CUP: Comprehensive User-Space Protection for C/C++

Nathan Burow
nburow@purdue.edu
Purdue University

Derrick McKee
mckee15@purdue.edu
Purdue University

Scott A. Carr
carr27@purdue.edu
Purdue University

Mathias Payer
mathias.payer@nebelwelt.net
Purdue University

ACM Reference Format:
Nathan Burow, Derrick McKee, Scott A. Carr, and Mathias Payer. 2018. CUP: Comprehensive User-Space Protection for C/C++. In *ASIA CCS '18: 2018 ACM Asia Conference on Computer and Communications Security, June 4–8, 2018, Incheon, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3196494.3196540

Abstract

Memory corruption vulnerabilities in C/C++ applications enable attackers to execute code, change data, and leak information. Current memory sanitizers do not provide comprehensive coverage of a program's data. In particular, existing tools focus primarily on heap allocations with limited support for stack allocations and globals. Orthogonally, existing tools focus on the main executable with limited support for system libraries. Existing tools also suffer from both false positives and false negatives.

We present Comprehensive User-Space Protection for C/C++, CUP, an LLVM sanitizer that provides complete spatial and probabilistic temporal memory safety for C/C++ programs on 64-bit architectures (with a prototype implementation for x86_64). CUP uses a hybrid metadata scheme that supports all program data including globals, heap, or stack and maintains Application Binary Interface (ABI) compatibility. Existing approaches have false positives and 8%-25% false negatives on the NIST Juliet test suite. In contrast, CUP has no false negatives or false positives. CUP instruments all user-space code, including libc and other system libraries, removing these libraries from the trusted computing base. Supporting all of user space allows CUP to treat a missed check as a failed check, leading to no false negatives for CUP. The overhead introduced by CUP is half that of the state-of-the-art full memory protection on benchmarks where both mechanisms run, and imposes 1.58x overhead when compared to baseline on all benchmarks. Consequently, CUP is intended as a sanitizer for use by system developers, and to protect truly critical systems.

1 INTRODUCTION

Despite extensive research into memory safety techniques, only a few mechanisms have been deployed, and exploits of memory corruptions remain common [22, 36, 38, 40]. These attacks rely on the fact that C/C++ require the programmer to manually enforce spatial safety (bounds checks) and temporal safety (lifetime checks). As the continuing stream of memory corruption Common Vulnerabilities and Exposures (CVEs) shows, these programmer added checks are often inadequate. Many of these bugs are in network facing code such as browsers, e.g., CVE-2016-5270, CVE-2016-5210, and servers, e.g., CVE-2014-0226, CVE-2014-0133, allowing attackers to illicitly gain arbitrary code execution on remote systems. Consequently, a memory safety sanitizer that *comprehensively* protects user-space is necessary to find and fix these bugs.

To correctly address memory safety in user-space, there are four main requirements. *Precision* addresses spatial safety by requiring that exact bounds are maintained for all allocations. *Object Awareness* prevents temporal errors by tracking whether the pointed-to object is currently allocated or not. These two requirements are sufficient to enforce memory safety. Adding *Comprehensive Coverage* expands this protection to all of user space by requiring that all data on the stack, heap and globals be protected. *Comprehensive Coverage* implies that all code must be instrumented with the sanitizer, including system libraries like libc. A sanitizer that meets these three requirements is powerful enough to find all memory corruption vulnerabilities in user-space programs. To be useful, such a sanitizer must also be practical. Requiring *Exactness* — no false positives and minimal false negatives — ensures that bugs reported by a sanitizer are real, and that all spatial and most temporal violations are found. We discuss these challenges in § 3.

The research community has come up with many approaches that attempt to address memory safety. State of the art defenses can be divided into two categories: probabilistic and deterministic. Probabilistic defenses [3, 32, 35] can be bypassed by a sophisticated attacker, but provide lower overhead. Deterministic defenses [12, 20, 28, 29] cannot be bypassed but come with significant overhead. Modern defenses respect the Application Binary Interface (ABI), but may alter the memory layout of the program by, e.g., injecting "red zones" [35] or imposing additional alignment constraints [12].

Modern defenses still do not fully address the requirement we identify for a memory safety solution. *Comprehensive Coverage* is largely an open problem, with prior work mostly ignoring the stack and neglecting support for system libraries like libc. Low-Fat Pointers [12] is the only work to protect the stack — providing only spatial safety. No existing work provides spatial and temporal safety comprehensively for all user-space data (stack, heap, globals) and code (program code, libc, libraries). Doing so requires a large amount of additional metadata to protect the extra allocations (§ 2.1), which existing schemes are unable to handle. Further, existing tools' overhead per check does not allow them to scale to handle the additional memory surface of the stack and libc. *Comprehensive Coverage* weaknesses are exacerbated by the fact that existing works [12, 28, 35] can miss a check without consequence as the execution continues normally. If a pointer is not instrumented, e.g., for hand written assembly code, external uninstrumented code, or pointers passed from the kernel to user-space, these existing schemes continue without raising a fault. In contrast, a defense mechanism that protects all of user space can require that all pointer dereferences fail by default. The failure then alerts the programmer to the issue, allowing her to annotate the inline assembly code, kernel system call, or recompile the uninstrumented code. By failing by default, a mechanism with *Comprehensive Coverage* that executes a program also guarantees that all pointer dereferences were checked.

Libc is the most commonly used third party library, and a particularly critical part of user-space to protect. It is prone to memory errors, notably the `mem*` and `str*` family of functions (e.g., `memcpy` or `strcpy`). SoftBound, for example, uses intrinsics for some of the functions to catch memory safety issues in their use. Memory errors in libc are not limited to these functions, however as shown by, e.g., GHOST [22], a stack overflow in `getaddrinfo` [36], and a one byte corruption in glibc's `malloc` which led to ASLR breaks and arbitrary execution [15]. Consequently, the *entire* libc needs to be protected, not just certain interfaces. Libc is also the primary interface with the kernel. This is a natural boundary to unprotected code for a user-space defense mechanism. Given its prevalence, vulnerability, and boundary to unprotected code, supporting libc gives strong evidence that a defense mechanism is robust enough to protect all of user-space.

Exactness shows how well a memory safety solution protects against vulnerabilities in practice. The U.S. National Institute of Standards and Technology (NIST) maintains the Juliet test suite. Juliet consists of thousands of examples of bugs, grouped by class from the Common Weakness Enumeration (CWE). Juliet reveals that existing, open source memory safety solutions [28, 29, 35] have both false positives and false negatives (§ 5.2).

CUP satisfies all four requirements for a powerful, usable memory sanitizer. We introduce a new hybrid metadata scheme which is capable of storing and using *per object* metadata for the stack, libc, heap, and globals. Our metadata is precise and does not require altering the program's memory layout. Additionally, we introduce a new way to check bounds that leverages hardware to increase our check's performance. Hybrid metadata allows us to meet the *Precision* and *Object Awareness* requirements. CUP presents a novel use of escape analysis to reduce the amount of instrumented stack allocations without loss of protection. This reduction allows scaling our mechanism to include all user-space data, satisfying the *Comprehensive Coverage* requirement. By modifying all pointers so that dereferences fail by default, CUP guarantees that every pointer dereference is checked. Further, CUP successfully handles all system libraries, including libc, the first memory sanitizer to do so. Consequently, we guarantee that all user-space pointers are always protected, even after passing them across the kernel boundary. Our evaluation on Juliet (§ 5.2) shows that we have no false positives or negatives, considerably advancing the state of the art for *Exactness*. Further, this precision is achieved with *half* the overhead of SoftBound+CETS, the state of the art for full memory safety, on SPEC CPU2006 benchmarks where both run (CUP runs a super set of the benchmarks that SoftBound+CETS does).

Contributions

We present the following contributions:

- A new hybrid metadata scheme capable of tracking runtime information for all object allocations, and show how it can be applied to memory safety.
- The first sanitizer to fully protect user-space, including libc
- By design, CUP guarantees that every pointer dereference is proven safe statically or checked at run time. Missing checks halt execution.
- A new static analysis for determining what stack variables require active protection, and present a local protection scheme for non-escaping stack variables
- Evaluation of a CUP prototype that, using our hybrid metadata model, results in (i) no false positives and no false negatives on the NIST Juliet C/C++ test suite and (ii) reasonably low overhead (in line with other sanitizers).

