence of such checks, simple programming errors can become the root cause of security vulnerabilities and exploits.

What is memory safety? Memory safety is a property of the program that ensures that all memory accesses are well-defined according to the language specification. When the compiler and/or the runtime enforces memory safety, the program terminates with a fail-stop exception when an access is not according to the specification. Such an enforcement can prevent a class of security vulnerabilities and exploits that rely on a memory safety error.

Spatial and temporal safety. A program can violate the memory safety property when the memory accesses are to locations that are either beyond the memory allocated for an object or to locations that have not been allocated or have been deallocated. The former is known as a spatial memory safety violation (*e.g.*, bounds errors) and the latter is known as a temporal or lifetime memory safety violation (*e.g.*, use-after-free errors or dangling pointer errors).

Security bugs due to memory safety errors. In the absence of mechanisms to enforce memory safety, simple errors can become serious security bugs. For example, Heartbleed was a result of an out-of-bound read. Matt Miller from the Microsoft Security Response Center in his 2019 BlueHat talk reports more than 70% of the security vulnerabilities addressed in Microsoft's security updates are due to memory safety errors. Similarly, the Chromium security team reports that approximately 70% of the high severity security bugs in the Chromium project are due to memory safety errors. Such errors persist even after continuous fuzz testing for multiple years. Further, the Chromium team reports that the number of serious security bugs that exploit temporal safety errors account for more than half of the memory safety bugs.

The ingenuity of the attacker determines how these vulnerabilities will be exploited. It involves crafting a suitable input in an appropriate context (*e.g.*, making the attack work in the context of address space randomization and non-executable regions). Any buffer overflow, use-after-free error, and/or low-level vulnerability resulting from memory safety violations can compromise the security of the system. Tools such as AddressSanitizer [14] have detected numerous memory safety errors in both user-space and Linux Kernel code by placing guards between objects and tracking whether an object is allocated or not. However, they can also miss many errors (*i.e.*, with large strides and with reallocations). Recent hardware extensions such as ARM's Memory Tagging Extensions and Pointer Authentication provide probabilistic detection of memory safety errors in production systems and make the task of exploiting memory safety errors difficult.

Use of memory-safe languages. Another approach to completely eliminate security bugs resulting from memory safety errors is to use a memory-safe language. Fortunately, these memory-safe languages

XXXX-XXX © 2024 IEEE
Digital Object Identifier 10.1109/XXX.0000.0000000

© 2024 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Security & Privacy. This is the author's version which has not been fully edited and
content may change prior to final publication. Citation information: DOI 10.1109/MSEC.2024.3363142

are getting adoption wherever applicable. Languages such as `Java`, `C#`, and `Go` enforce memory safety by using a strong type system and runtime checks for accesses that cannot be statically checked. Unsafe features such as the type casts are either disallowed or restricted to objects in the same object hierarchy. Automated memory management using a garbage collector ensures that any reachable object is not freed and eliminates temporal safety errors. `Rust`, which is getting adoption in some domains as a safe replacement for C, uses an ownership-based type system and a borrow checker to provide memory safety without garbage collection. Ideally, security critical code should be written in memory-safe languages in the future. In the interim, full memory safety for C will hopefully end the war in memory and enhance the security of our computing systems [15].

What makes enforcing memory safety for C challenging? Given an arbitrary pointer in a program's execution, typically there is no information about the region of memory that is safe to access with that pointer. This problem is exacerbated by the conflation of arrays and pointers (specifically pointers to a single object), frequent type casts between pointers (type punning), and type casts from integers to pointers. In many cases, pointer manipulation in the various dialects of C used in mainstream applications is undefined behavior according to the C standard [6]. Further, the semantics of pointer provenance for memory objects especially with pointer to integer casts is ambiguous in the best interpretation [5]. Reconciling the different interpretations in existing C code and the abstract semantics needed to perform high-level compiler optimizations with the C standard is an open research problem.

A consequence of various low-level features (sometimes undefined or implementation defined behavior) in the dialects of C used in mainstream applications is that a pointer can point to anything! To enforce memory safety, we need to maintain additional information with each pointer to check whether the access is safe. Further, this additional information should be propagated with every pointer operation (sometimes even with integer operations to be consistent with the de-facto C standards used by the applications). The crux of the problem in enforcing memory safety is maintaining enough information (*i.e.*, metadata) with pointers, propagating them on every operation, and checking them before a memory access.

