MemSafe: Ensuring the Spatial and Temporal Memory Safety of C at Runtime

Matthew S. Simpson, Rajeev K. Barua
Department of Electrical & Computer Engineering
University of Maryland, College Park
College Park, MD 20742-3256, USA
{*simpsom, barua*}@*umd.edu*

Abstract

Memory access violations are a leading source of unreliability in C programs. As evidence of this problem, a variety of methods exist that retrofit C with software checks to detect memory errors at runtime. However, these methods generally suffer from one or more drawbacks including the inability to detect all errors, the use of incompatible metadata, the need for manual code modifications, and high runtime overheads.

In this paper, we present a compiler analysis and transformation for ensuring the memory safety of C called MemSafe. MemSafe makes several novel contributions that improve upon previous work and lower the cost of safety. These include (1) a method for modeling temporal errors as spatial errors, (2) a metadata representation that combines features of both object- and pointer-based approaches, and (3) a dataflow representation that simplifies optimizations for removing unneeded checks. MemSafe is capable of detecting real errors with lower overheads than previous efforts. Experimental results show that MemSafe detects all memory errors in 6 programs with known violations and ensures complete safety with an average overhead of 87% on 30 large programs widely-used in evaluating error detection tools.

1. Introduction

Use of the C programing language remains common despite the well-known memory errors it allows. The features that make C desirable for many system-level programing tasks—namely its weak typing, low-level access to computer memory, and pointers—are the same features whose misuse cause the variety of difficult-to-detect memory access violations common among C programs. Although these violations often cause a program to crash immediately, their symptoms frequently go undetected long after they occur, resulting in data corruption and incorrect results while making software testing and debugging particularly onerous.

A commonly cited memory error is the buffer overflow, where data is stored to a memory location outside the bounds of the buffer allocated to hold it. Although these errors are well-understood [e.g. 1–7], they and other memory access violations still plague modern software and are a major source of recently reported security vulnerabilities. For example, according to the United States Computer Emergency Readiness Team, 67 (29%) of the 228 vulnerabilities from 2008–2009 were due to buffer overflow errors [8].

Several safety methods [4, 9–11] have characterized memory access violations as either *spatial* or *temporal* errors. A spatial error is a violation caused by dereferencing a pointer that refers to an address outside the bounds of its "referent." Examples include indexing beyond the bounds of an array; dereferencing pointers obtained from invalid pointer arithmetic; and dereferencing uninitialized, `NULL` or "manufactured" pointers.[1] A temporal error is a violation caused by using a pointer whose referent has been deallocated (e.g. with `free`) and is no longer a valid object. The most well-known temporal violations include dereferencing "dangling" pointers to dynamically allocated memory and freeing a pointer more than once. Dereferencing pointers to automatically allocated memory (stack variables) is also a concern if the address of the referent "escapes" and is made available outside the function in which it was defined. A program is *memory safe* if it does not commit any spatial or temporal errors.

Safe languages, such as Java and Euclid [12], ensure memory safety with a combination of syntax restrictions and runtime checks, and can be used when security is a major concern. Others, like Cyclone [13] and Deputy [14], preserve many of the low-level features of C, but require additional programmer annotations to assist in ensuring safety. Although the use of such languages may be ideal for safety-critical environments, the reality is that many of today's applications—including operating systems, web browsers, and database systems—are still implemented in C or C++ because of its efficiency, predictability, and access to low-level features. This trend will likely continue into the future.

An alternative to safe languages, sophisticated static analysis methods for C [e.g. 15–17] can be used alone, or in conjunction with other systems, to ensure the partial absence of spatial and temporal errors statically. However, while these techniques are invaluable for software verification and debugging, they can rarely prove the absence of all memory errors.

A growing number of methods rely primarily on runtime checks to detect memory errors. However, of the methods capable of detecting both spatial and temporal errors [1, 5, 6, 9–11, 18–22], they generally suffer from one or more practical drawbacks that have thus far limited their widespread adoption. These drawbacks (detailed in Section 2) can be summarized with the following:

- **Completeness.** Methods that associate metadata (the base and bound information required for checks) with objects [e.g. 1, 5, 6, 18–20]—rather than pointers to objects—cannot detect all spatial and temporal errors.

1. A *manufactured* pointer is a pointer created by means other than explicit memory allocation (e.g., `malloc`) or taking the address of a variable using the address-of operator. Incorrect pointer type-casting is a common example.

- **Compatibility.** The use of inline pointer metadata, such as multi-word "fat-pointers" [e.g. 10, 21] to store metadata breaks many legacy programs—and requires implicit language restrictions for new ones—because it changes the memory layout of pointers.
- **Code Modifications.** Some methods [e.g. 21] require non-trivial source code modifications to avoid the above compatibility issues or to prevent an explosion in runtime.
- **Cost.** Methods that are complete often suffer from high performance overheads [e.g. 9–11, 22]. This is commonly due to the cost of maintaining the required metadata and the use of a garbage collector.

In this paper, we describe an approach for ensuring both the *spatial* and *temporal* memory safety of C programs, which we call MemSafe. MemSafe is a whole-program compiler analysis and transformation that, like other runtime methods, utilizes a limited amount of static analysis to prove memory safety whenever possible, and then inserts checks to ensure the safety of the remaining memory accesses at runtime. MemSafe is *complete*, *compatible*, requires no *code modifications*, and generally has lower *cost* than other complete and automatic methods achieving the same level of safety.

MemSafe makes the following novel contributions to lower the runtime cost of dynamically ensuring memory safety:

- MemSafe uniformly handles memory violations by modeling temporal errors as spatial errors. Therefore, the use of separate mechanisms for detecting temporal errors (e.g. garbage collection and "temporal capabilities" [9–11, 20]) is no longer required.
- MemSafe captures the most salient features of object- and pointer-based spatial metadata in a hybrid representation that ensures its compatibility while allowing for the detection of temporal errors as well.
- MemSafe uniformly handles pointer data-flow in a representation that simplifies several performance-enhancing optimizations. Unlike previous methods that require runtime checks for all dereferences and the expensive propagation of metadata at every pointer assignment [e.g. 9–11, 22], MemSafe eliminates redundant checks and the propagation of unused metadata. This capability is further enhanced with whole-program analysis.

We have evaluated our implementation of MemSafe in terms of its completeness and cost. MemSafe was able to successfully detect all previously reported memory errors in 6 programs from the BugBench [23] benchmark suite. In terms of cost, MemSafe's average overhead was 87% on 30 large programs widely-used in evaluating error detection tools. As evidence of its compatibility, MemSafe compiled each program without requiring code modifications.

Table 1 summarizes previous software approaches for ensuring both spatial and temporal safety.[2] Each method is evaluated on its completeness, compatibility, lack of code modifications, use of whole-program analysis, and cost. For consistency, slowdown is reported for the Olden benchmarks [24] where results are available. MemSafe compares favorably in each category and has the lowest overhead among all existing complete and automatic methods. This result is primarily due to the novel contributions outlined above.

Approach	Complete	Compat.	Code Mods.	Whole Prog.	Slowdown
Purify [19]	no	yes	yes	yes	148.44*
Patil, Fischer [11]	yes	yes	yes	no	6.38†
Safe C [10]	yes	no	yes	no	4.88†
Fail-Safe C [22]	yes	yes	yes	no	4.64†
MSCC [9]	yes	yes	yes	no	2.33
Yong, Horwitz [20]	no	yes	yes	no	1.37‡
CCured [21]	yes	no	no	yes	1.30
MemSafe	**yes**	**yes**	**yes**	**yes**	**1.31**

Table 1: Related Work. A comparison of methods providing both spacial and temporal memory safety is given. Slowdown is reported for the Olden benchmarks [24] unless otherwise noted.

