CETS: Compiler-Enforced Temporal Safety for C

Santosh Nagarakatte Jianzhou Zhao Milo M. K. Martin Steve Zdancewic

Computer and Information Sciences Department, University of Pennsylvania

santoshn@cis.upenn.edu jianzhou@cis.upenn.edu milom@cis.upenn.edu stevez@cis.upenn.edu

Abstract

Temporal memory safety errors, such as dangling pointer dereferences and double frees, are a prevalent source of software bugs in unmanaged languages such as C. Existing schemes that attempt to retrofit temporal safety for such languages have high runtime overheads and/or are incomplete, thereby limiting their effectiveness as debugging aids. This paper presents CETS, a compile-time transformation for detecting all violations of temporal safety in C programs. Inspired by existing approaches, CETS maintains a unique identifier with each object, associates this metadata with the pointers in a disjoint metadata space to retain memory layout compatibility, and checks that the object is still allocated on pointer dereferences. A formal proof shows that this is sufficient to provide temporal safety even in the presence of arbitrary casts if the program contains no spatial safety violations. Our CETS prototype employs both temporal check removal optimizations and traditional compiler optimizations to achieve a runtime overhead of just 48% on average. When combined with a spatial-checking system, the average overall overhead is 116% for complete memory safety.

Categories and Subject Descriptors D.3.3.4 [*Programming Languages*]: Processors; D.2.5 [*Software Engineering*]: Testing and Debugging

General Terms Languages, Performance, Security, Reliability

Keywords memory safety, temporal errors, dangling pointers, C

1. Introduction

Unmanaged languages such as C/C++ are the de facto standard for implementing operating systems, virtual machine monitors, language runtimes, database management systems, embedded software, and performance-critical software of all kinds. Such languages provide low-level control of memory layout, explicit memory management, and proximity to the underlying hardware. However, all of this control comes at a price—lack of bounds checking leads to buffer overflows (spatial safety violations) and manual memory management leads to dangling pointer and double-free errors (temporal safety violations). Both types of memory errors can result in crashes, silent data corruption, and severe security vulnerabilities. Recognizing the gravity of the problem, there have been many proposals for detecting or preventing one or both kinds of errors [4–6, 9, 10, 13–15, 18–20, 22, 23, 25–28, 30, 31].

This paper focuses on debugging tools for runtime prevention of temporal safety violations. Temporal safety errors include: dangling pointer dereferences (referencing an object that has been deallocated), double `free`'s (calling `free()` on the same object multiple times), and invalid frees (calling `free()` with a non-heap address or pointer to the middle of a heap-allocated region). Figure 1 shows two examples of common kinds of temporal memory safety errors, including a dangling reference to a re-allocated heap locations (left) and a dangling pointer to the stack (right).

Heap based

```
int *p, *q, *r;
p = malloc(8);
...
q = p;
...
free(p);
r = malloc(8);
...
... = *q;
```

Stack based

```
int* q;
void foo() {
  int a;
  q = &a;
}
int main() {
  int *q;
  foo();
  ... = *q;
}
```

Figure 1. Dangling pointer errors involving the heap and stack. On the left, freeing `p` causes `q` to become a dangling pointer. The memory pointed to by `q` could be reallocated by any subsequent call to `malloc()`. On the right, `foo()` assigns the address of stack-allocated variable `a` to global variable `q`. After `foo()` returns, its stack frame is popped, thereby `q` points to a stale region of the stack, which any intervening function call could alter. In both cases, dereferencing `q` can result in garbage values or data corruption.

Although there has been considerable effort in detecting temporal errors [6, 14, 15, 18, 25, 26, 28], challenges remain. For example, consider the widely used Valgrind Memcheck tool [25]. Although an important debugging aid, Memcheck fails to detect dangling pointers to reallocated data locations, and it exhibits runtime overheads in excess of 10x [25], partly due to its use of dynamic binary instrumentation. Other approaches modify `malloc()` to allocate only one object per page and unmap the page at deallocation time, thus using the processor's virtual address translation hardware to detect dangling pointers dereferences. For programs with many small objects, this approach can inflate memory use dramatically (*e.g.*, allocating a 4k page for a 40-byte object causes a 100x increase in memory footprint). Recent work has reduced physical memory usage [14], but such approaches do not detect dangling references to stack locations, and the system calls per allocation/deallocation can results in significant runtime overheads for programs that frequently allocate memory [14]. Conservative garbage collection side-steps this problem for heap allocations, however it also fails to detect dangling pointers to the stack, and it is not suitable for all applications domains. Furthermore, it typically masks such errors (rather than reporting them), which is less useful in the context of a debugging tool.

Overall, prior proposals suffer from one or more of the following deficiencies: high runtime overheads, high memory overheads, failure to detect all temporal errors (for example, to the stack, to reallocated heap locations, or in the face of arbitrary casts), requiring annotations inserted by the programmer, or altering memory layout

ISMM'10, June 5–6, 2010, Toronto, Ontario, Canada.

(which breaks compatibility with existing C code). These drawbacks have typically led developers to employ such techniques selectively rather than using them by default throughout the software development and testing process.

Our proposal, CETS (Compiler Enforced Temporal Safety), is built upon the hypothesis that compiler-based instrumentation and efficient metadata structures can yield a low-overhead and highly compatible system for detecting all violations of temporal safety. CETS addresses the above deficiencies through a synergistic combination of two existing techniques. First, CETS uses an *identifier-based scheme*, which assigns a unique key for each allocation region to identify dangling pointers [5, 27, 30]. Second, this per-pointer metadata is tracked using a *disjoint shadowspace*, yielding high compatibility because memory layout is unchanged. Such shadowspace approaches for tracking per-pointer metadata have been applied previously in the context of spatial safety [22]. We demonstrate the correctness of this approach by describing a machine-checked formal proof that CETS instrumented programs detect all temporal errors for a small, but realistic, fragment of C.

For our CETS prototype implementation, we developed several simple temporal-check elimination optimizations (analogous to, but different from, bounds-check elimination optimizations [7, 17, 29]). Our experiments show that such static optimizations reduce the number of checks by 70% on average. The combination of these optimizations and standard optimizations applied after instrumentation results in a runtime overhead of 48% on average for a set of 17 SPEC benchmarks. Furthermore, when coupled with our prior work on spatial safety [22], the average runtime overhead is 116% for detecting all spatial and temporal errors.

To explore the design alternatives, we compare CETS' temporal checking to an allocation shadow bits scheme that is similar to the best-effort temporal checking performed by Valgrind's Memcheck, but implemented within the compiler rather than by binary instrumentation. Although the approach has lower overhead than when implemented using binary instrumentation, it is less complete and has higher runtime overhead than CETS. These results show the effectiveness of compiler-based instrumentation and optimization for creating checking tools with efficiency and compatibility sufficient for use in all stages of the software development life cycle.

2. Temporal Checking Background

Detecting or preventing dangling pointer dereferences has been a long-standing problem and several approaches have been proposed [5, 6, 14, 15, 18, 25–28, 30]. This section describes the approaches that CETS builds on, all of which track allocation information and perform extra run-time checking on pointer dereferences [5, 18, 20, 25, 25, 27, 28, 30]. Table 1 summarizes the comparison of these techniques, which fall into two basic categories: (1) *object location* approaches and (2) *allocation identifier* approaches. Each are described in detail in the next two subsections. We defer to Section 7 discussion of other approaches for handling temporal errors (*e.g.*, conservative garbage collection [8], probabilistically approximating an infinite heap [6, 26], or using virtual memory or segmentation hardware [14]).