What information should we maintain with each pointer? To completely enforce full spatial and lifetime safety with the de-facto C standards used in mainstream applications, we have to maintain metadata with each pointer and propagate them with operations.

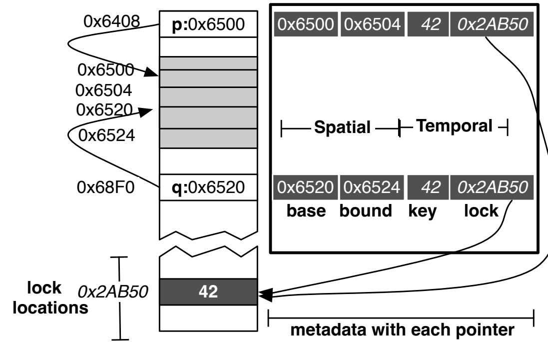

FIGURE 1. Metadata maintained with each pointer. The two pointers **p** and **q** point to two different sub-objects in an aggregate data type. They have different base and bound metadata (*i.e.*, pointer **p** can access memory locations `[0x6500, 0x6504)` and pointer **q** can access memory locations `[0x6520, 0x6524)`) but have the same lock and key metadata because they are part of the same allocation.

This metadata provides a view of memory that the pointer can legally access according to the language specification [10], [12]. To enforce spatial safety, each pointer maintains the start (*i.e.*, `base`) and the end (*i.e.*, `bound`) of the region of memory that the pointer can access. To ensure lifetime safety, we would need to track allocations, design a mechanism to identify all aliased pointers in memory, and invalidate their spatial metadata when a particular object is deallocated. However, identifying such aliased pointers requires additional data structures and is often expensive even with hardware support. In contrast, our approach is to provide a unique identifier (*i.e.*, `key`) and a location in memory (*i.e.*, `lock`) that stores the key on every memory allocation and associate them with the pointer. On a deallocation, we just invalidate the key at the lock associated with the pointer. The invariant is that the key associated with the pointer and the key at the lock associated with the pointer will match for all valid allocated objects [7], [11]. The key associated with the pointer and the key at the lock will mismatch for all pointers that violate lifetime safety. This is the approach used in our SoftBoundCETS project [7], [11], [10]. Figure 1 illustrates the metadata maintained with each pointer for full spatial and temporal safety.

Where should we maintain the additional information? One approach is to maintain the additional information co-located with the pointer (*i.e.*, fat pointer). However, interfacing with external libraries will require deep copies of data structures due to memory layout changes. It also makes incremental adoption difficult.

© 2024 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Security & Privacy. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/MSEC.2024.3363142

In contrast, we advocate for maintaining the metadata in a disjoint metadata space, which avoids the necessity for deep copies of data structures especially while enforcing the safety in the presence of insidious type casts (*e.g.*, the de-facto standards of C used by the applications). If the external library modifies the pointer, then any future use of such a pointer will trigger a memory safety violation because the metadata would not have not been updated.

To enable incremental adoption, we maintain the pointer in the metadata and allow the pointer to access any memory when the pointer produced by the program and the pointer in the metadata mismatch (*i.e.*, the pointer was updated by an external library and the runtime does not have valid information), which is inspired from Intel's Memory Protection Extensions [13]. This allows safety checking to be added incrementally to the code, albeit, at the cost of safety.

Trade-offs. Ideally, we want to have `always-on` full spatial and temporal memory safety for all legacy applications using various de-facto standards with negligible performance overhead. However, these are conflicting goals given the diverse de-facto C standards used in mainstream applications. Hence, any technique will have to make some trade-offs and probably use a combination of various hardware features. The performance overheads can be mitigated with some hardware support [8], [9], [13], [2]. Restricting some of the undefined behaviors statically by disallowing integer-to-pointer casts will require some source code modifications but can reduce the performance overheads significantly. Further, annotations from the programmer (*e.g.*, annotations with the `-fbounds-safety` approach widely used at Apple and Deputy [3]) can reduce the performance cost of enforcing safety. With implementation optimizations, some source code changes, and modest hardware support, full spatial and temporal memory safety can be achieved with approximately 15% performance overhead compared to an unsafe program execution.

The SoftBoundCETS Approach

To enforce full spatial and temporal safety, our approach is to maintain metadata with each pointer that identifies the region of memory that the pointer can safely access and check them before every access. The pointer together with the metadata can be considered as a capability in that the metadata allows the pointer to access a particular region of memory.

