SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2011; **00**:1–34
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

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

Matthew S. Simpson[*] and Rajeev K. Barua

Department of Electrical & Computer Engineering, University of Maryland, College Park, MD 20742-3256, USA

SUMMARY

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. This paper presents 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 data-flow 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 six programs with known violations as well as two large and widely-used open source applications. Finally, MemSafe ensures complete safety with an average overhead of 88% on 30 programs commonly used for evaluating the performance of error detection tools. Copyright © 2011 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: Memory Safety, Reliability, Verification, Programming Languages

1. INTRODUCTION

Use of the C programing language remains common despite the well-known memory errors it allows. The features that make C a desirable language 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 can frequently go undetected long after they occur, resulting in data corruption and incorrect results while making software testing and debugging a particularly onerous task.

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 buffer overflow errors have been understood as early as 1972 [2], 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 (US-CERT), 67 (29%) of the 228 vulnerability notes released in 2008–2009 were due to buffer overflow errors alone [3].

[†]An earlier version [1] of this paper was presented at the *10th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Timişoara, Romania, September 12–13, 2010.
[*]Correspondence to: 1431 A. V. Williams Building, Department of Electrical & Computer Engineering, University of Maryland, College Park, MD 20742-3256, USA. E-mail: simpsom@umd.edu

Several safety methods [4–7] 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.[2] A temporal error is a violation caused by using a pointer whose referent has been deallocated (e.g. by calling the `free` standard library function) and is no longer a valid memory object. The most well-known temporal violations include dereferencing "dangling" pointers to dynamically allocated memory and attempting to deallocate a pointer more than once. However, dereferencing pointers to automatically allocated memory (i.e., 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, ensure memory safety through a combination of syntax restrictions and runtime checks, and are widely-used when security is a major concern. Others, like Cyclone [8] and Deputy [9], preserve many of the low-level features of C, but require additional programmer annotations to assist in ensuring safety. Although the use of these languages may be ideal for safety-critical environments, the reality is that many of today's applications—including operating systems, web browsers, and database management systems—are still typically implemented in C or C++ because of its efficiency, predictability, and access to low-level features. This trend will likely continue into the future.

As an alternative to safe languages, sophisticated static analysis methods for C [10–14] can be used alone, or in conjunction with other systems, to ensure the partial absence of spatial and temporal errors statically. While these techniques are invaluable for software verification and debugging, they can rarely prove the absence of all memory errors and often require a significant amount of verification time due to the precision of their analyses.

A growing number of methods rely primarily on inserted runtime checks to detect memory access violations dynamically. However, the methods capable of detecting both spatial and temporal memory safety violations [4–6, 15–23] generally suffer from one or more practical drawbacks that have thus far limited their widespread adoption. These drawbacks can be summarized by the following qualities.

- **Completeness.** Methods that associate *metadata* (the base and bound information required for runtime checks) with objects [15–21]—rather than the pointers to these objects—generally do not detect two kinds of memory errors. First, because C supports the allocation of nested objects (e.g., an array of structures), spatial errors involving sub-object overflows are not detected since inner objects share metadata with the outer object. Second, if the system allocates an object to a previously deallocated location, temporal errors are not detected since dangling pointers to the deallocated object may still refer to a location within bounds of the newly allocated object.

- **Compatibility.** The use of alternate pointer representations, such as multi-word "fat-pointers" [4, 22] to store metadata raises compatibility concerns. Inline metadata breaks many legacy programs—and requires implicit language restrictions for new ones—because it changes the memory layout of pointers. For example, since their data types are the same size, programmers can cast pointers to integers to compute certain addresses. However, since fat-pointers alter memory layout, casting a single-word integer to a multi-word pointer, or casting a fat-pointer of one type to a pointer of another type, is no longer valid and can result in data corruption. Inline metadata also breaks the calling convention of external libraries whose parameters or return types involve pointers.

- **Code Modifications.** Some methods [22] require non-trivial source code modifications to avoid the above compatibility issues or to prevent an explosion in runtime. A common example is for a programmer to write "wrapper functions" that remove inline metadata in order to interface with external libraries.

[2]A *manufactured* pointer is a pointer created by means other than explicit memory allocation (e.g., by calling the `malloc` standard library function) or taking the address of a variable using the address-of operator (&). Type-casting an integral type to a pointer type is a common example. The various memory safety violations are discussed in detail in Section 2.

- **Cost.** Methods capable of detecting both spatial and temporal errors often suffer from high performance overheads [4–6, 23]. This is commonly due to the cost of maintaining the metadata required for ensuring spatial safety and the use of conservative garbage collection for ensuring temporal safety. High runtime overhead can make a method prohibitively expensive for deployment and can slow the development process when it is used for testing, especially if a program is to be executed many times to increase coverage.

This paper introduces MemSafe [1], a method for ensuring both the *spatial* and *temporal* memory safety of C programs at runtime. 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 runtime *cost* than other complete and automatic methods achieving the same level of safety. MemSafe makes the following contributions for lowering the runtime cost of dynamically ensuring memory safety:

- MemSafe uniformly handles all memory violations by modeling temporal errors as spatial errors. Therefore, the use of separate mechanisms for detecting temporal errors (e.g. garbage collection or explicit checks for temporal safety [4–6, 21]) is no longer required.
- MemSafe captures the most salient features of object and pointer metadata in a hybrid spatial metadata representation. MemSafe's handling of pointer metadata is similar to that of SoftBound [7], a previous technique for detecting spatial errors, and ensures MemSafe's completeness and compatibility. However, MemSafe's additional use of object metadata creates a novel synergy with pointer metadata that allows 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 checks for all dereferences and the expensive propagation of metadata at every pointer assignment [4–6, 23], MemSafe eliminates redundant checks and the propagation of unused metadata. This capability is further enhanced with whole-program analysis.

In order to achieve the above contributions, MemSafe exploits several key insights related to the flow of pointer values in a program. The following program behavior models form the foundation of MemSafe's approach.

1. *Memory deallocation can be modeled as an assignment.* For example, the statement `free(p)` can be represented by the statement `p=invalid`, where *invalid* is a special untyped pointer to a temporally "invalid" range of memory.

   This insight is useful because it enables spatial safety mechanisms to be reused to ensure temporal safety. In order to detect spatial safety violations, existing methods insert before pointer dereferences runtime checks that determine whether the pointers refer to a location within the base and bound addresses of their referents. If a dereferenced pointer refers to a location outside the region of memory occupied by its referent, a spatial safety violation is signaled. By assigning pointers to deallocated memory to be equal to the *invalid* pointer, they inherit the base and bound addresses of the "invalid" region of memory. If the base and bound addresses of this region are defined such that they represent some impossible address range (e.g., a block with a negative size), any legal pointer must refer to a location outside this range. Thus, dereferences of dangling pointers and multiple deallocation attempts can then be detected with the inserted checks for spatial safety.

2. *Indirect pointer assignments can be modeled as explicit assignments.* Statements of the form `ptr1=*p`, where both ptr_1 and p are pointers, make low-cost memory safety difficult to achieve since ptr_1*'s* set of potential referents is not known statically. Alias analysis can be used to narrow this set, and MemSafe makes the results of this analysis explicit in the program's Static Single Assignment (SSA) [24] form by using a new ϕ-like construct called the ϱ-function.

Approach	Complete	Compatible	No Code Modifications	Whole Program	Slowdown
Purify [20]	no	yes	yes	yes	148.44*
Patil, Fischer [5]	yes	yes	yes	no	6.38†
Safe C [4]	yes	no	yes	no	4.88†
Fail-Safe C [23]	yes	yes	yes	no	4.64†
MSCC [6]	yes	yes	yes	no	2.33
Yong, Horwitz [21]	no	yes	yes	no	1.37‡
CCured [22]	yes	no	no	yes	1.30
MemSafe	**yes**	**yes**	**yes**	**yes**	**1.29**

Table I. **Related work.** A comparison of methods providing both spatial and temporal memory safety is given. Slowdown is computed as the ratio of the execution time of the instrumented program to that of the original program. Slowdown is reported for the Olden benchmarks [28] 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.

For example, assume the statement `s0:*p=ptr0` is the only direct reaching definition of a pointer defined as `s1:ptr1=*p`. The statement `s2:*q=ptr2` may *indirectly* redefine ptr_1 if p and q may alias and control-flow may reach statement s_1 from s_2. Therefore, MemSafe models the statement `ptr1=*p` as `ptr1=`ϱ`(ptr0,ptr2)`, meaning the value of ptr_1 may equal that of ptr_0 or ptr_2 but only these two values.

This insight is useful because it enables MemSafe to construct a convenient data-flow graph that codifies both direct and indirect pointer assignments—in addition to memory deallocation with the insight above—as simple definition and use relationships. Thus, this representation greatly simplifies optimizations for reducing the cost of achieving memory safety.

A prototype implementation of MemSafe has been evaluated in terms of its completeness and runtime cost. MemSafe was able to successfully detect known memory violations in multiple versions of the Apache HTTP server [25] and the GNU Core Utilities [26] software package. Additionally, MemSafe detected all previously reported memory errors in six programs from the BugBench [27] benchmark suite. In terms of cost, MemSafe's average overhead was 88% on 30 large programs widely-used in evaluating error detection tools. Finally, as evidence of its compatibility, MemSafe compiled each of the above programs without requiring any code modifications or programmer intervention.

Table I summarizes previous software approaches for ensuring both spatial and temporal safety.[3] Each method is evaluated on its completeness, compatibility, lack of code modifications, use of whole-program analysis, and runtime cost. For consistency, slowdown is reported for the Olden benchmarks [28] 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 MemSafe's novel contributions based on the above insights.

Since MemSafe's performance overheads cannot necessarily be considered "low," MemSafe is deployable in systems whose primary concern is memory safety. In practice, it has been observed that many runtime checks can be avoided with MemSafe's simple optimizations, and for safety-critical applications, MemSafe's moderate runtime overheads can be an acceptable trade-off compared to redesigning systems in a safe language. However, for performance-critical applications, MemSafe is primarily useful as a dynamic bug detection tool.

[3]Other methods (e.g, CIT [29], DFI [30], WIT [31], SoftBound [7], SafeCode [32], "baggy" bounds checking [33], etc.) are excluded from Table I since they either are (1) not software-only mechanisms for detecting memory errors or (2) do not aim to ensure complete spatial and temporal safety. However, these methods are discussed in detail in Section 7.

Memory Safety	Violation	Example
Spatial Safety	Bounds violation	`struct { ... int array[100]; ... } s;`
		`int *p;`
		`...`
		`p = &(s.array[101]);`
		`... *p ...` ▷ *bounds violation*
	Uninitialized pointer	`int *p;`
		`...`
		`... *p ...` ▷ *uninitialized pointer dereference*
	`NULL` pointer	`int *p;`
		`...`
		`p = NULL;`
		`... *p ...` ▷ *null pointer dereference*
	Manufactured pointer	`int *p;`
		`...`
		`p = (int*) 42;`
		`... *p ...` ▷ *manufactured pointer dereference*
Temporal Safety	Dangling stack pointer	`int *p;`
		`...`
		`void f() {`
		`  int x;`
		`  ...`
		`  p = &x;`
		`}`
		`...`
		`void g() {`
		`  f();`
		`  ... *p ...` ▷ *dangling stack pointer dereference*
		`}`
	Dangling heap pointer	`int *p, *q;`
		`...`
		`p = (int*) malloc(10*sizeof(int));`
		`q = p;`
		`...`
		`free(p);`
		`... *q ...` ▷ *dangling heap pointer dereference*
	Multiple deallocations	`int *p, *q;`
		`...`
		`p = (int*) malloc(10*sizeof(int));`
		`q = p;`
		`...`
		`free(p);`
		`free(q);` ▷ *multiple deallocations*

Table II. Memory safety violations. Example code fragments demonstrating memory safety violations are presented grouped by whether they affect aspects of spatial or temporal safety.

2. BACKGROUND

Memory safety violations can be divided into two categories: violations of *spatial safety* and violations of *temporal safety*. A spatial safety violation is an error in which a pointer is used to access the data at a location in memory that is outside the bounds of an allocated object. The error is “spatial” in the sense that the dereferenced pointer refers to an incorrect location in memory. A temporal safety violation is an error in which a pointer is used in an attempt to access or deallocate an object that has already been deallocated. The violation is “temporal” in the sense that the pointer use occurs at an invalid instance during the execution of the program (i.e., after the object to which it refers has been deallocated). Table II lists spatial and temporal memory safety violations that MemSafe detects and gives examples of each.

As evidence of the significance of these memory safety violations, the instrumentation of C programs to ensure memory safety remains an actively researched topic. The remainder of this section reviews previous approaches for detecting some or all spatial and temporal safety violations, primarily focusing on the prior works' use of metadata. In the context of enforcing memory safety, *metadata* refers to the creation of additional data for describing the spatial or temporal properties of an object or pointer. For example, metadata often consists of the base and bound addresses that indicate the valid address range to which a pointer may refer.

2.1. Spatial safety

The goal of spatial safety is to ensure that every memory access occurs within the bounds of a known object. Spatial safety is typically enforced by inserting runtime checks before pointer dereferences. Alternatively, checking for bounds violations after pointer arithmetic is also possible [15–17, 33], 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 spatial 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 addresses 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 the propagation of metadata at every pointer assignment (see the discussion of pointer metadata below), and compatibility, since it does not change the layout of objects in memory or prohibit the use of pre-compiled libraries. Prominent methods employing this strategy include the work by Jones and Kelly [15], Ruwase and Lam [16], Dhurjati and Adve [17], Akritidis et al. [33], SafeCode [32] and SVA [19].

However, the use of object metadata as means of enforcing spatial safety results in several drawbacks. First, this approach prevents complete spatial safety. Since nested objects (e.g., an array of structures) are assigned a base and bound address that spans 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 operation for retrieving metadata from the object database. Dhurjati and Adve [17] improve the runtime cost associated with this lookup operation by partitioning the object database using Automatic Pool Allocation [34], and Akritidis et al. [33] improve runtime by constraining the size and alignment of allocated objects. However, these methods do not detect sub-object overflows or temporal errors.

