Runtime Detection of Temporal Memory Errors*

Kostyantyn Vorobyov, Nikolai Kosmatov, Julien Signoles, and Arvid Jakobsson

CEA, LIST, Software Reliability and Security Laboratory,
PC 174, 91191 Gif-sur-Yvette France
{kostyantyn.vorobyov,nikolai.kosmatov,julien.signoles}@cea.fr
arvid.jakobsson@gmail.com

Abstract. State-of-the-art memory debuggers have become efficient in detecting spatial memory errors – dereference of pointers to unallocated memory. These tools, however, cannot always detect errors arising from the use of stale pointers to valid memory (temporal memory errors). This paper presents an approach to reliable detection of temporal memory errors during a run of a program. This technique tracks allocated memory tagging allocated objects and pointers with tokens that allow to reason about their temporal properties. The technique further checks pointer dereferences and detects temporal (and spatial) memory errors before they occur. The present approach has been implemented in E-ACSL – a runtime verification tool for C programs. Experimentation with E-ACSL using TempLIST benchmark comprising small C programs seeded with temporal errors shows that the suggested technique detects temporal memory errors missed by state-of-the-art memory debuggers. Further experiments with computationally intensive runs of programs from SPEC CPU indicate that the overheads of the proposed approach are within acceptable range to be used during testing or debugging.

Keywords: Runtime analysis · Memory safety · Temporal Memory Error · Shadow memory · Frama-C/E-ACSL

1 Introduction

Low-level memory access and pointer arithmetic make the C programming language an attractive choice for system-level programming. The same features, on the other hand, make it vulnerable to memory access errors. In C, a programmer is permitted to access and modify memory directly using pointers. It is therefore crucial that each pointer is *valid* at the time of its dereference, as dereferencing an invalid pointer can compromise safety and security of a running program, lead to data corruption or crash the program altogether.

A pointer p is said to be valid if it points to an allocated memory object (also referred to as *memory block*) that p is allowed to dereference with respect to ISO C semantics. For instance, at a given program point p is valid if it has

* This work has received funding for the S3P project from French DGE and BPIFrance, and for the AnaStaSec project from the French National Research Agency (ANR, grant ANR-14-CE28-0014).

been made point to dynamically-allocated memory or a variable within an active scope. Once that memory has been deallocated all its pointers become invalid.

Motivation State-of-the-art memory debuggers [3,7,22,20] have become efficient in detecting *spatial* memory errors (accesses to unallocated memory). These tools track memory allocated by a program at runtime and report an error if a program attempts to access a memory location that has not been tracked. Memory debuggers, however, cannot detect uses of invalid pointers in all cases. Consider, for instance, the following code snippet.

```
1 int *p = malloc(sizeof(int));
2 free(p);
3 int *q = malloc(sizeof(int));
4 *p = 1;
```

Execution of the assignment at Line 4 leads to an error. This is because the block initially pointed to by `p` has been deallocated making `p` a stale pointer to unallocated memory. In a typical execution, however, the second call to `malloc` at Line 3 can reuse the freed space and place the newly allocated block at the same location as the first one implicitly making `p` point to the allocated memory. A memory debugger analysing the code observes an access to the allocated memory and does not raise an alarm. Such behaviour, however, is not enforced by ISO C semantics, thus another execution can put the second allocation in a different area. Dereference of a pointer to an allocated memory block that has not been made point to it leads to a *temporal* memory error.

Admittedly, memory debuggers try to avoid temporal errors by using so called quarantine zones, an approach that attempts to allocate new blocks at new addresses and make stale pointers refer to unallocated memory. With this strategy the allocator returns fresh addresses within some allocation buffer. However, once the address space of the buffer is exhausted, the allocator starts to reuse freed memory making occurrence of undetected temporal errors possible.