We need to identify what metadata to maintain for enforcing spatial and temporal safety, how to propagate them according to the provenance semantics of the de-facto C standards, and how to check them to enforce full spatial and temporal safety. Subsequently, we discuss how to ease the job of incremental adoption with easy interoperation with external libraries.

Metadata for spatial safety. For enforcing spatial safety, we maintain the base and bound of the underlying memory object that the pointer points to. The compiler and the runtime can easily identify the size of the memory region based on the underlying data type and the source or the intermediate code. Identifying such information from stripped binaries will be challenging. When the pointer is assigned to an object with the address-of operator (`int *p = &a;`), then the memory that is accessible through that pointer is restricted to the beginning and the end of the specific object. Similarly, for memory objects allocated on the heap dynamically (*i.e.*, using `malloc`), the base is set to the beginning of the allocation and the bound is set to the end of the allocation. By explicitly maintaining the base and bound metadata, we can support C code that creates out-of-bounds pointers and pointers to the internal elements of objects/structs and arrays. Figure 2(a) illustrates the spatial safety check on a pointer dereference once we maintain the base and the bound metadata.

Metadata for temporal safety. To enforce temporal safety, the runtime creates a unique identifier on every memory allocation both implicit (*e.g.*, stack frames and globals) and explicit (*i.e.*, dynamic memory allocation with `malloc`). The identifier is invalidated on memory deallocation. The idea is to maintain this identifier with each pointer and check whether the identifier is valid before every memory access. By maintaining unique identifiers on every allocation, this approach detects temporal safety violations even in the presence of reallocations and memory reuse.

To make the temporal safety check simple, we organize the identifier metadata as a *key* and a *lock*. The key is an unsigned integer (of an appropriate size) and the lock is the address of the memory location (which we call a lock location) that holds the key [11], [8]. The key and the value at the lock location will match if the underlying memory for the object is valid. The key and lock are also associated with each pointer. All aliased pointers receive the same key and lock. Freeing an allocated region changes the value at the lock location, thereby invalidating other aliased pointers (*i.e.*, dangling pointers) to the region. The lock locations can be reused after the allocation that it guards is deallocated because the keys are unique. Figure 2(b) illustrates the temporal safety check, which is a simple load from the lock and a comparison with the key.

© 2024 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Security & Privacy. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/MSEC.2024.3363142

(a) Spatial safety check

```
if (p < p_base || p + size >= p_bound){
    raise exception();
}
```

(b) Temporal safety check

```
if (p_key != *(p_lock)) {
    raise exception();
}
```

FIGURE 2. Spatial and temporary safety checks done before dereferencing a pointer **p**. The metadata associated with this pointer is `(p_base, p_bound, p_key, p_lock)`.

```
q = p + index;
q_base = p_base;
q_bound = p_bound;
q_key =  p_key;
q_lock = p_lock;
```

FIGURE 3. Metadata propagation on pointer arithmetic with an integer.

In summary, the pointer along with its metadata can be abstractly considered as a capability, `(p, base, bound, key, lock)`*; it allows the pointer to access a particular region of memory that is bounded by the base and the bound and satisfies the temporal safety invariant. The program only manipulates the pointer* `p` *and the other information is present to define whether the execution is safe or not.*

Metadata propagation. Once we have this metadata that defines the region of memory that the pointer can access safely, it needs to be propagated with various operations (*i.e.*, arithmetic, copying, casts, function calls, returns). The semantics of this propagation is straight-forward with pointer casts and when the pointer arithmetic is with integers (*i.e.*, one operand is an integer and other is a pointer). Pointer arithmetic and casts between pointer types just propagate the metadata from the source pointer to the destination pointer. Figure 3 illustrates metadata propagation with pointer arithmetic.