Figure 1 depicts the utility of object metadata (shown in red) in enforcing memory safety. In order to ensure memory safety, complete spatial and temporal safety must be enforced. Since all pointer dereferences are either object-level references or sub-object references, it follows that all object and sub-object references must be both spatially and temporally safe for a program to be memory safe. However, object-level base and bound information is only useful in enforcing object-level spatial safety, since sub-objects must share metadata with their corresponding outer objects.[4] Figure 1 will be referenced again when describing the remaining prior enforcement strategies.

Pointer metadata An alternative to using object metadata for enforcing spatial safety is to associate metadata with individual pointers. When a new pointer is created (i.e., with `malloc` or the address-of operator), its metadata is initialized to be the base and bound address of its referent, and when a pointer definition 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 assigning each pointer a unique base and bound address.

[4]Some methods [15–17, 19, 32, 33] are capable of using object metadata to detect some, but not all, temporal safety violations. However, if an object is deallocated and its space is reallocated for use by another object, dangling pointer dereferences to the original object will not be detected because they are within bounds of the new object. Thus, these methods are incapable of enforcing either complete object-level or sub-object temporal safety.

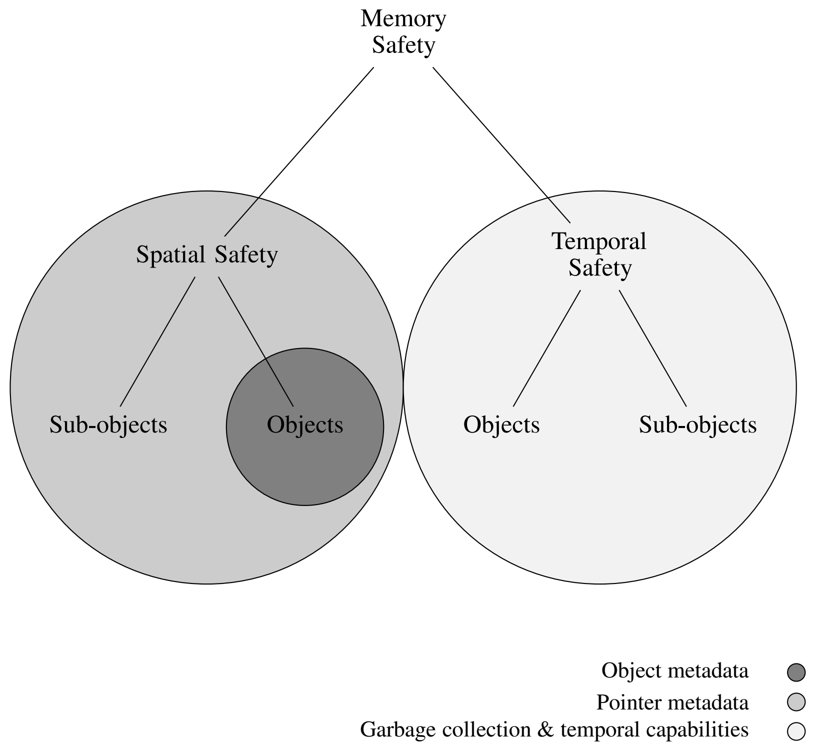


Figure 1. Prior enforcement methods. The use of spatial metadata, garbage collection, and temporal capabilities is shown for previous methods of enforcing memory safety.

Pointer metadata is commonly implemented using multi-word blocks of memory, called "fat-pointers," that record the required base and bound information inline with pointers. Each pointer in a program essentially becomes a `struct` containing three fields: the original pointer value and the base and bound address of its referent. Prominent methods employing this strategy include Safe C [4], Fail-Safe C [23] and CCured [22]. However, the use of inline metadata is not always compatible and breaks many programs. Since a pointer's size is no longer equal to 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.

Several pointer-based methods have developed approaches that avoid some of the compatibility issues of fat-pointers. CCured [22], MSCC [6], and Patil and Fischer [5] record metadata in disjoint structures that mirror the shape of the underlying data, but maintaining this representation increases runtime. Fail-Safe C [23] combines fat-pointers with fat-integers and virtual structure offsets, but this too increases cost. Finally, Softbound [7] maintains metadata for in-memory pointers in an efficient global lookup table, but this method only detects spatial violations.

Another disadvantage of the use of pointer metadata as a means of enforcing spatial safety is its runtime cost. While it avoids the need for expensive database lookups operations, metadata must instead be propagated at every pointer assignment. CCured [22] reduces metadata propagation by using a type system to infer pointer usage. CCured classifies pointers as `SAFE`, `SEQ` and `WILD` and optimizes the inserted checks and code for propagating metadata for each pointer kind. However, CCured requires manual code modifications to avoid the expensive bookkeeping of `WILD` pointers and to correct the compatibility issues of fat-pointers.

Figure 1 depicts the utility of pointer metadata (shown in blue) in enforcing memory safety. Since individual pointers can be associated with a unique base and bound address, pointer metadata can be used to enforce complete object-level and sub-object spatial safety. However, prior methods are not capable of utilizing pointer metadata in enforcing temporal safety.

MemSafe's approach MemSafe's use of metadata as a means for ensuring spatial safety avoids the drawbacks of the above approaches. 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-based metadata in an approach similar to that of SoftBound. However, to lower runtime cost, MemSafe models temporal errors as spatial errors and propagates pointer-based metadata only when it is needed for performing runtime checks. Additionally, MemSafe maintains some object-based metadata in a global database but performs lookup operations only when MemSafe's pointer-based metadata is insufficient for ensuring temporal safety.

2.2. *Temporal safety*

The goal of enforcing temporal safety is to ensure that every memory accesses refers to an object that has not been deallocated. As summarized in Table II, temporal safety violations occur when dereferencing pointers to stack objects, if the function in which they were defined has exited, and when dereferencing pointers to heap objects, if the object to which they refer has been deallocated with `free`. 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 commonly ignore calls to the `free` function and replace calls to `malloc` with the Boehm-Demers-Weiser conservative garbage collector [35]. To prevent dangling pointers to stack objects, local variables can be "heapified" and replaced with dynamically allocated objects that are managed by the garbage collector. This is the approach taken by CCured [22] and Fail-Safe C [23].

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 [36], increases address space requirements, reduces reference locality, and increases page fault and cache miss rates [37]. Moreover, since the collector must be conservative, some memory may never be reclaimed by the system, resulting in memory leaks. Finally, heapifying stack objects increases the runtime overhead of enforcing temporal safety since dynamic allocation is slower than automatic allocation.

Despite these drawbacks, conservative garbage collection is capable of enforcing complete temporal safety. This capability is depicted in Figure 1, where the use of garbage collection is shown in orange.

Temporal checks An alternative to using garbage collection for enforcing temporal safety is to insert explicit software checks that 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 temporal metadata that is created and propagated with spatial metadata links a pointer to the temporal capability of its referent. Methods employing this strategy include Safe C [4], MSCC [6], and the work by Patil and Fischer [5] and Yong and Horwitz [21].

There are advantages and disadvantages of using explicit temporal checks for enforcing temporal safety. The primary strength of this approach is that it retains C's memory allocation model and avoids the drawbacks associated with garbage collection. However, the inclusion of additional runtime checks and metadata significantly increases the runtime overhead beyond that of enforcing spatial safety alone. Figure 1 indicates that temporal capabilities (shown in orange), like garbage collection, can be used to enforce complete temporal safety.

MemSafe's approach One of MemSafe's main contributions is the modeling of temporal errors as spatial errors. Therefore, MemSafe does not require conservative garbage collection or explicit temporal checks, and it avoids the drawbacks of both approaches. Instead, MemSafe relies on spatial safety checks and the hybrid metadata representation mentioned above for ensuring temporal safety.

$$
\begin{array}{rrcl}
\text{Atomic Types} & \alpha & ::= & \textbf{int} \mid \tau* \\
\text{Types} & \tau & ::= & \alpha \mid \textbf{struct}\{d^{+}\} \mid \tau[n] \\
\text{Declarations} & d & ::= & \tau\ x; \\
\text{Functions} & f & ::= & \texttt{func}(x^{*})\ \{b^{*}\} \\
\text{Blocks} & b & ::= & p^{*}\ d^{*}\ s^{+} \\
\phi\text{-Functions} & p & ::= & x\ =\ \phi(x^{+}); \\
\text{LHS Expressions} & l & ::= & x \mid *l \mid l.y \\
\text{RHS Expressions} & r & ::= & r{+}r \mid l \mid \&l \mid (\alpha)r \mid \texttt{malloc}(r) \mid n \\
\text{Statements} & s & ::= & l{=}r; \mid l(r); \mid \textbf{for}(l{=}r;\ r{<}r;\ l{=}r)\ \{b\} \\
 & & \mid & \textbf{if}(r)\ \{b\}\ \textbf{else}\ \{b\} \mid \textbf{return}\ r; \mid \texttt{free}(r); \\
 & \text{where} & & x \in \text{variables} \\
 & & & y \in \text{structure field identifiers} \\
 & & & n \in \mathbb{N}
\end{array}
$$

Figure 2. Syntax for a simple SSA [24] language with procedures, pointers, control flow, and manual memory management.

3. MEMSAFE

This section describes MemSafe's unoptimized approach for ensuring the memory safety of C programs. 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.

3.1. Language Extensions and Assumptions

Because the C programming language, in its entirety, is both large and complex, the language defined in Figure 2 will be used for describing MemSafe's source code translations and the rules for runtime check insertion and metadata propagation. Figure 2 defines a small SSA [24] intermediate language that captures all the relevant pointer-related portions of C. Features of the language include, among others, syntax for pointer types, manual memory management, type-casting of pointer values, pointer arithmetic, and complex control-flow.

Without loss of generality, the following assumptions are made of the language presented in Figure 2. First, it is assumed that memory is only accessed with explicit load (e.g., `x=*ptr`) and store (e.g., `*ptr=x`) operations involving pointers. Second, it is assumed that pointer values are only created with the address-of operator (`&`) or by calling the `malloc` function. Recall that in C, a variable declared as an array of some particular type can act as a pointer to that type, and when used by itself, is a pointer that points to the first element of the array. To enforce the notion that pointers are only created through the two mechanisms above, all array accesses are represented as an indexing operation applied to the address of the first element of the array. For example, for the allocation of an array *a* of ten elements, an access of the fifth element `a[4]` is represented as `(&a[0])[4]`. That is, a pointer is created to the first element of the array, and then this pointer is used to compute the address of the fifth element. In this way, all new pointer values may only be created with the address-of operator and by calling the `malloc` system function.

Furthermore, MemSafe assumes all global variable definitions define a symbol that provides the address of an object instead of the actual object "contents." Since assignments of global variables must be conservatively accounted for in SSA, compiler intermediate representations (e.g., LLVM [38]) often represent global variables as pointers to statically allocated regions of memory. The advantage of this approach is that within a procedure, a global variable can be loaded from memory, renamed according to the SSA conversion algorithm, and then stored back to memory before control-flow reaches another procedure. Therefore, MemSafe identifies statically allocated objects by their location in memory. Note that MemSafe's representation of global variables is analogous to the discussion of arrays above in that the declaration of an object implicitly creates a pointer to that object.

MemSafe models both memory deallocation and pointer store operations as explicit assignments using syntax extensions to this C-like SSA language. The advantage of this approach is that it enables

MemSafe to ensure complete memory safety by reasoning solely about pointer definitions, which eliminates the need for separate mechanisms for detecting spatial and temporal errors and reveals optimization opportunities. The remainder of this section describes these abstractions.

3.2. Memory deallocation

Memory deallocation can implicitly change the object to which a pointer refers. If the region of memory that was occupied by a deallocated object is ever reallocated, the contents of the region may change, and any remaining pointers to the original object implicitly become invalid. This implicit redefinition of pointers can be made apparent by modeling both automatic and dynamic memory deallocation as an explicit pointer assignment. For example, MemSafe models the statement `free(p)` as `p=invalid`, where *invalid* is a special untyped pointer constant that points to an "invalid" region of memory. The base and bound address associated with this abstract memory region are defined by the impossible address range $[1,0]$. Thus, if the spatial metadata of p is located at address $addr_p$ in memory, then p could be associated with the base and bound of the *invalid* pointer by the statements `addr_p->base=1` and `addr_p->bound=0`. Since the size of this block is -1, spatial safety checks involving the base and bound of the *invalid* pointer are guaranteed to always report a safety violation. Therefore, temporal safety violations can be detected with runtime checks inserted for enforcing spatial safety. The rules for inserting assignments of the *invalid* pointer are given below.

Automatic memory deallocation If the address of a stack-allocated object is taken with the address-of operator (&), the pointer to this object may "escape" and be made available outside the function in which the object is allocated. Such an occurrence is possible, for example, if a local variable's address is stored in a global or heap variable. While this is a legal operation in the C programming language, a common consequence of escaping pointers is the program committing a temporal safety violation. When a function exits, its local variables are automatically deallocated, and any escaping pointers to these deallocated objects become dangling. To make the implicit redefinition of these pointers explicit, MemSafe inserts assignments of the *invalid* pointer at the end of a procedure for each of its local variables whose address is taken, as shown in the following example.

```
1: void f() {
2:   struct { ... int array[100]; ... } s, *p;
3:   ...
4:   p = &s;                  ▷ address of s is taken and may escape
5:   ...
     p = invalid;             ▷ MemSafe models deallocation as an explicit
6: }                            pointer assignment of 'invalid'
```

In this example, the nested structure s is allocated automatically on the stack as a local variable of function f. In line 4, the address of s is taken and stored in pointer p. It is assumed that p may escape to another procedure and result in a dangling pointer when function f exits. Therefore, MemSafe assigns p the value of the *invalid* pointer before the function exits, indicating that the pointer now refers to a temporally "invalid" region of memory. After this assignment, the base and bound of p would be updated to be equal to that of the *invalid* pointer, and any pointer derived from or aliased with p would inherit this metadata as well (see Section 3.5).

Dynamic memory deallocation If a pointer's referent is deallocated dynamically by a program calling the `free` function, all pointers that refer to this object become dangling pointers. The subsequent dereference of a dangling pointer results in a temporal safety violation. To make the redefinition of these pointers explicit, MemSafe inserts assignments of the *invalid* pointer after calls to `free` for the pointer used in deallocating the object, as shown in the following example.

```
1: int *p;
2: ...
3: p = malloc(size);
4: ...
5: free(p);                   ▷ MemSafe models deallocation as an explicit
```

```
   p = invalid;                pointer assignment of ‘invalid’
```

In this example, an object of *size* bytes is dynamically allocated by a program with the `malloc` function, and the base address of object is assigned to pointer p. In line 5, the object to which p refers is deallocated by the program with the `free` function. Therefore, MemSafe assigns p the value of the *invalid* pointer to indicate that the pointer now refers to a temporally "invalid" region of memory.