2 CHALLENGES AND BACKGROUND

Our requirements for *Precision* and *Object Awareness* are designed to enforce spatial and temporal memory safety, which we define here and then use to introduce the notion of a capability ID.

Spatial Vulnerabilities, also known as bounds-safety violations are over/under-flows of an object. Over/under-flows occur when a pointer is incremented/decremented beyond the bounds of the object that it is currently associated with. Even if the out-of-bounds pointer points to another valid object, it does not have the capability to access that object, and the operation results in a spatial memory safety violation. However, this violation is *only* triggered on a dereference of an out-of-bounds pointer. The C standard specifically allows out-of-bounds pointers to exist.

Temporal Vulnerabilities, or lifetime-safety violations, occur when the object that a pointer's capability refers to is no longer allocated and that pointer is dereferenced. For stack objects, this is because the stack frame of the object is no longer valid (the function it was created in returned); for heap objects, this happens as a result of a free. These errors do not *necessarily* cause segmentation faults (accesses to unmapped memory), because the memory may have been reallocated to a new object. Similarly, we cannot simply track what memory is currently allocated, because the object at a particular address can change, which still results in a temporal safety violation. Temporal bugs are at the heart of many recent exploits, e.g., for Google Chrome or Mozilla Firefox as shown in the pwn2own contests [39].

Violating either type of memory safety can be formulated as a capability violation. In our terminology, an object is a discrete memory area, created by an allocation regardless of location (stack, heap, data, or bss under the Linux ELF format). A capability identifies a specific object, along with information about its bounds

Memory Type	Allocations	
global	3,900	0.00%
heap	425,433	0.07%
stack	621,043,816	99.90%

Table 1: Allocation distribution in SPEC CPU2006.

and allocation status. Pointers retain a capability ID that identifies the capability of the object that was most recently assigned — either directly from the allocation or indirectly by aliasing another pointer [16]. Capabilities form a contract, upon dereference: (i) the pointer must be in bounds, and (ii) the referenced object must still be allocated. Violating the terms of this contract leads to spatial or temporal memory safety errors respectively.

2.1 Comprehensive Coverage Challenges

Understanding the scope of the challenge presented by *Comprehensive Coverage* is critical to understanding CUP's design. To illustrate this challenge, we show how intensively programs use different logical regions of memory. While the operating system presents applications with a contiguous virtual memory address space, that address space is partitioned into three logical groups for data: global, heap, and stack spaces.

Stack allocations account for almost all (99.9%) of memory allocations in SPEC CPU2006 (see Table 1), and are not fully handled by existing work § 7. This measurement *includes* allocations made in libc. While memory usage in SPEC CPU2006 may not be representative of deployed applications, it is pervasively used to compare new defense techniques. Defenses that only handle heap objects thus gain a significant advantage by protecting 1,000x fewer objects.

Stack allocations matter in practice as well. The latest data from van der Veen, et. al. [40, 43] show that stack-based vulnerabilities are responsible for an average of over 15% of memory related CVEs annually since tracking began in November 2002. By comparison, heap-based vulnerabilities account for an average of 25% of memory related CVEs over the same time period. Given the stack's exploitability and prevalence, which stresses memory safety designs, protecting it is a key design challenge for memory safety solutions.

3 DESIGN

CUP provides precise, complete spatial memory safety and stochastic temporal memory safety by protecting all program data, including libc (and any other library). Safety is enforced, for all program data, by dynamically maintaining information about the size and allocation status of all objects that are vulnerable to memory safety errors. This information is recorded through our novel hybrid metadata scheme (§ 3.1). A compiler-based instrumentation pass is used to add code that records and checks metadata at runtime (§ 3.2). We provide a detailed argument for why our instrumentation guarantees memory safety in § 3.3.

A powerful usable memory sanitizer must comply with the following requirements:

I *Precision.* The solution must enforce exact object bounds, ideally without changing the memory layout (i.e., spatial safety).

II *Object Awareness.* The solution must remember the allocation state of any object accessed through pointers (i.e., temporal safety).

III *Comprehensive Coverage.* The solution must fully protect a program's user-space memory including the stack, heap, and globals, requiring instrumentation and analysis of all code, including system libraries such as libc (i.e., completeness).

IV *Exactness.* The solution must have no false positives, and any false negatives must be the result of implementation limitations, not design limitations (i.e., usefulness).

These requirements drive the design of CUP. Fully complying with the *Precision* and *Object Awareness* requirements requires creating metadata for all allocated objects. While it is possible [2, 12] to do alignment based spatial checks without metadata, these schemes lose precision, alter memory layout, and cannot support *Object Awareness*. *Object Awareness* for temporal checks requires metadata to lookup whether the object is still valid [29] or to find all pointers associated with an object and mark them invalid upon deallocations [20]. Consequently, CUP is a metadata based sanitizer.

CUP provides *Comprehensive Coverage*, and in particular protects globals, the heap, and stack by instrumenting all code, including libc. Our hybrid metadata scheme scales to handle the required number of allocations (§ 2.1), and our bounds check leverages the x86_64 architecture (§ 4.2.2) to perform the required volume of checks quickly enough to be usable. Additionally, our compiler pass is robust enough to handle libc (§ 4.3), making CUP the first memory sanitizer to do so. Protecting libc allows CUP to increase coverage to all user space code, reducing the TCB to the kernel and the CUP runtime library. Further, by protecting libc, CUP demonstrates that it can correctly handle the kernel interface, and guarantee that all user-space pointers are always protected.

Exactness is achieved, and *Comprehensive Coverage* verified, by *failing closed*, making a missed check equivalent to a failed check. We modify the initial pointer returned by object allocation (§ 3.2), and our modification marks it illegal for dereference. This modification propagates through aliasing and all other operations naturally. Consequently, we *must* check all uses of the pointer for the program to execute correctly, enabled by our comprehensive coverage of program data, and support of libc. Such an approach results in optimal precision, and requires novel design decisions as shown in § 4, but removes all false negatives. False positives are prevented by maintaining accurate metadata, and having it propagate automatically.

3.1 Hybrid Metadata Scheme

To provide *Precision*, *Object Awareness*, and *Comprehensive Coverage*, we introduce a new hybrid metadata scheme that lets us embed a capability ID in a pointer without changing its bit width. This capability ID ties a pointer to the capability metadata for its underlying object. *Precision* is provided by the metadata containing exact bounds for every object, and by not rearranging the memory layout. *Object Awareness* results from having a unique metadata entry for each capability ID.

Providing *Comprehensive Coverage* requires assigning a capability ID to all vulnerable objects in order to associate their pointers with the object's capability metadata. However, the capability ID space is fundamentally limited by the width of pointers. To address this limit, we allow capability IDs to be reused. Consequently, our capability ID space only needs to support the maximum number of *simultaneously* allocated objects. This allows CUP to *comprehensively cover* globals, the heap, and stack for all allocations in long running applications.

Our metadata scheme that draws inspiration from both fat-pointers and disjoint metadata (and is thus a "hybrid" of the two) for 64-bit architectures. We conceptually reinterpret the pointer as a structure with two fields, as illustrated by Listing 1. The first field contains the pointer's capability ID. The second field stores the offset into the object. This does not change the size of the pointer, thus maintaining the ABI. Further, when pointers are assigned, the capability ID automatically transfers to the assigned pointer without further instrumentation. Enriching pointers in this manner causes unchecked dereferences to fail by default (see § 4.1), verifying at runtime that all necessary checks have been performed.

Hybrid metadata rewrites pointers to include the capability ID of their underlying object and current offset, creating *enriched* pointers. The size of the offset field limits the size of supported object allocations. The tradeoffs of the field sizes and our implementation decisions are discussed in § 4.1. While we use it for memory safety, this design allows access to arbitrary metadata, and could be applied for, e.g., type safety, or any property that requires runtime information about object allocations.