The thorny issues are the corner cases of the C standard. Creating a pointer beyond the end of the memory object is undefined behavior according to the C standard (one-past the end is allowed). Legacy C applications use their own de-facto C standards and

(a) Pointer loads

```
int **p, *q;
...
scheck(p, p_base, p_bound);
tcheck(p_key, p_lock);
q = *p;
q_base = lookup(p)->base;
q_bound = lookup(p)->bound;
q_key = lookup(p)->key;
q_lock = lookup(p)->lock;
```

(b) Pointer stores

```
int **p, *q;
...
scheck(p, p_base, p_bound);
tcheck(p_key, p_lock);
*p = q;
lookup(p)->base = q_base;
lookup(p)->bound = q_bound;
lookup(p)->key = q_key;
lookup(p)->lock = q_lock;
```

FIGURE 4. Metadata propagation through disjoint metadata space on (a) pointer loads and (b) pointer stores. Here, `scheck` and `tcheck` are the spatial safety and temporal safety checks in Figure 2(a) and Figure 2(b), respectively.

create out-of-bound pointers. Further, the propagation of metadata depends on the pointer provenance semantics [5], which is ambiguous in the C standard. Consider the scenario where a pointer is cast to an integer, arithmetic performed over integers, and then cast back to a pointer. What should be the metadata of the resultant pointer? In our approach, we do not track metadata with integers and other non-pointer data types. We support various type of casts between different types of pointers.

Sub-object bounds. Another issue that is ambiguous in the C standard concerning the memory model and the pointer provenance is the creation of pointers to sub-fields of an aggregate data type. What should be the bounds of the resulting pointer? Should it be bounds of the entire memory object or the just bounds of the sub-object? Our approach, by default, narrows the bounds of pointers to sub-objects, which in turn allows us to prevent internal object overflows. Narrowing of bounds can result in false violations for some C use cases. One common example is in the Linux kernel. Linux uses the ANSI C `offsetof()` macro to create a `container_of()` macro, which is used when creating a pointer to an enclosing container structure based only on a pointer to an internal sub-field. Another example is to create a pointer to the specific element of an integer array and then use pointer arithmetic to access the other elements (*e.g.*, `int* p = &arr[4]; b = *(p+1);`. When we narrow the bounds for sub-objects, the above idioms will raise an exception. The programmer can disable narrowing of bounds for such use cases by providing specific attributes to our compiler.

Metadata in disjoint metadata space for pointers in memory. To prevent malicious corruption of metadata and to leave the memory layout of the pro-

© 2024 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Security & Privacy. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/MSEC.2024.3363142

gram intact that eases interfacing with external libraries without deep copies, we maintain the metadata for pointers in memory in a disjoint metadata space. Our runtime maps the pointer in memory to its metadata in the disjoint metadata space, which can be organized using various data structures such as a linear array or a trie. To minimize accesses to the disjoint metadata space, we maintain metadata in registers (*i.e.*, temporaries in the context of the compiler) for pointers in registers. The disjoint metadata space is accessed only when pointers are loaded from or stored to memory. Hence, the cost of accessing the disjoint metadata space is primarily experienced by pointer-chasing code (*i.e.*, linked data structures). Accessing an element of a dynamic allocated array using a pointer just pays the cost of spatial safety and temporal safety checks. We use the address of the pointer to index into the disjoint metadata space (more importantly it is not what the pointer points to). When the program does a dereference to load a pointer (*i.e.*, `q = *p;`), the metadata for pointer `q` is loaded from the disjoint metadata space. The lookup is done using the address of pointer `q`, which is `p`. Figure 4(a) and Figure 4(b) illustrate the disjoint metadata accesses when a pointer is loaded from or stored to memory.

Safety in the presence of type casts. Our approach does not access the disjoint metadata space when non-pointer values are written (read) to (from) memory. One side effect of instrumenting only pointer operations is that a pointer can be manufactured from an integer implicitly through memory (*e.g.*, using type casts between pointers of two different structure types and writes through memory in some legacy C code). However, our approach allows such a dereference through a manufactured pointer only when the resulting pointer belongs to the same allocation and is within bounds of the object pointed by the pointer before the cast.

Our approach enforces comprehensive memory safety even when we propagate metadata only with pointers because (1) metadata is manipulated/accessed only through the extra instrumentation added, (2) metadata is not corrupted and accurately depicts the region of memory that a pointer can legally access, and (3) all memory accesses are conceptually checked before a dereference [7]. A store operation using a pointer involved in an unsafe type cast can only overwrite pointer values but not the metadata in the disjoint metadata space. When pointers involved in arbitrary casts are subsequently dereferenced, the pointer is checked with respect to its metadata. As the checks use the metadata to ascertain the validity of the memory access and the metadata is never corrupted, our approach provides comprehensive detection of memory safety errors.