∗. Checks are only inserted for heap objects.
†. Slowdown is the average of all results reported by the authors.
‡. Checks are only inserted for store operations.

Since MemSafe's performance overheads cannot be considered "low" at this point, we recommend it be deployed only in situations where safety is a primary concern. In our experience, many checks can be avoided with our simple optimizations, and for safety-critical applications, MemSafe's moderate overheads can be an acceptable trade-off compared to redesigning systems using a safe language. However, for performance-critical applications, MemSafe should primarily be used as a dynamic bug detection tool.

2. Background

As evidence of the drawbacks mentioned above, the instrumentation of C programs to ensure memory safety remains an actively researched topic. In this section, we review previous approaches, primarily focussing on their use of metadata.

2.1. Spatial Safety

The goal of spatial safety is to ensure every memory access occurs within bounds of a known object. Safety is typically enforced by inserting checks before pointer dereferences. Alternatively, checking for bounds violations after pointer arithmetic is also possible [e.g. 1, 5–7], but requires care since pointers in C are allowed to be out-of-bounds so long as they are not dereferenced. The metadata required for safety checks can be associated either with objects or pointers, and there are strengths and weaknesses of each approach.

Object Metadata Methods that utilize object metadata usually record the base and bound information of objects as they are allocated in a global database that relates every address in an allocated region to the metadata of its corresponding object. Advantages of this approach include efficiency, since it avoids propagating metadata at every pointer assignment, and compatibility, since it does not change the layout of memory or prohibit the use of precompiled libraries. Prominent methods employing this strategy include the work by Jones and Kelly [1], Ruwase and Lam [5], Dhurjati and Adve [6], Akritidis et al. [7], SafeCode [28] and SVA [18].

2. We exclude from Table 1 other methods (e.g, CIT [25], DFI [26], WIT [27], SoftBound [4], SafeCode [28], "baggy" bounds checking [7], etc.) since they either are not software-only or do not aim to ensure complete spatial and temporal safety. However, we discuss these in Section 6.

However, the use of object metadata results in several drawbacks. First, this approach prevents complete safety. Since nested objects (e.g., an array of structures) are assigned a base and bound that span the entire allocated region, it is impossible to detect sub-object overflows if an out-of-bounds pointer to an inner object remains *within* bounds of the outer object. Second, this approach requires a runtime lookup to retrieve metadata from the object database.

Pointer Metadata An alternative to using object metadata is to associate metadata with pointers. When a new pointer is created (with `malloc` or the address-of operator), its metadata is initialized to the bounds of its referent, and when a pointer uses the value of another pointer (e.g., pointer arithmetic), its metadata is inherited from the original pointer. Advantages of this approach include avoiding costly database lookups and the ability to ensure complete safety, since sub-object overflows can be detected by giving each pointer a unique base and bound. Prominent methods employing this strategy include Safe C [10], Fail-Safe C [22] and CCured [21].

However, the use of pointer metadata also results in several drawbacks. First, this approach is often not compatible with and breaks many programs. A common implementation of pointer metadata relies on multi-word blocks of memory, called fat-pointers, that record base and bound information inline with pointers. Since this increases a pointer's size beyond that of the word size of the target architecture, many programming idioms no longer work as expected. Additionally, interfacing with external libraries becomes difficult and requires wrapper functions to pack and unpack fat-pointers at boundaries with uninstrumented code. A second drawback is cost: while avoiding expensive database lookups, pointer metadata must be propagated at every pointer assignment.

MemSafe's Approach MemSafe captures the most salient features of object and pointer metadata in a hybrid representation. To ensure complete and compatible spatial safety, MemSafe maintains disjoint pointer metadata in an approach similar to that of SoftBound [4]. However, to lower cost, MemSafe propagates pointer metadata only when needed for runtime checks. Additionally, MemSafe maintains some object metadata but performs lookups only when pointer metadata is insufficient for ensuring temporal safety.

2.2. Temporal Safety

The goal of temporal safety is to ensure every memory accesses refers to an object that has not been deallocated. Violations occur when dereferencing pointers to stack objects if their function has exited and when dereferencing pointers to heap objects if they have been freed. Temporal safety is typically enforced with garbage collection or by software checks. Like the methods for ensuring spatial safety, there are strengths and weaknesses of each approach.

Garbage Collection Methods using garbage collection to prevent dangling pointers to heap objects ignore `free` and replace `malloc` with the Boehm-Demers-Weiser conservative collector [29]. To prevent dangling pointers to stack objects, local variables can be "heapified" and moved to the heap to be managed by the collector. This is the approach taken by CCured [21] and Fail-Safe C [22].

```
    Atomic Types α  ::= int | τ*
           Types τ  ::= α | struct{d⁺} | τ[n]
    Declarations d  ::= τ x;
       Functions f  ::= func(x*) {s⁺ b*}
          Blocks b  ::= p* s⁺
     ϕ-Functions p  ::= x = ϕ(x⁺);
 LHS Expressions l  ::= x | *l | l.y
 RHS Expressions r  ::= r+r | l | &l | (α)r | malloc(r) | n
      Statements s  ::= d | l=r; | l(r); | for(l=r; r<r; l=r) {b}
                     |  if(r) {b} else {b} | return r; | free(r);
                where   x ∈ variables
                        y ∈ structure field identifiers
                        n ∈ ℕ
```

Figure 1: Syntax. Syntax is givien for a simple SSA[32] language with pointers, control flow and manual memory management.

However, garbage collection negates several of C's primary benefits, including its predictability and low-level access to memory. Garbage collection voids real-time guarantees [30], increases address space requirements, reduces reference locality, and increases page fault and cache miss rates [31]. Moreover, since the garbage collector must be conservative, some memory may never be reclaimed, resulting in memory leaks. Finally, heapifying stack objects increases cost since dynamic allocation is slower than automatic allocation.

Temporal Checks An alternative to garbage collection is to insert explicit software checks to test the temporal validity of referenced objects. To achieve this, a "capability store" is commonly used to record the temporal capability of objects as they are created and destroyed. Additional metadata created and propagated with spatial metadata links a pointer to the temporal capability of its referent. Methods employing this strategy include Safe C [10], MSCC [9], and the work by Patil and Fischer [11] and Yong and Horwitz [20].

There are advantages and disadvantages of explicit temporal checks. The primary strength of this approach is that it retains C's memory allocation model and avoids the drawbacks of garbage collection. However, the inclusion of additional checks and metadata significantly increases runtime beyond that of spatial safety alone.

MemSafe's Approach Since MemSafe models temporal errors as spatial errors, it does not require garbage collection or explicit temporal checks. Instead it uses spatial safety checks to ensure temporal safety, which avoids the above drawbacks.

3. MemSafe

In this section, we describe MemSafe's approach for ensuring the memory safety of C at runtime. MemSafe is a compiler analysis and transformation that inserts software checks before memory accesses to detect spatial and temporal violations. It requires a limited amount of static analysis (a flow- and context-insensitive alias analysis) to avoid unnecessary checks and metadata propagation for memory accesses that it can statically verify to be safe.

Figure 1 defines a small SSA[32] intermediate language capturing the relevant pointer-related portions of C. We will use it in describing MemSafe's translations for check insertion and metadata propagation. Without loss of generality, we assume memory is only accessed with explicit load (`x=*ptr`) and store (`*ptr=x`) operations, and an unlimited number of "virtual" registers exist for holding temporary values. The following discussion presents (1) syntax extensions for MemSafe's data-flow representation, (2) MemSafe's checks and metadata, (3) program translations for propagating metadata, and (4) optimizations for reducing cost.