An application's address space is not affected by our hybrid metadata scheme. All available bytes (2^{48} in 64-bit architectures) remain usable. We simply reinterpret what a pointer means, but the pointer can conceptually be stored anywhere within an application's address space.

Capability IDs in our hybrid-metadata scheme are indexes into a metadata table. Each entry in this table is a tuple of the *base* and *end* addresses for the memory object, required for spatial safety checks. Each object that is currently allocated has an entry in the table, leading to a memory overhead of 16 bytes per allocated object. Note that we do not require per-pointer metadata due to our hybrid scheme. To reduce the number of required IDs to the number of *concurrently* active objects, we allow capability IDs to be reused. Allowing ID reuse thus allows us to protect long running programs, as our limit is on concurrent pointers, not total allocations supported. The security impact of ID reuse is evaluated in § 6.

The metadata table provides strong probabilistic *Object Awareness.* For a temporal safety violation to go undetected, two conditions must hold. First, the capability ID must have been reused. Second, the accessed memory must be within the bounds of the new object. Current heap grooming techniques [14, 37] already require a large number of allocations to manipulate heap state. Adding the requirement that the same capability ID also be used makes temporal violations harder. § 6 contains other suggestions to further increase the difficulty.

We are aware of two memory safety concerns for hybrid metadata: (i) arithmetic overflows from the offset to the capability ID, and (ii) protecting the metadata table. The first concern is addressed by operating on the two fields of the pointer separately. By treating them like separate variables — while maintaining them as one entity in memory — we prevent under/over flows from the offset field modifying the capability ID. The second concern is not relevant

```
struct pointer_fields {
  int32 enriched : 1;
  int32 id : 31;
  int32 offset;
}

union enriched_ptr {
  struct pointer_fields capability;
  void *ptr;
}
```

Listing 1: Enriched Pointer

for CUP— if all memory accesses are checked, then the metadata table cannot be modified through a memory violation. As CUP is designed to protect all of user space, including libraries like libc, *all* memory accesses are actually checked.

3.2 Static Analysis

Our static analysis identifies when objects are allocated or deallocated, and when pointers are dereferenced through an intra-procedural analysis. All pointers passed inter-procedurally are instrumented using our metadata scheme, including all heap allocations (which are manually identified, see § 4.3).

Our analysis divides protected stack allocations into (i) escaping and (ii) non-escaping allocations. An allocation does not escape if the following holds: (i) it does not have any aliases, (ii) it is not assigned to the location referenced by a pointer passed in as a function argument, (iii) it is not assigned to a global variable, (iv) it is not passed to a sub-function (our analysis is intra-procedural excluding inlining), and (v) is not returned from the function. For those that escape, we use our usual metadata scheme so that the bounds information can be looked up in other functions. For those that do not escape, we use an alternate instrumentation scheme.

The optimized instrumentation for non-escaping stack variables creates local variables with base and bounds information. Since these allocations are *only* used within the body of the function, we use local variables for checks instead of looking up the bounds in the metadata table. This reduces pressure on our capability IDs, helping us to achieve *Comprehensive Coverage.*

All other allocation sites requiring metadata are instrumented to assign the object the next capability ID and to create metadata (recording its precise *base* and *end* addresses) — returning an enriched pointer. We create metadata at allocation because it is the only time that we are guaranteed to know the size of the object.

Identifying deallocations for objects is straightforward. Global objects are never deallocated over the lifetime of the program. Heap objects are explicitly deallocated by, e.g., `free()` or `delete`. Stack objects are implicitly deallocated when their allocating function returns. Deallocations are instrumented to mark associated metadata invalid and to reclaim the capability ID.

Pointer dereferences are found by traversing the use-def chain of identified pointers. Dereferences are analyzed intra-procedurally, so we include pointers from function arguments (including variadic arguments) and pointers returned by called functions in the set of allocations for this analysis. We instrument dereferences with a bounds check. Note that the bounds check implicitly checks that the pointer's capability ID identifies the correct object. See § 3.3 for a discussion of the safety guarantees.

CUP also inserts instrumentation to handle int to pointer casts. These are commonly inserted by LLVM during optimization, and have matching pointer to int casts in the same function. In this case, and any others where we can identify a matching pointer to int cast, we restore the original capability ID to the pointer. To date, we have not found a case where an int is cast to a pointer without a matching pointer to int cast.

3.3 Memory Safety Guarantees

We discuss how CUP guarantees spatial memory safety and probabilistically provides temporal safety. We assume that all code is instrumented and capability IDs are protected against arithmetic overflow (as proposed).

For code that we instrument, we keep a capability ID (and thus metadata) for every memory object that can be accessed via a pointer. This subset is sufficient to enforce spatial memory safety. Objects that are not accessed via pointers are guaranteed to be safe by the compiler (if you are reading an `int`, it will always emit instructions to read the correct 4 bytes from memory).

Pointers can be used to read or write arbitrary memory. Further, the address that they reference is often determined dynamically. Thus, pointers require dynamic checks at runtime for memory safety guarantees. As defined in § 2, memory objects define capabilities for pointers. These capabilities include the size and validity of the object. We only create capabilities when objects are allocated at runtime. Objects can change size due to, e.g., `realloc()` calls, in which case we update our metadata appropriately by changing *base* and *end* to the new values (§ 4.2.2). Thus, we always have correct metadata for every object that has been created since the start of execution. The metadata for objects that have not been created yet is invalid by default.

Pointers can receive values in five ways. First, pointers can be directly assigned from the memory allocation, e.g., through a call to `malloc()`. We have instrumented all allocations to return instrumented pointers. Second, they can receive the address of an existing object, via the & operator. We treat this as a special case of object allocation and instrument it. Third, pointers can be assigned to the value of another pointer. As all existing pointers have been instrumented under the first two scenarios, this case is covered as well. Fourth, pointers can be assigned the result of pointer arithmetic. This is handled naturally, with our separate loads preventing overflows into the capability ID.

The fifth scenario is a cast from an int to a pointer. This is exceedingly rare in well written user-space code. However, the compiler frequently inserts these operations in optimized code. As a result, we have to allow these operations. We assume that all ints casted to pointers were previously pointers, and thus instrumented.

Variadic arguments are also given a capability ID when passed to variadic functions. This ensures that variadic functions can only read the arguments explicitly given, and, since all pointers passed to functions are individually given their own capability ID, all writes (e.g., when `%n` is used in the format string given to `printf`) are equally protected.

So far we have established that all pointers are enriched with capabilities that accurately reflect the state of the underlying memory object. Memory safety violations occur when pointers are dereferenced [38]. We instrument every dereference to check the pointers capability and ensure that the dereference is valid. Because each pointer has a capability and each capability is up-to-date this ensures full memory safety.

A program is memory safe before any pointer dereference happens. We have shown that each type of pointer dereference is protected. Consequently, if our checks are correct, every pointer dereference is valid. Thus, showing memory safety for the program depends on showing that our checks are correct.

Spatial safety requires a simple bounds check. The correctness of this check depends only on the validity of the bounds used. We have shown that the capability ID associated with a pointer, and the metadata that ID references are always correct, which in turn implies that the spatial safety check is correct.

Temporal safety in our system requires that either: i) the capability ID has not been reused, or ii) the pointer not be in bounds for the new capability ID. CUP allows capability ID reuse, so our temporal guarantees depend on how difficult exploiting ID reuse is. To successfully reuse an ID, the attacker needs to dictate the location of the new object assigned to the ID that she wants to reuse. Randomized memory allocators such as DieHard [3, 32] make controlling the memory location of a particular allocation extremely difficult. Randomizing ID reuse adds another independent variable for an attacker to control. As an example, with 20 bits of entropy from the allocator and 10 bits of entropy from randomized IDs, a total of 2^{30} allocations would be required to defeat our temporal protection.