Most importantly, while the present techniques can catch certain instances of temporal errors on a program's heap, they provide little protection against stack-based temporal issues[1]. Consider the example program in Listing 1.

```
1 int main() {
2     int *p;      // ‘p‘ is invalid
3     {
4         int i = 9;
5         p = &i; // ‘p‘ made point to ‘i‘
6     }           // ‘p‘ becomes invalid
7     int j = 8;  // ‘j‘ is allocated at the same address as ‘i‘
8     *p = 1;     // the assignment changes the value of ‘j‘ through ‘p‘
9 }               // even though ‘p‘ has never been made point to ‘j‘
```

Listing 1. Example program containing a temporal error.

In the assignment `*p = 1` at Line 8 `p` is likely to be invalid temporally but valid spatially. This is because a typical C compiler (such as GCC) allocates `i` and `j` at the same address (since `i` is released before `j` is allocated). The assignment

[1] Recent versions of AddressSanitizer provide experimental support for detecting stack-based temporal errors [18,27]. However, at the time of developing the proposed approach this feature was not enabled in the mainstream version of the tool.

at Line 8 therefore results in a temporal error and changes the value of `j` via `p` even though `p` has never been set to point to `j`.

In a typical execution of the program shown in Listing 1 the temporal error at Line 8 is undetected because allocation and deallocation of program variables is delegated to a compiler. Adopting a different allocating strategy by tampering with automatic allocation is not feasible and has the same problems with detecting temporal errors as the heap-based approach.

Other techniques to detecting temporal memory issues exist. For instance, Safe C [2] detects temporal memory errors by associating memory blocks and pointers with *capabilities* that identify allocations uniquely. A pointer is valid if its capability matches the capability of the memory block it points to. Safe C detects temporal errors, but it also modifies pointer representation that breaks legacy programs. MemSafe [24] detects temporal errors by adding additional assignments that convert temporal issues into spatial. This presents a difficulty during dynamic analysis because pointer addresses can be computed at runtime.

Approach This paper proposes a solution to reliable detection of temporal errors that does not require compiler modifications or customized allocation strategies. Instead, the proposed technique associates tracked allocations with capability-like identifiers that allow to reason about temporal validity of pointers and detect temporal errors.

The present approach has been implemented in the E-ACSL [4] plug-in within the Frama-C [13] source code analysis platform. To assess error detection capabilities of the proposed technique its authors have created a benchmark called TempLIST comprising 15 small C programs seeded with temporal memory errors. Experimentation with E-ACSL using TempLIST programs has shown that the present approach detects errors missed by such state-of-the-art memory debuggers as AddressSanitizer [20] or MemCheck [22]. Further experimentation with programs from SPEC CPU benchmarks [25] has shown an average runtime overhead of 42% compared to conservative checking for spatial violations only.

Contributions

- An approach to runtime detection of temporal memory errors in C programs.
- An implementation of the proposed approach using E-ACSL.
- A C benchmark called TempLIST for evaluating precision of runtime analysers with respect to detection of temporal memory errors.
- Empirical evaluation of the proposed approach using TempLIST and SPEC CPU benchmarks.

2 Temporal Error Detection

The present technique instruments an input program P with statements that monitor its execution and detect temporal errors before they occur. A run of an instrumented program P' tracks every memory block B allocated by P using a metadata storage M that tags B with a unique identifier called an *origin* number and records its bounds. Origin numbers allow to distinguish between different allocations even if these allocations occupy the same address space. Block bounds

make it possible to identify whether a given address belongs to an allocated memory block and compute the start address and the origin number associated with that block. M also tracks program pointers. Each pointer variable (that is, a left-value of a pointer type) is mapped to a referent number – an origin number of the memory block the pointer (should) refer to. During its execution the instrumented program P' observes assignments that involve aliasing and updates referent numbers. Temporal errors are detected by checking pointer validity before dereference. For every pointer p to an allocated memory block B the monitor compares the tracked referent number of p to the origin number of B. A temporal error is detected if these numbers do not match.