3.1. A Data-flow Graph for Pointers

MemSafe represents memory deallocation and pointer store operations as explicit assignments. The advantage of this approach is that it enables MemSafe to ensure complete safety by reasoning solely about pointer definitions, which eliminates the need for separate mechanisms for detecting spatial and temporal errors and reveals optimization opportunities. We describe these abstractions below.

First, we assume all temporary in-register pointers have been demoted to memory such that references of an in-register pointer q become references of $*p$. In-memory pointers are promoted back to registers following MemSafe's analysis.[3]

Memory Deallocation MemSafe models both explicit and automatic memory deallocation as in-memory pointer assignments. For example, the statement `free(*p)` is represented as `*p=invalid`, where *invalid* is a special untyped pointer constant that points to an "invalid" region of memory. We define the base and bound of this region to be the impossible address range $[1,0]$. Since the size of this block is -1, spatial checks involving the base and bound of *invalid* are guaranteed to report a violation. Therefore, temporal errors can be detected with spatial safety checks.

MemSafe inserts stores of *invalid* after calls to `free` and at the end of every function for each of its stack-allocated variables. MemSafe removes these stores after instrumenting the program with the required safety checks.

Pointer Stores MemSafe models in-memory pointer assignments (including those induced for *invalid*) as explicit register assignments using alias analysis and a ϕ-like SSA extension called the ϱ-function. For example, assume the statement `*p=ptr0` (s_0) is the only direct reaching definition of `ptr1=*p` (s_1). The statement `*q=ptr2` (s_2) may *indirectly* redefine ptr_1 if p and q may alias and control flow may reach statement s_1 from s_2. Therefore, we can model `ptr1=*p` as `ptr1=`ϱ`(ptr0,ptr2)`, meaning ptr_1 may equal ptr_0 or ptr_2 but only these two values. In this way, all indirect pointer assignments and object deallocations are represented as direct assignments of the pointers that are potentially modified.

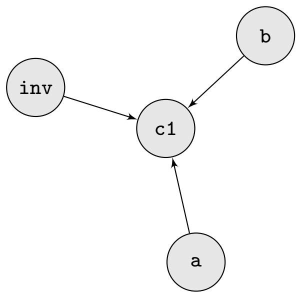
Figure 2(a) shows an example code fragment with our syntax extensions. At the call to `free` in line 7, we insert a store of *invalid* to indicate *p's* referent has been deallocated. Since p and q may alias, this store and the ones in lines 3 and 5 may define the pointer loaded in line 9. Therefore, pointers a, b and *invalid* are added to the ϱ-function for pointer c, meaning c may equal any of these values.

MemSafe inserts ϱ-functions after all pointer loads, and like the inserted stores of *invalid*, they are removed after inserting the required checks. Crucially, MemSafe does not insert ϱ-functions for loads of non-pointer values since they are

3. Demotion is required so that ϱ-functions will propagate *invalid* to pointers that may refer to to deallocated objects (discussed later).

```
1: int  *a,  *b,  *c;
2: int **p, **q;
   ▷ assume p and q may alias
   ...
3: *q = a;
4: if (condition) {
5:     *p = b;
6: } else {
7:     free(*p);
       *p = invalid;
8: }
9: c0 = *q
   c1 = ϱ(a, b, invalid);
```

(a) Example ϱ-extension



(b) DFPG

Figure 2: DFPG Construction. (a) A code fragment with the ϱ-extension is shown with (b) its corresponding DFPG. Numbered lines indicate original code.

not required for our analysis and could potentially lead to a large increase in code size.

The DFPG Utilizing the above abstractions, MemSafe creates a whole-program Data-Flow for Pointers Graph (DFPG). The DFPG is a def-use graph for all in-register pointers: if the definition of pointer p uses the value of pointer q, there is an edge from q to p. Since the ϱ-function encodes alias information as direct assignments, connected components represent disjoint alias sets. Figure 2(b) shows the DFPG for 2(a).

3.2. The Required Checks and Metadata

MemSafe utilizes a unique combination of object and pointer metadata to ensure both spatial and temporal memory safety.

Pointer Metadata For every pointer definition, MemSafe creates a 3-tuple $\langle base, bound, id\rangle$ of intermediate values. Together, *base* and *bound* indicate the sub-range of memory the pointer is permitted to access, and *id* associates the pointer with the object metadata of its referent (to be discussed).

Pointer Bounds Check (PBC) MemSafe inserts a Pointer Bounds Check before pointer dereferences (`*ptr`) that cannot be verified safe statically. `PBC` is defined by:

```
inline void PBC(ptr, base, bound, size) {
  if ((ptr < base) or (ptr + size > bound))
    signal_safety_violation();
}
```

For example, MemSafe utilizes the pointer metadata of *ptr* $\langle base, bound, id\rangle_{ptr}$ to ensure its safe dereference.

```
PBC(ptr, base_ptr, bound_ptr, sizeof(*ptr));
... *ptr ...
▷ some load or store operation involving ptr
```

Here, MemSafe signals a safety violation if `*ptr` will access a location outside the range specified by $[base_{ptr}, bound_{ptr})$. No costly database lookup is required for $base_{ptr}$ and $bound_{ptr}$ since they are uniquely named symbols in the inserted code.

This procedure is capable of not only ensuring complete spatial safety, but also temporal safety with a *single check*. Temporal safety is enforced because, had *ptr's* referent been deallocated, *ptr's* metadata would have been set to that of *invalid*, which will always cause the `PBC` to signal a safety violation. However, the `PBC` is insufficient for ensuring complete temporal safety. Since a nested object is deallocated using its base address, only object-level references are assigned *invalid*, and the metadata of sub-object references will not be updated. Thus, object metadata is required to associate pointers to *inner* objects with the base and bound of their *outer* objects. We introduce it below.

Object Metadata For every object allocation, MemSafe creates and assigns a unique *id* to the object and records a tuple $\langle base, bound\rangle$ for the allocated region in a global metadata facility. MemSafe removes entries for objects from the metadata facility when they are deallocated. The object metadata facility maps an object's *id* to its base and bound address and is defined by the partial function:

$$\begin{aligned} omd &: I \rightarrow O \\ id &\mapsto \langle base, bound\rangle_{id} \end{aligned}$$

where I is the set of *ids* and O is the set of object metadata. For convenience, we can also represent *omd* as the relation $\mathcal{R}_O$, where $(id, \langle base, bound\rangle_{id}) \in \mathcal{R}_O$.

Object Bounds Check (OBC) If a pointer may refer to a sub-object (determined by traversing the DFPG), MemSafe inserts an Object Bounds Check for temporal safety in addition to the PBC for spatial safety. OBC is defined by:

```
inline void OBC(ptr, id, size) {
  $\langle base, bound\rangle_{id} = omd(id)$
  if ((ptr < $base_{id}$) or (ptr + size > $bound_{id}$))
    signal_safety_violation();
}
```

Here, the OBC uses *ptr's id* to retrieve the object metadata of *ptr's* referent: $\langle base, bound\rangle_{id}$. Temporal safety is enforced because, had *ptr's* referent been deallocated, it would have been unmapped in the object metadata facility $\mathcal{R}_O$, causing $omd(id)$ to fail and MemSafe to signal a violation. If the detection of sub-object overflows is not a requirement, the PBC can be eliminated since the OBC also verifies *ptr* is within bounds of its outer object.