4 IMPLEMENTATION

We implemented CUP on top of LLVM version 4.0.0-rc1. Our compiler pass is ≈2,500 LoC (lines of code), the runtime is another ≈300 LoC for ≈2,800 LoC total. The line count excludes modifications to our libc, which required only light annotations (§ 4.3). Our pass runs after all optimizations, so that our instrumentation does not prevent compiler optimizations. This also reduces the total amount of memory locations that must be protected, reducing capability ID pressure.

Here we discuss the technical details of how we implemented CUP in accordance with our design (§ 3). We first discuss how our hybrid metadata scheme is implemented. Next we present how we find the sets of allocations and dereferences required by our design. With the metadata implementation in mind, we then show how we instrument allocations and dereferences. With these details established, we discuss the modifications required to libc for it to work with CUP.

4.1 Metadata Implementation

Our metadata scheme consists of four elements: (i) a table of information, (ii) a bookkeeping entry for the next entry to use in that table, (iii) a free list (encoded in the table) that enables us to reuse entries in the table, and (iv) how to divide the 64 bits in a pointer between the capability ID and offset in our enriched pointers (§ 3.1). Our metadata table is maintained as a global pointer to a `mmap`'d

```c
uint_32 next_entry = 1;

//This is done inline, functions are illustrative
void *on_allocation(size_t base, size_t end){
    size_t offset = table[next_entry].base;
    table[next_entry].base = base;
    table[next_entry].end = end;
    uint_32 ret = 0x80000000 & next_entry;
    next_entry = next_entry + offset + 1;
    return (void *)(ret << 32);
}
void on_deallocation(int id){
    table[id].base = next_entry - id - 1;
    table[id].end = 0;
    next_entry = id;
}
```

Listing 2: Free List

region of memory. Similarly, the next entry in that table is a global variable known as *next_entry*.

By `mmap`'ing our metadata table, we allow the kernel to lazily allocate pages, limiting our effective memory overhead. Further, our ID reuse scheme reduces fragmentation of our metadata since it will always reuse a capability ID before allocating a new one. This also helps improve the locality of our metadata lookups, reducing cache pressure. Alternative reuse schemes with better temporal security are discussed in § 6.

To implement our capability ID reuse scheme, we update *next_entry* using our free list. The first entry in our metadata table is reserved (§ 4.2.2), so *next_entry* is initially one. The free list is encoded in the *base* fields of each free entry in the table. These are all initialized to zero. When an entry is free'd, the *base* field is set to the *offset* to the next available table entry. When we add a metadata entry, *next_entry* is incremented, and the offset is added. When an object is deallocated, we have to update the *base* field for its corresponding capability ID (*ID*) to maintain the free list correctly. This requires calculating the *offset* to the next free entry. C code illustrating these operations is in Listing 2.

The final implementation decision for our metadata scheme is how to divide the 64 bits of the pointer between the capability ID and offset. We use the high order 32 bits to store our enriched flag and capability ID (Listing 1). This leaves the low order 32 bits for the offset. Limiting the offset to 32 bits does limit individual object size to 4GB under our current design (with up to 2^{31} such allocations). However, hardware naturally supports 32-bit manipulations, improving the performance of our implementation. Further, having a 31-bit capability ID space is crucial for protecting the entire application (§ 2.1). The enriched bit plays two roles. First, it causes all dereferences of enriched pointers to fail. Consequently, the fact that a program runs guarantees that all pointer dereferences were correctly checked. Second, it enables us to support compatibility with unprotected code, by allowing the correct dereference of unenriched pointers by means of a mask that voids our pointer reconstruction. Unprotected code is not supported by default, however the mechanics are described here, and the implications of enabling unprotected code are discussed in § 6.

Note that no matter how a 64-bit pointer is divided, no limits are placed on the application address space. The entire base pointer is stored in the metadata, and so the offset can reach any part

```c
struct Metadata{
    size_t base;
    size_t end;
}

struct Metadata *table;

//This is done inline, functions are illustrative
static inline size_t check_bounds(size_t base, size_t end
    , size_t check){
    //High order bit is 0 if check passes, 1 otherwise
    size_t valid = (check - base) | (end - (check + size)
    );
    valid = valid & 0x8000000000000000;
    return valid;
}
static inline void *check(void *ptr, uint_32 size){
    size_t tmp = (size_t)ptr;
    //Mask Supports compatibility mode, otherwise omitted
    size_t mask = ptr >> 63;
    uint_32 id = (tmp >> 32) & 0x7fffffff;
    id = id & mask;
    size_t base = table[id].base;
    size_t end = table[id].end;
    size_t check = base + (uint_32)ptr;
    return (void *)(check_bounds(base, end, check) &
    check);
}
void set(int *x, int val){
    //Size of 4 inferred by compiler for int type
    *(check(x, 4)) = val;
}
//example of dereferencing an escaping and a local stack
    array
int main(void){
    int escapes[5];
    escapes = on_allocation(escapes, escapes+5*4);
    set(escapes[2], 10);

    int local[5];
    size_t local_base = local;
    size_t local_end = local+5*4;
    *(local & check_bounds(local_base, local_end, local+2
    *4)) = 10;

    on_deallocation(escapes);
}
```

Listing 3: Instrumentation Example

of the address space. The only limit is the size of an individual object, which is restricted by the space available for the offset in the enriched pointer.

With a minimal allocation size of 8 bytes, a 31-bit ID allows for at least 16 GB of allocated memory. In practice, much more memory can be allocated as objects are usually larger than 8 bytes. When fully allocated, our metadata table uses `2GB * sizeof(struct Metadata)`, see Listing 3. Note that CUP only allocates pages for ID's that are actually used.

4.2 Compiler Pass

Our LLVM compiler pass operates in two phases: (i) analysis, and (ii) instrumentation. As per our design, the analysis phase first determines a set of code points that require us to add code to perform our runtime checks, and the instrumentation phases adds these checks. These checks have been optimized to let the hardware detect bounds violations rather than doing comparisons in software. Listing 3 has a running example for stack objects.

4.2.1 Analysis Implementation.

The first task of our pass is to find the set of object allocations that we must protect to guarantee spatial safety § 3. Heap-based allocations via `malloc` are found by our instrumented musl libc (see § 4.3). Stack-based allocations are found by examining `alloca` instructions in the LLVM Intermediate Representation (IR). These are used to allocate all stack local variables. We only target allocations which can be indirectly accessed via, e.g., pointers. In practice, this means that we need to protect arrays and address-taken variables on the stack, all others are accessed via fixed offsets from the frame pointer. Arrays are trivially found by checking the type of `alloca` instruction as LLVM's type system for their IR includes arrays. LLVM's IR has no notion of the & operator. However, clang (the C/C++ front end) inserts markers — `llvm.lifetime.start` — which we use to identify stack allocations that have their address taken.

CUP also protects global variables. We only need to protect global arrays (non-pointer globals are inherently memory safe). Global arrays present a challenge for our instrumentation scheme. CUP relies on changing pointer values. Unfortunately in C/C++ it is illegal to assign to a global array once it has been allocated. This means that we cannot change the pointer's value. To address this challenge, we create a new global pointer to the first element in the array, and instrument that pointer. We then replace all uses of the global array with our new pointer that can be manipulated as described above. The new pointer must be initialized at runtime, once the address of the global array it replaces has been determined. To do this, we add a new global constructor that initializes our globals. The constructor is given priority such that it runs before any code that relies on our globals.

4.2.2 Instrumentation.

Our compiler is required to add two types of instrumentation to the code: (i) allocation, and (ii) dereference. Allocation instrumentation is responsible for assigning a capability ID to the resulting pointer, creating metadata for it, and returning the enriched pointer. A subcategory of allocation instrumentation is handling deallocations — where metadata must be invalidated and the free list updated. Dereference instrumentation is responsible for performing our bounds check, and returning a pointer that can be dereferenced. While our runtime library provides functions for the functionality described in this section (for use in manual annotations), all of our compiler added checks are done inline. Listing 2 and Listing 3 contain examples for these operations for stack based allocations.