Example Consider an execution of the program shown in Listing 1 where `i` and `j` are allocated at the same address. The temporal error occurring at Line 8 is detected via the proposed approach as follows. Declarations of variables `p`, `i` and `j` (allocated on a program's stack) assign unique origin numbers to each of these variables. Assume `p`, `i` and `j` are assigned `1`, `2` and `3` respectively. Direct pointer assignment `p = &i` at Line 5 (which makes `p` point to `i`) stores the origin number of `i` as the referent number of `p`. That is, after the execution of the assignment the referent number of p becomes `2`. Finally, the temporal error at Line 8 is detected by comparing the referent number of `p` with the origin number of the memory block `p` actually points to. Since `p` has the referent number of `2` while pointing to `j` (whose origin number is `3`), the analysis concludes that even though `p` points to an allocated memory block its dereference leads to a temporal error.

The following sections now describe the present technique in greater detail.

2.1 Memory Allocation and Validity

At runtime a C program allocates memory in units called *memory blocks* that typically describe memory allocated automatically for program variables or heap allocations via functions such as `malloc`. A memory block is represented by a contiguous memory region described by its start and end addresses. The start address of a memory block B (denoted $start(B)$) is less than or equal to its end address (denoted $end(B)$). The *length* of B is the difference between the block's end and start addresses increased by one. A memory address a is said to *belong* to a memory block B (and B is said to *contain* a) if a is greater than or equal to the start address of B and less than or equal to its end address.

A pointer p is said to *point* to a memory block B (and B is said to be pointed to by p) if p stores an address belonging to B. To simplify the presentation it is assumed that pointer arithmetics operations use byte offsets. For instance, for pointer p of any type that stores address 0x100, $p + 5$ is 0x105.

Let p be a pointer of type $t*$ storing a memory address a. Let sz denote the number of bytes in the pointed location $*p$ (denoted $sizeof(*p)$). At a given program point p is said to be *spatially* valid if a belongs to an allocated memory block B such that $a + sizeof(*p) - 1 \leq end(B)$. That is, the dereference of p (that reads $sizeof(*p)$ bytes from a) should not exceed the bounds of B. At a given program point p is said to be *temporally* valid if it is valid spatially and B has not been deallocated after p has been made point to B.

The notion of spatial and temporal validity is extended to pointer arithmetic expressions of the form $p + i$, where i is an integer expression and p is a pointer (called the *base* of the pointer expression). The dereference of a pointer expression $p + i$ is *spatially* valid if $p + i$ is spatially valid and addresses given by the base p and the expression itself (i.e., $p + i$) belong to the same allocated memory block, and invalid otherwise. A dereference of a spatially invalid expression $p + i$ leads to a spatial memory error. The dereference of a spatially valid pointer expression $p + i$ is *temporally* valid if p is temporally valid. The dereference of a temporally invalid expression $p + i$ leads to a temporal memory error.

For every allocated memory block B, a unique identifier, called the *origin number* of B and denoted $\alpha(B)$, is associated with B. If a is a memory address belonging to B, then $\alpha(a) = \alpha(B)$. For a pointer variable p (identified by its address $\&p$) that has been made point to block B via address a (e.g. via $p = a$), the *referent number* of p, denoted $\gamma(\&p)$, is defined as the orgin number of a, that is, $\gamma(\&p) = \alpha(a)$.

Dynamic metadata storage M that tracks information about memory allocation during a run of an instrumented program is represented by the substructures M_b and M_p. M_b keeps track of every memory block B allocated by the program at runtime recording its bounds and origin number $\alpha(B)$. M_p tracks program pointers such that each pointer p in M_p, identified by its address $\&p$, is mapped to a referent number $\gamma(\&p)$. Relevant implementation details of M are further discussed in Section 3.

2.2 Key Instrumentations

Tracking allocated memory, updating referent numbers and checking validity of program pointers before their dereference is delegated to specific functions called from the instrumented program. This section discusses details of these functions and shows relevant examples of their use.