3.3. Pointer stores

Having inserted assignments of the *invalid* pointer to make the redefinition of pointers to deallocated memory explicit, MemSafe then transforms the program to make indirect pointer store operations explicit assignments as well.

Indirect assignments are problematic in SSA form and make the representation of pointer stores nonintuitive. A key property of SSA is that each assignment is given a unique name (hence, the "single assignment" condition of Static Single Assignment). However, this property does not hold for in-memory assignments of the form `*p=x`. In this case, `*p` is not given a unique name when it is assigned the value of x, and it is unclear what values loaded from memory can be equal to x.

To address this problem for the indirect assignment of pointer values, MemSafe models in-memory pointer assignments (including those induced for the *invalid* pointer) as explicit assignments using alias analysis and a ϕ-like SSA extension called the ϱ-function. In the same way that the ϕ-function of SSA is used to resolve control-flow uncertainty, thereby giving a unique name to conditional assignments at the point where control-flow paths merge, MemSafe uses the ϱ-function to resolve the data-flow uncertainty of pointer values, thereby giving a unique name to indirect pointer assignments at the point where pointers are loaded from memory.

For example, assume the statement `s0:*p=ptr0` is the only direct reaching definition of a pointer defined as `s1:ptr1=*p`. The statement `s2:*q=ptr2` may *indirectly* redefine ptr_1 if p and q may alias and control-flow may reach statement s_1 from s_2. Therefore, MemSafe models `ptr1 = *p` as `ptr1=ϱ(ptr0,ptr2)`, meaning the value of ptr_1 may equal that of 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.

The following code fragment provides a more concrete example of the ϱ-function and demonstrates how this syntax extension creates a synergy with MemSafe's representation of memory deallocation that results in the propagation of the data-flow associated with the *invalid* pointer.

```
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;          ▷ MemSafe models deallocation as an explicit
8: }                          pointer assignment of ‘invalid’
   ...
9: c0 = *q
   c1 = ϱ(a, b, invalid);   ▷ MemSafe models in-memory data-flow with the
                              ϱ-function. All subsequent uses of c0 are
                              replaced with uses of c1.
```

In this example, all pointer values exist in memory, and pointer assignments are made possible by pointer store and load operations. After the call to the `free` function in line 7, an in-memory assignment of the *invalid* pointer is inserted to indicate that the referent of `*p` has been deallocated. Since p and q are assumed to potentially alias, this store operation and the ones in lines 3 and 5 may define the pointer loaded and assigned to the variable c_0 in line 9. Therefore, MemSafe resolves this uncertainty in pointer data-flow and gives a unique name to these assignments by inserting the

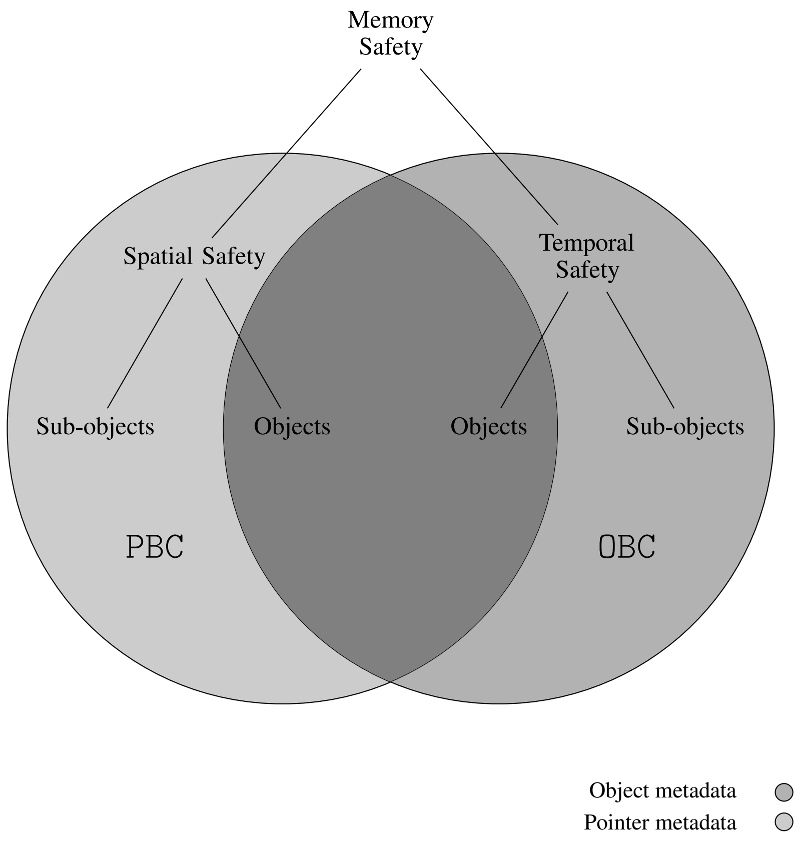


Figure 3. Hybrid metadata representation. The use of object and pointer spatial metadata is shown for MemSafe's method of enforcing memory safety.

ϱ-function after line 9 and assigning it to the variable c_1. Pointers a, b and *invalid* are added to the ϱ-function assigned to pointer c_1, meaning c_1 may be equal to any of these three values, but only these values. All subsequent uses of pointer c_0 are replaced with uses of c_1.

By default, MemSafe utilizes flow- and context-insensitive pointer alias information to determine the arguments of ϱ-functions. However, MemSafe is capable of using more precise alias analyses, and in general, their use results in ϱ-functions with smaller arity. In the case of the former, MemSafe performs a simple reachability analysis to improve the results of alias analysis. For example, consider the pointer store operation `*ptr1=p0` and the pointer load operation `p1=*ptr2`. If alias analysis indicates that the pointers ptr_1 and ptr_2 may alias, p_0 would be added to the ϱ-function inserted for p_1. However, if there is no control-flow path from the store operation to the load operation, this is unnecessary since there is no program execution in which the stored value can modify the loaded value. MemSafe does not include stored pointers in the ϱ-functions inserted for loaded pointers if the store cannot reach the load. Note that MemSafe's reachability analysis does not result in a flow-sensitive alias analysis.

3.4. The Required Checks and Metadata

After inserting code for modeling memory deallocation and pointer stores as explicit assignments, MemSafe then inserts the runtime checks and metadata necessary for enforcing memory safety. This section describes the pointer- and object-based checks and metadata that MemSafe requires.

Figure 3 depicts MemSafe's unique combination of object- and pointer-based metadata. In contrast to the enforcement methods shown in Figure 1, MemSafe utilizes this hybrid metadata representation for ensuring both the spatial and temporal memory safety of C programs.

The differences between Figures 1 and 3 can be explained as follows. First, because MemSafe models temporal errors as spatial errors, MemSafe avoids the drawbacks associated with the use of conservative garbage collection and the use of additional checks and metadata for enforcing temporal safety (shown in orange in Figure 1). Second, since MemSafe's hybrid metadata representation captures the most salient features of object and pointer metadata, MemSafe avoids the drawbacks associated with the use of each in enforcing spatial safety, and gains the ability to reuse this metadata for enforcing temporal safety as well. As shown in Figure 3, MemSafe utilizes pointer-based metadata for enforcing complete spatial and partial temporal safety, and MemSafe utilizes object-based metadata for enforcing complete temporal and partial spatial safety. `PBC` and `OBC` are MemSafe's runtime checks that utilize this metadata. These checks, in addition to MemSafe's object- and pointer-based metadata, are discussed below.

Pointer Metadata For the definition of a new pointer p (i.e., a pointer created with `malloc` or the address-of operator), MemSafe creates pointer metadata in the form of a 3-tuple $\langle base, bound, id\rangle_p$ of intermediate values. Together, $base_p$ and $bound_p$ indicate the range $[base, bound)$ of memory p is permitted to access. id_p is a unique key that is assigned to *p's* referent object, and it is used to associate p with the metadata of its referent (discussed in Section 3.4). MemSafe maintains pointer metadata in memory and allocates at runtime an address $addr_p$ from a set of unused addresses A for storing $\langle base, bound, id\rangle_p$. These values are stored to memory with an explicit dereference operation, represented by $M[addr_p] \leftarrow \langle base, bound, id\rangle_p$, where $M[addr_p]$ holds the value at address $addr_p$ in memory.

In addition to the pointer metadata described above, MemSafe also creates a tuple $\langle addr, id\rangle_p$ of intermediate values. These values are created for the definition of each pointer p in a program (i.e., not just those pointers created with `malloc` or the address-of operator), and are statically named such that there is a known compile-time association with p. Unlike the metadata described above, no dereference is required at runtime for retrieving these values. As previously described, $addr_p$ is the location in memory containing the base and bound addresses that indicate the range of memory p is permitted to access. Finally, in order to allow the reuse of location $addr_p$ (discussed later), a copy of the id associated with *p's* referent is also maintained with this statically associated tuple.

Pointer bounds check MemSafe utilizes pointer metadata for performing a Pointer Bounds Check (PBC). MemSafe inserts a `PBC` before each pointer dereference that cannot be verified to be safe statically (see Section 4.2 for optimizations that reason about dereferences that must be safe). `PBC` is the forcibly inlined procedure defined by:

```
inline void PBC(ptr, size, addr, id) {
  $\langle base, bound, id\rangle_{ptr} \leftarrow M$[addr]
  if ((id != $id_{ptr}$) || (ptr < $base_{ptr}$) || (ptr + size > $bound_{ptr}$)) {
    signal_safety_violation();
  }
}
```

In the above check, ptr, $base_{ptr}$, and $bound_{ptr}$ are all pointers to the type `unsigned char` and $size$ is the size in bytes (as indicated by the `sizeof` operator) of *ptr's* referent.[5] MemSafe utilizes the pointer metadata of a pointer ptr to ensure the safety of its dereference at runtime:

```
PBC(ptr, sizeof(*ptr), $addr_{ptr}$, $id_{ptr}$);
... *ptr ...
$\triangleright$ some load or store operation involving ptr
```

[5]These pointers are implicitly type-cast to be pointers to type `unsigned char` because `sizeof(unsigned char)` is defined to always equal one byte [39]. This is required for the pointer arithmetic performed in the body of the bounds check to be valid.

In this example, MemSafe will abort the program and report a violation of memory safety (by calling `signal_safety_violation`) if the dereference `*ptr` will access a location outside the range specified by $[base_{ptr}, bound_{ptr})$. Because the `PBC` only utilizes pointer metadata, no costly database lookup is required to retrieve $base_{ptr}$ and $bound_{ptr}$, as $\langle addr, id\rangle_{ptr}$ are uniquely named symbols in the inserted code.

As depicted in Figure 3, the `PBC` and MemSafe's pointer-based metadata are capable of not only ensuring complete spatial safety, but also temporal safety with a *single check*. Whenever a pointer p is assigned the value of the *invalid* pointer, its pointer metadata is updated as $M[addr_p] \leftarrow \langle base, bound, id\rangle_{invalid}$, which will always cause the `PBC` to signal a safety violation since the *invalid* pointer refers to an impossible address range.

As mentioned above, since addresses in A can be reused, a copy of the *id* associated with a pointer p must be included with *p's* pointer metadata. Whenever the metadata associated with the *invalid* pointer is stored to a particular address, this address is marked for potential reuse. Thus, $addr_p$ may be reused for storing the pointer metadata of another pointer if *p's* referent is deallocated. To ensure that $addr_p$ has not been reused, the `PBC` checks whether the *id* associated with the dereferenced pointer p, is equal to the *id* located at $addr_p$. If it is not, $addr_p$ has been reused for the pointer metadata of another pointer and *p's* referent is temporally invalid.

However, the `PBC` is insufficient for ensuring complete temporal safety. Since a nested object (e.g., an array of structures or a structure containing and array field) is deallocated using a pointer to its base address, only pointers that refer to the outer object are assigned the value of the *invalid* pointer upon the object's deallocation. The pointer metadata of any potential sub-object references are not updated in this way (see the rules for metadata propagation in Section 3.5). Thus, object metadata is required to associate pointers to *inner* objects with the base and bound address of their corresponding *outer* object. Object metadata is introduced 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 object 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 formally defined by the partial function:

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

where I is the set of *ids* and O is the set of object metadata. For notational convenience, the function *omd* can also be represented more generally as the relation $\mathcal{R}_O$, where $(id, \langle base, bound\rangle_{id}) \in \mathcal{R}_O$ if the object associated with *id* is a valid memory object that has yet to be deallocated. A discussion of the implementation of the object metadata facility is deferred until Section 5.2 in order to separate the presentation of MemSafe's method from its prototype implementation.

Object bounds check MemSafe utilizes object metadata for performing an Object Bounds Check (OBC). MemSafe inserts an `OBC`, in addition to the `PBC` described above, before each pointer dereference that may access a sub-object if the pointer cannot be statically verified to be temporally safe.[6] `OBC` is the forcibly inlined procedure defined by:

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

[6]MemSafe determines the set of potential sub-object references by traversing its data-flow graph, which is described in Section 4.1.

In the above check, *ptr* is a pointer to type `unsigned char`, *id* is a component of *ptr's* pointer metadata, and *size* is the size in bytes of *ptr's* referent. MemSafe utilizes the object metadata of pointer *ptr's* referent, denoted $\langle base, bound\rangle_{id}$, to ensure the safety of its dereference at runtime:

```
PBC(ptr, sizeof(*ptr), $addr_{ptr}$, $id_{ptr}$);
OBC(ptr, sizeof(*ptr), $id_{ptr}$);
... *ptr ...
▷ some load or store operation involving ptr
```

In this example, MemSafe will abort the program and report a violation of memory safety if the dereference `*ptr` will access a location outside the range specified by $[base_{id}, bound_{id})$. The `OBC` uses the *id* field of *ptr's* pointer metadata to retrieve the object metadata of its referent from the object metadata facility. Assuming pointer *ptr* refers to a sub-object, the temporal safety of *ptr's* dereference is ensured because, had *ptr's* referent been previously deallocated, its entry would have been unmapped in the object metadata facility $\mathcal{R}_O$, causing *omd*(*id*) to fail and MemSafe to signal a safety violation.