Metadata propagation with function calls and returns. Metadata propagation also needs to happen with pointer arguments and return values. The mechanism is important especially when the calls are across the application binary interface and to external libraries. In many instruction set architectures (ISAs), arguments are passed in registers. Furthermore, indirect calls through function pointers can create unsafe type casts.

We explore two methods: (1) passing metadata as extra arguments and (2) passing metadata using a shadow stack. For functions that are not involved in indirect calls and are amenable to the fast calling convention, we pass the metadata as extra arguments. Otherwise, we use a shadow stack for all other function calls including variadic functions. Our shadow stack implementation provides dynamic typing between the arguments pushed at the call site and those retrieved by the callee [7]. The shadow stack prevents the callee from dereferencing a non-pointer value pushed by the caller in the call stack by treating it as a pointer value. In compliance with the de-facto standards, we trigger the exception only when such pointers are dereferenced but not when they are created.

Interfacing with external libraries. The above approach provides full spatial and temporal safety. It also supports separate compilation and link-time optimizations. Separate compilation allows creation of memory-safe libraries. In many cases where we cannot recompile external libraries, we need a mechanism to provide metadata for pointers that are returned and those that are updated in memory (*e.g.*, `qsort` function invocation to sort pointers). In the prior versions of our SoftBoundCETS prototype, we developed wrapper code to interface with external libraries. When an external library function returns a pointer or updates pointers in memory, wrappers provide the glue code between the instrumented code and the external library. Although writing wrappers with our approach does not require deep copies of data structures, it can be tedious to develop when the pointers are updated in memory by the external library function.

Given that incremental adoption of memory safety enforcement is necessary for adoption with legacy code, our latest prototype uses the Intel's MPX idea of redundantly storing the pointer value produced by the program in the disjoint metadata space along with the other metadata. When the external library, which is not hardened with our approach, updates the pointer in memory, it will not touch the disjoint metadata space. When the program subsequently loads the pointer from

© 2024 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Security & Privacy. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/MSEC.2024.3363142

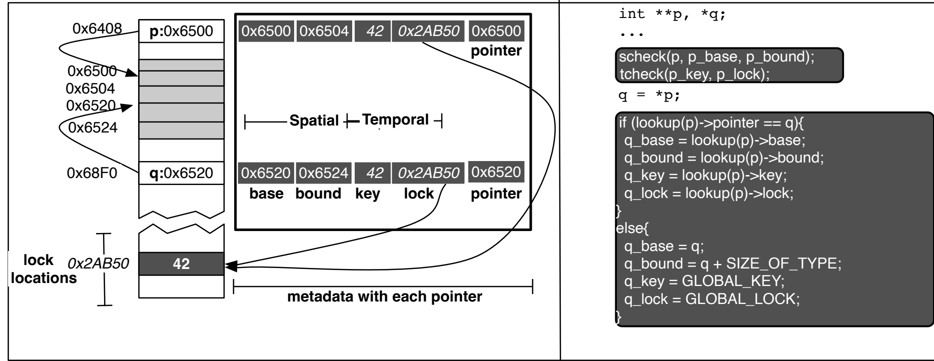

FIGURE 5. Disjoint metadata with the pointer value in the metadata space along with base, bound, key, and lock for every pointer in memory. It enables selective and incremental adoption of memory safety checking. The semantics of safety checking on pointer loads changes a bit to check if the pointer produced by the program matches the pointer in the metadata space to identify updates to memory by external libraries.

memory in the instrumented code, it loads the metadata from the disjoint metadata space. The pointer produced by the program and the one in the disjoint metadata space will mismatch. The programmer can appropriately set bounds for such memory accesses (*e.g.*, treat it as singleton access or even access the entire heap). Figure 5 illustrates the pointer value in the metadata space along with other metadata and check done on pointer loads. We have found this approach to be extremely useful to incrementally apply dynamic monitoring tools in a variety of contexts (*e.g.*, race detection, numerical error detection) beyond memory safety.

Optimizations and Hardware Support

The approach described in the previous section can be accelerated by optimizing the encoding of the metadata in the metadata space to reduce the memory footprint, reducing the number of both spatial and temporal safety checks with static analysis, and accelerating various operations with hardware extensions.

Compressed encoding of the metadata. To keep the exposition simple, we described our approach to maintain five 64-bit pieces of metadata per-pointer on a 64-bit machine. Using 40-bytes of metadata for every pointer can significantly increase the memory usage. To reduce the size of the metadata, we compress our metadata using the following insights: (1) only 48-bits of the virtual address space is currently used on Linux x86-64 machines, (2) pointers are word-aligned, (3) our compiler and the runtime can add padding to make each allocation to be sized as a power of two, and (4) the entire pointer is not needed to identify whether an external library modified the pointer in memory.