3.3. Metadata Propagation

Having presented the safety checks MemSafe inserts before pointer dereferences, in this section we describe MemSafe's translations for creating and propagating the required metadata. In doing so, we assume the program has already been transformed to include our extensions for deallocation and pointer stores (Section 3.1).

Allocation MemSafe creates entries in the object metadata facility as objects are allocated. Since the number and size of global objects are known statically, $\mathcal{R}_O$ is initialized with their metadata. For stack-allocated objects, MemSafe generates a new *id* and maps it to their base and bound address:

```
struct { ... int array[100]; ... } s;
  $\mathcal{R}_O = \mathcal{R}_O \cup \{(id \in I, \langle \&s, \&s + \mathbf{sizeof}(s)\rangle)\}$
```

For heap-allocated objects, MemSafe updates $\mathcal{R}_O$ as before but also sets the pointer metadata of the pointer returned by malloc since malloc creates a new pointer as well as an object. (If the pointer returned by malloc is NULL, its metadata is set to that of *invalid*):

```
p0 = malloc(size);
  $\langle base, bound, id\rangle_{p_0} = \begin{cases} \langle base, bound, id\rangle_{invalid} & \text{if } p_0 = null, \\ \langle p_0, p_0 + size, id \in I\rangle & \text{otherwise} \end{cases}$
  $\mathcal{R}_O = \mathcal{R}_O \cup \left\{\left(id_{p_0}, \langle base, bound\rangle_{id_{p_0}}\right)\right\}$
```

Deallocation When the referent of a pointer to a heap- or stack-allocated object is deallocated (e.g., with free), MemSafe removes its entry from $\mathcal{R}_O$, and sets the pointer's metadata to that of *invalid* ("\" denotes set difference):

```
p0 = *ptr
  ...
  $\mathcal{R}_O = \mathcal{R}_O \setminus \left\{\left(id_{p_0}, omd\left(id_{p_0}\right)\right)\right\}$
  $\langle base, bound, id\rangle_{p_0} = \langle base, bound, id\rangle_{invalid}$
free(p0);              ▷ MemSafe models deallocation as an in-
*ptr = invalid;          memory pointer assignment of "invalid"
```

Recall that if id_{p_0} has been unmapped in $\mathcal{R}_O$, $omd\left(id_{p_0}\right)$ will fail, causing MemSafe to signal a safety violation that, in this example, would indicate a double free error.

Address-of Like malloc, the address-of operator (&) creates a pointer to a new location, so MemSafe sets the pointer metadata of the newly created pointer:

```
struct { ... int array[100]; ... } s;
  ...
p0 = &(s.array[42]);
  $\langle base, bound, id\rangle_{p_0} = \langle id_s, \&s.array[0], \mathbf{sizeof}(s.array)\rangle$
```

This example demonstrates MemSafe's ability to detect sub-object overflows. Note that p_0 inherits the *id* of the outer object s, yet its *base* and *bound* are associated with *array*.

Pointer Arithmetic Pointers defined using simple assignments or in terms of arithmetic (e.g., array and structure indexing) inherit the pointer metadata of the original pointer:[4]

```
p1 = p0 + x;
  $\langle base, bound, id\rangle_{p_1} = \langle base, bound, id\rangle_{p_0}$
```

ϱ-functions Since the value produced by a ϱ-function is not known statically, MemSafe must "disambiguate" it for the returned pointer to inherit the correct metadata. Thus, MemSafe requires an additional metadata facility. Like the object metadata facility, the pointer metadata facility maps the address of an in-memory pointer to its pointer metadata and is defined by the partial function:

$$\begin{aligned} pmd &: M \rightarrow P \\ ptr &\mapsto \langle base, bound, id\rangle_{*ptr} \end{aligned}$$

where M is the set of addresses and P is the set of pointer metadata. For convenience, we also use the relation $\mathcal{R}_\mathcal{P}$ to represent *pmd*, where $(ptr, \langle base, bound, id\rangle_{*ptr}) \in \mathcal{R}_\mathcal{P}$. MemSafe retrieves metadata for the result of a ϱ-function:

```
p0 = *ptr1;                ▷ MemSafe models in-memory data-
p1 = ϱ(a0, b0, ...);        flow with the ϱ-function
  $\langle base, bound, id\rangle_{p_1} = pmd(ptr_1)$
```

For each argument of the ϱ-function (including *invalid*), MemSafe updates $\mathcal{R}_\mathcal{P}$ at the location it is stored to memory:

```
*ptr2 = a0;      ▷ $ptr_2$ may alias $ptr_1$ above
  $\mathcal{R}_\mathcal{P} = (\mathcal{R}_\mathcal{P} \setminus \{(ptr_2, pmd(ptr_2))\}) \cup \left\{\left(ptr_2, \langle base, bound, id\rangle_{a_0}\right)\right\}$
```

Pointer Casts Pointer casts and unions do not require any additional metadata propagation. The new pointer simply inherits the metadata of the original one. Pointers defined as NULL or as a cast from an integer to a pointer inherit the base and bound of *invalid*. Although this may result in false positives, we have observed this to be rare. For memory-mapped I/O, MemSafe requires a target's backend to specify the *base* and *bound* of valid address ranges.

Function Calls MemSafe requires an additional metadata facility in order to propagate pointer metadata for pointers

4. ϕ-functions are no different than other pointer assignments. For example, p = ϕ(a, b) assigns a to p at the end of the basic block producing a and b to p at the end of the basic block producing b.

passed as arguments to functions. The function metadata facility maps a formal argument to the pointer metadata of its actual argument and is defined by the partial function:

$$\begin{aligned} \mathit{fmd} &: A \rightarrow P \\ \mathit{formal} &\mapsto \langle \mathit{base}, \mathit{bound}, \mathit{id} \rangle_{\mathit{actual}} \end{aligned}$$

where A is the set of arguments and *formal* is a tuple $\langle \&f, i \rangle$ specifying the argument of function f at offset i. For convenience, we can also represent *fmd* as the relation $\mathcal{R}_{\mathcal{F}}$, where $(\mathit{formal}, \langle \mathit{base}, \mathit{bound}, \mathit{id} \rangle_{\mathit{actual}}) \in \mathcal{R}_{\mathcal{F}}$. MemSafe updates the function metadata facility before function calls:

$$\begin{aligned} \mathcal{R}_{\mathcal{F}} = (\mathcal{R}_{\mathcal{F}} &\setminus \{(\langle \&\mathit{func}, 0 \rangle, \mathit{fmd}(\langle \&\mathit{func}, 0 \rangle))\}) \\ &\cup \left\{ \left(\langle \&\mathit{func}, 0 \rangle, \langle \mathit{base}, \mathit{bound}, \mathit{id} \rangle_{p_0} \right) \right\} \end{aligned}$$

```
func(p0);
```

Similarly, in the body of *func*, MemSafe retrieves the metadata for p_0 with $\mathit{fmd}(\langle \&\mathit{func}, 0 \rangle)$. This approach is sufficient for handling function pointers, variable-argument functions, and precompiled libraries. For complete safety, libraries must be compiled with MemSafe's safety checks, but a safe application is capable of interfacing with unsafe libraries as well.

3.4. Optimizations of the Basic Approach

MemSafe utilizes the DFPG to perform several optimizations that reduce the cost of memory safety. Since the DFPG blurs the distinction between spatial and temporal errors, MemSafe's optimizations (described below) affect aspects of both.

Dominated Dereferences Multiple dereferences of the same pointer require safety checks only for the dereference that dominates the others.