As depicted in Figure 3, the `OBC` and MemSafe's object-based metadata are capable of not only ensuring complete temporal safety, but also partial spatial safety with a *single check*. Thus, if the detection of sub-object overflows is not a requirement, the `PBC` in the above example can be eliminated since the `OBC` also verifies *ptr* is within bounds of its outer object.

3.5. Propagation of the Required Metadata

Having presented the runtime checks that MemSafe requires for ensuring memory safety, this section describes MemSafe's translations for creating and propagating the required metadata. In doing so, it is assumed that the program has already been transformed such that it includes the syntax extensions for modeling memory deallocation and pointer stores as explicit pointer assignments (see Section 3.1). In the discussion below, the rules for propagating the required metadata are addressed according to the way in which pointers are defined.

Memory Allocation As described previously, MemSafe creates entries in the global object metadata facility as objects are allocated. For automatic memory allocation (i.e., the allocation of stack variables), MemSafe generates a new *id* for the allocated object and maps it to the object's base and bound address in $\mathcal{R}_O$, as shown below.

```
struct { ... int array[100]; ... } s;
```

$$\mathcal{R}_O = \mathcal{R}_O \cup \{(id \in I, \langle \texttt{\&s}, \texttt{\&s+}\textbf{sizeof}\texttt{(s)}\rangle)\} \tag{1}$$

In this example, a structure containing an array field is allocated on the procedure stack. Therefore, MemSafe obtains a new *id* for the allocated object and maps it to the base and bound address of the allocated region in $\mathcal{R}_O$ (1).

For dynamic memory allocation (i.e., the allocation of objects on the heap), MemSafe updates $\mathcal{R}_O$ as it does for automatic memory allocation, but it also creates pointer metadata for the pointer returned by `malloc`, since the `malloc` function is responsible for creating a new object as well as a new pointer to the allocated object. If the pointer returned by `malloc` is equal to the `NULL` pointer, the pointer inherits the metadata of the *invalid* pointer. MemSafe creates the required object and pointer metadata for heap-allocated objects as shown below.

```
int *p;
...
p = (int*) malloc(size);
```

$$\langle addr, id\rangle_p = \langle addr \in A, id \in I\rangle \tag{2}$$

$$\langle base, bound\rangle_{id_p} = \begin{cases} \langle base, bound\rangle_{id_{invalid}} & \text{if } p = null, \\ \langle \texttt{p}, \texttt{p+}\mathit{size}\rangle & \text{otherwise} \end{cases} \tag{3}$$

$$\mathcal{R}_O = \mathcal{R}_O \cup \left\{ \left(id_p, \langle base, bound \rangle_{id_p} \right) \right\} \tag{4}$$
$$M[addr_p] \leftarrow \langle base_{id_p}, bound_{id_p}, id_p \rangle \tag{5}$$

In this example, an object of *size* bytes is allocated dynamically by calling `malloc`, and the address returned by `malloc` is assigned to the pointer p. After the program allocates the object, MemSafe obtains an address for holding the pointer metadata of p and obtains a new unique *id* for the allocated object (2). If the value returned by `malloc` is equal to `NULL`, the object metadata associated with id_p is set to the base and bound address of the "invalid" region of memory. Otherwise, the metadata associated with the object is defined such that it refers to the space occupied by the allocated region of memory (3). Finally, MemSafe associates the object's metadata with id_p in $\mathcal{R}_O$ (4) and stores the metadata of p at its associated address (5).

For static memory allocation (i.e., the allocation of global variables), MemSafe initializes the object metadata facility to include entries for the base and bound address of each allocated region, since the number and size of global variables in known at compile-time.

Memory Deallocation Whenever an object is deallocated, MemSafe removes its entry from $\mathcal{R}_O$ and sets the pointer metadata of the pointer that refers to the object to be equal to that of the *invalid* pointer. Stack-allocated objects are deallocated when the function in which they are defined exits. Therefore, MemSafe removes their entries from $\mathcal{R}_O$ before the end of the procedure, as shown below.

```
1: void f() {
2:   struct { ... int array[100]; ... } s;
3:   int *p;
4:   ...
5:   p = &(s.array[42]);
6:   ...
```

$$\mathcal{R}_O = \mathcal{R}_O \setminus \{(id_s, omd\,(id_s))\} \tag{6}$$
$$M[addr_p] \leftarrow \langle base, bound, id \rangle_{invalid} \tag{7}$$

```
7:   p = invalid;          ▷ MemSafe models deallocation as an explicit
8: }                         pointer assignment of 'invalid'
```

In this example, structure s is an automatic variable of function f and contains an array sub-object that is nested within it. In line 5, pointer p is assigned the address of an element of the structure's *array* field, and it is assumed that p may escape to another procedure. Before the procedure exits, MemSafe removes the entry for s from $\mathcal{R}_O$ (6) using the unique *id* associated with s ("\" denotes set difference). Since p may escape, it is assigned the value of the *invalid* pointer in line 7, and its pointer metadata is updated to refer to the metadata associated with *invalid* (7).

Heap-allocated objects are deallocated dynamically with the `free` function. Similar to the above rule for automatic memory deallocation, MemSafe updates object and pointer metadata for dynamic memory deallocation, as shown below.

```
1: int *p;
2: ...
```

$$\mathcal{R}_O = \mathcal{R}_O \setminus \left\{ \left(id_p, omd\left(id_p \right) \right) \right\} \tag{8}$$
$$M[addr_p] \leftarrow \langle base, bound, id \rangle_{invalid} \tag{9}$$

```
3: free(p);
4: p = invalid;          ▷ MemSafe models deallocation as an explicit
                           pointer assignment of 'invalid'
```

In this example, a pointer p to a heap-allocated object is used to deallocate its referent dynamically by the program calling the `free` function. Before the call to `free` in line 3, MemSafe removes the entry for the deallocated object from $\mathcal{R}_O$ with id_p (8) and sets the pointer metadata of p to be equal to that of the *invalid* pointer (9). Pointer p is assigned the value of *invalid* in line 4.

If id_p had been previously unmapped in the object metadata facility (indicating that *p's* referent was already deallocated before the call to `free`), the lookup operation represented by $omd\left(id_p\right)$ would fail. In this case, MemSafe would signal a temporal safety violation to indicate the multiple deallocation attempt.

Address-of Operator Like dynamic memory allocation, the address-of operator (&) creates a pointer to a new location. Therefore, having already updated $\mathcal{R}_O$ for an object's allocation, MemSafe creates pointer metadata for pointers to the object, as shown below.

```
struct { ... int array[100]; ... } s;
int *p;
...
p = &(s.array[42]);
```

$$\langle addr, id\rangle_p = \langle addr \in A, id_s\rangle \tag{10}$$

$$M[addr_p] \leftarrow \langle \texttt{\&s.array[0]}, \textbf{sizeof}\texttt{(s.array)}, id_p\rangle \tag{11}$$

In this example, as in previous examples, a pointer p is assigned the address of an element of the *array* field of structure s. Because the program creates a new pointer, MemSafe obtains a new address for storing the pointer metadata of p (10). MemSafe then creates and stores pointer metadata for p to indicate that it refers to the base and bound address of the *array* field of s (11).

This example also demonstrates MemSafe's ability to detect sub-object overflows. Although p refers to a location within object s (indeed, p inherits the id of s), *p's* base and bound address are associated with the *array* field of s.

Pointer Copies and Arithmetic Pointers defined as simple pointer copies or in terms of pointer arithmetic (e.g., array and structure indexing) inherit the pointer metadata of the original pointer, as shown below.[7]

```
int x, *p0, *p1;
...
p1 = p0 + x;
```

$$\langle addr, id\rangle_{p_1} = \langle addr, id\rangle_{p_0} \tag{12}$$

In this example, since pointer p_1 is defined in terms of pointer arithmetic, it simply inherits the pointer metadata associated with pointer p_0 (12).

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:

$$\begin{aligned} pmd &: A \rightarrow P \\ ptr &\mapsto \langle addr, id\rangle_{*\texttt{ptr}} \end{aligned}$$

where A is the set of addresses, and P is the set of pointer metadata. For convenience, the function *pmd* can also be represented more generally as the relation $\mathcal{R}_{\mathcal{P}}$, where $(ptr, \langle addr, id\rangle_{*\texttt{ptr}}) \in \mathcal{R}_{\mathcal{P}}$.

For pointer loads, MemSafe creates a new definition for the loaded value and assigns it the result of a ϱ-function, which indicates the set of values to which the loaded value may potentially be equal. For a pointer *ptr* whose pointed-to location is loaded in defining another pointer p, MemSafe retrieves from the pointer metadata facility the required pointer metadata for p with the lookup operation $pmd(ptr)$, as shown below.

[7]SSA ϕ-functions that involve pointer values are no different than ordinary pointer copies. For example, `p2=`ϕ`(p0,p1)` copies p_0 to p_2 at the end of the basic block defining p_0 and copies p_1 to p_2 at the end of the basic block defining p_1.

```
int **ptr1, *p0, *p1, ...;
...
p0 = *ptr1;                      ▷ MemSafe models in-memory data-flow with
p1 = $\varrho$(a0, b0, ...);     the $\varrho$-function
```

$$\langle addr, id\rangle_{p_1} = pmd(ptr_1) \tag{13}$$

In this example, an in-memory pointer is loaded and assigned to pointer p_0. MemSafe then creates a new pointer p_1 and assigns it the result of a ϱ-function indicating the values the in-memory pointer may potentially equal. The pointer metadata for p_1 is retrieved from the pointer metadata facility with the $pmd(ptr_1)$ lookup operation (13), and all uses of p_0 are replaced with uses of p_1.

For each argument of the ϱ-function (including the *invalid* pointer), MemSafe saves their pointer metadata in $\mathcal{R}_{\mathcal{P}}$ at the locations each pointer is stored to memory, as shown below.

```
1: int **ptr2, *a0;
2: ...
3: *ptr2 = a0;        ▷ ptr2 may alias ptr1 from above
```

$$\mathcal{R}_{\mathcal{P}} = (\mathcal{R}_{\mathcal{P}} \setminus \{(ptr_2, pmd(ptr_2))\}) \cup \{(ptr_2, \langle addr, id\rangle_{a_0})\} \tag{14}$$

In this example, pointer ptr_2 is assumed to potentially alias with pointer ptr_1 from the previous example. Thus, pointer a_0 appears in the ϱ-function defined above for pointer p_1 because of the pointer store in line 3. Here, MemSafe maps pointer ptr_2 to the pointer metadata of a_0 in $\mathcal{R}_{\mathcal{P}}$ (14). If ptr_1 equals ptr_2, the pointer metadata of a_0 would be retrieved in the previous example.

`NULL` and Manufactured Pointers Pointer type-casts and unions do not require any additional metadata propagation. The new pointer simply inherits the pointer metadata of the original pointer, as in the rule for pointer copies and arithmetic. However, pointers defined as `NULL` or as a cast from a non-pointer type must inherit the base and bound of the *invalid* pointer, as shown below.[8]

```
int *p;
...
p = (int*) 42;
```

$$\langle addr, id\rangle_{p} = \langle addr, id\rangle_{invalid} \tag{15}$$

In this example, pointer p is defined as a type-cast from the integer 42. Thus, MemSafe defines the pointer metadata for p to be equal to that of the *invalid* pointer (15). The result would have been the same if p had been assigned the value of `NULL`.

Function Arguments and Return Values MemSafe requires an additional metadata facility in order to propagate pointer metadata for pointers passed as arguments to functions or returned from functions. Let *callee* values refer to formal pointer arguments and pointer values that are returned from functions. Similarly, let *caller* values refer to actual pointer arguments and local pointer values to be returned from functions. The function metadata facility maps a *callee* value to the pointer metadata of its corresponding *caller* value and is defined by the partial function:

$$\begin{aligned} fmd &: C \to P \\ callee &\mapsto \langle addr, id\rangle_{caller} \end{aligned}$$

where C is the set of caller values, P is the set of pointer metadata, and *callee* is a tuple $\langle \texttt{\&f}, i\rangle$ indicating the i^{th} pointer associated with function *f*. Pointers are statically assigned an index i based on their usage: the return value of a function is assigned index zero, and the pointer arguments of a function are assigned an index equal to their offset in the function's argument list, beginning at one.

[8]Although this may result in false positives, they have been observed to be rare occurrences in practice. For reading and writing to memory-mapped I/O locations, MemSafe requires a target's backend to specify the base and bound address of all valid address ranges.

For notational convenience, the function *fmd* can also be represented more generally as the relation $\mathcal{R}_{\mathcal{F}}$, where $(callee, \langle addr, id\rangle_{caller}) \in \mathcal{R}_{\mathcal{F}}$.

For function calls, MemSafe creates an entry in the function metadata facility for pointer arguments passed to the function. Similarly, MemSafe defines the pointer metadata of a pointer returned from the function call by performing a lookup operation of $\mathcal{R}_{\mathcal{F}}$. MemSafe updates and defines pointer metadata for function calls as shown below.

```
int *p0, *p1;
...
```

$$\mathcal{R}_{\mathcal{F}} = (\mathcal{R}_{\mathcal{F}} \setminus \{(\langle \texttt{\&f}, 1\rangle, fmd(\langle \texttt{\&f}, 1\rangle))\}) \cup \left\{\left(\langle \texttt{\&f}, 1\rangle, \langle addr, id\rangle_{p_0}\right)\right\} \tag{16}$$

```
p1 = f(p0);
```

$$\langle addr, id\rangle_{p_1} = fmd(\langle \texttt{\&f}, 0\rangle) \tag{17}$$

In this example, a pointer p_0 is passed as an argument to function *f* and pointer p_1 is assigned the returned value. The return value of *f* is statically associated with the index "0," and its single pointer argument is given an index of "1." Thus, before the function call, the pointer metadata of p_0 is associated with the tuple $\langle \texttt{\&f}, 1\rangle$ in $\mathcal{R}_{\mathcal{F}}$ (16). That is, a key represented by the address of *f* and the integer "1" is mapped to the pointer metadata of p_0. Similarly, after the call returns, the pointer metadata for p_1 is retrieved from $\mathcal{R}_{\mathcal{F}}$ with the tuple $\langle \texttt{\&f}, 0\rangle$ (17).