2.1 Location-Based Temporal Checking

Object location-based approaches (*e.g.*, [18, 20, 25, 28]) use the location (address) of the object to determine whether it is allocated or not. An auxiliary data structure records the allocated/deallocated status of each location. This data structure is updated on memory allocations (*e.g.*, `malloc()`) and deallocations (*e.g.*, `free()`). On a memory access, these auxiliary structures are consulted to determine whether the dereferenced address is currently valid (allocated) memory. As long as deallocated memory locations are never reallocated, this approach will detect all dangling pointer dereferences. However, if a location has been reallocated, this approach erroneously allows the pointer dereference to proceed—the information tracked by this approach is insufficient to determine that the pointer's original allocation area was freed and is now being used again for a potentially unrelated purpose. Thus, although such techniques are able to detect many dangling pointer dereferences, they cannot detect all temporal errors.

Tree-based location lookup. One approach to implementing the auxiliary data structure is to record all allocated regions of memory in a tree structure [20]. On a pointer dereference, a range lookup in the tree identifies whether the address pointed to by the pointer is currently allocated (when a mapping is found) or unallocated (when no mapping is found). The memory overhead of this approach is low because it is proportional to the number of live objects, but it requires a potentially slow range lookup (*i.e.*, splay tree walk) for each dereference check.

Shadowspace-based location lookup. An alternative approach is to use a *shadowspace* in which allocation/deallocation status is maintained with each byte (or word) of memory [18, 24, 28]. A shadowspace may be implemented as a large, directly accessed memory region, a hashtable [22], or a trie-based data structure [24]. Accessing a shadow space entry is typically an efficient $O(1)$ operation, and thus provides fast checks. The memory usage is proportional to the size of the memory rather than just the number of allocated objects (as in the tree-based scheme), but tracking a bit per word is only a few percent memory overhead.

2.2 Identifier-Based Temporal Checking

An alternative approach is the *allocation identifier approach*, which associates a unique identifier with each memory allocation.[1] Each allocation is given a unique identifier and identifiers are never reused. To ensure that this unique identifier persists even after the object's memory has been deallocated, the identifier is associated with pointers. On a pointer dereference, the system checks that the unique allocation identifier associated with the pointer is still valid.

Per-pointer metadata via fat pointers. One implementation of pointer-based metadata is to expand all pointers into multi-word *fat pointers*. Although use of fat pointers has been proposed in variety of contexts [5, 19, 23, 30], fat pointers have the drawback of changing memory layout in programmer-visible ways, and the modified memory layout makes interfacing with libraries challenging [10, 23]. Furthermore, the combination of fat pointers and arbitrary casts can lead to metadata corruption, which weakens the guarantees provided or complicates their implementation. To mitigate the problems with fat pointers, using a disjoint pointer-based metadata shadowspace has been proposed in the context of bounds checking [22], but this specific technique has not previously been applied to the detection of dangling pointer dereferences.

Set-based identifier checking. A set data structure (such as a hash table) is one approach for determining whether an allocation identifier is still valid (*i.e.*, the object has not been deallocated) [5]. The allocation identifier is inserted into the set during allocation (*e.g.*, `malloc()`) and removed from the set at deallocation (*e.g.*, `free()`), and thus the set contains an identifier if and only if the identifier is valid. Although set lookups can take just $O(1)$ time, performing a hash table lookup on every memory reference has the potential for introducing significant runtime overheads.

Lock-and-key identifier checking. To avoid a set lookup on each check, an alternative is to pair each pointer with two pieces of

[1] Some prior work has referred to these unique allocation identifiers as "pointer capabilities" [5, 30].

Temporal checking taxonomy		Approach	Instrumentation method	Runtime overhead	Metadata organization	Handles arbitrary casts	Detects errors with reallocations
Location based	Shadowspace	Memcheck [25]	Binary	> 10x	Disjoint	Yes	No
	Tree	J&K [20]	Compiler	> 10x	Disjoint	Yes	
Identifier based	Set	SafeC [5]	Source	> 10x	Inline	No	**Yes**
	Lock & Key	P&F [27]	Source	5x	Split	No	
		MSCC [30]	Source	2x	Split	No	
		CETS	**Compiler**	**< 2x**	**Disjoint**	**Yes**	

Table 1. Comparison of representative location-based and identifier-based approaches.

metadata: an allocation identifier—the *key*—and a *lock address* that points to a special *lock location* in memory [27, 30].[2] The key and lock value will match if and only if the pointer is valid. A dereference check is then a direct lookup operation—a simple load and compare—rather than a hash table lookup. Freeing an allocation region changes the lock value, thereby invalidating any other (now-dangling) pointers to the region. Because the keys are unique, a lock location itself can be reused after the space it guards is deallocated.

2.3 Analysis of the Temporal Checking Design Space

The two general approaches described above have complementary strengths and weaknesses. Using identifiers permits detection of dangling pointer dereferences even if the memory has been reallocated to a new object. The disadvantages of identifiers stem from tracking per-pointer metadata, which adds potentially significant overhead to loads and stores of pointer values. Fat-pointer implementations also change data layout, which can reduce source compatibility and make interaction with libraries difficult. Fat pointers also interact poorly with arbitrary type casts, because the in-line metadata can potentially be overwritten via casts.

Location-based checking is attractive because its disjoint metadata does not change the program's memory layout. By re-linking with a different `malloc()` library, this approach works even for objects allocated within libraries, and thus is highly compatible—using it requires fewer source program changes. However, because this approach tracks only allocations, it does not detect pointers that erroneously point to reallocated regions of memory.

2.4 Program Instrumentation

Given the approaches explained above, there are several different options for adding the checks necessary to enforce memory safety.

Binary instrumentation. One option is to instrument the program at the binary level, after compilation. Mechanisms for doing so range from completely static binary rewriting [18] (*i.e.*, produce a new executable from the old one) to partial emulation at the instruction level, intercepting control-flow transfers and interpolating new code [25]. The benefit of binary translation is that it operates on unmodified binary code and dynamically linked libraries, even when the source code is unavailable. However, the emulation and instrumentation overhead can contribute to high runtimes (10x slowdown is common [25]), in part because it is difficult to perform high-level optimizations. These tools are also inherently tied to a specific instruction-set architecture.

Hardware-assisted instrumentation. Hardware support for monitoring and checking execution (*e.g.*, [12, 28]) captures the desirable properties of binary instrumentation but with potentially lower runtime overhead and good backwards-compatability, at the expense of requiring new hardware.

[2] We have intentionally reversed the meaning of lock and key from original paper [27], because we find this terminology more intuitive—there can be multiple copies of the key for each lock location.

Source-level instrumentation. A third option is source-level transformation to insert runtime checks (*e.g.*, [5, 23, 30]). This approach allows for use of source-code type information, and the resulting instrumented source is independent of any specific C compiler or instruction-set architecture. The instrumentation is applied before the code is optimized by the compiler (*e.g.*, before variables are register allocated or code is hoisted out of loops). Unfortunately, once the instrumentation code has been added, the additional memory operations introduced may limit the effectiveness of subsequent compiler optimization passes.

Compiler-based instrumentation. The final option is for the compiler to instrument the code during compilation (*e.g.*, GCC's compiler-inserted profiling instrumentation). This approach is similar to source-level instrumentation, but adds two key advantages: (1) it can introduce instrumentation after the compiler has applied standard code optimizations and (2) the compiler can then reapply the same optimizations to reduce the runtime overhead.