Allocation Instrumentation. CUP requires us to rewrite the pointer for every allocation that we identify as potentially unsafe. Our rewritten or "enriched" pointer contains the assigned capability ID, has the enriched bit set, and the lower 32 bits (which encode the distance from the *base* pointer) are set to 0. All uses of the original pointer are then replaced with the new, instrumented pointer. At allocation time, we use the *next_entry* global variable as the capability ID, and then update *next_entry* as described in § 4.1. See the `escapes` variable in `main()` in Listing 3.

Note that this creates a pointer which cannot be dereferenced on x86_64, which requires that the high order 16 bits all be 1 (kernel-space) or 0 (user-space). As a result, any dereference without a check will cause a hardware fault. Consequently, for any program that runs correctly we can guarantee that all pointers to protected allocations are checked on dereference. This is in contrast to other schemes [28] that fail open, i.e., silently continue, when a check is missed, sacrificing precision. In LLVM-IR allocations that need protection are relatively easy to identify. However, only a subset of `load` and `store` instructions actually correspond to dereferences that need to be checked. By failing closed, we can aggressively limit the `loads` and `stores` that are checked, while still knowing that all dereferences are being checked at runtime.

When an object is deallocated, we insert code to update the free list in our metadata table as per § 4.1 and Listing 2. Further, we mark the *end* address 0 to invalidate the entry.

Dereference Instrumentation. To dereference a pointer, two things need to be done. First, we need to recreate the unenriched pointer. Then, using the metadata from the enriched pointer's capability ID, we need to make sure that the unenriched pointer is in bounds.

To create the unenriched pointer, we first retrieve the high order bit (which indicates whether the pointer is enriched or not). When supporting unprotected code, we create a 64-bit mask with the value of this bit. We then extract the capability ID, and `AND` it with this mask. If the pointer was *not* enriched, this yields an ID of 0 and otherwise preserves the capability ID. This step is skipped by default, as we assume all code is protected. We then lookup the *base* pointer for the capability ID, and add the offset to it. See `check` in Listing 3.

In the case where the pointer was not enriched, we lookup the reserved entry 0 in our metadata table. This entry has *base* and *end* values that reflect all of user-space (`0` to `0x7fffffffffff`). Thus, performing our unenrichment on an unenriched pointer has no effect, and our spatial check below simply sandboxes it in user-space.

Our spatial check performs the requisite lower and upper bounds check. Note, however, that on the upper bound we have to adjust for the number of bytes being read or written. This adjustment adds significantly to our improved precision against other mechanisms (§ 5.2). To illustrate its importance consider the following. An `int` pointer is being dereferenced, meaning the last byte used is the pointer base + 4 bytes - 1, while for a `char` pointer, the last byte used is the pointer base + 1 byte - 1. Equation 1 shows our bounds check formula, the size of the pointer dereferenced is `element_size`.

$$base \leq ptr + element\_size - 1 < base + length \tag{1}$$

Hardware Enforcement. The check in Equation 1 could naively be implemented with comparison instructions and a jump — resulting in additional overhead. We propose a novel way to leverage hardware to perform the check for us. We observe that the difference between the adjusted pointer and the *base* address should always be greater than or equal to 0. Similarly, the *end* address minus the adjusted pointer should always be greater than or equal to 0. Consequently, the high order bit in the differences should always be 0 (x86_64 with two's complement arithmetic). We `OR` these two differences together, mask off the low order 63 bits, and then `OR` the result into the unenriched pointer. If it passed the check, this changes nothing. If it failed the check, it results in a invalid pointer, causing a segmentation fault when dereferenced. Listing 3 shows this optimized check in `check_bounds()`.

4.3 LIBC

Libc is the foundation of nearly every C program and therefore linked with nearly every executable. Unfortunately, many of its most popular functions such as the str* and mem* functions are highly prone to memory safety errors. They make assumptions about program state (e.g., null terminated strings, buffer sizes) and rely on them without checking that they hold. Prior work [12, 28, 35] assumes that libc is part of the trusted code base (TCB).

In contrast, CUP removes libc, including kernel invoking functions like malloc and free, from the TCB by instrumenting libc with our compiler. The majority of the instrumentation is automatic, with few exceptions such as the memory allocator, system calls, and functions implemented in assembly code. Dealing correctly with all of these cases demonstrates the CUP is robust enough to protect all of user space for real world code, and does not contain hidden design flaws that would prevent its adoption as a development tool. The most mature libc implementation that we are aware of that compiles with Clang-4.0.0-rc1 is musl [24]. Our instrumented musl libc is part of CUP.

4.3.1 Dynamic Memory Allocator. The dynamic memory allocator is responsible for requesting memory for the process from the kernel, and returning it. To do so efficiently, most allocators — including musl — modify user requests. In particular, musl rounds up the number of bytes requested. Further, musl maintains metadata inline on the heap in the form of headers before each allocated segment. Consequently, the allocator's view of memory is different than that of the program.

To compensate for this difference, we manually instrumented musl's allocator. We ensure that pointers returned to the application are instrumented with the programmer specified length, not the rounded length. Doing so requires two annotations in malloc and free to adjust the pointer's capability to allow for the headers and rounded length when the pointer is used by the allocator. Doing so allows us to prevent vulnerabilities that rely on corrupting heap metadata [15], while still removing the allocator from the trusted computing base by instrumenting its internals.

An interesting corner case is realloc(). By design it changes the *end* address. Additionally, it can change the *base* address if it was forced to move the allocation to find sufficient room. We manually intervene in both cases (one annotation per) to keep our metadata table up-to-date.

4.3.2 Kernel Boundary. User-space code interacts with the kernel through the system call API. For CUP to correctly protect all user-space code, this boundary must be handled so that pointers passed to the kernel are unenriched, and pointers returned from the kernel are enriched. Almost all system calls are made through libc. Consequently, to show that CUP correctly handles system calls, we instrument them in musl. System calls are made through a dedicated API containing inline assembly in musl. We initially instrumented this API to ensure that no enriched pointers are passed to the kernel. Unfortunately, this is insufficient as structs containing pointers are passed to the kernel. Consequently, we required more context than the narrow system call API provided. This forced us to find the actual system call sites and add one annotation per pointer contained in the struct to unenrich these pointers. When pointers are returned from the kernel through, e.g., mmap or brk, they are annotated to instrument them with their known size. To avoid this (limited) manual annotation effort, future work could define the system call boundary in, e.g., a list that specifies function names, and the instrumentation to be added.

The same level of effort would be required for any boundary to uninstrumented code. As library boundaries are well defined, we consider this level of effort reasonable, particularly for developers, who have intimate knowledge of their code bases.

4.3.3 Assembly Functions. musl implements many of the mem*, str* functions in assembly for x86. As a result, clang is unaware of these functions as they are directly assembled and linked in. We therefore manually instrumented these functions. Our manual effort was approximately ten assembly instructions per function. These instructions call our runtime support library, while preserving register state after our intervention. Setting up these calls would be trivial for any developer writing inline assembly. Supporting them automatically would require a binary only tool, with a corresponding overall loss of precision. Consequently, CUP accepts this level of manual effort.

4.3.4 Other Libc Concerns. printf() can be used to modify memory via the %n format. Similarly, attackers commonly pass format strings that cause memory beyond the supplied arguments to be read, resulting in memory leaks that are used to bypass, e.g., ASLR. Both of these attacks rely on using pointer's outside of their assigned capability, and so are prevented by CUP.

Libc, and for C++ libc++, support non standard control flow via setjmp, longjmp, and exceptions. setjmp and longjmp are implemented in assembly, and rely on pointers. Consequently, we manually added instructions to call our runtime library to unenrich the pointers. This required the same level of effort as the other assembly function in libc. Exception handling relies on libc++, and in particular libunwind for stack unwinding. Recompiling these with CUP causes exceptions to be handled correctly.

4.3.5 Manual Annotation Effort. Any pointer passed to uninstrumented code must be unenriched, and similarly, any pointer returned from uninstrumented code should be enriched. The programming effort needed to perform these actions is one line of code for enrichment (on_allocation) and one to two lines of code for unenrichment (check) for every pointer going to and returning from the uninstrumented code. Two lines of code are needed for unenrichment when the unprotected code can access more than one byte, e.g., read. In this case, the programmer must check that both the start of the buffer and the end of the buffer (as determined by the number of bytes the caller intends to access) are in bounds. As an example of the cumulative level of effort required, we manually modified 37 lines of code in musl, including the assembly functions discussed above. The annotations we added to libc places malloc and free in the TCB.