Temporally Safe Dereferences If a pointer p is not reachable from *invalid* in the DFPG, then it must refer to a temporally valid object. Therefore, a dereference of p does not require an `OBC`. Recall that since MemSafe models temporal errors as spatial errors, the `PBC` ensures spatial *and* temporal safety for object-level references. However, if p may refer to a sub-object, its dereference requires the `OBC` in addition to the `PBC` to ensure temporal safety. *p's* potential referents are represented by the set of nodes in DFPG^{T} that are reachable from p and have no children. DFPG^{T} is the transpose of the DFPG (i.e., the DFPG with its edges reversed).

Non-incremental Dereferences If a pointer p must refer to a temporally valid object and is not reachable from a path in the DFPG representing pointer arithmetic, then p must refer to the base of a valid object or sub-object. If p is physically sub-typed [33] with each of its potential referents (i.e., their types are compatible for assignment), *p's* dereference does not require a `PBC`. If p is reachable from only constant increments (e.g., structure field accesses), MemSafe performs static checks instead of inserting the `PBC`.

Monotonically Addressed Ranges A pointer whose value is a monotonic function of a loop induction variable refers to a monotonically addressed range of memory. If the pointer is dereferenced in a loop having a computable number of iterations, MemSafe inserts a $\underline{\text{M}}$onotonically $\underline{\text{A}}$ddressed $\underline{\text{R}}$ange $\underline{\text{C}}$heck (`MARC`) in the loop preheader, and eliminates the check within the loop. For the sake of discussion, we assume loops have been transformed to have a single canonical induction variable that is initialized to zero and incremented by one. For example, MemSafe inserts the following code to ensure the safety of a pointer dereference within a loop:

```
MARC(ptr, base_ptr, bound_ptr, sizeof(*ptr), N);
for (i = 0; i < N; i++) {
    ... *(ptr + i) ... ;
    ▷ some store or load operation involving ptr
}
```

where `MARC` is the forcibly inlined procedure defined by:

```
inline void MARC(ptr, base, bound, size, trip_count) {
  ptr_max = ptr + trip_count;
  if ((ptr < base) || (ptr_max + size > bound))
    signal_safety_violation();
}
```

In this example, MemSafe signals a safety violation if the dereference `*(ptr+i)` will access a location outside the range specified by $\langle \mathit{base}, \mathit{bound}, \mathit{id} \rangle_{\mathit{ptr}}$ on any iteration of the loop and eliminates *ptr's* `PBC` within the loop. If *ptr* may refer to a sub-object, MemSafe hoists the `OBC` within the loop to the location before the `MARC` in the loop preheader.[5]

Unused Metadata Connected components in the DFPG represent disjoint alias sets. Therefore, if MemSafe eliminates checks for all pointers in a particular connected component, then their metadata is unused, and MemSafe eliminates it as well. This is more aggressive than dead code elimination since MemSafe not only removes unused metadata, but also code that *updates* the metadata facilities.

4. Implementation

Having described MemSafe's approach for inserting and optimizing checks and metadata, in this section we describe the prototype implementation of MemSafe and its limitations.

4.1. MemSafe's analysis and transformation

MemSafe is implemented in the LLVM [34] compiler infrastructure. LLVM's intermediate form is a low-level, typed SSA form that is both ISA- and language-independent. Thus, MemSafe's transformation is not specific to a particular architecture and can be used to ensure safety for languages other than C, but we have not tested this. Our implementation requires Andersen's alias analysis [35], but MemSafe can be used with any analysis compatible with LLVM.

MemSafe consists of several analysis and transformation passes that together implement the approach described in Section 3. These include passes to (1) insert assignments of *invalid*, (2) insert ϱ-functions with the aid of alias analysis, (3) construct the DFPG, and (4) insert optimized safety checks and metadata determined by traversing the DFPG.

4.2. Metadata Facilities

The facilities MemSafe requires for maintaining object and pointer metadata ($\mathcal{R}_{\mathcal{O}}$, $\mathcal{R}_{\mathcal{P}}$ and $\mathcal{R}_{\mathcal{F}}$) are implemented as dynamically resized hash tables for efficiency. Collisions are resolved using separate chaining and a simple move-to-front heuristic favoring reference locality. Hash functions are the modulo of the key with the size of the table, which becomes an efficient bitwise and operation by restricting table sizes to powers of two. The unique object *ids* used as keys for $\mathcal{R}_{\mathcal{O}}$ are generated using a global counter for simplicity.

5. Although MemSafe uses the `MARC` to essentially hoist checks out of loops, array bounds check elimination could be used with MemSafe to completely eliminate some of these checks.

Benchmark		Size		Detected All
Suite	Program	LOC	Derefs	
BugBench	099.go	29246	16632	yes
	129.compress	1934	232	yes
	bc-1.06	14288	2474	yes
	gzip-1.2.4	9076	1722	yes
	ncompress-4.2.4	1922	838	yes
	polymorph-0.4.0	716	65	yes

Table 2: Detected Violations. MemSafe's ability to detect memory errors is shown for BugBench [23] programs.

4.3. Limitations

Although MemSafe's approach is complete and compatible with most C programs, in practice MemSafe is not without limitations. MemSafe's most significant limitation is its use of whole-program analysis to remove unnecessary checks and metadata propagation. While whole-program analysis is used to enhance the effectiveness of MemSafe's optimizations, it negates the advantages of separate compilation and can be difficult in common build environments. However, the use of whole-program analysis is not required for MemSafe to ensure safety, and it can be turned off when not desirable.

5. Results

In this section, we describe the evaluation of our prototype implementation of MemSafe. We investigate (1) MemSafe's completeness by applying our approach to programs with known memory errors, (2) MemSafe's cost by comparing its runtime overhead to that of related methods, and (3) results related to MemSafe's static analysis. In doing so, we will demonstrate that MemSafe is compatible with a variety of C programs and does not require code modifications. Additionally, we will show that MemSafe's key contributions—modeling temporal errors as spatial errors, hybrid metadata, and our data-flow representation—are effective tools for reducing the cost of dynamically ensuring memory safety.

5.1. Effectiveness in Detecting Errors

We have used MemSafe to detect real errors in a variety of programs including the Apache HTTP server, GNU Coreutils, and programs from Bugbench [23]. BugBench is a collection of programs containing various documented software bugs that was expressly created to evaluate the effectiveness of error detection tools. Table 2 shows that MemSafe is capable of detecting all known memory errors in 6 programs from BugBench. We excluded programs that only contain errors *not* related to spatial or temporal safety (e.g., memory leaks). The size of each program is given in lines of code (LOC) and the number of static dereferences.

5.2. Runtime Performance

We measured MemSafe's runtime overhead on 30 programs from the Olden [24], PtrDist [10] and SPEC [36] suites. Programs from Olden and PtrDist are known for being allocation intensive, while those from SPEC are larger and more computation intensive. The programs were executed on a 3GHz Pentium 4 processor with 2GB of memory. Execution times were obtained by taking the lowest of three runtimes obtained using the `time` command. Due in part to LLVM's research-quality implementation of Andersen's analysis, our current implementation of MemSafe is not yet robust enough to compile all the SPEC programs, and we present results in this section for the subset that compiles correctly.

Overheads The "Runtime" and "Slowdown" columns of Table 3 summarize the runtime performance of MemSafe's fully optimized approach. MemSafe ensured complete spatial and temporal safety for all 30 programs with an average overhead of 87%. In general, we observed MemSafe's overhead to be comparable to that of CCured[21]: On the allocation intensive Olden benchmarks, MemSafe's overhead was 31% versus CCured's 30%, and on CCured's entire set of reported benchmarks, MemSafe overhead was 59% versus CCured's 80%. Not including *bc* (on which CCured's overhead was particularly high) reduces these to 54% and 30%, respectively. Due to CCured's need for manual code modifications, we did not obtain results for CCured on additional programs.