3. CETS Design and Implementation

CETS is intended to achieve several goals. Its primary goal is to detect *all* dangling pointers dereference bugs (a property of identifier-based approaches described in Section 2.2) while leaving memory layout unchanged to provide high compatibility with existing C source (a property associated with location-based approaches described in Section 2.1). Furthermore, to provide higher utility as a debugging aid, the runtime overhead introduced by CETS should be low enough (less than 2x) for default deployment throughout the software development and testing process.

To meet these goals, CETS employs a lock-and-key identifier-based approach, but CETS avoids the compatibility problems of fat pointers by borrowing the shadowspace mechanisms commonly used in location-based approaches to support per-pointer metadata without changing memory layout (see Figure 2). This disjoint metadata is tracked and propagated for each pointer, but memory layout remains unchanged. To achieve its runtime efficiency goals, CETS uses compiler-based instrumentation and directly-accessed data structures. To reduce the runtime impact of instrumentation, CETS invokes standard compiler optimizations both preceding and following its instrumentation pass. The pass also performs the temporal check elimination optimizations described in Section 5.

The remainder of this section describes CETS' implementation of lock-and-key temporal checking and its mechanisms for preventing double-frees (Sections 3.1 and 3.2) and CETS' shadowspace mechanisms for disjoint per-pointer metadata (Section 3.3). As discussed below, CETS's temporal safety properties can be guaranteed only if spatial safety is enforced too (otherwise metadata corruption could occur). Section 4 formalizes a spatially-safe version of CETS and sketches a proof of CETS' temporal safety guarantees.

3.1 CETS Lock-and-Key Implementation

CETS augments each pointer with two additional word-sized fields: (1) a unique *allocation key* and (2) a *lock address* that points to a *lock location*, which is queried on temporal checks [27, 30]. By

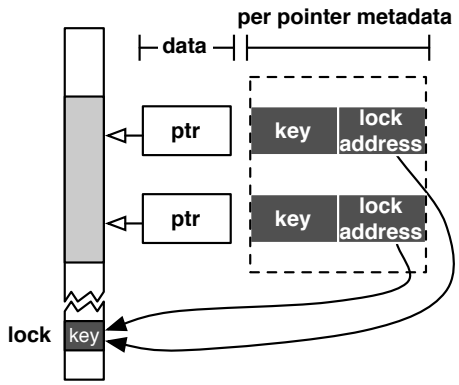

Figure 2. CETS pointer metadata for two pointers in memory. In this example, the pointers point to locations within the same object and thus have the same lock address and have the same key value.

incrementing a 64-bit *next_key* counter, each allocation is given a unique key and key values are never reused.[3] The lock provides an efficient mechanism for determining whether the memory allocated for the pointer has been deallocated. When memory is deallocated, CETS "changes the lock" on the memory, preventing pointers with the now-stale key from accessing the memory (analogously to how a landlord might change the lock on a door after tenants move out).

Whenever memory is allocated: (1) a unique key is associated with the region of memory, (2) a lock location is allocated, and (3) the lock value is initialized to the key value. The pointer is not a dangling pointer if and only if the pointer's key and the value of the lock location (the location pointed to by the pointer's lock address) match. When the region is deallocated, the lock location is set to `INVALID_KEY` and marked for possible reuse. A lock location can be reused, but only as another lock location—never for some other purpose. Even when the lock location is reused, because all the keys are unique, the value of a lock location will never again match the key of the now-dangling pointer. The lock address associated with each source pointer makes the dangling-pointer check just a few instructions (*i.e.*, load, compare, and branch).

It is a temporal error to call `free()` on a pointer that was not returned by `malloc()` or to call `free()` twice on the same pointer.[4] To catch these errors, CETS maintains a mapping from keys to *freeable* pointers—given a key that protects some region of memory, there is at most one freeable pointer associated with that key (*i.e.* the one originally returned by `malloc()`). CETS checks the freeable pointer map upon deallocation requests.

3.2 Metadata Operations

For every pointer in the program (*e.g.*, `ptr`), CETS instrumentation introduces a key (*e.g.*, `ptr_key`) and a lock address (*e.g.*, `ptr_lock_addr`). This metadata must be propagated through registers and memory along with the underlying pointer value. Discussion of the encoding of pointers in memory is deferred to Section 3.3, for now we describe how the metadata is created and used.

Initialization. The allocation key value zero is reserved as `INVALID_KEY`. `INVALID_LOCK_ADDR`, the lock location corresponding to `INVALID_KEY` is allocated and set to any non-zero value, so that any pointer with this key will fail its temporal check. The 64-bit global counter variable `next_key` is initialized to one.

[3] Even if a new object was allocated each cycle on a 3Ghz processor, it would still take approximately 200 years to allocate 2^{64} objects.

[4] Unless `malloc()` has recycled the memory region and returns the same heap address again. In this case CETS will reject attempts to call `free()` on stale (old) pointers but permit the new pointer to be freed.

Heap allocation. For any code that allocates memory on the heap (*e.g.*, `malloc()` or `mmap()`), CETS inserts the code (highlighted in gray, below) to: (1) associate a new key with the allocation pointer by incrementing `next_key`, (2) obtain a new lock location, (3) write the key into the lock location, and (4) record that the pointer returned is freeable.

```
ptr = malloc(size);
ptr_key = next_key++;
ptr_lock_addr = allocate_lock();
*(ptr_lock_addr) = ptr_key;
freeable_ptrs_map.insert(ptr_key, ptr);
```

Pointer metadata propagation. All pointers derived from a pointer inherit its key and lock address. CETS operates under the assumption that either the program is free of spatial (bounds) errors or that CETS is coupled with a system for spatial safety [22]. By assuming spatial safety, CETS knows that any pointer manipulation or arithmetic yields a pointer to the *same* memory allocation as the original pointer, and thus the new pointer has the same key and lock address:

```
newptr = ptr + offset;  // or &ptr[index]
newptr_key = ptr_key;
newptr_lock_addr = ptr_lock_addr;
```

Dangling pointer check. CETS inserts code that checks for dangling pointers on memory accesses. The check passes only if the lock value (accessed via the lock address) has the same value as the pointer's key:

```
if (ptr_key != *ptr_lock_addr) { abort(); }
value = *ptr;     // original load
```

Heap deallocation. For code that deallocates heap memory (*e.g.*, `free()` or `unmmap()`), CETS inserts code to: (1) check for double-free and invalid-free by querying the freeable pointers map, removing the mapping if the free is allowed, (2) setting the lock's value to `INVALID_KEY`, and (3) deallocating the lock location:

```
if (freeable_ptrs_map.lookup(ptr_key) != ptr) {
  abort();
}
freeable_ptrs_map.remove(ptr_key);
free(ptr);
*(ptr_lock_addr) = INVALID_KEY;
deallocate_lock(ptr_lock_addr);
```

Allocation/deallocation of lock addresses. Lock address allocation/deallocation overhead is kept low by organizing the unallocated lock addresses as a LIFO free list (`allocate_lock` pops an address from the list; `deallocate_lock` pushes the address back on to the list). The free list can share the memory space with lock locations, so long as the lock addresses are disjoint from the key values, for example, by requiring all keys to have the highest bit set (effectively reducing the size of the key space to 2^{63} keys).