For the declaration of a function with pointer arguments, MemSafe retrieves the pointer metadata for each incoming pointer by performing a lookup operation of $\mathcal{R}_{\mathcal{F}}$. Similarly, if a function returns a pointer value, MemSafe creates an entry in $\mathcal{R}_{\mathcal{F}}$ for its pointer metadata just before the function returns. MemSafe updates and defines pointer metadata for function declarations as shown below.

```
int* f(int *q) {
  int *r;
  ...
```

$$\langle addr, id\rangle_{q} = fmd(\langle \texttt{\&f}, 1\rangle) \tag{18}$$

```
  ...
```

$$\mathcal{R}_{\mathcal{F}} = (\mathcal{R}_{\mathcal{F}} \setminus \{(\langle \texttt{\&f}, 0\rangle, fmd(\langle \texttt{\&f}, 0\rangle))\}) \cup \{(\langle \texttt{\&f}, 0\rangle, \langle addr, id\rangle_{r})\} \tag{19}$$

```
  return r;
}
```

In this example, pointer *q* is a formal argument of function *f*, and pointer *r* is returned at the end of the procedure. Since *q* is declared to be the first pointer in the function's argument list, MemSafe retrieves the pointer metadata for *q* from $\mathcal{R}_{\mathcal{F}}$ with the tuple $\langle \texttt{\&f}, 1\rangle$ at the beginning of the procedure (18). Similarly, since MemSafe statically assigns pointer return values the index "0," the pointer metadata of *r* is associated with the tuple $\langle \texttt{\&f}, 0\rangle$ in $\mathcal{R}_{\mathcal{F}}$ before the procedure exits (19).

MemSafe's approach for propagating metadata for pointer arguments and return values is quite robust. It is sufficient for interfacing with pre-compiled libraries, handling variable-argument functions, and passing metadata through poorly-typed function pointers. For complete safety, pre-compiled libraries must have been compiled with MemSafe's safety checks, but a safe application is capable of interfacing with unsafe libraries as well.

Memory Copying Functions The memory copying functions (e.g. `memcpy` and `memmove`) defined in the `string.h` standard library generally copy a specified number of bytes from a location indicated by a source pointer *src* to the location referred to by a destination pointer *dest*. Although these procedures result in multiple read and write operations, MemSafe only needs to perform bounds checks for the source and destination buffers once before the operation begins.

However, any in-memory pointer values that are located in the source buffer and copied to the destination buffer must also have their associated pointer metadata copied. Recall that pointer metadata that is associated with in-memory pointers is maintained in the pointer metadata facility $\mathcal{R}_{\mathcal{P}}$. Thus, any mapped address in $\mathcal{R}_{\mathcal{P}}$ that is within the range $[base, bound)$ of the source buffer must have its metadata copied and associated with a new address that is located at a distance of $dest - src$ bytes from the original address. MemSafe updates and defines pointer metadata for the memory copying functions of `string.h` as shown below.

```
1: memcpy(v, u, size);
```

$$S = \Big\{(ptr, \langle addr, id\rangle_{*\mathtt{ptr}}) \in \mathcal{R}_{\mathcal{P}} : base_{src} \leq ptr < bound_{src}\Big\} \tag{20}$$

$$D = \Big\{(ptr + (dest - src), \langle addr, id\rangle_{*\mathtt{ptr}}) : (ptr, \langle addr, id\rangle_{*\mathtt{ptr}}) \in S\Big\} \tag{21}$$

$$\mathcal{R}_{\mathcal{P}} = \mathcal{R}_{\mathcal{P}} \cup D \tag{22}$$

In this example, the definition of S selects the pointer metadata associated with the source buffer that must be copied (20), and the definition of D associates the metadata with a new address within bounds of the destination buffer (21). Finally, $\mathcal{R}_{\mathcal{P}}$ is updated to contain the copied metadata (22). To avoid the runtime overhead of performing the metadata copy, MemSafe attempts to infer if the source buffer contains any in-memory pointer values by reasoning about its type and usage. Although this may lead to the pointer metadata facility not being properly updated, instances of in-memory pointers being copied with the `string.h` functions have been observed to be rare in practice.

4. REDUCING THE RUNTIME COST OF ENFORCING MEMORY SAFETY

Because the runtime overhead of MemSafe's basic approach can be prohibitively expensive for use in real systems, this section develops several tools and optimizations for reducing the cost associated with the inserted checks for memory safety and the code required for propagating metadata. First, a novel pointer data-flow representation is presented that is made possible by the modeling of both memory deallocation and pointer store operations as explicit pointer assignments. Then, this data-flow representation is used as the foundation of several optimizations that identify and eliminate unneeded runtime checks and code for propagating unused metadata.

4.1. A Data-flow Graph for Pointers

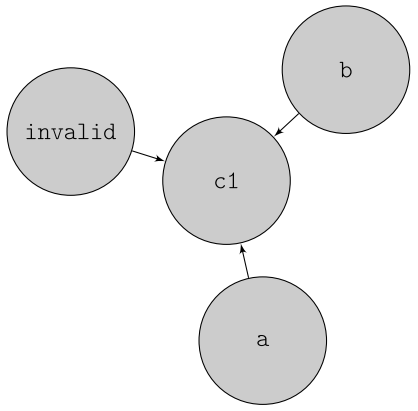
By utilizing the abstractions developed in Section 3.1 for memory deallocation and pointer stores, MemSafe creates a whole-program Data-Flow of Pointers Graph (DFPG). The DFPG is a definition-use graph for the flow of all pointer metadata in a program. Since the *invalid* pointer creates a direct pointer assignment for each deallocated object, and since the ϱ-function creates a pointer assignment for indirect pointer stores, every pointer assignment, whether it is an explicit or implicit assignment, is given a unique name in the SSA representation of the program. Therefore, if the pointer metadata associated with a pointer p is copied to that of another pointer q, there is a directed edge from p to q in the DFPG.[9] Similarly, if the pointer metadata of q is loaded from memory, a ϱ-function is inserted that indicates the pointers whose metadata may equal that of q, and these pointers are represented in the DFPG as predecessors of q. In general, since the data-flow of pointers may flow recursively, cycles are possible in the DFPG. Recall that since the data-flow associated with pointer loads and stores is represented by the ϱ-function, the definitions and uses associated with these operations are not included in the DFPG.

Figure 4 shows (a) an example code fragment and (b) its associated DFPG. In this code fragment, which was introduced during the discussion of the *invalid* pointer and ϱ-function (see Section 3.3), the only pointer definition that is not a pointer load or store operation is that of c_1, which occurs after line 9. Since the definition of c_1 uses the value of three other pointers (a, b, and *invalid*), there is an edge in the DFPG from each of these pointers to c_1.

[9]For simplicity, nodes in the DFPG are identified by named pointer values. However, in actuality, DFPG nodes represent the pointer metadata associated with these pointers, and not the pointers themselves.

```
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) Syntax Extensions



(b) DFPG

Figure 4. DFPG construction. A code fragment with MemSafe's syntax extensions (a) and its corresponding DFPG (b). Numbered lines indicate original code.

4.2. *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. Dominated dereferences do not require checking.

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 [40] 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 compile-time checks to eliminate 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. For the dereference of such a pointer, MemSafe removes the pointer's PBC from within the loop body, and inserts a Monotonically Addressed Range Check (MARC) in the loop pre-header. If the pointer's dereference also requires an OBC, this check is placed in the pre-header as well. MARC is the forcibly inlined procedure defined by:

```
inline void MARC(ptr, size, addr, id, trip_count) {
  $\langle base, bound, id\rangle_{ptr} \leftarrow M$[addr]
  ptr_max = ptr + trip_count;
  if ((id != $id_{ptr}$) || (ptr < $base_{ptr}$) || (ptr_max + size > $bound_{ptr}$)) {
    signal_safety_violation();
6:   }
7: }
```

In the above check, *ptr*, $base_{ptr}$, and $bound_{ptr}$ are all pointers to type `unsigned char`, and *size* is the size on byptes of *ptr's* referent. MemSafe inserts the following code to ensure the safety of a pointer dereference within a loop:

```
1: MARC(ptr, sizeof(*ptr), $addr_{ptr}$, $id_{ptr}$, N);
   for (i = 0; i < N; i++) {
      ... *(ptr + i) ... ;
      ▷ some store or load operation involving ptr
   }
```

In this example, MemSafe signals a safety violation if the dereference `*(ptr+i)` will access a location outside the range specified by $\langle base, bound, id\rangle_{ptr}$ 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.[10]

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.

5. IMPLEMENTATION

Having described MemSafe's approach for inserting and optimizing the runtime checks needed for ensuring memory safety, this section describes the implementation of the MemSafe compiler and the global data structures it requires for maintaining metadata. Additionally, implementation issues related to the C language and the typical C programming development process are also considered.

5.1. MemSafe's Analysis and Transformation

MemSafe is implemented within the Low Level Virtual Machine (LLVM) [38] compiler infrastructure. LLVM's intermediate representation is a low-level, typed SSA [24] form that is language-independent and also independent of any instruction set architecture. Thus, the implementation of MemSafe's transformation for ensuring memory safety is not specific to a particular computer architecture, and in theory, could also be used to enforce safety for languages other than C. However, MemSafe has not been tested for this purpose. By default, MemSafe uses Andersen's analysis [42] for interprocedural flow- and context-insensitive points-to information, but MemSafe is compatible with any alias analysis implementation that operates within the LLVM infrastructure.

MemSafe consists of a collection of analyses and transformations that each contribute a portion of the overall approach described in Section 3. These include compilation passes for (1) inserting assignments of the *invalid* pointer at deallocation sites, (2) inserting ϱ-functions with the aid of alias analysis, (3) constructing the DFPG, (4) inserting optimized safety checks and code for metadata propagation, and (5) aggressively removing all previously inserted assignments of *invalid* (but not its associated metadata) and all ϱ-functions since these are only used for MemSafe's analysis. We run the above passes after the program has been linked and optimized with LLVM's standard set of optimizations. Applying MemSafe's transformation after LLVM's optimizations improves the results of alias analysis and ensures that MemSafe avoids inserting unnecessary checks. We run LLVM's standard optimizations once more after MemSafe has completed its transformation to further

[10]Although MemSafe uses the `MARC` to essentially hoist checks out of loops, array bounds check elimination [41] can be used with MemSafe to completely eliminate some of these checks.

optimize the inserted checks and to eliminate any dead code MemSafe may have introduced during its analysis.

5.2. Metadata Facilities

The global facilities ($\mathcal{R}_{\mathcal{O}}$, $\mathcal{R}_{\mathcal{P}}$ and $\mathcal{R}_{\mathcal{F}}$) MemSafe requires for maintaining object and pointer metadata can be implemented using any data structure that supports efficient insertion, deletion, and retrieval operations. For simplicity and ease of implementation, MemSafe uses dynamically resized hash tables for all three metadata facilities. Collisions are resolved using separate chaining. Hash functions are a 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 prototype implementation of MemSafe makes two simplifications in the process described for creating pointer and object metadata. Fist, MemSafe acquires addresses $addr \in A$ for storing a pointer's pointer metadata using `malloc`, and this storage is released back to the system using `free` when a pointer's metadata is updated to be that of the *invalid* pointer. Second, MemSafe acquires a unique $id \in I$, which is used as a key for the object metadata facility by incrementing a global counter. While this ensures that each generated *id* is unique, this places a finite limitation on the number of objects that can be allocated by a program. However, note that a 4GHz computer would take 136 years to overflow a 64-bit counter allocating a new object on every clock cycle.

5.3. Limitations

Although MemSafe's method of ensuring the memory safety of C is complete and compatible with most programs, given C's weak typing guarantees and the typical application development process, in practice, MemSafe is not free from limitations. For example, the implementation of MemSafe currently does not support inline assembly instructions and does not allow self-modifying code. For programs requiring assembly, MemSafe could be extended with the appropriate rules for handling these instructions, but this would likely limit the effectiveness of MemSafe's optimizations.

MemSafe's most significant limitation is its use of whole-program analysis to limit the number of required checks and to avoid unnecessary metadata propagation. Although analyzing the entire program is essential for reducing the cost of software-provided memory safety, it negates the advantages of separate compilation, and can be problematic for use in common build environments. However, MemSafe's whole-program analysis, which is based the construction of the DFPG, is not required for enforcing safety. The checks and metadata propagation described in Sections 3.4–3.5 are fully compatible with separate compilation, and MemSafe's optimizations can be turned off for programs where whole-program analysis is infeasible. Section 6 presents performance overheads with and without using whole-program analysis.

Another limitation is the requirement that external libraries be compiled with MemSafe in order to achieve complete safety. This limitation is necessary not only for ensuring the safety of the libraries but also for correctly propagating metadata to the applications. However, for various compatibility issues, it is not uncommon for systems to maintain multiple versions of pre-compiled libraries, and in such cases, it may be reasonable to accommodate safe and unsafe library versions as well. As mentioned in Section 3.5, the design of MemSafe's metadata facilities enables both safe and unsafe libraries to link with a safe application. When linking a safe application with unsafe libraries, MemSafe ensures that externally defined pointers inherit the metadata of the *invalid* pointer. Although this may result in some false positives, we have observed this to be uncommon in practice.

6. RESULTS

Having discussed the prototype implementation of the MemSafe compiler, this section presents a thorough evaluation of MemSafe's approach for ensuring the memory safety of C programs at runtime. Specifically, this section evaluates (1) MemSafe's completeness by demonstrating that it is capable of detecting known memory safety violations in several large programs, (2) MemSafe's cost by measuring its runtime overhead on a variety of programs and comparing this slowdown with

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 III. MemSafe's ability to detect all memory violations in the BugBench [27] programs. Program size is measured in lines of code (LOC) and the number of static dereferences.

that of prior methods, and (3) the effectiveness of MemSafe's static analysis by measuring quantities related to MemSafe's data-flow representation and the number of required checks and performed optimizations.

In performing the above evaluation, it will be demonstrated that MemSafe is compatible with a variety of C programs and that it does not require any source code modifications or programmer intervention. Additionally, it will be shown that MemSafe's key contributions—namely, the modeling of temporal violations as spatial violations, the use of a hybrid metadata representation, and MemSafe's data-flow representation—are effective tools for reducing the runtime cost of dynamically ensuring memory safety.

6.1. *Effectiveness in Detecting Errors*

To provide evidence of its completeness and ability to detect real errors, MemSafe was evaluated on programs containing known memory errors from the BugBench [27] suite of programs. BugBench is a collection of programs containing various documented software bugs that was expressly created to evaluate the effectiveness of error detection tools. Table III shows that MemSafe is capable of detecting all known memory errors in six programs from BugBench. BugBench programs that were excluded from Table III include programs that only contain errors that are *not* related to spatial or temporal safety (e.g., memory leaks and race conditions). Thus, the programs in Table III are representative of all memory safety violations in BugBench. The size of each program is given in lines of code (LOC) and the number of static dereferences.

MemSafe's ability to detect real-world memory violations was further validated by it compiling two large applications and successfully detecting the known memory errors. Table IV summarizes the memory safety violations detected by MemSafe in various versions of the Apache HTTP server [25] and the GNU Core Utilities [26] software package. The Apache HTTP server is a widely-used open source web server, and the GNU Core Utilities is a GNU software package that provides the basic text, shell, and file utilities (e.g., `cat`, `expr`, and `cp`) common among virtually all Unix-like operating systems. To reproduce the known errors in these programs, the online development archive and bug database of each was consulted in order to identify particular versions of the software that contain memory safety violations and the runtime conditions necessary for producing them. Having discovered the known violations, MemSafe was then used to compile each version of the software, and the programs were executed to verify that the inserted runtime checks successfully detected the violations. The size of each program in Table IV is given in lines of code.

6.2. *Runtime Performance*

MemSafe's increase in runtime and memory consumption was measured on a total of 30 programs from the Olden [28], PtrDist [4] and SPEC [43] benchmark suites. Programs from Olden and PtrDist suites are known for being memory allocation intensive, while those from SPEC are larger and generally more computationally intensive. The programs were executed on a system running the Ubuntu 8.04 LTS Desktop operating system with Linux kernel version 2.6.24. The system contains a single 3GHz Pentium 4 processor and 2GB of main memory. Program execution times were

Application	Version	LOC	Component	Detected Violation
Apache HTTP Server[*]	2.0.39	262487	mod_ext_filter	null dereference
	2.0.40	266741	mod_env	null dereference
	2.0.46	282682	mod_ssl	dangling pointer
	2.0.48	284627	mod_ssl	null dereference
	2.0.50	262266	mod_rewrite	buffer overflow
	2.0.52	263513	mod_auth_ldap	null dereference
	2.0.54	265243	mod_auth_ldap	null dereference
	2.0.59	267783	mod_rewrite	uninitialized pointer
	2.2.0	310283	mod_proxy	double free
	2.2.2	311235	mod_dbd	double free
	2.2.6	314531	mod_proxy_balancer	buffer overflow
	2.2.8	316713	mod_log_config	null dereference
	2.2.9	332867	mod_ldap	null dereference
	2.3.4	206590	mod_proxy	null dereference
GNU Core Utilities[†]	5.2.1	103659	fts	double free
	5.2.1	103659	copy	buffer overflow
	5.2.1	103659	who	buffer overflow
	5.3.0	107147	cut	double free
	5.9.0	112781	regexec	buffer overflow
	6.10	69491	mkfifo	null dereference
	6.10	69491	mknod	null dereference
	6.10	69491	ptx	buffer overflow

Table IV. Known real-world memory violations from the Apache HTTP Server [25] and GNU Core Utilities [26] that MemSafe sucessfuly detects. Program size is measured in lines of code (LOC).

[*]Source of violations: https://issues.apache.org/bugzilla/
[†]Source of violations: http://lists.gnu.org/archive/html/bug-coreutils/

determined by taking the lowest of three times obtained using the GNU/Linux `time` command, and memory usage was measured by instrumenting all allocation and deallocation instructions to record the number of allocated objects and their sizes. Due in part to LLVM's research-quality implementation of Andersen's analysis, the current implementation of MemSafe is not yet robust enough to compile the entire set of SPEC benchmarks. The results presented in this section pertain to the subset that MemSafe correctly compiles.

Increase in Runtime Table V summarizes the runtime and memory consumption overheads of MemSafe's fully optimized approach. While this section discusses the increase in runtime of programs compiled with MemSafe's safety checks, the discussion of their increase in memory consumption is deferred until Section 6.2.

The "Runtime" and "Slowdown" columns of Table V show that MemSafe ensured complete spatial and temporal safety for all 30 programs with an average overhead of 88%. In general, MemSafe's overhead was observed to be comparable to that of CCured [22]: on the allocation intensive Olden benchmarks, MemSafe's overhead was 29% versus CCured's 30%, and on CCured's entire set of reported benchmarks, MemSafe overhead was 69% versus CCured's 80%. Not including *bc* (on which CCured's overhead was particularly high) reduces these to 65% and 30%, respectively. While the runtime cost of MemSafe is similar to that of CCured, MemSafe does not incur the drawbacks associated with the use of CCured—the need for manual modifications and the compatibility issues arising from the use of "fat pointer." Due to CCured's need for manual code modifications, results for CCured on additional programs were not obtained.

Additionally, MemSafe demonstrated a significant and consistent improvement over the reported performance of MSCC [6], 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 (29%) was roughly 1/4 that of MSCC (133%), and on the entire set of MSCC's reported benchmarks, MemSafe's overhead (44%) was roughly 1/3 that of MSCC (137%).

Benchmark		**Size**		**Runtime (s)**		**Memory (MB)**		**Slowdown**		
Suite	Program	LOC	Derefs	Base	MemSafe	Base	MemSafe	MemSafe	CCured	MSCC
Olden	bh	2073	284	4.64	5.34	14.67	17.70	1.15	1.44	2.82
	bisort	350	76	1.31	1.59	214.73	275.12	1.21	1.45	1.76
	em3d	688	187	5.11	6.95	54.84	55.15	1.36	1.87	1.79
	health	502	236	0.47	0.70	36.63	53.61	1.48	1.29	2.72
	mst	428	57	0.31	0.36	24.90	24.92	1.17	1.06	1.76
	perimeter	484	258	0.36	0.48	37.33	80.12	1.34	1.09	3.37
	power	622	285	4.09	4.70	2.91	5.14	1.15	1.07	1.22
	treeadd	245	26	0.38	0.59	48.00	182.92	1.55	1.10	3.23
	tsp	582	194	3.83	4.44	144.00	278.65	1.16	1.15	2.28
	average	*716*	*178*	*2.28*	*2.79*	*64.22*	*108.15*	*1.29*	*1.30*	*2.33*