We use the above insights to pack the entire metadata in 128-bits. Rather than maintaining the virtual address of the bound, we just maintain the size of the allocation. When every allocation is sized to be a power of two, we just the need to remember the power of two for the bound, which is inspired from the baggy bounds project [1]. Just six bits are sufficient to represent allocation sizes up to 2^{63} bytes (*i.e.*, $bound = base + 2^{size}$). We use only 13-bits of the pointer in the metadata space to identify whether external library modified the pointer in memory or not, which is inspired from random property testing literature. Finally, the compressed base is just 45-bits of the virtual address representing the beginning of the memory object. Using this compression, we have reduced the metadata space required per pointer in the disjoint metadata space for spatial safety to 64-bits. Rather than using the actual virtual address for the locks, we use a 32-bit offset from the start of the lock locations to represent the lock. We also use 32-bit identifiers as the key. Even when we reuse the lock locations and use 32-bit identifiers for the key, the probability of conflict due to reuse is very small because the lock and key together constitute the unique identifier for the allocation.

This encoding reduces the memory overhead significantly even for pointer intensive applications. We do not change the sizes of sub-objects especially when they are used in aggregate types to minimize memory layout changes. Hence, we disable sub-object bounds

© 2024 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Security & Privacy. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/MSEC.2024.3363142

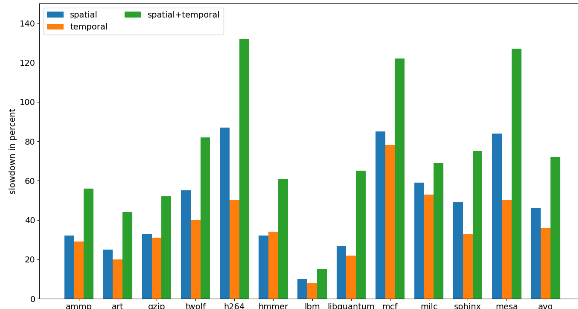

FIGURE 6. Performance overhead for enforcing spatial safety, temporal safety, and full spatial+temporal safety with the latest SoftBoundCETS prototype for a collection of SPEC benchmarks when compared to a native execution without any instrumentation.

to enable metadata compression when the underlying object is not sized to be power of 2. In some cases, this metadata compression can cause false violations because the library updated the pointer but left the 13-bits of the pointer exactly identical.

Optimizations to reduce the number of checks. A wide range of check optimizations based on loop invariant code motion and loop peeling can reduce the number of spatial safety checks. Specifically, we hoist the spatial check inside the loop outside by computing a new check using the weakest-precondition, which can reduce the number of checks by 30% in some applications. In contrast to spatial safety checks, optimizing temporal safety checks need different kinds of static analysis that identifies lifetimes.

Performance overhead of a software-only prototype with compressed metadata. Figure 6 reports the performance overhead of our latest software-only prototype of SoftBoundCETS, which is based on LLVM-17.6, to enforce only spatial safety (left-most bar), only temporal safety (middle bar), and the full memory safety (right-most bar) over an uninstrumented native execution with a collection of SPEC applications. On average, enforcing only spatial safety incurs a performance overhead of 46% over an uninstrumented application. Enforcing only temporal safety incurs a performance overhead of 36% on average. Enforcing full spatial and temporal safety incurs a performance overhead of 72% on average when compared to a native execution without any instrumentation. We found that metadata compression to reduce memory overheads and inlining of all instrumentation are instrumental in reducing the performance overheads.

Hardware support. The above overheads may not be appealing to some applications. Hardware support can further reduce these overheads. We, in the past, have investigated various degrees of hardware support to accelerate the task of enforcing memory safety [8], [9]. Intel's MPX adopts some of the same design decisions from our project. Dedicated metadata registers, specific instructions for spatial and temporal safety checks with compression, and dedicated load and store instructions with the above encoding can enforce memory safety for a wide class of applications at 10-20% performance overhead compared to an unsafe execution. To ensure these bounds checks are not bypassed with speculative execution, the compiler also needs to introduce appropriate barriers (*e.g.*, `LFENCE` and bounds clipping instructions on Intel machines).