Call stack allocations and deallocations. To detect dangling pointers to the call stack, a key and corresponding lock address is also associated with each stack frame. This key and lock address pair is given to any pointer derived from the stack pointer (and thus points to an object on the stack). Performing a stack allocation is much like calling `malloc()`, except that stack pointers are not freeable, so `freeable_ptrs_map` is unchanged. Stack deallocation is analogous to calling `free()`. To reduce overhead of allocating/reallocating lock addresses for stack frames, CETS uses a separate pool of lock addresses that is itself managed as a stack:

```
void func() {
  // Function prologue
  local_key = next_key++;
  local_lock_addr++; // allocate lock address
  *(local_lock_addr) = local_key;
  int var;  // local variable
  ptr = &var;
  ptr_key = local_key;
  ptr_lock_addr = local_lock_addr;
  // Function epilogue
  *(local_lock_addr) = INVALID_KEY;
  local_lock_addr--; // deallocate lock address
}
```

To further reduce overhead, CETS elides the prologue and epilogue code for functions whose stack variables are all register allocated (or, more generally, any function in which no pointers to a local variable can escape).

Metadata propagation with function calls. When pointers are passed as arguments or returned from functions, the key and lock address must also travel with them. If all arguments were passed and returned on the stack (*i.e.*, via memory and not registers), the shadowspace approach for handling in-memory metadata would suffice. However, the function calling conventions of most ISAs specify that function arguments are generally passed in registers.

To address this issue, CETS uses procedure cloning to transform every function declaration and function call site to include additional arguments for key and lock address. For each pointer argument, key and lock address arguments are added to the end of the list of the function's arguments. Functions that return a pointer are changed to return a three-element structure by value that contains the pointer, its key, and its lock address.

3.3 Disjoint Pointer Metadata to Avoid Fat Pointers

Fat pointers and other in-line metadata encodings interact poorly with arbitrary casts—if casts can overwrite metadata, the system is not guaranteed to detect all temporal errors. CETS avoids fat pointers by using disjoint pointer metadata, which allows arbitrary casts while retaining strong safety guarantees. Separating metadata from the main program memory also eliminates user-visible memory layout-changes that reduce compatibility with existing C source code and pre-compiled libraries.

Trie-based shadowspace implementation. CETS's metadata facility decouples the metadata from the pointers in memory. At its simplest, CETS must map an address to the key and lock address metadata for the pointer at that address (*i.e.*, the lookup is based on the location of the pointer being loaded or stored, not the value of the pointer that is loaded or stored). CETS implements its shadowspace using a two-level *lookup trie* data structure, which provides the ability to shadow every word in memory efficiently [24]. A trie is a mapping structure much like a page table in which each level of the trie is accessed using a set of bits from the index being accessed (see Figure 3). The current CETS prototype assumes a 48-bit virtual address space and that pointers are word-aligned, for a total of 45 bits to index the trie. CETS uses 2^{22} first level entries and 2^{23} second level entries. Each second-level entry contains the key and lock address (128 bits total). Each such trie lookup for a pointer load/store is approximately eleven x86 instructions: four loads, three shifts, two adds, compare, and branch.

Disjoint metadata operation. For pointer metadata associated with register-allocated variables and temporaries, CETS simply inserts the associated metadata as additional temporaries into the compiler's intermediate representation (IR) (see the paragraph

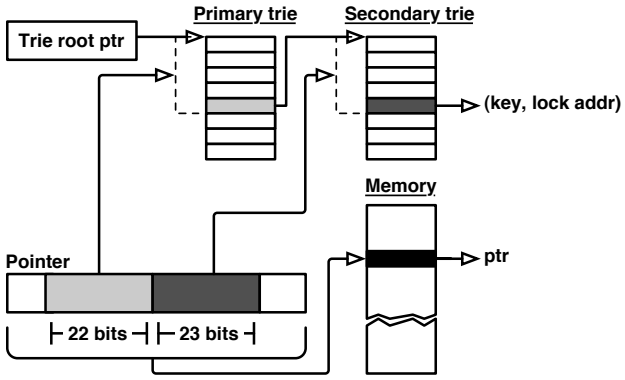

Figure 3. Organization of the trie for pointer metadata

labeled "pointer metadata propagation" above). Loading a pointer value from memory, however, requires CETS to insert code that accesses the shadowspace and loads the corresponding key and lock address pair:

```
int** ptr;
int* newptr;
if (ptr_key != *ptr_lock_addr) { abort(); }
newptr = *ptr;       // original load
newptr_key = trie_lookup(ptr)->key;
newptr_lock_addr = trie_lookup(ptr)->lock_addr;
```

CETS handles writes of pointers to memory analogously:

```
int** ptr;
int* newptr;
if (ptr_key != *ptr_lock_addr) { abort(); }
(*ptr) = newptr;     // original store
trie_lookup(ptr)->key = newptr_key;
trie_lookup(ptr)->lock_addr = newptr_lock_addr;
```

Only loads and stores of pointers are instrumented; loads and stores of non-pointer value are unaffected.

Globals variables. As global variables are never deallocated, CETS associates the constant identifier `GLOBAL_KEY` and the always-valid lock location `GLOBAL_LOCK_ADDR` with any pointer derived from a pointer to a global:

```
int var; // global variable
ptr = &var;
ptr_key = GLOBAL_KEY;
ptr_lock_addr = GLOBAL_LOCK_ADDR;
```

For pointer values that are in the global space and initialized to non-zero values, CETS adds code that explicitly initializes the in-memory keys and lock addresses. This initialization is implemented with the same compiler hooks that C++ uses to run code for constructing global objects (functions with constructor attributes).

3.4 Support for Multithreading

[Fixme: New section. Needs careful reread.] When the program is multi-threaded, it is necessary to ensure that the allocation of `key` and `lock` is properly synchronized. To avoid synchronization when allocating key identifiers, each thread can be given its own counter (for example, using a thread-private "__thread" variable). Each counter ranges over a disjoint region of the 64-bit key identifier space. The allocation of lock locations also requires a a per-thread pool similar to the local/global heap maintained by the Hoard memory allocator. It is also necessary to ensure that the

pointer–metadata loads and stores are atomic. In a race free program, the program's existing synchronization provides the needed atomicity, but, as with all other previous schemes, CETS has problems handling programs with races.

3.5 Integration with Spatial Checking and Casts

Spatial safety. As mentioned above, CETS assumes that the underlying C program is free of spatial safety violations. This is necessary to ensure that the key and lock address metadata is not clobbered accidentally (or maliciously) by using casts or pointer arithmetic operations. Though CETS alone will provide improved temporal safety, especially for debugging purposes (it is unlikely that the shadowspace metadata will be accidentally modified in a way that causes CETS to fail silently), CETS's protection is complete only when spatial safety is also enforced. Our prototype therefore builds upon the SoftBound [22] prototype, which provides the necessary spatial safety guarantees. The disjoint metadata space approach used by CETS is the same strategy used in SoftBound, so the two techniques mesh well together.

Arbitrary casts and unions. C supports arbitrary type conversion by explicit casts and implicit conversions via unions. CETS defers to the spatial safety mechanisms to ensure that these casts do not corrupt the disjoint metadata structures. CETS alone ensures that a pointer dereference can only succeed for locations that are allocated and not stale; it does not provide any assurance about the types of those memory locations.

By default CETS sets the key and lock address metadata of a pointer created from a non-pointer value (such as an integer) to `INVALID_KEY` and `INVALID_LOCK_ADDR`, respectively. This is a safe default (any dereference of such a pointer will trigger a safety violation), but may cause false positives in particularly perverse C programs. To handle such cases—although we have not encountered any code that requires this feature in the applications we have examined—and to support custom memory allocators, CETS provides an interface that allows the programmer to manually set the pointer key and lock address metadata.