PtrDist	anagram	650	113	1.56	2.96	0.24	0.25	1.90	1.43	–
	bc	7297	3927	1.34	3.15	0.72	0.72	2.35	9.91	–
	ft	1766	246	2.04	3.61	3.24	9.20	1.77	1.03	–
	ks	782	239	1.54	2.97	0.02	0.09	1.93	1.11	–
	yacr2	3986	1000	1.96	5.98	29.88	30.17	3.05	1.56	–
	average	*2896*	*1105*	*1.69*	*3.73*	*6.82*	*8.09*	*2.20*	*3.01*	–
SPEC'95	099.go	29246	16632	0.62	1.26	0.00	0.01	2.03	1.22	2.60
	129.compress	1934	232	0.01	0.02	0.00	0.00	2.20	1.17	1.85
	130.li	7597	4905	0.06	0.12	19.91	19.91	1.93	1.70	–
	147.vortex	67202	25135	0.00	0.00	96.41	96.41	–	–	–
	average	*26495*	*11726*	*0.17*	*0.35*	*29.08*	*29.08*	*2.05*	*1.36*	–
SPEC'00	164.gzip	8605	1499	20.72	43.10	187.93	187.94	2.08	–	1.46
	175.vpr	17729	5386	8.34	16.26	44.35	44.37	1.95	–	3.53
	181.mcf	2412	534	11.34	21.89	99.86	99.46	1.93	–	2.85
	186.crafty	24975	7579	14.93	34.94	7.09	7.14	2.34	–	–
	255.vortex	67213	25134	3.96	8.28	96.46	96.46	2.09	–	–
	256.bzip2	4649	1254	22.33	45.55	191.95	191.96	2.04	–	–
	300.twolf	20459	11741	7.52	15.34	6.39	12.74	2.04	–	–
	average	*20863*	*7590*	*12.73*	*26.48*	*90.58*	*91.49*	*2.07*	–	–
SPEC'06	401.bzip2	8293	4013	6.20	17.55	855.79	855.79	2.83	–	–
	445.gobmk	197215	27614	0.29	0.66	28.50	28.66	2.27	–	–
	456.hmmr	35992	7582	7.82	17.52	59.82	59.89	2.24	–	–
	458.sjeng	13847	5832	10.12	22.26	179.63	179.64	2.20	–	–
	473.astar	5842	1873	0.00	0.00	313.15	313.15	–	–	–
	average	*52238*	*9383*	*4.89*	*11.60*	*287.38*	*287.43*	*2.39*	–	–
Average		**18394**	**5136**	**4.77**	**9.62**	**93.31**	**106.92**	**1.88**	**–**	**–**

Table V. Dynamic results with whole-program analysis. Program size is measured in lines of code and the number of static dereferences, runtime is measured in seconds, and memory consumption is measured in megabytes. Slowdown is computed as the ratio of the execution time of the instrumented program to that of the original program. Slowdown for MemSafe with all optimizations is shown in comparison with CCured [22] and MSCC [6] where results are available.

In order to provide a direct comparison with MSCC (instead of relying on published results) on the same computer hardware, an attempt was made to compile our entire set of benchmarks with MSCC. However, perhaps due to MSCC having not been actively maintained since its publication, it was found to be difficult to compile MemSafe's entire set of 30 benchmarks with MSCC. Figure 5 compares the slowdown of MemSafe's fully optimized approach for spatial and temporal safety with that of MSCC on the set of benchmarks MSCC compiled correctly. MemSafe's average overhead for these benchmarks (74%) was roughly 1/6 that of MSCC (486%). While these results show a dramatic increase in runtime overhead for MSCC, the overall trend is similar to the reported results for MSCC shown in Table V. Comparisons with additional methods on the Olden benchmarks is presented in Section 1.

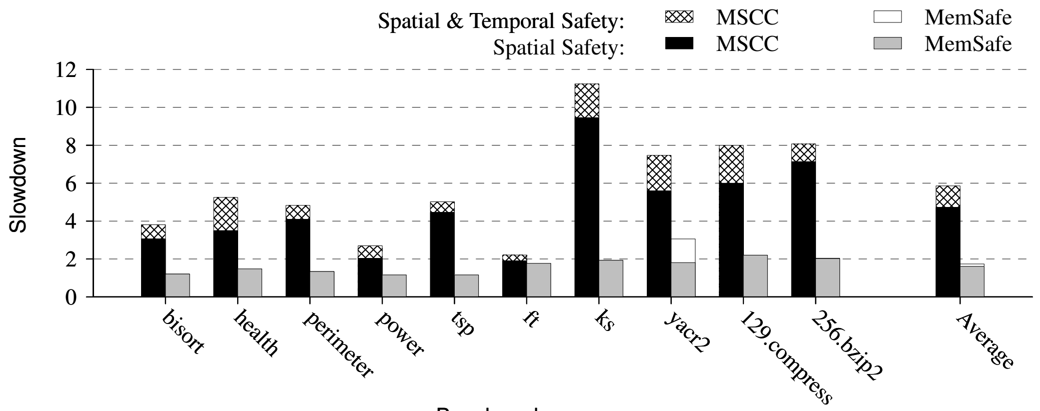


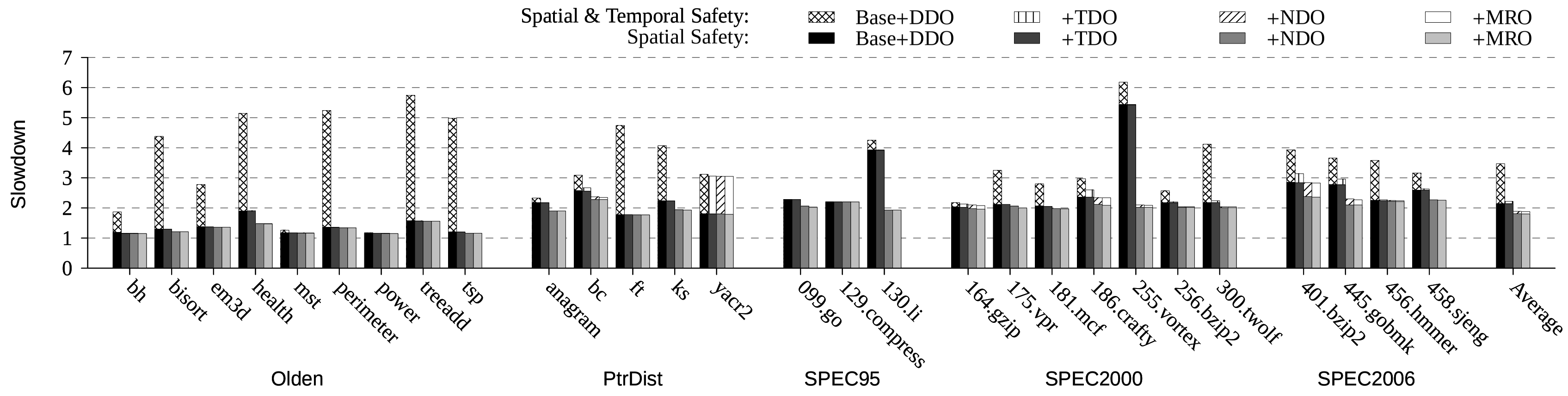
Figure 5. Runtime Comparison With MSCC. Slowdown for MemSafe and MSCC [6] is shown for spatial and temporal and spatial-only safety.

MemSafe's optimized approach improves the runtime cost required for memory safety in comparison to that of prior work for the following reasons: (1) MemSafe's data-flow representation enables performance-enhancing optimizations that reduce overhead from 253% to 88% (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 10% 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 deallocate all dynamically allocated memory at once before terminating. Thus, by determining that there is no control-flow path from the deallocation to other points in the program, MemSafe is able to eliminate the propagation of the metadata associated with the *invalid* pointer and remove all object bounds checks. Deallocated memory at the end of a program is a common programming style when objects are required to have an unlimited lifespan or when memory reallocation is not needed.

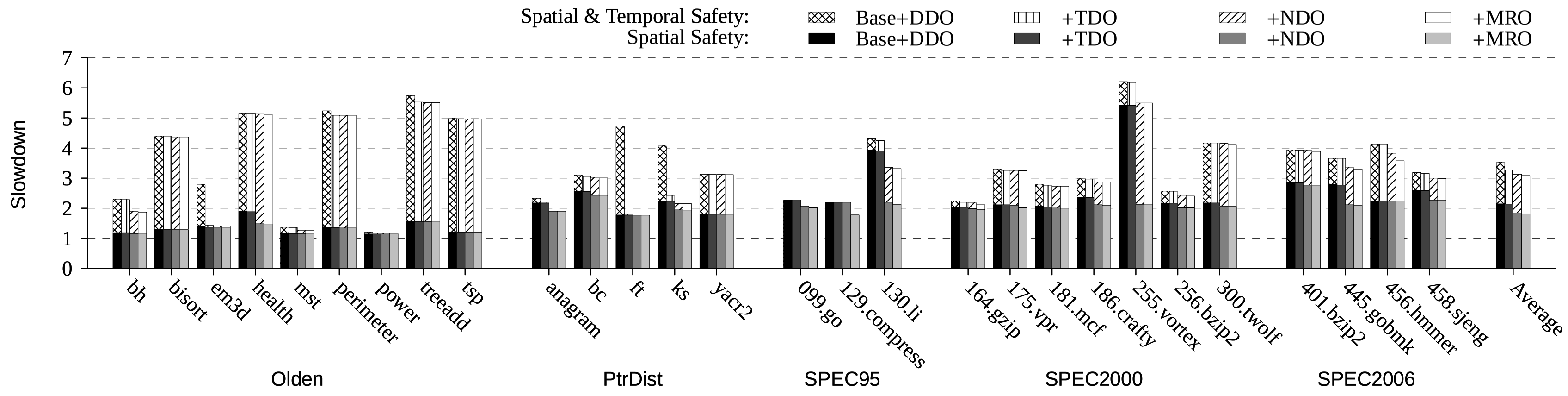
Increase in Memory Consumption The "Memory" column of Table V reports the memory consumption of each program when compiled with LLVM MemSafe. MemSafe ensured complete spatial and temporal safety for all 30 programs with an average increase in memory of 13.61MB, which is equal to 48.60% of the programs' original memory requirements. MemSafe's average memory consumption overhead is significantly higher for the Olden (73.52%) and PtrDist (130.32%) benchmark suites compared to the three SPEC (8.46%) benchmark suites. Since MemSafe requires the metadata of each allocated object to be mapped in the object metadata facility, allocation intensive programs, like those in the Olden and PtrDist suites, require more memory for maintaining metadata than computationally intensive programs. The memory required for MemSafe's metadata is determined by the number of allocated objects rather than their total size.

Effectiveness of Optimizations Figure 6(a) shows that MemSafe's optimizations and whole-program analysis are effective tools for reducing the runtime overhead required for ensuring memory safety. Shown in the "Average" histogram, MemSafe's optimizations reduced its average runtime overhead from 253% to 88%. Since the optimization for dominated dereferences (DDO) is minimally effective, it is presented in Figure 6(a) as the baseline. The optimization for temporally-safe dereferences (TDO) reduced overhead by 102%, and the optimization for non-incremental dereferences (NDO) reduced overhead by 37%. Combined with the optimization for unused metadata, which is included with both, NDO and TDO accounted for the greatest reduction in overhead. The optimization for monotonically addressed ranges (MRO) was marginally effective and reduced overhead by only 1%.

Figure 6(b) shows the effectiveness of MemSafe's optimizations without utilizing whole-program analysis. When restricted to not use interprocedural information, MemSafe's optimizations reduced the overhead from 253% to 209%. TDO reduced overhead by 11%, NDO reduced overhead by 7%,



(a) Optimization Effectiveness With Whole-Program Analysis



(b) Optimization Effectiveness Without Whole-Program Analysis

Figure 6. Optimization Effectiveness. Slowdown with whole-program analysis (a) and without whole-program analysis (b) 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).

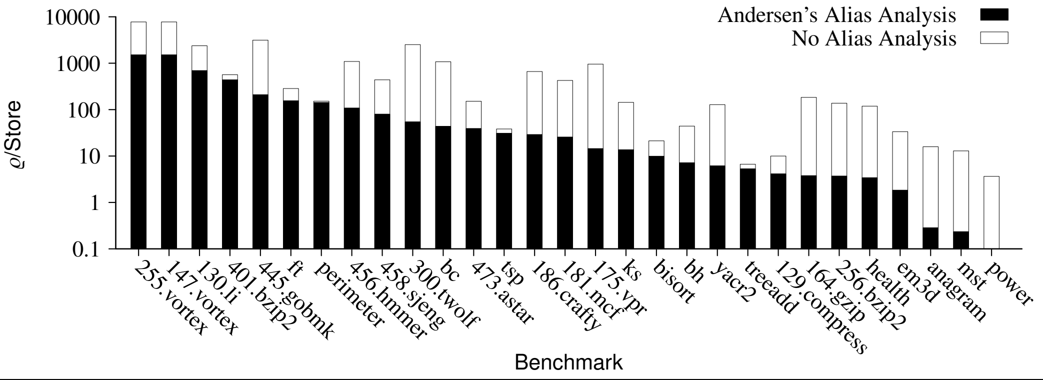


Figure 7. Effect of aliasing. The average number of ϱ-nodes modifyable by each pointer store is shown with Andersen's alias analysis versus no alias analysis.

and MRO reduced overhead by an additional 2%. Hence, MemSafe's average overhead with separate compilation was 209% versus 88% with whole-program analysis. MemSafe's seemingly large improvement in runtime overhead when given the ability to perform interprocedural optimizations is not by chance: By representing memory deallocation and pointer stores as direct assignments, MemSafe makes whole-program optimizations much more effective. Thus, MemSafe's overheads are lower than those of existing methods that cannot benefit in this way.

Additional Cost of Temporal Safety Figure 6(a) also quantifies the additional cost required for MemSafe to ensure temporal safety. The "Average" histogram shows that MemSafe's overhead for both spatial and temporal safety (88%) is comparable to the runtime overhead MemSafe requires for ensuring spatial safety (80%). Thus, for the 30 programs tested, MemSafe ensured complete temporal safety with a modest 10% increase in the average overhead for achieving spatial safety.

Finally, the additional cost of ensuring temporal safety with MemSafe is a significant reduction in the cost of achieving temporal safety with MSCC. On MSCC's set of reported benchmarks, the additional cost of ensuring temporal safety with MemSafe (1%) is a reduction in the additional runtime cost required for MSCC to ensure tempral safety (62%) by a factor of 62. For the set of programs that were successfully compiled with MSCC on the same platform as MemSafe (shown in Figure 5), the additional cost of ensuring temporal safety with MemSafe (13%) is a reduction in the additional runtime cost required for MSCC to ensure temporal safety (114%) by a factor of nearly 9. Thus, the difference in the additional overhead required for MemSafe achieve temporal safety is further evidence that by modeling temporal errors as spatial errors, MemSafe's optimizations are effective tools for reducing the cost of temporal safety.

6.3. Static analysis

The "Checks," "Opts.," and "DFPG" columns of Table VI describe results related to MemSafe's whole-program analysis. First, the "Checks" column shows the static number of required checks, organized by check type, as a percentage of the static number of total pointer dereferences. Second, the "Opts." column shows the static number of checks (i.e., `PBC` and `OBC`) that were eliminated by MemSafe's optimizations, organized by optimization type. Finally, the "DFPG" column summarizes the DFPG with the percentage of nodes reachable from the node representing the *invalid* pointer and with the $\varrho/store$ quantity. 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. Thus, these two quantities are a static estimate of the uncertainty in pointer data-flow. Figure 7 demonstrates that Andersen's alias analysis [42] is often capable of reducing $\varrho/store$ by several orders of magnitude.

Finally, Figure 8 shows MemSafe's compile-time slowdown for each program. Slowdown is computed as the ratio of the compilation time required by MemSafe to that of the base LLVM

Benchmark		Size		Checks (%)			Opts. (%)			DFPG	
Suite	Program	LOC	Derefs	PBC	OBC	MARC	DDO	TDO	NDO	Invalid (%)	ϱ/Store
Olden	bh	2073	284	19.01	0.00	1.76	27.11	70.07	52.11	0.00	7.19
	bisort	350	76	9.21	0.00	0.00	35.53	64.47	55.26	0.00	9.93
	em3d	688	187	31.55	0.00	3.21	7.49	82.35	57.75	0.00	1.84
	health	502	236	22.46	0.00	0.00	15.25	84.32	62.29	0.00	3.42
	mst	428	57	19.30	0.00	5.26	12.28	77.19	63.16	0.00	0.24
	perimeter	484	258	19.77	0.00	0.00	0.00	100.00	80.23	0.00	142.61
	power	622	285	37.89	0.00	0.00	22.46	75.79	39.65	0.00	0.00
	treeadd	245	26	39.47	0.00	0.00	0.00	100.00	60.53	0.00	5.33
	tsp	582	194	15.98	0.00	0.00	31.96	68.04	52.06	0.00	31.07
	average	*716*	*178*	*23.85*	*0.00*	*1.14*	*16.90*	*80.25*	*58.12*	*0.00*	*22.40*
PtrDist	anagram	650	113	33.63	0.00	0.00	21.24	52.21	45.13	0.00	0.29
	bc	7297	3927	15.76	3.79	1.12	32.85	57.70	50.27	8.99	43.62
	ft	1766	246	30.49	0.00	0.00	24.80	70.33	44.72	3.92	155.43
	ks	782	239	28.03	0.00	0.00	27.62	49.37	44.35	0.00	13.71
	yacr2	3986	1000	34.70	5.00	3.90	34.60	51.20	26.80	4.85	6.15
	average	*2896*	*1105*	*28.52*	*1.76*	*1.00*	*28.22*	*56.16*	*42.25*	*3.55*	*43.84*
SPEC'95	099.go	29246	16632	57.54	0.00	5.96	25.44	11.06	11.06	0.00	–
	129.compress	1934	232	16.38	0.00	4.74	40.95	37.93	37.93	0.00	4.13
	130.li	7597	4905	14.92	0.00	0.06	27.26	70.89	57.76	0.00	694.18
	147.vortex	67202	25135	7.35	0.25	0.04	34.36	64.25	58.25	13.18	1511.59
	average	*26495*	*11726*	*24.05*	*0.06*	*2.70*	*32.00*	*46.03*	*41.25*	*3.30*	*736.63*
SPEC'00	164.gzip	8605	1499	21.35	0.47	4.34	44.70	30.02	29.62	0.96	3.79
	175.vpr	17729	5386	21.59	0.32	2.32	22.08	71.67	54.01	7.07	14.52
	181.mcf	2412	534	7.30	0.19	1.31	23.60	69.10	67.79	9.61	25.74
	186.crafty	24975	7579	38.49	11.11	0.01	15.64	46.15	45.86	23.77	29.08
	255.vortex	67213	25134	7.35	0.25	0.04	34.37	64.24	58.24	13.18	1511.61
	256.bzip2	4649	1254	43.46	0.40	3.03	41.23	14.83	12.28	1.12	316.95
	300.twolf	20459	11741	16.73	0.88	0.25	20.82	72.24	62.21	9.39	3.71
	average	*20863*	*7590*	*22.32*	*1.95*	*1.61*	*28.92*	*52.61*	*47.14*	*9.30*	*54.72*
SPEC'06	401.bzip2	8293	4013	17.29	8.07	1.20	12.09	75.18	69.42	27.83	440.48
	445.gobmk	197215	27614	38.51	8.52	1.96	19.49	41.80	40.05	12.70	209.98
	456.hmmr	35992	7582	27.13	8.76	1.58	18.40	65.63	52.89	14.21	108.66
	458.sjeng	13847	5832	26.29	2.54	0.22	28.21	48.47	45.28	18.37	80.29
	473.astar	5842	1873	7.90	2.62	0.32	19.38	75.71	72.40	18.91	39.38
	average	*52238*	*9383*	*23.42*	*6.10*	*1.06*	*19.51*	*61.36*	*56.01*	*18.40*	*133.89*
Average		**18394**	**5136**	**24.23**	**1.77**	**1.42**	**24.04**	**62.07**	**50.31**	**6.48**	**177.68**

Table VI. Static Results with Whole-program Analysis. Program size is measured in lines of code and the number of static dereferences. 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.

compiler using default optimizations. In general, the compile-time requirements of MemSafe are modest. For 28 of the benchmarked programs, MemSafe was able to ensure memory safety with an increase in compile-time by less than a factor of two. The compilation time required by *ft* and *130.li* surpassed this threshold due to the time required for inserting ϱ-functions. Despite these two programs, MemSafe's average increase in compile-time over all 30 benchmarks was 62%.

7. RELATED WORK

Most prior techniques related to the enforcement of memory safety were presented in Section 2 after having described the various types of spatial and temporal memory errors. Therefore, this section

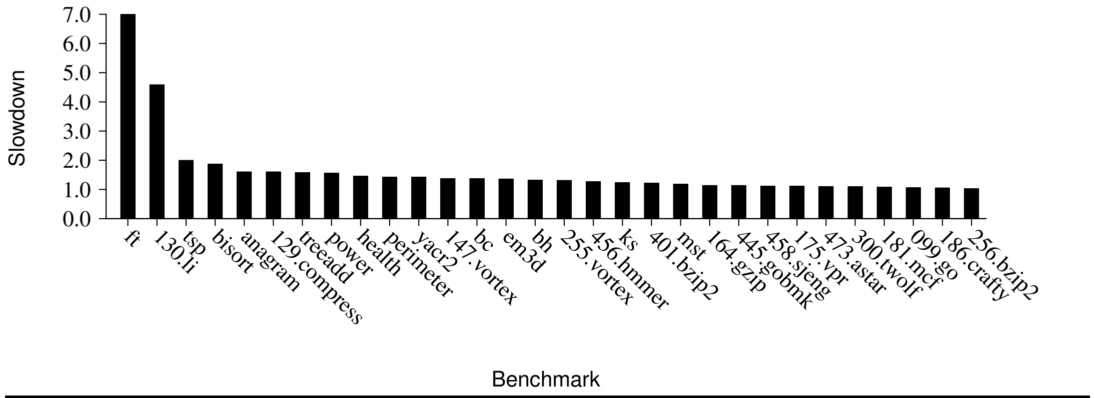


Figure 8. Compile-time slowdown. For each program, slowdown is computed as the ratio of the compilation time required by MemSafe to that of the base LLVM compiler using the default set of optimizations.

will not repeat that content, but it will discuss additional details for methods capable of detecting both spatial and temporal violations as well as techniques that can only detect one type of error. This section also presents a discussion of previous work related to MemSafe's data-flow analysis.

7.1. *Spatial and Temporal Safety*

While generally not enforcing complete memory safety, several methods are capable of detecting both spatial and temporal errors. Purify [20] operates on binaries, but only ensures the safety of heap-allocated objects. Yong and Horwitz [21] present a similar approach and improve its cost with static analysis, but this method only checks store operations. Safe C [4] ensures complete safety but is incompatible due to its use of fat-pointers. Patil and Fischer [5] address these issues by maintaining disjoint metadata and performing checks in a separate "shadow process," but this requires an additional CPU. CCured [22] 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 [6] is highly compatible and complete but is unable to handle some downcasts. Fail-Safe C [23] maintains complete compatibility with ANSI C but incurs significant runtime overhead. Finally, Clause et al. [44] describe an efficient technique for detecting memory errors, but it requires custom hardware.

7.2. *Spatial Safety*