Additionally, MemSafe demonstrated a significant and consistent improvement over MSCC [9], the tool with the lowest overhead among all existing complete and automatic methods that detect both spatial and temporal errors. On the Olden benchmarks, MemSafe's average overhead was 1/4 that of MSCC (133%) and on the entire set of MSCC's reported benchmarks, MemSafe's average overhead (57%) was less than 1/2 that of MSCC (137%). We present comparisons with other methods in Table 1 of Section 1.

MemSafe is able to improve cost for the following reasons: (1) MemSafe's data-flow representation enables performance-enhancing optimizations that reduce overhead from 236% to 87% (explained later). (2) MemSafe's modeling of temporal errors as spatial errors, combined with a hybrid metadata representation, enables MemSafe to ensure temporal safety with only a 24% increase in the overhead of spatial safety alone (also explained later). In particular, MemSafe's large improvement versus MSCC on the Olden benchmarks is due to the fact that these programs postpone deallocating memory until terminating. Thus, MemSafe eliminates the propagation of *invalid* and all `OBC` checks. This is a common programming style when reallocation is not needed.

Optimizations Figure 3 shows that MemSafe's optimizations are effective tools for reducing cost. By observing the "Average" histogram, we see that MemSafe's optimizations reduced the average runtime overhead from 236% to 87%. Since the optimization for dominated dereferences (DDO) is minimally effective, we present it as a baseline. The optimization for temporally-safe dereferences (TDO) reduced overhead by 38%, and the optimization for non-incremental dereferences (NDO) reduced overhead by 23%. Combined with the optimization for unused metadata, which we include with both, NDO and TDO accounted for the greatest reduction in overhead. The optimization for monotonically addressed ranges (MRO) reduced overhead by 2%.

Though much of MemSafe's improvement stems from interprocedural optimizations, this is not by chance: By representing memory deallocation and pointer stores as direct assignments, MemSafe improves the effectiveness of whole-

Benchmark		Size		Runtime (s)		Slowdown			Checks (%)			Opts. (%)			DFPG	
Suite	Program	LOC	Derefs	Base	MemSafe	MemSafe	CCured	MSCC	PBC	OBC	MARC	DDO	TDO	NDO	Inv. (%)	ϱ/Store
Olden	bh	2073	284	4.68	5.34	1.14	1.44	2.82	4.93	0.00	1.76	27.11	2.11	43.31	0.00	7.19
	bisort	350	76	1.32	1.66	1.26	1.45	1.76	3.95	0.00	0.00	35.53	0.00	32.89	0.00	9.93
	em3d	688	187	5.14	7.92	1.54	1.87	1.79	15.51	0.00	3.21	7.49	0.00	44.92	0.00	1.84
	health	502	236	0.47	0.81	1.72	1.29	2.72	0.85	0.00	0.00	15.25	0.00	37.29	0.00	3.42
	mst	428	57	0.31	0.36	1.15	1.06	1.76	12.28	0.00	5.26	12.28	0.00	42.11	0.00	0.24
	perimeter	484	258	0.36	0.54	1.50	1.09	3.37	8.53	0.00	0.00	0.00	0.00	59.30	0.00	142.61
	power	622	285	4.09	4.62	1.13	1.07	1.22	3.51	0.00	0.00	22.46	1.75	25.61	0.00	0.00
	treeadd	245	26	0.38	0.46	1.20	1.10	3.23	18.42	0.00	0.00	0.00	0.00	47.37	0.00	5.33
	tsp	582	194	3.83	4.37	1.14	1.15	2.28	5.15	0.00	0.00	31.96	0.00	51.03	0.00	31.07
	average	*716*	*178*	*2.29*	*2.90*	*1.31*	*1.30*	*2.33*	*8.13*	*0.00*	*1.14*	*16.90*	*0.43*	*42.65*	*0.00*	*22.40*
PtrDist	anagram	650	113	1.57	2.87	1.83	1.43	–	25.66	0.00	0.00	21.24	11.50	23.89	0.00	0.29
	bc	7297	3927	1.34	3.18	2.37	9.91	–	13.09	3.41	1.12	32.85	0.03	13.01	8.99	43.62
	ft	1766	246	2.07	3.48	1.68	1.03	–	5.28	0.00	0.00	24.80	3.66	24.80	3.92	155.43
	ks	782	239	1.52	2.99	1.97	1.11	–	24.27	0.00	0.00	27.62	19.67	18.83	0.00	13.71
	yacr2	3986	1000	1.96	3.65	1.86	1.56	–	33.80	4.00	3.90	34.60	1.80	11.20	4.85	6.15
	average	*2896*	*1105*	*1.69*	*3.23*	*1.94*	*3.01*	–	*20.42*	*1.48*	*1.00*	*28.22*	*7.33*	*18.35*	*3.55*	*43.84*
SPEC'95	099.go	29246	16632	0.62	1.26	2.03	1.22	2.60	53.76	0.00	5.96	25.44	52.18	3.94	0.00	–
	129.compress	1934	232	0.01	0.02	1.60	1.17	1.85	12.07	0.00	4.74	40.95	5.17	11.21	0.00	4.13
	130.li	7597	4905	0.06	0.11	1.89	1.70	–	8.28	0.00	0.06	27.26	0.06	21.06	0.00	694.18
	147.vortex	67202	25135	0.00	0.00	–	–	–	6.40	0.72	0.04	34.36	0.00	5.32	13.18	1511.59
	average	*26495*	*11726*	*0.17*	*0.35*	*1.84*	*1.36*	–	*20.13*	*0.18*	*2.70*	*32.00*	*14.35*	*10.38*	*3.30*	*736.63*
SPEC'00	164.gzip	8605	1499	20.62	62.68	3.04	–	1.46	18.35	15.48	4.34	44.70	0.00	5.94	0.96	3.79
	175.vpr	17729	5386	8.35	14.70	1.76	–	3.53	20.92	3.08	2.32	22.08	0.09	14.17	7.07	14.52
	181.mcf	2412	534	11.22	20.20	1.80	–	2.85	7.12	1.50	1.31	23.60	2.06	38.01	9.61	25.74
	186.crafty	24975	7579	14.93	44.34	2.97	–	–	36.09	21.04	0.01	15.64	0.04	3.81	23.77	29.08
	255.vortex	67213	25134	3.95	8.18	2.07	–	–	6.40	0.72	0.04	34.37	0.00	5.32	13.18	1511.61
	256.bzip2	4649	1254	22.32	59.37	2.66	–	–	37.88	3.03	3.03	41.23	19.94	4.78	1.12	316.95
	300.twolf	20459	11741	7.53	14.01	1.86	–	–	14.10	3.67	0.25	20.82	0.00	24.21	9.39	3.71
	average	*20863*	*7590*	*12.70*	*31.92*	*2.31*	–	–	*20.12*	*6.93*	*1.61*	*28.92*	*3.16*	*13.75*	*9.30*	*54.72*
SPEC'06	401.bzip2	8293	4013	6.23	13.58	2.18	–	–	14.40	2.09	1.20	12.09	0.07	4.73	27.83	440.48
	445.gobmk	197215	27614	0.29	0.55	1.90	–	–	37.31	23.54	1.96	19.49	0.00	13.87	12.70	209.98
	456.hmmr	35992	7582	7.82	16.89	2.16	–	–	23.79	1.25	1.58	18.40	0.00	19.11	14.21	108.66
	458.sjeng	13847	5832	10.13	28.36	2.80	–	–	24.98	18.66	0.22	28.21	0.19	6.28	18.37	80.29
	473.astar	5842	1873	0.00	0.00	–	–	–	7.90	1.17	0.32	19.38	0.00	14.95	18.91	39.38
	average	*52238*	*9383*	*4.89*	*11.88*	*2.26*	–	–	*21.68*	*9.34*	*1.06*	*19.51*	*0.05*	*11.79*	*18.40*	*133.89*
Average		**18394**	**5136**	**4.77**	**10.88**	**1.87**	**–**	**–**	**16.83**	**3.45**	**1.42**	**24.04**	**4.01**	**22.41**	**6.48**	**177.68**

Table 3: Summary of Results. Program size is measured in lines of code and the number of static dereferences. Slowdown is shown in comparison with CCured [21] and MSCC [9] where results are available. The static number of required checks and optimizations are measured as a percentage of dereferences. The DFPG is measured by the percentage of nodes reachable from *invalid* and the average number of ϱ-nodes modifiable by each pointer store.