4. Formal Model of Memory Safety

As described above, CETS' temporal safety properties can only be guaranteed if spatial safety is also enforced. This section presents a formal model for CETS, which builds on the prior formalization for spatial safety in SoftBound [22] and shows that the combination completely captures both spatial and temporal safety. These claims have been mechanized using the Coq proof assistant [11], and all the Coq proof developments are available online [2].

The grammar in Figure 4 gives the fragment of C we consider. To account for temporal errors, we extend the earlier syntax with `free` and a simple form of function call that has no arguments or results, but does allow stack pointers to escape via the heap. The presentation proceeds in three steps. First, we formalize a non-standard *partial* operational semantics for (simplified, straight-line, and single-threaded) C programs that tracks information about which memory addresses have been allocated. It is undefined whenever a program would cause a memory-safety violation. Second, we augment the operational semantics to abstractly model the results of combined spatial and temporal safety instrumentation of the C program. Finally, we define a well-formedness predicate on syntax that captures invariants ensured by the instrumented program, and prove that starting from a well-formed initial program state, the instrumented version will never get stuck trying to access memory in a way that is either spatially or temporally unsafe. This approach is similar to those used in both CCured's [23] and Cyclone's [16] formal developments.

Atomic Types	a	$::= \texttt{int} \mid p\texttt{*}$
Pointer Types	p	$::= a \mid s \mid n \mid \texttt{void}$
Struct Types	s	$::= \texttt{struct\{}\cdots;a_i:id_i;\cdots\texttt{\}}$
Functions	f	$::= id\texttt{()\{}a_1:x_1\cdots a_n:x_n;c\texttt{\}}$
LHS Expressions	lhs	$::= x \mid \texttt{*}lhs \mid lhs\texttt{.}id \mid lhs\texttt{->}id$
RHS Expressions	rhs	$::= i \mid rhs\texttt{+}rhs \mid \texttt{malloc(}rhs\texttt{)} \mid \texttt{\&}lhs \mid lhs \mid \texttt{(}a\texttt{)}\,rhs \mid \texttt{sizeof(}p\texttt{)}$
Commands	c	$::= c\texttt{ ; }c \mid lhs\texttt{ = }rhs \mid f\texttt{()} \mid \texttt{free(}lhs\texttt{)}$

Figure 4. Syntax for the formal development.

4.1 Operational Semantics

The operational semantics for this C fragment relies on an environment E, that has four components: a map, M, from addresses to values (modeling memory), a map, G, from variable names to their addresses and atomic types (modeling global variables), a map, B, from start addresses of allocated memory regions to size of regions (modeling runtime memory regions), and a stack, S which models function frames, where each frame fr maps variable names to their addresses and atomic types.

A memory M is defined only for addresses that have been allocated to the program by the C runtime. Our formalism axiomatizes properties of six primitive operations for accessing memory: `read`, `write`, `malloc`, `free`, function frame allocation `push`, and function frame deallocation `pop`. The axioms of `malloc` and `free` enforce properties like "malloc returns a pointer to a region of memory that was previously unallocated, and stores this region in B", "free only deallocates a memory region stored at B, and removes this region from B", and "malloc and free do not alter the contents of irrelevant locations." `push` allocates a new frame on the top of the stack S without changing allocated memory. `pop` simply removes the latest frame from the stack. `read` and `write` fail if they try to access unallocated memory; `malloc` and `push` fail if there is not enough space; `free` fails if it tries to deallocate a memory region that is not at B; `pop` fails if the runtime removes a frame from an empty stack.

To model the instrumentation behavior, we extend the memory model with three additional components: a map, $Meta$, from addresses to metadata (modeling metadata storage), a map, L, from `lock address` to `lock` (modeling lock table), and a nk, which is `next_key`. Each allocated location has associated metadata including a `base` (b), `bound` (e), `key` (k), and `lock address` (la). A location with `INVALID_KEY` (0) indicates an unallocated address. The `GLOBAL_KEY` (1) is assigned to all global objects along with a unique gla(`GLOBAL_LOCK_ADDR`). Each frame fr in a stack has its own key and lock address.

[Fixme: Do we really need this paragraph?]

With the above machinery in place, we formalized the combination of CETS and SoftBound [22] in three large-step evaluation rules. Most of the rules are straightforward, and available in our Coq proofs. Left-hand-sides evaluate to a result r (which must be an address l if successful) and its atomic type a: $(E, lhs) \Rightarrow_l r:a$ with no effect on the environment. Evaluating a right-hand-side expression also yields a typed result, but it may also modify the environment E via memory allocation, causing it to become E': $(E, rhs) \Rightarrow_r (r:a, E')$ (where r must be a value with its metadata $v_{(b,e,k,la)}$ if successful). Commands simply evaluate to a result, which must be `OK` if successful, and final environment: $(E, c) \Rightarrow_c (r, E')$. In all cases, r is `Abort` when memory-safety check fails, or `SysError` when `malloc`, `free`, `push` or `pop` fails.

4.2 Safety

The safety result relies on showing that certain well-formedness invariants are maintained by the instrumented program. A well-formed environment $\vdash_E E$ consists of a well-formed global storage G, which ensures that global variables are allocated with well-formed type information, a well-formed stack S, which ensures that local variables of a function are allocated at a frame with well-formed type information, and later frames are of larger key than earlier frames, well-formed allocated memory regions B that must be disjoint from each other, a nk that is larger than 1 (so it is neither invalid or global), **[Fixme: Should this be "larger than 1" or "larger than zero"?]** and a well-formed memory M. A memory M is well-formed when the metadata associated with each allocated location l is well-formed. If l is accessible, its value $v_{(b,e,k,la)}$ satisfies the properties:

1. $(k < nk) \wedge ((b = 0) \vee ((b \neq 0) \wedge (k \neq 0) \wedge (L(la) = k \implies \forall l \in [b,e).\ \texttt{val}\ M\ l)))$. Here, $\texttt{val}\ M\ l$ is a predicate that holds when a location l is allocated in memory M.
2. If k is equal to the key of another accessible value, then their lock addresses are also equal. Particularly, if k is 1, la must be gla; if k is equal to the key of a frame fr, la is same as the lock address of that frame.
3. $[b,e)$ is within the range of global variables if and only if $k = 1 \wedge la = gla$. Similarly, $[b,e)$ is within the range of a frame on the stack if and only if k and la are equal to the key and lock address of that frame.
4. If $[b,e)$ is within an allocated memory region in B, then for any accessible value $v'_{(b',e',k',la')}$ that is also within this range, $k \neq k' \implies (L(la) \neq k \vee L(la') \neq k')$; If $[b,e)$ is within the range of a frame on the stack, and k is not equal to the key of that frame, then $L(la) \neq k$.

With the above well-formedness conditions, the type safety theorems show that CETS+SoftBound can detect memory violations at runtime. $E.G, E.S \vdash_c c$ defines a well-formed command that ensures that each variable in a command c has an atomic type assigned by global variables G, or stack frames S in an environment E.

THEOREM 4.1 (Preservation). *If* $\vdash_E E$, $E.G, E.S \vdash_c c$ *and* $(E,c) \Rightarrow_c (r, E')$, *then* $\vdash_E E'$.