Multithreading. To support multithreaded programs, our approach has to ensure the atomicity of the checks and the memory access to prevent time-of-check-to-time-of-use errors and atomicity of the metadata accesses and the memory access. If a pointer operation, temporal safety check and the metadata accesses occur non-atomically, interleaved execution and race conditions can report false errors and miss errors. To avoid them, the compiler instrumentation must ensure that: (1) a pointer load/store's data and metadata access execute atomically, (2) checks execute atomically with the load/store operation, and (3) allocation of metadata is thread safe. We can satisfy requirements #1 and #2 for data-race free programs by inserting metadata access and check instructions within the same synchronization region as the pointer operation. For programs with data races, hardware support in the form atomic wide load/store can help, which needs more research from the community to be practical.

Related Work

There is a large body of work on enforcing memory safety for C. Initially, majority of the techniques were focused on spatial safety. Recently, there is increasing interest to enforce temporal safety by detecting use-after-free errors either using spatial safety metadata or through virtual memory mechanisms. A survey, albeit ten years old, details the various techniques [15].

Our approach is inspired by the seminal work on enforcing spatial memory safety in the CCured [12] and Cyclone [4] projects. Cyclone [4] provided full safety with reasonable performance overheads but it required significant code changes to be compatible with region-based memory management. CCured [12] provided comprehensive spatial safety with a fat pointer representation. It relied on a garbage collector to provide temporal safety. With the use of a fat-pointer,

© 2024 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Security & Privacy. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/MSEC.2024.3363142

CCured required deep copies of data structures to remove metadata while interfacing with external libraries through wrappers. The pointer metadata could be potentially overwritten with unsafe type casts. CCured used a whole program inference to detect such pointers involved in casts (*e.g.*, WILD pointers), which prevented separate compilation and WILD pointers had higher performance overhead. To mitigate these issues, the programs had to be changed/rewritten to avoid such type casts or use run-time type information extensions. Our approach is an effort to address compatibility issues with CCured in an effort to make it easily usable with large code bases while avoiding garbage collection.

Intel Memory Protection Extensions (MPX) provided hardware acceleration with additional instructions for metadata accesses and checks for enforcing spatial safety using disjoint metadata similar to our approach. Intel's pointer checker, which is a software-only prototype, uses compiler instrumentation similar to our SoftBound prototype [10]. We believe that MPX's feature to enable incremental adoption by maintaining the pointer in the disjoint metadata space is a great contribution to the class of dynamic checking tools, which we use with our latest prototype. One reason for MPX's failure to get significant adoption, in our opinion, is the lack of compiler and tooling support in open-source compilers, which resulted in significant performance overheads. Further, Intel's own compiler did not generate optimal instrumentation to effectively use MPX.

CHERI is a hardware capability machine that can provide spatial safety and compartmentalization by converting every pointer into a capability [2]. Each pointer on a 64-bit machine becomes a 128-bit capability and a tag to confirm the validity of the capability. Conceptually, CHERI's capability is similar to CCured's WILD pointer enforced in hardware. The 1-bit validity tag in CHERI exactly has the same purpose as the tag used with CCured's WILD pointers. The key difference is CHERI's hardware enforcement rather than the runtime enforcement with CCured. Our metadata is also a form of capability, albeit without the need for additional validity tags because of the disjoint metadata space. By default, CHERI does not provide temporal safety. Recent proposals in the CHERI project prevent dangling pointers by deferring deallocation on a `free` and reclaiming memory after a sweep to revoke the capabilities. CHERI also explores metadata compression to store base and bound along with the pointer in a 128-bit capability. It also needs to resolve the exact same issues about pointer provenance and the de-facto C standards followed by legacy C code.

ARM recently has introduced Memory Tagging Extensions (MTE) to enforce probabilistic memory safety. MTE maintains a 4-bit tag with each 16-byte block of memory. These tags are stored in a dedicated region of memory. Any pointer that points to the memory block has the 4-bit tag stored in the upper unused bits of the pointer. The memory access is safe if the tag in the upper bits of the pointer and the tag associated with the memory pointed by the pointer match. By assigning different tags for adjacent memory allocations, MTE can enable detection of adjacent spatial safety errors. MTE is a limited form of lock-and-key checking that we described in our approach where lock locations are in a dedicated memory region and the metadata is in the upper bits of the pointer. Given the small number of distinct 4-bit tags, MTE can miss bugs and the protection is probabilistic. However, it is a useful addition to the set of hardware extensions explored for always-on low-overhead enforcement of some memory safety.