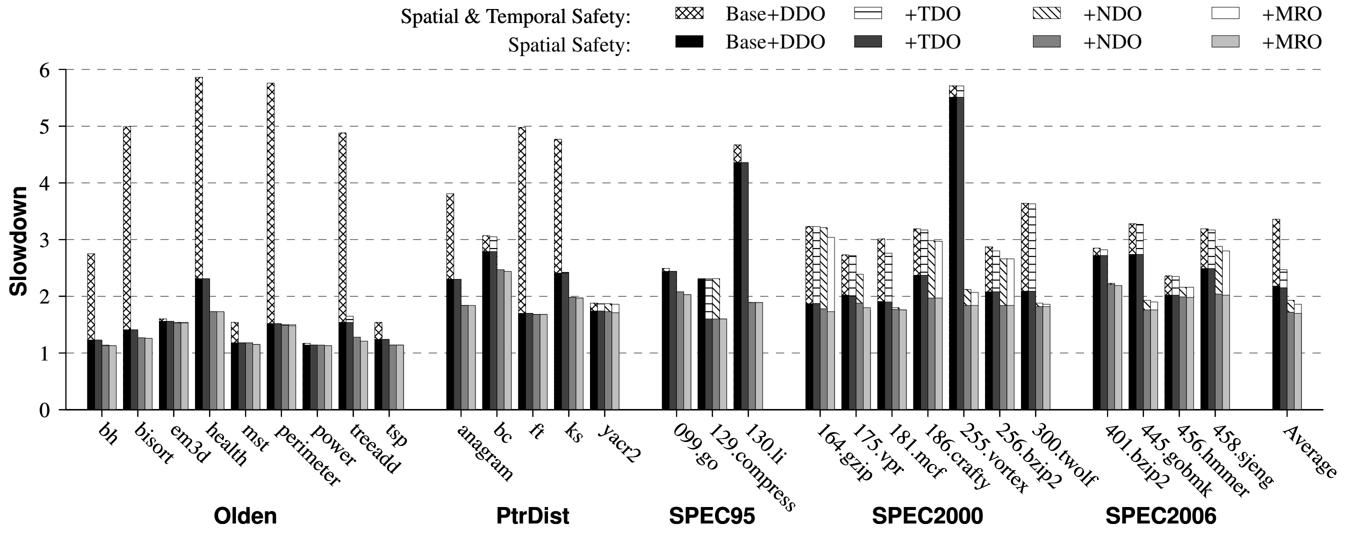


Figure 3: Optimizations. Slowdown is shown for spatial and temporal and spatial-only safety. Optimizations include dominated dereferences (DDO), temporally-safe dereferences (TDO), non-incremental dereferences (NDO), and monotonically addressed ranges (MRO).

program optimization. Thus, MemSafe's overheads are lower than those of existing methods that cannot benefit in this way.

Cost of Temporal Safety Figure 3 also quantifies the additional cost required to ensure temporal safety. Observing the last bar in the "Average" histogram, we see that MemSafe's overhead for spatial and temporal safety (87%) is comparable to that of just spatial safety (70%). Thus, MemSafe ensured complete temporal safety with a modest 24% increase in the overhead of spatial safety alone.

Comparing the additional cost of ensuring temporal safety with MemSafe versus that of MSCC on MSCC's set of reported benchmarks, MemSafe's overhead for spatial and temporal safety (57%) is a 19% increase in that of spatial safety (48%), whereas MSCC's overhead for spatial and temporal safety (137%) is an 83% increase in that of spatial safety (75%) [9]. While the overheads for spatial safety are comparable, this result demonstrates that by modeling temporal errors as spatial errors, MemSafe's optimizations are effective tools for reducing the additional cost of temporal safety.

5.3. Static Analysis

The "Checks," "Opts.," and "DFPG" columns of Table 3 describe results related to MemSafe's static analysis. First, we show the static number of required and optimized checks as a percentage of total dereferences. Second, we summarize the DFPG by the percentage of nodes reachable from *invalid*, and $\varrho/store$. The former indicates the portion of pointers that may refer to temporally invalid objects, and the latter indicates the average number of loaded memory locations that each pointer store may potentially modify.

6. Related Work

We have already described many methods related to memory safety in Section 2. In this section, we will not repeat that content but present additional details for particular methods.

Complete Safety Several methods are capable of detecting both spatial and temporal errors. Purify [19] operates on binaries, but only ensures the safety of heap-allocated objects. Yong and Horwitz [20] present a similar approach and improve its cost with static analysis, but this method only checks store operations. Safe C [10] ensures complete safety but is incompatible due to its use of fat-pointers. Patil and Fischer [11] address these issues by maintaining disjoint metadata and performing checks in a separate "shadow process," but this requires an additional CPU. CCured [21] utilizes a type system to eliminate checks for safe pointers and reduce metadata bookkeeping. However, CCured's use of fat-pointers causes compatibility issues, and some programs require code modifications to lower cost. MSCC [9] is highly compatible and complete but is unable to handle some downcasts. Fail-Safe C [22] maintains complete compatibility with ANSI C but incurs significant runtime overhead. Finally, Clause et al. [37] describe an efficient technique for detecting memory errors, but it requires custom hardware.

Spatial Safety Methods that primarily detect bounds violations are numerous. Notable is the work by Jones and Kelly [1] since it maintains compatibility with precompiled code. However, this method has high overhead and results in false positives. Ruwase and Lam [5] extend this method to track out-of-bounds pointers to avoid false positives. Additionally, Dhurjati and Adve [6] utilize Automatic Pool Allocation [38, ch.5] to improve cost, and Akritidis et al. [7] constrain the size and alignment of allocated regions to further improve cost. However, these methods do not detect temporal violations and are unable to detect sub-object overflows.

HardBound [39] is a hardware-assisted approach for ensuring spatial safety with low overhead. This method encodes fat-pointers in a special "shadow space" and provides architectural support for checking and propagating metadata. SoftBound [4] is a related technique that records pointer metadata in disjoint data structures similar to our representation. However, while these methods ensure complete spatial safety, they do not ensure temporal safety, and HardBound requires custom hardware to achieve low overhead.

Temporal Safety Few methods are designed primarily for the detection of temporal violations. Dhurjati and Adve [40] describe a technique based the Electric Fence [41] `malloc` debugger: Their system assigns a unique virtual page to every dynamically allocated object and relies on hardware page protection to detect dangling pointer dereferences. This approach is improved with Automatic Pool Allocation [38, ch. 5] and a customized address mapping. However, this method does not detect spatial violations and only detects temporal violations of heap objects.