THEOREM 4.2 (Progress). *If* $\vdash_E E$ *and* $E.G, E.S \vdash_c c$, *then* $\exists\ E'.\ (E,c) \Rightarrow_c (ok, E')$ *or* $\exists\ E'.\ (E,c) \Rightarrow_c (\text{SystemError}, E')$ *or* $\exists\ E'.\ (E,c) \Rightarrow_c (\text{Abort}, E')$.

THEOREM 4.3 (Backward Simulation). *If* $\vdash_E E$, $E.G, E.S \vdash_c c$ *and* $(E,c) \Rightarrow_c (ok, E')$, *then the original C program will not cause any memory violation.*

5. Temporal Check Elimination

Although bounds check elimination is an established and long-studied problem (*e.g.*, [7, 17, 29]), there is little published work on eliminating temporal safety checks. This section describes and evaluates simple optimizations we developed for removing temporal checks. These optimizations can apply to other temporal checking schemes (such as those described in Section 2), but we have implemented and studied their effectiveness in the context of CETS. Our temporal check optimizations fall into two categories: removal of unnecessary checks and removal of redundant checks.

5.1 Unnecessary check elimination

Some temporal checks are unnecessary. No temporal checks are required for any pointer that is directly derived from the stack pointer within the corresponding function call, because the stack frame is guaranteed to live until the function exits. In the same vein, checking stack spills and restores is unnecessary. As CETS works on an intermediate representation (IR) with infinite registers, spills and restores are not visible at the IR level, so such checks are simply not inserted. In addition, performing temporal checks to pointers known to point to global objects is unnecessary, because global objects are never deallocated. CETS uses simple dataflow analysis to identify these pointers and elides their checks.

5.2 Redundant check elimination

A temporal check is redundant if: (1) there is an earlier check to pointers that share the same values for key and lock address (*i.e.*, points to the same object) and (2) the check is not killed by a call to `free()`. Our algorithm for finding redundant checks constructs a dominator tree and standard dataflow analysis to identify the root object accessed by each reference. Any check dominated by an earlier check that is not killed along the path is removed. In our current implementation a temporal check is killed when there is any function call along the path—that is, we conservatively assume that any function could potentially `free()` any pointer. This analysis could be further improved by using an interprocedural analysis to identify which functions call `free()`.

Example. Consider the code snippet below:

```
if (ptr1_key != *ptr1_lock_addr) { abort(); }

... = *(ptr1+offset1);
*(ptr2) = ...;   // potentially aliasing store

if (ptr1_key != *ptr1_lock_addr) { abort(); }

... = *(ptr1+offset2);
```

Naive temporal check insertion would insert two temporal checks (shaded). Our temporal-check elimination optimization removes the second check as redundant, because it knows that (1) the checks are to the same object and (2) the intervening code does not kill the validity of temporal check (a store can not deallocate memory and thus cannot cause `ptr1` to dangle). The store to `ptr2` however blocks standard compiler optimizations from removing the redundant loads of `*ptr1_lock_addr`. However, if the store was not present (or with better alias analysis) standard compiler optimizations alone would easily eliminate the second temporal check.

This example also illustrates two key differences between spatial and temporal check elimination. First, even though the two loads are to different addresses, the temporal check is redundant whereas a bounds check would generally not be redundant. Second, a function call between the two loads would block a temporal check removal, whereas the redundancy of spatial check to the same address is independent of the intervening code.

5.3 Check removal effectiveness

The graph in Figure 5 shows the percentage of runtime temporal checks eliminated as a percentage of runtime checks preformed by CETS without such temporal-check specific optimizations—taller bars are better. Unnecessary-check elimination (the striped segment of each bar) removes on average 46% of the temporal checks. For benchmarks like `go`, `crafty` and `sjeng`, which operate on large number of global arrays, more than 80% of the checks are eliminated. The solid segments of each bar in Figure 5 represent the percentage of temporal checks removed as a result of redundant check elimination. In benchmarks such as `lbm` and `ammp`, more than 75% of the temporal checks are removed. On an average for the benchmarks, the combination of both redundant and unnecessary check elimination removes 70% of the temporal checks.

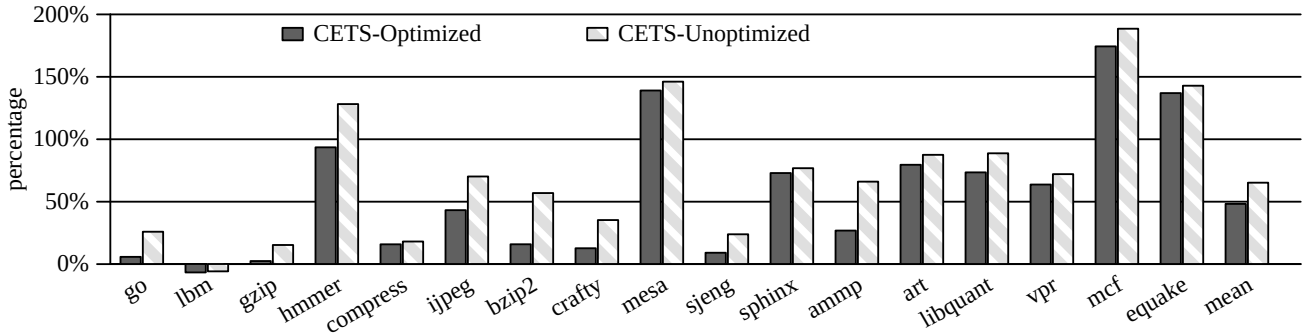

Figure 6. Normalized execution time overhead of CETS with (left bar) and without (right bar) the optimizations described in Section 5.

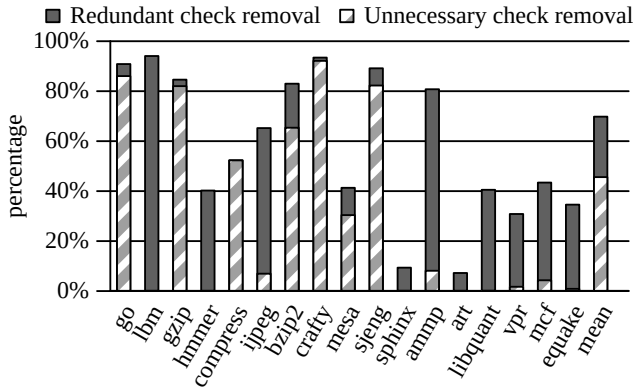

Figure 5. Percentage of temporal dereference check eliminated with various optimizations over CETS without optimizations.

6. Experiments

This section describes the CETS prototype and experimental evaluation. The goal of the evaluation is (1) to show that CETS is effective in detecting temporal violations, (2) to measure the runtime overhead, (3) to show the impact of optimization in reducing the runtime overhead, (4) to contrast it with our implementation of other proposals for temporal checking and (5) to evaluate the overhead of providing complete memory safety for C by coupling CETS with SoftBound [22].

6.1 Prototype

We implemented CETS by modifying the publicly available SoftBound code base because CETS employs a similar disjoint per-pointer metadata approach. We replaced SoftBound's existing metadata space code with our own trie-based shadowspace implementation. The prototype is built upon LLVM [21] version 2.6 and operates on the LLVM's typed single static assignment (SSA) form. LLVM invokes a standard suite of optimizations both before and after the CETS pass. The instrumentation code inserted by CETS is written as C functions that are subsequently forcibly inlined by LLVM. The CETS pass itself is approximately 7K lines of C++ code, and we plan to make the source code publicly available [2]. CETS operates on LLVM's ISA-independent intermediate form so it is architecture independent, but we selected 64-bit x86 as the ISA and a 2.66Ghz Intel Core 2 Duo for reporting runtime results.