Tracking Memory Blocks Tracking allocated memory during a run of an instrumented program is enabled via functions *store* and *delete*. *store*(a, s) records a memory block of s bytes in length and a start address of a to the metadata storage M_b. Calls to *store* are added after declarations of variables or calls to memory allocating functions (e.g., `malloc`). *delete*(a) removes a memory block with start address a from M_b. Calls to *delete* are added at the end of scope of variable declarations or before calls to memory-deallocating functions such as `free`. The present approach also takes into account abrupt scope terminations (e.g., via `goto` statements) and emits additional *store* or *delete* statements as required. An example showing the use of *store* and *delete* in an instrumented program is shown in Section 2.4.

Updating Referent Numbers Referent numbers of pointers are updated using functions *mapOrigin* and *mapReferent*. *mapOrigin*$(\&p, a)$ associates a pointer variable p identified by its address $\&p$ with an origin number of address a (i.e., $\gamma(\&p) = \alpha(a)$). For the case when a lies outside of a program's allocation the referent number of a is mapped to an invalid referent number that indicates that dereferencing of p would result in a spatial memory error. *mapReferent*$(\&p, \&q)$

sets the referent number of p to the referent number carried by another pointer variable $\&q$ identified by its address $\&q$ (that is, $\gamma(\&p) = \gamma(\&q)$).

The following snippet shows instrumentation example using *mapOrigin* and *mapReferent* functions. In the original program (shown to the left), pointer p

Original program

```
1 char *p = malloc(sizeof(char));
2 char *q = p;
```

Instrumented program

```
1 char *p = malloc(sizeof(char));
2 mapOrigin(&p, p);     // γ(&p) = α(p)
3 char *q = p;
4 mapReferent(&q, &p); // γ(&q) = γ(&p)
```

is first assigned an allocated memory block and then q is made point to the same block via p. To update the referent number of p after the first assignment program instrumentation adds function call $mapOrigin(\&p, p)$ that updates the referent of p to the origin number of the allocated block. Further, the instrumentation uses $mapReferent(\&q, \&p)$ to set the referent number of q to that of p because the aliasing of q is performed indirectly using p. This paper further describes rules by which referent numbers are updated in Section 2.3.

Verifying Validity of Pointers Validity of pointers is established by functions *svalid* and *tvalid* that check spatial and temporal validity of pointer expressions.

Function call $svalid(a, base, sz)$ checks spatial validity of pointer expressions and returns a non-zero value if memory addresses a, $a + sz - 1$ and *base* belong to the same allocated memory block and zero otherwise. Consider an expression $p + i$, where p is a pointer and i is an integer expression. Spatial validity of this expression can then be established using $svalid(p + i, p, sizeof(*p))$. That is, *svalid* checks whether $p + i$ belongs to the same allocated block as p (i.e., $\alpha(p + i) = \alpha(p)$) and dereferencing of $p + i$ does not exceed the bounds of the memory block it points to. This approach facilitates detection of overflows into allocated areas, as if p and $p + i$ refer to different blocks, then $p + i$ accesses memory outside of bounds of an allocated object (i.e., memory block pointed to by p). Similarly, if p belongs to unallocated area, then it is invalid and thus using p to access another memory address (even if that address belongs to allocated space) also results in a memory error.

Temporal validity of pointers is determined using function *tvalid*. Function call $tvalid(\&p, a)$, where $\&p$ is the address of a pointer variable p and a is an address, checks whether a belongs to a block p should point to. In other words, $tvalid(p, a)$ returns a non-zero value if $\alpha(a)$ and $\gamma(\&p)$ have the same value and zero otherwise. *tvalid* expects p to be spatially valid. For instance, for an assignment `*(p + 5) = 1`, where `p` is a pointer variable of type `char*`, the instrumentation adds the following assertions:

```
assert(svalid(p + 5, p, sizeof(*p))); // Spatial validity error
assert(tvalid(&p, p));                // Temporal validity error
```

2.3 Rules for Updating Referent Numbers

This section now discusses the key rules for updating pointer referent numbers. Their purpose is to track the origin number of the block that was referred to at the moment *when the reference was initially created.* Assignments considered by

the rules $\mathcal{R}2$, $\mathcal{R}3$ and $\mathcal{R}5$ below create a new reference to a block and thus the block's origin number is stored as a referent number. The rules $\mathcal{R}1$ and $\mathcal{R}4$ copy (and potentially modify) an existing reference, therefore the referent number is copied as well. Rule $\mathcal{R}6$ is derived from rules $\mathcal{R}1$–$\mathcal{R}5$.