Other Approaches Several methods use software checks to enforce various security-related policies. Abadi et al. [25] describe a technique to prevent software attacks by enforcing control-flow integrity. Similarly, Castro et al. [26] enforce data-flow integrity with an analysis based on reaching definitions, and WIT [27] enforces write-integrity by ensuring each write operation accesses an object from a static set of legally modifiable objects. Although these techniques are capable of preventing many memory access violations, they do not ensure complete spatial and temporal safety.

DieHard [42] is a memory allocator capable of preventing many heap-related errors. It uses random object placement within a larger-than-normal heap to prevent invalid frees and probabilistically avoid heap buffer overflows. However, this method cannot ensure complete spatial and temporal safety.

7. Conclusion

MemSafe is a compiler analysis and transformation for ensuring the spatial and temporal memory safety of C at runtime. MemSafe builds upon previous work to enable its completeness and compatibility, capturing the most salient features of object and pointer metadata in a new hybrid representation. To improve cost, MemSafe exploits a novel mechanism for modeling temporal errors as spatial errors, and a new dataflow representation that simplifies optimizations for removing unneeded checks and metadata.

We verified MemSafe's ability to detect real errors with lower overhead than previous methods. MemSafe detected all documented memory errors in 6 programs with known bugs. Additionally, it ensured complete safety with an average overhead of 87% on 30 programs widely-used in evaluating error detection tools. Finally, MemSafe's average runtime overhead on the Olden benchmarks was 1/4 that of the tool with the lowest overhead among all existing complete and automatic methods that detect both spatial and temporal errors.

References

[1] R. W. M. Jones and P. H. J. Kelly, "Backwards-compatible bounds checking for arrays and pointers in C programs," in *Proceedings of the 3rd International Workshop on Automatic Debugging*, 1997, pp. 13–26.

[2] C. Dahn and S. Mancoridis, "Using program transformation to secure c programs against buffer overflows," in *Proceedings of the 10th Working Conference on Reverse Engineering*, 2003, p. 323.

[3] L. Wang, J. R. Cordy, and T. R. Dean, "Enhancing security using legality assertions," in *Proceedings of the 12th Working Conference on Reverse Engineering*, 2005, pp. 35–44.

[4] 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 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009, pp. 245–258.

[5] O. Ruwase and M. S. Lam, "A practical dynamic buffer overflow detector," in *Proceedings of the Network and Distributed System Security Symposium*, 2004, pp. 159–169.

[6] 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*, 2006, pp. 162–171.

[7] P. Akritidis, M. Costa, M. Castro, and S. Hand, "Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors," in *Proceedings of the 18th USENIX Security Symposium*, 2009.

[8] *US-CERT Vulnerability Notes Database*, U.S. Computer Emergency Readiness Team, http://www.kb.cert.org/vuls/.

[9] 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*, 2004, pp. 117–126.

[10] T. M. Austin, S. E. Breach, and G. S. Sohi, "Efficient detection of all pointer and array access errors," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1994, pp. 290–301.

[11] H. Patil and C. Fischer, "Low-cost, concurrent checking of pointer and array accesses in C programs," *Software: Practice & Experience*, vol. 27, no. 1, pp. 87–110, 1997.

[12] D. B. Wortman, "On legality assertions in euclid," *IEEE Transactions on Software Engineering*, vol. 5, no. 4, pp. 359–367, 1979.

[13] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang, "Cyclone: A safe dialect of C," in *Proceedings of the USENIX Annual Technical Conference*, 2002, pp. 275–288.

[14] J. P. Condit, "Dependent types for safe systems software," Ph.D. dissertation, University of California, Berkeley, 2007.

[15] T. Ball and S. K. Rajamani, "The SLAM project: Debugging system software via static analysis," in *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002, pp. 1–3.

[16] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The software model checker BLAST: Applications to software engineering," *International Journal on Software Tools for Technology Transfer*, vol. 9, no. 5-6, pp. 505–525, 2007.

[17] M. Das, S. Lerner, and M. Seigle, "ESP: Path-sensitive program verification in polynomial time," in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, 2002, pp. 57–68.

[18] J. Criswell, A. Lenharth, D. Dhurjati, and V. Adve, "Secure virtual architecture: A safe execution environment for commodity operating systems," in *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, 2007, pp. 351–366.

[19] R. Hastings and B. Joyce, "Purify: A tool for detecting memory leaks and access errors in C and C++ programs," in *Proceedings of the USENIX Winter Technical Conference*, 1992, pp. 125–138.

[20] S. H. Yong and S. Horwitz, "Protecting C programs from attacks via invalid pointer dereferences," in *Proceedings of the 9th European Software Engineering Conference Held Jointly With the 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2003, pp. 307–316.

[21] 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*, vol. 27, no. 3, pp. 477–526, 2005.

[22] Y. Oiwa, "Implementation of the memory-safe full ANSI-C compiler," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009, pp. 259–269.

[23] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou, "BugBench: Benchmarks for evaluating bug detection tools," in *Workshop on the Evaluation of Software Defect Detection Tools*, 2005.

[24] A. Rogers, M. C. Carlisle, J. H. Reppy, and L. J. Hendren, "Supporting dynamic data structures on distributed-memory machines," *ACM Transactions on Programming Languages and Systems*, vol. 17, no. 2, pp. 233–263, 1995.

[25] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security*, 2005, pp. 340–353.

[26] 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*, 2006, pp. 147–160.

[27] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing memory error exploits with WIT," in *IEEE Symposium on Security and Privacy*, 2008, pp. 263–277.

[28] D. Dhurjati, S. Kowshik, and V. Adve, "SAFECode: Enforcing alias analysis for weakly typed languages," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2006, pp. 144–157.

[29] H.-J. Boehm and M. Weiser, "Garbage collection in an uncooperative environment," *Software: Practice & Experience*, vol. 18, no. 9, pp. 807–820, 1988.

[30] D. F. Bacon, P. Cheng, and D. Grove, "Garbage collection for embedded systems," in *Proceedings of the 4th ACM International conference on Embedded Software*, 2004, pp. 125–136.

[31] B. Zorn, "The measured cost of conservative garbage collection," *Software: Practice & Experience*, vol. 23, no. 7, pp. 733–756, 1993.

[32] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *ACM Transactions on Programming Languages and Systems*, vol. 13, no. 4, pp. 451–490, 1991.

[33] S. Chandra and T. Reps, "Physical type checking for C," in *Proceedings of the 1999 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 1999, pp. 66–75.

[34] 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 (GCO)*, 2004, pp. 75–87.

[35] L. O. Andersen, "Program analysis and specialization for the C programming language," Ph.D. dissertation, University of Copenhagen, 1994.

[36] *SPEC CPU Benchmarks*, Standard Performance Evaluation Corporation, http://www.spec.org/.

[37] J. Clause, I. Doudalis, A. Orso, and M. Prvulovic, "Effective memory protection using dynamic tainting," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, 2007, pp. 284–292.

[38] C. Lattner, "Macroscopic datastructure analysis and optimization," Ph.D. dissertation, University of Illinois, Urbana-Champaign, 2005.

[39] 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 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008, pp. 103–114.

[40] 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*, 2006, pp. 269–280.

[41] B. Perens, "Electric fence malloc debugger," http://perens.com/FreeSoftware/ElectricFence/.

[42] E. D. Berger and B. G. Zorn, "DieHard: Probabilistic memory safety for unsafe languages," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2006, pp. 158–168.