Benchmarks We used 17 benchmarks selected from the SPECint and SPECfp benchmark suites to evaluate CETS's performance. The CETS prototype is not yet robust enough to compile all of the C programs in the SPEC benchmark suites, but we present data for the benchmarks we examined for which our prototype compiles correctly. In all the graphs presented in this paper, the benchmarks are ordered (from left to right) by the ratio of the number of pointer metadata accesses to the number of pointer dereferences carried out by the program.

Effectiveness To evaluate the effectiveness of CETS in detecting temporal violations, we applied CETS to a set of thirty programs with temporal errors from the NIST-SAMATE benchmark suite [1] and our own suite consisting of several test programs exercising various temporal errors involving both the stack and the heap. CETS successfully detected all the temporal errors in these two test suites. No false violations were reported for the suites or the SPEC benchmarks used in our runtime experiments.

Runtime Overhead Figure 6 presents the percentage runtime overhead of CETS (smaller bars are better as they represent lower runtime overheads). This graph contains a pair of bars for each benchmark. The left bar of each group show that CETS's overall average runtime overhead for detecting all violations of temporal safety is 48%. As 70% of temporal checks on average are eliminated (Section 5) the temporal checks themselves are not the overwhelming source of overhead for many benchmarks.

The non-check overheads come from a variety of different sources. The benchmarks on the right side of the graph have a higher percentage of loads and stores of pointers, which corresponds to higher instruction overhead and cache pressure caused by frequent shadowspace accesses. To estimate another source of overhead, we measured the runtime overhead of just allocating/deallocating the lock locations and unique keys. We generally found this overhead to be negligible (one exception was the call-intensive benchmark `mesa`, with an overhead of 52% just for such allocations/deallocations). To better understand the remaining sources of overhead, we recorded the number of memory operations to the stack in the baseline and CETS configurations. We found that some benchmarks exhibited a dramatic increase in the number of stack accesses (5x in some cases). These results and examination of the assembly code generated by LLVM indicate that tracking the key and lock location values increases register pressure enough—x86-64 has only 16 registers—to introduce spills and restores to comprise a significant component of the remaining overheads for some benchmarks.

Impact of Compile-Time Optimizations The right bar in each group in Figure 6 shows the runtime overhead of CETS without the temporal-specific optimizations described in Section 5. Some benchmarks see substantial performance improvements (67% to 26% for `ammp` and 56% to 16% for `bzip2`), and the overall runtime overhead improves from 66% to 48%. The relatively smaller impact than suggested by the 70% reduction in temporal checks

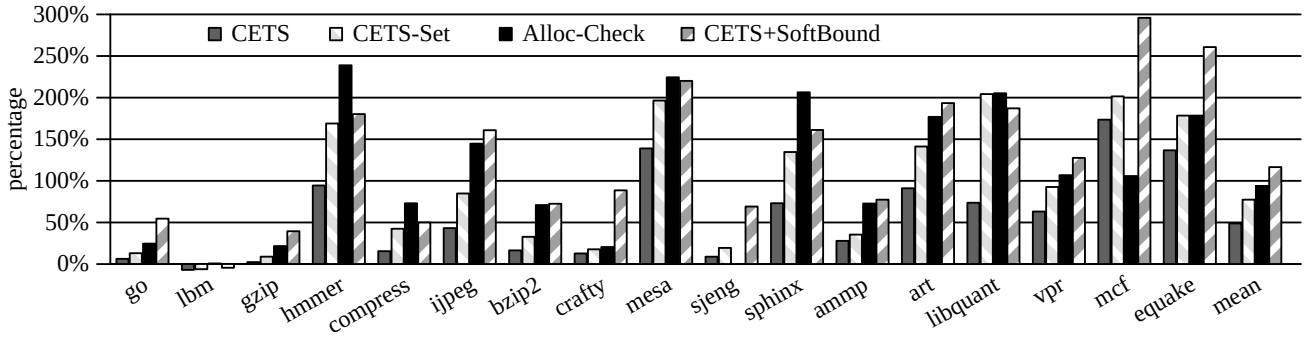

Figure 7. Normalized execution time overhead of CETS, CETS-Set, Alloc-Check and CETS+SoftBound over the baseline.

(Section 5) is mostly likely explained by two effects. First, as discussed above, other non-check overheads are significant in many benchmarks. Second, the standard compiler optimizations applied after instrumentation (enabled for both configurations in Figure 6) alone eliminates many redundant checks.

6.2 Design Alternatives

We also compared CETS instrumentation some alternative temporal checking designs. To provide an informative comparison, these alternatives were implemented within the compiler and benefit from the temporal check optimization (Section 5).

Comparison to CETS-Set. CETS-Set is a set-based variant (Section 2.2) that performs a hash table lookup on each temporal check to determine whether the key is valid. This approach maintains just the key (no lock address) with each pointer, reducing the metadata per pointer by half in exchange for a more expensive temporal check. The second bar of each group in Figure 7 illustrates the runtime overhead of CETS-Set. The runtime overhead of such a scheme is higher than CETS—77% versus 48% on average—justifying our selection of the lock-and-key approach for CETS.

Comparison to Alloc-Check. Alloc-Check is a shadowspace-based location lookup approach to temporal checking (described in Section 2.1). This design point is intended to represent a compiler-based implementation of the incomplete temporal checking provided by tool such as Valgrind's Memcheck. As this is a location-based approach, dangling pointers to reallocated locations are not detected. This scheme maintains one bit with each word in memory to record the allocation status of the word. The metadata space is implemented using the same two-level trie as CETS uses for its per-pointer metadata shadowspace. Every dereference check involves a trie lookup to confirm the memory being accessed is allocated. The third bar of each benchmark in Figure 7 presents the runtime overhead of Alloc-Check. The runtime overhead of Alloc-Check is 94% on average, which is higher overhead than CETS while providing weaker detection. **[Fixme: New prose]**An alternative implementation of Alloc-Check with a splay tree has an average overhead of 314%, which is higher than the one bit with each word implementation. Alloc-Check also provides some detection of spatial errors as well, so we next evaluate CETS combined with spatial checking.

6.3 Combined Spatial and Temporal Checking

We also tested CETS in combination with SoftBound because together they provide detection of all temporal and spatial violations of memory safety. The combined approach extends the metadata maintained with each pointer from two to four words: `base`, `bound`, `key` and `lock address` to enable both spatial and temporal checks. The fourth bar of each group in Figure 7 shows the average runtime overhead for detecting all spatial and temporal errors is 116%. Spatial checking alone (not shown) has 83% overhead, so adding CETS's temporal checking to SoftBound increases overhead by only 33%. When compared to Alloc-Check's overhead of 94%, CETS's overhead is somewhat higher, but Alloc-Check does not provide complete detection of either spatial or temporal errors.

7. Related Work

In addition to the various approaches described in Section 2, we describe other related approaches in this section.

Garbage collection. Garbage collection (either conservative garbage collection [8] or reference counting [10]) is an alternative approach to eliminating dangling pointers for application domains in which garbage collection is suitable. When combined with "heapification" [23] of escaping stack objects, garbage collection can eliminate all dangling pointers. Garbage collection masks temporal errors rather than reporting them, so it may not be ideal in a debugging context.

Infinite heap abstraction based approaches. DieHard [6] provides probabilistic memory safety by approximating an infinite sized heap using the runtime. It uses a randomized memory manager which places objects randomly across a heap. Randomized allocation makes it unlikely that a newly freed object will soon be overwritten by a subsequent allocation thus avoiding dangling pointer errors. Exterminator [26] builds on top of DieHard, and it carries out error detection and correction based on data accumulated from multiple executions without programmer intervention.