($\mathcal{R}1$) **Pointer Assignments** For an assignment of pointers of the form $p = q+i$, where both p and q are pointer variables and i in an integer expression, the referent number of p is set to the referent number of q, i.e., $\gamma(\&p) = \gamma(\&q)$. The referent number of p is set to invalid if either q or $q + i$ points to unallocated memory, or if q and $q + i$ refer to different memory blocks.

($\mathcal{R}2$) **Pointer-from-Address Assignments** Assignments of the form $p = \&lval$, where p is a pointer and $lval$ is a left value expression evaluating to some address a, sets the referent number of p to the origin number of the memory block containing a. That is, $\gamma(\&p) = \alpha(\&lval)$. Assignments involving pointer arithmetic in their left-hand-side parts (e.g., $p = \&lval + i$ or $p = \&lval[i]$, where i is an integer expression) are handled similarly in that $\alpha(\&lval)$ is taken for the referent number of p. For the cases when address $\&lval + i$ evaluates to an address that does not belong to an allocated block or if addresses $\&lval$ and $\&lval + i$ belong to different memory blocks, the referent number of p is set to an invalid value indicating that p is invalid and its dereference will lead to a memory error.

($\mathcal{R}3$) **Pointer-from-non-Pointer Assignments by Cast** If an assignment is of the form $p = rval$, where p is a pointer and $rval$ is a right-value expression of non-pointer type which has been casted to it, then the referent number of p is set to the origin number of $rval$, that is, $\gamma(\&p) = \alpha(rval)$. Notably, such an update is potential source of imprecision as it make it possible to bypass temporal errors. We further discuss this issue in Section 3.

($\mathcal{R}4$) **Non-Pointer Assignments** Non-pointer assignments of the form $lval = rval$, where $lval$ and $rval$ are left- and right-value expressions are handled based on the type $lval$ as follows. If $lval$ is of integral or a floating point type, or if $lval$ is of struct or a union type that contains no pointer (sub-)fields then no referent numbers are updated. Otherwise, if the assignment leads to copying pointer (sub-)fields, then all referent information is copied as well. For instance, in the assignment `s1 = s2`, where `s1` and `s2` have type `struct { int *p; int *q; }`, the referent number of `s2.p` is copied to `s1.p` and that of `s2.q` is assigned to `s1.q`. That is, $\gamma(\&$`s1.p`$) = \gamma(\&$`s2.p`$)$ and $\gamma(\&$`s1.q`$) = \gamma(\&$`s2.q`$)$.

($\mathcal{R}5$) **Library Calls** If an assignment is of the form $p = f(arg)$, where p is a pointer and $f(arg)$ is a call to a library or external function whose source code is not available then the referent number of p is set to the origin number of the memory block returned by $f(arg)$ (i.e., $\gamma(\&p) = \alpha(p)$). This is suitable for many common functions returning a pointer to a newly allocated block (such as `malloc`) or returning a pointer to a pre-existing block (e.g. `strchr`) as long as f itself does not create temporally invalid pointer expressions. Approximations required to handle library functions are further discussed in Section 3.

($\mathcal{R}6$) **Functions** Function calls with arguments are treated as implicit assignments where transfer of referent numbers to parameters is performed using the above rules. Return values are treated similarly. Section 3 further discusses related implementation details.