Methods that primarily detect bounds violations are numerous. Notable is the work by Jones and Kelly [15] since it maintains compatibility with pre-compiled libraries. However, this method has high overhead and results in false positives. Ruwase and Lam [16] extend this method to track out-of-bounds pointers to avoid false positives. Additionally, Dhurjati and Adve [17] utilize Automatic Pool Allocation [34] to improve cost, and Akritidis et al. [33] 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 [45] 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 [7] is a related technique that records pointer metadata in disjoint data structures similar to MemSafe's representation. However, while these methods ensure complete spatial safety, they do not ensure temporal safety, and HardBound requires custom hardware to achieve low overhead.

7.3. *Temporal Safety*

Few methods are designed primarily for detecting temporal violations. Dhurjati and Adve [46] describe a technique based on the Electric Fence [47] `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 [34] and a customized address mapping. However, this method does not detect spatial violations and only detects temporal violations of heap objects. CETS [48] inserts temporal safety checks before pointer dereferences and utilizes an efficient lock-and-key mechanism, instead of hash tables, for accessing the required temporal metadata. However, this method also does not detect spatial violations and must be combined with an existing spatial safety mechanism to guarantee complete temporal safety.

7.4. *Software Debugging Tools*

While not intended for deployment in production-quality applications, automated debugging tools can be used to detect memory errors during software development and testing. Valgrind [49] is a heavyweight dynamic binary instrumentation framework providing the Memcheck [50] tool for debugging memory accesses and leaks, and Mudflap [18] is a compiler approach for debugging memory accesses implemented in the GCC [51] compiler infrastructure. However, these tools are incapable of ensuring complete spatial and temporal memory safety and incur significant runtime overheads. For example, both Memcheck and Mudflap are unable to detect spatial safety errors where an out-of-bounds pointer to one object happens to fall within bounds of another object. They are also unable to detect temporal safety errors when the runtime system reallocates memory to a previously deallocated location. Moreover, Memcheck does not aim to ensure spatial or temporal safety for stack-allocated objects and increases runtime by a factor of 10–30.

7.5. *Other Methods of Memory Protection*

Several methods utilize software checks to enforce various security-related policies. Abadi et al. [29] describe a technique to prevent software attacks by enforcing control-flow integrity. Similarly, Castro et al. [30] enforce data-flow integrity with an analysis based on reaching definitions, and WIT [31] 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 [52] is a dynamic memory allocator capable of preventing many heap-related errors. It uses random object placement within a larger-than-normal heap to prevent invalid deallocations and probabilistically avoid heap buffer overflows. However, this method is incapable of ensuring complete spatial and temporal memory safety.

Other methods seek to provide minimal memory protection guarantees to programs executed on systems lacking hardware virtual memory. Simpson et al. [53] developed a low-overhead method for achieving memory segmentation using compiler-inserted runtime checks. Like paging, segmented virtual memory is a common approach for providing coarse-grained memory protection. In another method, Biswas et al. [54] developed a technique for avoiding out-of-memory errors with compiler-inserted runtime checks, memory reuse, and the compression of unused data. Finally, Middha et al. [55] developed a similar method for avoiding out-of-memory errors in embedded systems by sharing stack space among the executing tasks of multitasking workloads.

7.6. *SSA Extensions*

Various methods have extended SSA [24] to incorporate alias information. IPSSA [56] is an interprocedural, Gated SSA [57] that uses alias analysis to replace indirect stores with ϕ-like functions whose semantics are similar to the ϱ-function. However, IPSSA represents *all* indirect stores as direct assignments, whereas MemSafe's ϱ-function is only used for *pointer* stores. Other extensions include the χ- and μ-extensions [58], which model may-def and may-use information, but unlike the ϱ-function, do not keep track of the defining values. Finally, Cytron and Gershbein [59] describe a demand-driven algorithm for incrementally incorporating alias information with SSA to avoid a large increase in program size.

8. 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 data-flow 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 as well as 2 large open source applications. Additionally, it ensured complete safety with an average overhead of 88% on 30 programs commonly used for 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. Simpson MS, Barua RK. MemSafe: Ensuring the spatial and temporal safety of C at runtime. *Proceedings of the Tenth IEEE International Working Conference on Source Code Analysis and Manipulation*, 2010; 199–208.
2. Anderson JP. Computer security technology planning study. *Technical Report ESD-TR-73-51, Vol. II*, United States Air Force Electronic Systems Division 1972.
3. U.S. Computer Emergency Readiness Team. *US-CERT Vulnerability Notes Database*. http://www.kb.cert.org/vuls/.
4. Austin TM, Breach SE, Sohi GS. Efficient detection of all pointer and array access errors. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1994; 290–301.
5. Patil H, Fischer C. Low-cost, concurrent checking of pointer and array accesses in C programs. *Software: Practice & Experience* 1997; **27**(1):87–110.
6. Xu W, DuVarney DC, Sekar R. An efficient and backwards-compatible transformation to ensure memory safety of C programs. *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2004; 117–126.
7. Nagarakatte S, Zhao J, Martin MMK, Zdancewic S. SoftBound: Highly compatible and complete spatial memory safety for C. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009; 245–258.
8. Jim T, Morrisett JG, Grossman D, Hicks MW, Cheney J, Wang Y. Cyclone: A safe dialect of C. *Proceedings of the USENIX Annual Technical Conference*, 2002; 275–288.
9. Condit JP. Dependent types for safe systems software. PhD Thesis, University of California, Berkeley 2007.
10. Ball T, Rajamani SK. The SLAM project: Debugging system software via static analysis. *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002; 1–3.
11. Beyer D, Henzinger TA, Jhala R, Majumdar R. The software model checker BLAST: Applications to software engineering. *International Journal on Software Tools for Technology Transfer* 2007; **9**(5-6):505–525.
12. Engler D, Chen DY, Hallem S, Chou A, Chelf B. Bugs as deviant behavior: A general approach to inferring errors in systems code. *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, 2001; 57–72.
13. Das M, Lerner S, Seigle M. ESP: Path-sensitive program verification in polynomial time. *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, 2002; 57–68.
14. Dillig I, Dillig T, Aiken A. Static error detection using semantic inconsistency inference. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007; 435–445.
15. Jones RWM, Kelly PHJ. Backwards-compatible bounds checking for arrays and pointers in C programs. *Proceedings of the 3rd International Workshop on Automatic Debugging*, 1997; 13–26.
16. Ruwase O, Lam MS. A practical dynamic buffer overflow detector. *Proceedings of the Network and Distributed System Security Symposium*, 2004; 159–169.
17. Dhurjati D, Adve V. Backwards-compatible array bounds checking for C with very low overhead. *Proceedings of the 28th International Conference on Software Engineering*, 2006; 162–171.
18. Eigler FC. Mudflap: Pointer use cheking for C/C++. *Proceedings of the GCC Developers Summit 2003*, 2003; 57–70.
19. Criswell J, Lenharth A, Dhurjati D, Adve V. Secure virtual architecture: A safe execution environment for commodity operating systems. *Proceedings of the 21st ACM Symposium on Operating Systems Principles*, 2007; 351–366.
20. Hastings R, Joyce B. Purify: A tool for detecting memory leaks and access errors in C and C++ programs. *Proceedings of the USENIX Winter Technical Conference*, 1992; 125–138.
21. Yong SH, Horwitz S. Protecting C programs from attacks via invalid pointer dereferences. *Proceedings of the 9th European Software Engineering Conference Held Jointly With the 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2003; 307–316.
22. Necula GC, Condit J, Harren M, McPeak S, Weimer W. CCured: Type-safe retrofitting of legacy software. *ACM Transactions on Programming Languages and Systems* 2005; **27**(3):477–526.
23. Oiwa Y. Implementation of the memory-safe full ANSI-C compiler. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009; 259–269.
24. Cytron R, Ferrante J, Rosen BK, Wegman MN, Zadeck FK. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems* 1991; **13**(4):451–490.

25. Apache Software Foundation. *Apache HTTP Server*. http://httpd.apache.org/.
26. GNU Project. *GNU Core Utilities*. http://www.gnu.org/software/coreutils/.
27. Lu S, Li Z, Qin F, Tan L, Zhou P, Zhou Y. BugBench: Benchmarks for evaluating bug detection tools. *Workshop on the Evaluation of Software Defect Detection Tools*, 2005.
28. Rogers A, Carlisle MC, Reppy JH, Hendren LJ. Supporting dynamic data structures on distributed-memory machines. *ACM Transactions on Programming Languages and Systems* 1995; **17**(2):233–263.
29. Abadi M, Budiu M, Erlingsson Ú, Ligatti J. Control-flow integrity. *Proceedings of the 12th ACM Conference on Computer and Communications Security*, 2005; 340–353.
30. Castro M, Costa M, Harris T. Securing software by enforcing data-flow integrity. *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, 2006; 147–160.
31. Akritidis P, Cadar C, Raiciu C, Costa M, Castro M. Preventing memory error exploits with WIT. *IEEE Symposium on Security and Privacy*, 2008; 263–277.
32. Dhurjati D, Kowshik S, Adve V. SAFECode: Enforcing alias analysis for weakly typed languages. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2006; 144–157.
33. Akritidis P, Costa M, Castro M, Hand S. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. *Proceedings of the 18th USENIX Security Symposium*, 2009.
34. Lattner C. Macroscopic datastructure analysis and optimization. PhD Thesis, University of Illinois, Urbana-Champaign 2005.
35. Boehm HJ, Weiser M. Garbage collection in an uncooperative environment. *Software: Practice & Experience* 1988; **18**(9):807–820.
36. Bacon DF, Cheng P, Grove D. Garbage collection for embedded systems. *Proceedings of the 4th ACM International conference on Embedded Software*, 2004; 125–136.
37. Zorn B. The measured cost of conservative garbage collection. *Software: Practice & Experience* 1993; **23**(7):733–756.
38. Lattner C, Adve V. LLVM: A compilation framework for lifelong program analysis & transformation. *Proceedings of the International Symposium on Code Generation and Optimization*, 2004; 75–87.
39. International Organization for Standardization. *ISO/IEC 9899: Programming Languages—C* 1999.
40. Chandra S, Reps T. Physical type checking for C. *Proceedings of the 1999 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, 1999; 66–75.
41. Bodik R, Gupta R, Sarkar V. ABCD: Eliminating array bounds checks on demand. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2000; 321–333.
42. Andersen LO. Program analysis and specialization for the C programming language. PhD Thesis, University of Copenhagen 1994.
43. Standard Performance Evaluation Corporation. *SPEC CPU Benchmarks*. http://www.spec.org/.
44. Clause J, Doudalis I, Orso A, Prvulovic M. Effective memory protection using dynamic tainting. *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, 2007; 284–292.
45. Devietti J, Blundell C, Martin MMK, Zdancewic S. HardBound: Architectural support for spatial safety of the C programming language. *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2008; 103–114.
46. Dhurjati D, Adve V. Efficiently detecting all dangling pointer uses in production servers. *Proceedings of the International Conference on Dependable Systems and Networks*, 2006; 269–280.
47. Perens B. Electric fence malloc debugger. http://perens.com/FreeSoftware/ElectricFence/.
48. Nagarakatte S, Zhao J, Martin MM, Zdancewic S. Cets: Compiler enforced temporal safety for C. *Proceedings of the 2010 International Symposium on Memory Management*, 2010; 31–40.
49. Nethercote N, Seward J. Valgrind: A framework for heavyweight dynamic binary instrumentation. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2007; 89–100.
50. Seward J, Nethercote N. Using Valgrind to detect undefined value errors with bit-precision. *Proceedings of the USENIX Technical Conference*, 2005; 17–30.
51. GNU Project. *GNU Compiler Collection*. http://gcc.gnu.org/.
52. Berger ED, Zorn BG. DieHard: Probabilistic memory safety for unsafe languages. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2006; 158–168.
53. Simpson M, Middha B, Barua R. Segment protection for embedded systems using run-time checks. *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, 2005; 66–77.
54. Biswas S, Carley T, Simpson M, Middha B, Barua R. Memory overflow protection for embedded systems using run-time checks, reuse, and compression. *ACM Transactions on Embedded Computing Systems* November 2006; **5**(4):719–752.
55. Middha B, Simpson M, Barua R. MTSS: Multitask stack sharing for embedded systems. *ACM Transactions on Embedded Computing Systems* August 2008; **7**(4):46:1–46:37.
56. Livshits VB, Lam MS. Tracking pointers with path and context sensitivity for bug detection in C programs. *Proceedings of the 9th European Software Engineering Conference Held Jointly With the 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2003; 317–326.
57. Ottenstein KJ, Ballance RA, MacCabe AB. The program dependence web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1990; 257–271.
58. Chow F, Chan S, Liu SM, Lo R, Streich M. Effective representation of aliases and indirect memory operations in SSA form. *Proceedings of the International Conference on Compiler Construction*, 1996; 253–267.
59. Cytron R, Gershbein R. Efficient accommodation of may-alias information in SSA form. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1993; 36–45.