Using page-level permissions. Other proposals provide infinite heap abstractions using the virtual address translation mechanism to detect temporal errors without inserting any checking code. The open-source tools Electric Fence, PageHeap, and DUMA allocate each object on a different physical and virtual page. Upon deallocation, the access permission on individual virtual pages are disabled, causing high runtime overheads for allocation-intensive programs [14]. As these schemes allocate one object per virtual and physical page, large memory overheads can result for programs with many small objects. More recently, proposals have addressed the physical memory overhead issue by placing multiple objects per physical page, but mapping a different virtual page for each object to the shared page [14].

Indirect temporal safety. Other relevant proposals are those that perform checking aimed at other types of errors and the same checks will also catch a subset of temporal safety violations, but temporal safety is not the primary aim of these proposals. Examples include: control flow integrity [3], dataflow integrity based on reaching definition analysis calculated statically [9], or using

pointer analysis to compute the approximate set of objects written by each instruction [4].

Another approach based on automatic pool allocation [15] assigns all heap-allocated objects to type homogenous pools, where each member has the same type and alignment. On reallocation of freed memory, new object will have the same type and alignment as the old object. Thus, dereferencing dangling pointers to reallocated memory cannot cause violations of type safety. This approach does not prevent dereferences through dangling pointers but only ensures that such dereferences do not violate type safety.

8. Conclusion

This paper proposed CETS, a compile time instrumentation for detecting all violations of temporal safety for C. The primary goal was to explore the design choices and the optimizations for a compiler based system. In this regard, CETS uses a `key` and `lock address` with each pointer in a disjoint metadata space and checks pointer dereferences. A mechanized formal proof showed that the CETS's approach provided complete temporal safety even in the presence of arbitrary casts and reallocations. Further, CETS post-optimization based instrumentation, subsequent optimizations passes, and temporal check removal optimizations, enabled it to achieve an average execution overhead of 48%. When coupled with a system providing complete spatial safety, CETS provides complete memory safety for C at an average execution time overhead of 116%. This runtime overhead meets our goals by providing a memory error checking tool that has low enough runtime overhead as to be practical for used by default in all stages of the software development life cycle.

References

[1] NIST SAMATE Reference Data Set. http://samate.nist.gov/SRD/.

[2] SoftBound website. http://www.cis.upenn.edu/acg/softbound/.

[3] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-Flow Integrity. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, Nov. 2005.

[4] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro. Preventing Memory Error Exploits with WIT. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, 2008.

[5] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient Detection of All Pointer and Array Access Errors. In *Proceedings of the SIGPLAN 1994 Conference on Programming Language Design and Implementation*, June 1994.

[6] E. D. Berger and B. G. Zorn. DieHard: Probabilistic Memory Safety for Unsafe Languages. In *Proceedings of the SIGPLAN 2006 Conference on Programming Language Design and Implementation*, June 2006.

[7] R. Bodík, R. Gupta, and V. Sarkar. ABCD: Eliminating Array Bounds Checks on Demand. In *Proceedings of the SIGPLAN 2000 Conference on Programming Language Design and Implementation*, June 2000.

[8] H.-J. Boehm. Space Efficient Conservative Garbage Collection. In *Proceedings of the SIGPLAN 1993 Conference on Programming Language Design and Implementation*, June 1993.

[9] M. Castro, M. Costa, and T. Harris. Securing Software by Enforcing Data-Flow Integrity. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, Nov. 2006.

[10] J. Condit, M. Harren, Z. Anderson, D. Gay, and G. C. Necula. Dependent Types for Low-Level Programming. In *Proceedings of the 16th European Symposium on Programming*, Apr. 2007.

[11] The Coq Development Team. *The Coq Proof Assistant Reference Manual (Version 8.2pl1)*, 2009.

[12] J. Devietti, C. Blundell, M. M. K. Martin, and S. Zdancewic. Hardbound: Architectural Support for Spatial Safety of the C Programming Language. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, Mar. 2008.

[13] D. Dhurjati and V. Adve. Backwards-Compatible Array Bounds Checking for C with Very Low Overhead. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, 2006.

[14] D. Dhurjati and V. Adve. Efficiently Detecting All Dangling Pointer Uses in Production Servers. In *Proceedings of the International Conference on Dependable Systems and Networks*, June 2006.

[15] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. Memory Safety Without Runtime Checks or Garbage Collection. In *Proceedings of the 2003 ACM SIGPLAN Conference on Language, Compiler, adn Tool for Embedded Systems (LCTES)*, 2003.

[16] D. Grossman. *Safe Programming at the C Level of Abstraction*. PhD thesis, Cornell University, Aug. 2003.

[17] R. Gupta. A Fresh Look at Optimizing Array Bound Checking. In *Proceedings of the SIGPLAN 1990 Conference on Programming Language Design and Implementation*, June 1990.

[18] R. Hastings and B. Joyce. Purify: Fast Detection of Memory Leaks and Access Errors. In *Proceedings of the Winter Usenix Conference*, 1992.

[19] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A Safe Dialect of C. In *Proceedings of the 2002 USENIX Annual Technical Conference*, June 2002.

[20] R. W. M. Jones and P. H. J. Kelly. Backwards-Compatible Bounds Checking for Arrays and Pointers in C Programs. In *Third International Workshop on Automated Debugging*, Nov. 1997.

[21] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, 2004.

[22] S. Nagarakatte, J. Zhao, M. M. K. Martin, and S. Zdancewic. SoftBound: Highly Compatible and Complete Spatial Memory Safety for C. In *Proceedings of the SIGPLAN 2009 Conference on Programming Language Design and Implementation*, June 2009.

[23] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer. CCured: Type-Safe Retrofitting of Legacy Software. *ACM Transactions on Programming Languages and Systems*, 27(3), May 2005.

[24] N. Nethercote and J. Seward. How to shadow every byte of memory used by a program. In *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2007.

[25] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the SIGPLAN 2007 Conference on Programming Language Design and Implementation*, June 2007.

[26] G. Novark, E. D. Berger, and B. G. Zorn. Exterminator: Automatically Correcting Memory Errors with High Probability. In *Proceedings of the SIGPLAN 2007 Conference on Programming Language Design and Implementation*, June 2007.

[27] H. Patil and C. N. Fischer. Low-Cost, Concurrent Checking of Pointer and Array Accesses in C Programs. *Software — Practice & Experience*, 27(1):87–110, 1997.

[28] G. Venkataramani, B. Roemer, M. Prvulovic, and Y. Solihin. MemTracker: Efficient and Programmable Support for Memory Access Monitoring and Debugging. In *Proceedings of the 13th Symposium on High-Performance Computer Architecture*, Feb. 2007.

[29] T. Würthinger, C. Wimmer, and H. Mössenböck. Array Bounds Check Elimination for the Java HotSpot Client Compiler. In *Proceedings of the 5th international symposium on Principles and practice of programming in Java*, 2007.

[30] W. Xu, D. C. DuVarney, and R. Sekar. An Efficient and Backwards-Compatible Transformation to Ensure Memory Safety of C Programs. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, 2004.

[31] S. H. Yong and S. Horwitz. Protecting C Programs From Attacks via Invalid Pointer Dereferences. In *Proceedings of the 11th ACM SIG-*

SOFT International Symposium on Foundations of Software Engineering (FSE), 2003.