2.4 Instrumentation Example

Listing 2 shows the resulting instrumented program for the program in Listing 1. The statements added by the program instrumentation are shown in italic font.

```
1  int main() {
2      char *p;
3      store(&p, sizeof(char*));      // Record ‘p‘ to M with origin ‘1‘
4      mapOrigin(&p, NULL);           // Set referent of ‘p‘ to be invalid
5      {
6          char i = 9;
7          store(&i, sizeof(char));   // Record ‘i‘ to M with origin ‘2‘
8          p = &i;
9          mapOrigin(&p, &i);         // Set referent of ‘p‘ to origin of ‘i‘
10         delete(&i);                // Remove ‘i‘ from M */
11     }
12     char j = 8;
13     store(&j, sizeof(char));       // Record ‘j‘ to M with origin ‘3‘
14     /* Spatial: Does ‘p‘ belong to allocated memory? Pass  */
15     assert(svalid(p, p, sizeof(*p)));
16     /* Temporal: Are referent of ‘p‘(2) and origin of ‘p‘(3) equal? Fail */
17     assert(tvalid(&p, p));
18     *p = 1;
19     delete(&j);                  // Remove ‘j‘ from M
20     delete(&p);                  // Remove ‘p‘ from M
21 }
```

Listing 2. Example of instrumentation for the program from Listing 1.

The instrumentation process proceeds as follows. It first adds calls to *store* and *delete* that track memory blocks allocated by the program. Statements recording allocated memory blocks to dynamic meta-storage M (via definitions of `p`, `i` and `j`) are shown at Lines 3, 7 and 13. Calls to *delete* that remove these variables from tracking are shown via Lines 10, 19 and 20.

Statements that update referent number of `p` are shown via calls to *mapOrigin* at Lines 4 and 9. The first call to *mapOrigin* at Line 4 sets the referent number of `p` to invalid (since `p` has no initializer). Further, once the pointer assignment which aliases `p` to `i` at Line 8 is executed, the call to *mapOrigin* at Line 9 updates the referent number of `p` to the origin number of `i`.

The final step of the instrumentation process adds assertions that check validity of pointer expressions before their dereferences. In the example shown in Listing 2 the assignment at Line 18 that dereferences `p` is guarded by two assertions: the assertion at Line 15 verifies the spatial validity of `p` and the assertion at Line 17 checks whether `p` is valid temporally before its dereference.

Since at the time of its dereference `p` is a stale pointer to a valid memory block described by `j`, its dereference leads to a temporal error. During the execution of the instrumented program shown above this violation is detected via the assertion at Line 17 that fails since the referent number $\gamma(\&p) = 2$ carried by `p` and the origin number of its actual pointee $\alpha(p) = 3$ differ.

3 Implementation Details

The present approach to detection of temporal errors has been implemented in the E-ACSL [4] plug-in within the Frama-C [13] source code analysis framework. E-ACSL plug-in is a runtime verification tool that accepts a C program P annotated with formal specifications written in the E-ACSL specification language and generates a new program P' that fails at runtime whenever an annotation is violated. A run of P' that satisfies all annotations is functionally equivalent to the run of P. In other words the E-ACSL plug-in instruments a program with inline monitors generated from formal specifications that can either be provided by the end-user or generated automatically by another tool. For instance, the RTE plug-in [8] of Frama-C can automatically generate such annotations for most undefined behaviours (e.g., out-of-bounds errors or arithmetic overflows). Among others, E-ACSL specifications include memory-related annotations such as `\valid(p)` and `\base_addr(p)` that respectively denote the validity of a pointer p and the start address of the memory block p belongs to. To support its memory predicates E-ACSL relies on a C runtime memory library (RTL). Memory modifications of the program are recorded by the monitor. Before program instrumentation, static analysis can be performed to safely remove unnecessary instrumentations and improve efficiency of the monitor [9,10].

Metadata Storage RTL is based on a specialized shadow memory scheme that allows to capture boundaries of allocated blocks [28]. The key feature of this scheme is that given an address it can identify whether that address belongs to allocated memory and if so compute the start address and the length of the block containing that address. For the purpose of this presentation the shadow memory space implementing this scheme is referred to as *spatial* shadow space.