Apart from MTE, another recent ARM feature, Pointer Authentication (PAC), stores a short 16-bit cryptographic signature in the upper unused bits of the pointer. The signature is computed using the pointer value, a discriminator to diversify signed pointers, and a signing key. The signature is checked on a pointer dereference to ensure integrity. When used with function pointers and return addresses, it can prevent a class of return-or-jump oriented programming attacks.

Recent efforts such as the `-fbounds-safety` transformation from Apple that is being mainstreamed into LLVM and the CheckedC project incrementally add annotations to make C spatially safe. The `-fbounds-safety` pass requires the developer to provide annotations (*e.g.*, to identify bounds, singletons) to reduce the performance overhead. The main insight in this effort, which was also identified by the Deputy project [3], is that code often has bounds information already present in it. The accompanying code may not be correctly checking the accesses. These annotations enable the runtime to enforce memory safety using a combination of static analysis and runtime checks with a low performance overhead.

Conclusion

Substantial work by the community, including our approach, has developed methods that can be used by the language implementation with or without hardware support to enforce full spatial and temporal safety. Appropriately restricting the de-facto C standards, incremental adoption, and annotations distinguishing pointers to arrays from pointers to non-array objects will

© 2024 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.

This article has been accepted for publication in IEEE Security & Privacy. This is the author's version which has not been fully edited and content may change prior to final publication. Citation information: DOI 10.1109/MSEC.2024.3363142

make the task of hardening legacy C code easier. Hardware support coupled with an efficient streamlined implementation can make the performance overheads of full spatial and temporal safety negligible and can make always-on deployment a reality. Recent efforts such as `-fbounds-safety` and the CheckedC project are promising directions to make new C code safe and incrementally make legacy C code safer.

ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation under Grant 2110861 and research gifts from the Intel corporation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation or the Intel corporation.

REFERENCES

1. Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the 18th USENIX Security Symposium*, August 2009.
2. David Chisnall, Colin Rothwell, Robert N.M. Watson, Jonathan Woodruff, Munraj Vadera, Simon W. Moore, Michael Roe, Brooks Davis, and Peter G. Neumann. Beyond the pdp-11: Architectural support for a memory-safe c abstract machine. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, page 117–130, New York, NY, USA, 2015. Association for Computing Machinery.
3. Jeremy Condit, Matthew Harren, Zachary Anderson, David Gay, and George C. Necula. Dependent types for low-level programming. In *Proceedings of the 16th European Symposium on Programming*, 2007.
4. Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of c. In *Proceedings of the 2002 USENIX Annual Technical Conference*, 2002.
5. Kayvan Memarian, Victor B. F. Gomes, Brooks Davis, Stephen Kell, Alexander Richardson, Robert N. M. Watson, and Peter Sewell. Exploring c semantics and pointer provenance. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019.
6. Kayvan Memarian, Justus Matthiesen, James Lingard, Kyndylan Nienhuis, David Chisnall, Robert N. M. Watson, and Peter Sewell. Into the depths of c: Elaborating the de facto standards. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, page 1–15, New York, NY, USA, 2016. Association for Computing Machinery.
7. Santosh Nagarakatte. *Practical Low-Overhead Enforcement of Memory Safety for C Programs*. PhD thesis, University of Pennsylvania, 2012.
8. Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Watchdog: Hardware for safe and secure manual memory management and full memory safety. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, 2012.
9. Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Watchdoglite: Hardware-accelerated compiler-based pointer checking. In *12th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO '14*, page 175, 2014.
10. Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In *Proceedings of the SIGPLAN 2009 Conference on Programming Language Design and Implementation*, 2009.
11. Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. Cets: Compiler enforced temporal safety for c. In *Proceedings of the 2010 International Symposium on Memory Management*, 2010.
12. George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems*, 27(3), May 2005.
13. Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel mpx explained: A cross-layer analysis of the intel mpx system stack. In *Abstracts of the 2018 ACM International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '18, page 111–112, New York, NY, USA, 2018. Association for Computing Machinery.
14. Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. Addresssanitizer: A fast address sanity checker. In *Proceedings of the USENIX Annual Technical Conference*, 2012.
15. Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, 2013.

© 2024 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission. See https://www.ieee.org/publications/rights/index.html for more information.