5 EVALUATION

We evaluate CUP along two axes. First, we show that its performance is competitive with existing sanitizers - and slightly better on benchmarks where both CUP and the existing tools run. Further, our performance is markedly better than other tools that provide

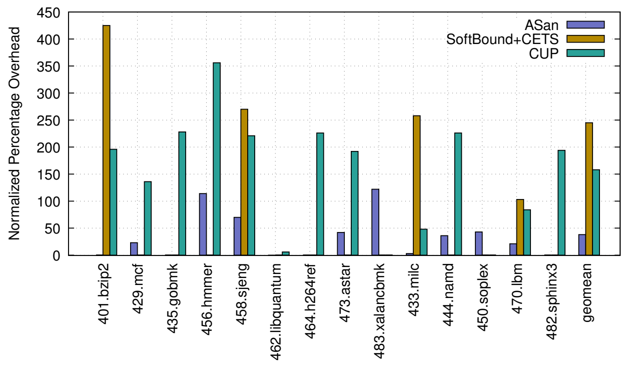

Figure 1: Performance overhead as percentage normalized to baseline using musl. Benchmarks missing entries indicate the mechanism failed to run. Geometric means are calculated over benchmarks where the mechanism ran, and are not directly comparable.

both *Precision* and *Object Awareness*, given that CUP provides *Comprehensive Coverage* (we do more checks in the same amount of time). Second, we demonstrate our *Exactness*, showing that we have no false positives or false negatives on the NIST Juliet test suite. The robustness of CUP and its ability to cope with the full spectrum of corner cases found in real code is demonstrated by supporting libc, making CUP the first sanitizer to do so.

All experiments were run on Ubuntu 14.04 with a 3.40GHz Intel Core i7-6700 CPU and 16GB of memory. For SoftBound+CETS we used the latest version [1]. For AddressSanitizer [35], we used the version based on LLVM-4.0.0-rc1.

5.1 Performance

Performance is an important requirement for any usable sanitizer. We evaluate the performance of CUP with musl. For comparison, we also measure the performance overhead of similar sanitizers, using glibc as the baseline. Musl has effectively the same performance as glibc on SPEC CPU2006, with minor variations on different test cases. ASan is the closest open source related work that is compatible with LLVM-4.0.0-rc1. SoftBound+CETS is open source, but relies on LLVM-3.4 and can only run a small subset of the SPEC CPU2006 benchmarks. Its performance results are reported here, but are not directly comparable. Low-Fat Pointers (developed concurrently with this work) was open sourced after the submission of this work.

Figure 1 summarizes the performance results for CUP. We have 158% overhead vs baseline, compared to 38% for ASan. Note that ASan only provides probabilistic safety, i.e., if a pointer is incremented past guard zones then violations are not detected compared to CUP which catches all spatial memory safety violations by tracking the validity and bounds of pointers. The geometric means in the figure are not directly comparable, as they are calculated over different sets of benchmarks (those where the mechanism actually ran). On benchmarks where both run, CUP has half the overhead of SoftBound+CETS's (126% vs. 245%). Further, compared to these existing tools, we offer stronger, more precise coverage, while being able to run more SPEC CPU2006 benchmarks. Note that both ASan and SoftBound+CETS do not instrument the libc standard library. As we show in § 5.2, CUP has 10x the coverage of SoftBound+CETS, for less overhead. Low-Fat Pointers [12] reports 16% to 62% overhead depending on their optimization level. They achieve this by using clever alignment tricks to avoid metadata look ups. While resulting in low performance overhead, this approach has two limitations: (i) they round allocations up to the nearest power of two, losing precision for bounds checks, and (ii) their design cannot support temporal checks which require metadata.

	False Neg.	False Pos.
CUP	0 (0%)	0 (0%)
SoftBound+CETS	1032 (25%)	12 (0.3%)
AddressSanitizer	315 (8%)	0 (0%)
Total Tests	4038	

Table 2: Juliet Suite Results.

5.2 Juliet Suite

NIST maintains the Juliet test suite, a large collection of C source code that demonstrates common programming practices that lead to memory vulnerabilities, organized by Common Weakness Enumeration (CWE) numbers [19]. Every example comes with two versions: one that exhibits the bug and one that is patched. We extracted the subset of tests for heap and stack buffers[2]. We compiled all sources with our pass, as well as with SoftBound+CETS and with ASan. Every patched test should execute normally. If a memory protection mechanism prematurely kills a patched test, we call it a false positive. Conversely, every buggy test should be stopped by the memory safety mechanism. All three memory protection mechanisms kill the process in case of a memory violation. If the process is not killed, we count it as false negative.

We found four architecture dependent bugs in the Juliet test suite. These tests attempt to allocate memory for a structure containing exactly two `int`s. However, erroneously, the examples use the size of a pointer to the two `int` structure when allocating memory. (e.g., `malloc(sizeof(TwoIntStruct*))`). On architectures which do not define pointers as twice the size of `int`s (including 32-bit x86), such a mistake would lead to a memory violation when reading or writing the second int. On the x86_64 architecture, though, the size of a pointer and the size of the two `int` structure are the same. These test cases show a type violation and not a memory safety violation for x86_64. Thus, while semantically incorrect, no true memory violation occurs.

Table 2 summarizes the results. Out of 4,038 tests that should not fail, we incur no false positives. ASan and SoftBound+CETS show a 0% and 0.3% respectively. We produce no false negatives, while ASan produces an 8% false negative rate, and SoftBound+CETS has a 25% false negative rate.

[1] https://github.com/santoshn/softboundcets-34/9a9c09f04e16f2d1ef3a906fd138a7b89df44996

[2] The tests that match the following regular expression: CWE(121|122)+((?!socket).)*(\.c)$

```
   int main() {
 2     int pos = 0;
       char data = 0;
 4     scanf("%d", &pos);
       char buff2[10] = "abc";
 6     char buff[10];
       while(pos < 10 && data != '\n') {
 8         scanf("%c", &data);
           buff[pos] = data;
10         printf("%d\n", pos);
           pos++;
12     }
   }
```

Listing 4: Example of code ASan fails to protect

Most of the false positives that SoftBound+CETS registers comes from variations in how the `alloca()` function call is handled. `alloca()` is compiler dependent [21]. The failing examples use `alloca` which is wrapped around a static function. SoftBound+CETS uses clang 3.4 as the underlying compiler, and CUP uses clang 4.0. Consequently, SoftBound+CETS handles the examples differently, and sees the memory from `alloca` as invalid, while CUP does not. However, there were four false positives that involve `memcpy`, which SoftBound+CETS claims to handle. An implementation bug is likely the cause of these failing tests, as the Juliet tests cover a wide spectrum of corner cases. While SoftBound+CETS theoretically has no false positives, this issue highlights the importance of implementation.

ASan and SoftBound+CETS higher false negative rate results from their incomplete *coverage*. In particular, many of the Juliet examples involve calling libc functions to copy strings and buffers (e.g., `strcpy` and `memcpy`). Neither ASan nor SoftBound+CETS are able to protect against unsafe libc calls. Our instrumentation of libc (§ 4.3) allows us to properly detect memory violations in these calls. Further, our adjustment for read / write size (§ 4.2.2) allows us to catch additional memory safety violations.

Given its support for separate compilation, and its theoretical soundness, SoftBound+CETS should be able to properly protect libc. However, as shown in § 2.1, supporting all data across the entirety of user space significantly stresses the design of memory safety mechanisms. Our annotations in libc § 4.3 allow us to protect all of user space. SoftBound+CETS has yet to prove that it can do so.

Listing 4 provides a representative Juliet test that CUP properly handles, and which ASan handles incorrectly[3]. The code allocates two 10 byte buffers on the stack, reads input from the user, and starts writing user input to one of the buffers. ASan protects the stack by surrounding stack objects with poison zones. However, depending on the value of of `pos`, an integer overflow bug allows an attacker to skip over the poison zone, and to write to an address higher in the stack. CUP enforces strict boundaries on a per-object basis, so the write at line 9 would be blocked.

6 DISCUSSION

We discuss several design aspects of CUP, how we handle specific corner cases, potential for optimization, and further extensions.

[3]Adapted from https://github.com/ewimberley/AdvancedMemoryChallenges/blob/master/4.cpp

Reducing instrumentation. Prior work on reducing the amount of required runtime checks has focused on type systems. CCured [30] infers three types of pointers: safe, sequential, and wild. Safe pointers are statically proven to stay in bounds. Sequential pointers are only incremented (or decremented) — e.g., iterating over an array in a loop. All remaining pointers are classified as wild. Leveraging CCured-style type systems to further optimize memory safety solutions is left as an open problem.

Optimization through LTO. CUP does not depend on Link Time Optimization (LTO). However, LTO makes it possible to inline functions across source files, and generally flattens code. Inlining would increase the effectiveness of our stack optimization and further reduce the amount of instrumented stack variables, reducing the number of IDs that a program consumes. Reducing the IDs a program uses reduces the overhead for ID management and resources used by CUP.

Arithmetic overflow. Our hybrid metadata scheme stores the capability ID as part of the pointer. Pointer arithmetic can potentially modify the ID, allowing an adversary to chose the metadata the pointer is checked against. To prevent this attack vector, the upper and lower 32 bits of the pointer are loaded separately. The compiler enforces that any arithmetic operations only happen on the lower 32 bits, which contain the pointer's offset. This protects our capability ID from manipulation by an adversary.

Stronger temporal protection. As discussed in § 3.1, it is possible (albeit difficult) for an attacker to perform a temporal attack on software instrumented with CUP. If CUP were deployed as an active defense, the difficulty of a successful temporal attack could be increased by utilizing a randomized memory allocator like DieHard [3] or DieHarder [32]. Such allocators randomize heap allocations, making heap grooming [14, 37] much more difficult. Beyond getting the addresses to line up, the increased number of required allocations makes matching the capability IDs even harder. This renders a successful use-after-free attack highly unlikely, with minimal additional overhead. Additionally, a “lock and key” scheme [29] can be added to our metadata. Alternatively, our metadata can be extended to include a DangNull [20]-style approach that records how many references are still pointing to an object and either explicitly invalidating those references or waiting until the last reference has been overwritten before reusing IDs.

Third-Party Code. CUP supports linking with uninstrumented libraries. Enriched pointers are the same size as regular pointers, maintaining the ABI. Dereferencing enriched pointers in uninstrumented code results, by design, in a segmentation fault. A segmentation fault handler can, on demand, dereference individual pointers. As the memory allocator is instrumented, memory allocations in uninstrumented code will return enriched pointers. Stack allocations on the other hand will not be protected in uninstrumented code. While this option allows compatibility, it clearly results in high performance overhead. An alternate solution is to manually annotate pointers passed to uninstrumented code (one annotation per pointer) and sacrifice protection, while gaining performance. This solution, however, exposes our metadata table to corruption from uninstrumented code. All memory safety solutions share this vulnerability to unprotected code. CUP alleviates this situation by recompiling all user-space code, including the libc.

Memory errors in unprotected code. While it is possible to use CUP with unprotected code, we cannot guarantee the safety of applications that use unprotected code. Any memory violation in unprotected code can lead to unsafe modifications to user-space applications. Since CUP has no knowledge of the behavior of unprotected code, it is up to the programmer to ensure that all relevant buffers are properly checked before they are used. In our instrumentation of musl, we have already done this (see § 4.3) for kernel system calls.

Assembly code. CUP automatically instruments all code written in high level languages; our analysis pass runs on LLVM IR. Our analysis does not (and cannot) instrument inline assembly and assembly files due to missing type information. We rely on the programmer to either instrument the assembly code accordingly or to fall back on supporting uninstrumented code as mentioned above.

Multithreading. CUP does not protect against inter-thread races of updates to metadata (e.g., one thread frees an object while the other thread is dereferencing a pointer). We leave the design of a low-overhead metadata locking scheme as future work. Note that this limitation is shared with other sanitizers.

Code Pointers. A substantial research effort has gone into protecting code pointers [1]. CUP does not need to directly protect code pointers, including function pointers, C++ virtual table pointers, and C++ virtual tables. Function pointers cannot be used to read or write memory themselves. Further, by enforcing memory safety, CUP guarantees that they point to the correct targets.

7 RELATED WORK

Precision is required to enforce spatial memory safety (bounds checks). There is a class of memory safety solutions that only approximately enforce this property [2, 12, 35]. These solutions make use of techniques such as poisoned zones — detecting spatial violations within limits, or rounding allocation size — which causes the executed program to differ from the programmers intent, and results in challenges when trying to handle intra-array and intra-struct checks. By changing the memory layout and not enforcing *exact* bounds, these solutions are not faithful to the programmer's intent. SoftBound [28] is the existing solution which best satisfies this requirement, while lacking comprehensive coverage (see below).

Object-based memory checking [8, 10, 13] keeps track of metadata on a per-object basis. Since the meta-data is associated on a per object level, every pointer to the object shares the same metadata. If a pointer is incremented, it may suddenly point to another valid object and therefore be (illegally) assigned to that object. Object layout in memory is generally left unchanged, which increases ABI compatibility. However, pointer casts and pointers to subfield struct members are unhandled [27]. SAFECode [11] is an example for efficient object-based memory checking.

Recently, Intel has started to add memory safety extensions, called Intel MPX, to their processors, starting with the Skylake architecture [17]. These extensions add additional registers to store bounds information at runtime. While effective at detecting spatial memory violations, MPX is incapable of detecting temporal violations [33], and typecasts to integers are not protected. In addition, current implementations of MPX incur a large memory penalty of up to 4 times normal usage [23]. Other ISA extensions include Watchdog [26], WatchdogLite [25], HardBound [9], and Chuang et al. [7]. As a software only solution, CUP does not require extra hardware or ISA extensions.

Object Awareness is required to prevent temporal memory safety violations (lifetime errors). This property requires remembering for every pointer whether the object to which it is assigned is still allocated. AddressSanitizer [35] and Low-Fat Pointers [12] make no attempt to do this (and their metadata does not support this property), while SoftBound+CETS [29] enforces this property. AddressSanitizer and Low-Fat Pointers do not maintain metadata either per object or per pointer. *Object Awareness* requires either per object or per pointer metadata. Consequently, they fundamentally *cannot* enforce temporal safety because their (current) metadata *cannot* be object aware. CUP's contributions on top of SoftBound+CETS lie primarily in Comprehensive Coverage.

Temporal only detectors include DangNull [20] and Undangle [6] from Microsoft. DangNull automatically nullifies all pointers to an object when it is freed. Undangle uses an early detection technique to identify unsafe pointers when they are created, instead of being used. CUP only provides a probabilistic temporal defense, however, DangNull and Undangle lack any spatial protection. Other probabilistic approaches [3, 32] change the memory allocator to reduce the frequency with which memory is reallocated.

Comprehensive Coverage is required to fully protect the program. As shown in § 2.1 stack objects are the overwhelming majority of allocations, and to this day a significant portion of memory safety Common Vulnerabilities and Exposures (CVE) are stack related. Our evaluation of SoftBound+CETS (§ 5.2) shows that it has poor coverage — missing many stack vulnerabilities. AddressSanitizer and Low-Fat Pointers do better. AddressSanitizer protects the stack through the use of poisoned zones, and, as illustrated in § 5.2 cannot handle all invalid stack memory accesses. Additionally, neither of them supports compiling libc — leaving the window open for vulnerabilities such as GHOST [22]. Tripwires [31, 34, 41, 44] are a way to detect some spatial and temporal memory errors [35, 42]. Tripwires place a region of invalid memory around objects to avoid small stride overflows and underflows. Temporal violations are caught by registering memory freed as invalid, until reclaimed. Tripwires, however, miss long stride memory errors, and thus cannot be said to be completely secure.

The state-of-the-art C/C++ pointer-based memory safety scheme is SoftBound+CETS [27]. Other pointer-based schemes include CCured [30] and Cyclone [18]. CCured uses a fat pointer to store metadata, as well as programmer annotations for indicating safe casts. Unfortunately, fat pointers break the ABI, and programmer annotations can significantly increase developer time. Even with annotations, CCured fails to handle structure changes. Cyclone also uses a fat pointer scheme, but does not guarantee full memory safety.

Control-Flow Hijacking Defenses. Mechanisms like control-flow integrity (CFI) [1, 4] or Object Type Integrity [5] check if a code pointer has been modified to point to an illegal address before it is used, e.g., for an indirect function call. CFI mechanisms assume that memory safety violations can happen and only checks the integrity of code pointers. CFI is a low-overhead approach to protect programs against illicit uses of a corrupted pointer (without protecting the integrity of the code pointer itself) while memory safety protects against the pointer corruption in the first place.

8 CONCLUSION

We present CUP, a C/C++ memory safety mechanism that provides full user-space protection, including libc, and strong probabilistic temporal protection. It is the first such mechanism that satisfies all requirements for a complete memory safety solution, while incurring only modest performance overhead compared with the state-of-the-art. CUP is exact, object aware, comprehensive in its coverage, and precise. We fully protect all user-space memory, including the stack, which, despite being the largest source of pointers, remained largely unprotected. Finally, we produce zero false negatives and zero false positives in the NIST Juliet Vulnerability example suite, which represents a significant advancement over existing memory safety mechanisms.

Our prototype is available at https://github.com/HexHive/CUP.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their insightful feedback that greatly improved the final paper. This material is based, in part, upon work supported by the National Science Foundation under awards CNS-1513783 and CNS-1657711 and the Intel corporation. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of our sponsors.

REFERENCES

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow Integrity. In *CCS '05*.
[2] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. 2009. Baggy Bounds Checking: An Efficient and Backwards-compatible Defense Against Out-of-bounds Errors. In *SEC'09*.
[3] Emery D. Berger and Benjamin G. Zorn. 2006. DieHard: Probabilistic Memory Safety for Unsafe Languages. *SIGPLAN Not.* (2006).
[4] Nathan Burow, Scott A Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. 2017. Control-flow integrity: Precision, security, and performance. *CSUR'17* (2017).
[5] Nathan Burow, Derrick McKee, Scott A Carr, and Mathias Payer. 2018. CFIXX: Object Type Integrity for C++. In *NDSS'18*.
[6] Juan Caballero, Gustavo Grieco, Mark Marron, and Antonio Nappa. 2012. Undangle: early detection of dangling pointers in use-after-free and double-free vulnerabilities. In *ISSTA'12*.
[7] Weihaw Chuang, Satish Narayanasamy, and Brad Calder. 2007. Accelerating meta data checks for software correctness and security. *Journal of Instruction-Level Parallelism* (2007).
[8] John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve. 2007. Secure virtual architecture: A safe execution environment for commodity operating systems. In *OSR'07*.
[9] Joe Devietti, Colin Blundell, Milo MK Martin, and Steve Zdancewic. 2008. Hardbound: architectural support for spatial safety of the C programming language. In *ACM SIGARCH Computer Architecture News*.
[10] Dinakar Dhurjati and Vikram Adve. 2006. Backwards-compatible Array Bounds Checking for C with Very Low Overhead. In *ICSE '06*.
[11] Dinakar Dhurjati and Vikram Adve. 2006. Backwards-compatible array bounds checking for C with very low overhead. In *ICSE'06*.
[12] Duck, Yap, and Cavallaro. 2017. Stack Bounds Protection with Low Fat Pointers. In *NDSS'17*.
[13] Frank Ch Eigler. 2003. Mudflap: Pointer Use Checking for C/C+. In *GCC Developers Summit*.
[14] Chris Evans. Feb 2017. https://googleprojectzero.blogspot.com/2015/06/what-is-good-memory-corruption.html. (Feb 2017).
[15] Chris Evans. May 2017. https://scarybeastsecurity.blogspot.com/2017/05/further-hardening-glibc-malloc-against.html?m=1. (May 2017).
[16] Michael Hicks. 2014. What is memory safety? (2014). http://www.pl-enthusiast.net/2014/07/21/memory-safety/
[17] Intel 2015. IntelÂő Software Development Emulator. https://software.intel.com/en-us/articles/intel-software-development-emulator. (2015).
[18] Trevor Jim, J. Greg Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C. In *ATC'02*.
[19] juliet 2013. National Institute of Standards and Technology Juliet C/C++ Test Suite. https://samate.nist.gov/SARD/testsuite.php. (2013).
[20] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu, and Wenke Lee. 2015. Preventing Use-after-free with Dangling Pointers Nullification.. In *NDSS'15*.
[21] man7.org. Feb 2017. http://man7.org/linux/man-pages/man3/alloca.3.html. (Feb 2017).
[22] Mitre. Feb 2017. https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2015-0235. (Feb 2017).
[23] mpx 2016. AddressSanitizerIntelMemoryProtectionExtensions. https://github.com/google/sanitizers/wiki/AddressSanitizerIntelMemoryProtectionExtensions. (2016).
[24] musl 2016. musl libc. http://www.musl-libc.org/. (2016).
[25] Santosh Nagarakatte, Milo MK Martin, and Steve Zdancewic. 2014. Watchdoglite: Hardware-accelerated compiler-based pointer checking. In *CGO'14*.
[26] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2012. Watchdog: Hardware for Safe and Secure Manual Memory Management and Full Memory Safety. In *ISCA '12*.
[27] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2015. Everything You Want to Know About Pointer-Based Checking. In *SNAPL'15)*.
[28] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2009. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *PLDI '09*.
[29] Santosh Nagarakatte, Jianzhou Zhao, Milo M.K. Martin, and Steve Zdancewic. 2010. CETS: Compiler Enforced Temporal Safety for C. In *ISMM '10*.
[30] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. CCured: Type-safe Retrofitting of Legacy Software. *TOPLAS'05* (2005).
[31] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*.
[32] Gene Novark and Emery D. Berger. 2011. DieHarder: Securing the Heap. In *WOOT'11*.
[33] Christian W Otterstad. 2015. A brief evaluation of Intel® MPX. In *SysCon'15*.
[34] Feng Qin, S. Lu, and Yuanyuan Zhou. 2005. SafeMem: exploiting ECC-memory for detecting memory leaks and memory corruption during production runs. In *HPCA'05*.
[35] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A fast address sanity checker. In *ATC'12*.
[36] Fermin J. Serna and Kevin Stadmeyer. Feb 2017. https://security.googleblog.com/2016/02/cve-2015-7547-glibc-getaddrinfo-stack.html. (Feb 2017).
[37] Alexander Sotirov. 2007. Heap feng shui in javascript. *Black Hat Europe* (2007).
[38] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *SP'13*.
[39] TrendMicro. Feb 2017. http://blog.trendmicro.com/pwn2own-day-1-recap/. (Feb 2017).
[40] Victor van der Veen, Nitish dutt Sharma, Lorenzo Cavallaro, and Herbert Bos. 2012. Memory Errors: The Past, the Present, and the Future. In *RAID'12*.
[41] G. Venkataramani, B. Roemer, Y. Solihin, and M. Prvulovic. 2007. MemTracker: Efficient and Programmable Support for Memory Access Monitoring and Debugging. In *HPCA'07*.
[42] Guru Venkataramani, Brandyn Roemer, Yan Solihin, and Milos Prvulovic. 2007. MemTracker: Efficient and Programmable Support for Memory Access Monitoring and Debugging. In *HPCA '07*.
[43] vvdveen. Feb 2017. https://github.com/vvdveen/memory-errors/. (Feb 2017).
[44] Suan Hsi Yong and Susan Horwitz. 2003. Protecting C Programs from Attacks via Invalid Pointer Dereferences. In *ESEC/FSE-11*.