To enable detection of temporal errors RTL has been extended to include an additional *temporal* shadow space that captures origin numbers of memory blocks and pointer referent numbers. Assuming a 64-bit architecture with pointers comprising 8 bytes, the temporal shadow space represents an application memory block B by 8-byte segments. Application blocks are tracked by shadow blocks of the same size such that an application segment is tracked via its shadow counterpart. Origin and referent numbers are represented by 32-bit integers. The first 4 bytes of a shadow block B' tracking application block B are used to capture the origin number of B. The 4 higher bytes of each 8-byte segment of B' store a referent number. Stack blocks whose size is less than 4 bytes are aligned at a boundary of 4. This is to provide enough space to capture an origin number in the shadow. Blocks whose sizes are between 1 and 7 bytes do not require storing referent numbers because they do not provide sufficient space to store a memory address. In 32-bit architectures, where the size of a program pointer is only 4 bytes, the shadow compression ratio increases meaning that 4 bytes of application memory need to be tracked using 8 bytes of shadow memory.

Generating origin numbers is delegated to a counter incremented each time a memory block is recorded to the meta-storage. To store an origin number x associated with a memory block B the approach writes x to a shadow memory location corresponding to the start address of B. Given an address a from block

B one can retrieve its origin number by first computing the start address s of B (using capabilities of the spatial shadow space) and then reading an integer stored in a shadow location mapped to s. To map a pointer at address a to a referent number x one computes the shadow address s tracking a, increments it by 4 bytes and writes x at that location. Retrieval of a referent number is similar except one reads a referent number instead of writing it.

Source Code Availability The present technique operates at a source code level of the C programming language, therefore it should make approximations when handling calls to functions for which the source code is not available. The present implementation assumes that the change to a pointer structure after such a call can only occur through calls to `memcpy`, `memccpy`, `memmove`, and `memset` functions, while treating the rest of the functions as *safe* in that their execution does not require updates in referent numbers. It handles a call to `memcpy(dest, src,n)` that copies `n` bytes from address `src` to address `dest` as follows. Since `memcpy` can duplicate pointers the monitor also updates referent numbers in the shadow memory to reflect this change. Calls to `memccpy` and `memmove` are treated in a similar way. `memset(s, c, n)` that fills the first `n` bytes of the memory area pointed to by `s` with the constant byte `c` effectively destroys all points-to relationships in that area. Once a call to `memset` is executed the monitor nullifies the portion of shadow space capturing referent numbers.

Function Calls As indicated in Section 2.3 function calls with arguments are treated as implicit assignments. The transfer of referent numbers from arguments to function parameters, however, poses a practical problem due to a change of stack frame. This is because the address of an argument can only be known before the call. The location where the parameter will be stored, however, is not known before the new stack-frame of the called function has been created. Transfer of a referent number requires both addresses. The instrumentation solves this problem by using a global buffer to transfer referent numbers between function calls. This is such that before each call referent numbers of the function arguments are placed to this buffer. Further, once the function is called and the new stack frame is created the instrumentation transfers referent values from the global buffer to function parameters. Referent numbers associated with return values are transferred similarly.

Limitations One limitation of the present approach stems from imprecision in referent number updates via rule $\mathcal{R}3$, Section 2. Consider the following snippet.

```
uintptr_t i  = (uintptr_t)q;  // Assume q is temporally invalid
int *p  = (int*)i;
```

Let pointer q be temporally invalid. The first assignment stores the address of the q's pointee in the integer i. Then, by Rule $\mathcal{R}3$ p is assigned the origin number of the memory block whose address was given via i. This makes p temporally valid even though it has been assigned from a temporally invalid pointer. Another limitation arises from referent number updates through calls to external functions (rule $\mathcal{R}5$ in Section 2). Indeed, an assignment $p = f()$, associates p with the origin number of the memory block address a returned by *f()* belongs to. The dereference of p is valid as long as the a belongs to

allocated space. In practice *f()* can return an address through a temporally invalid pointer. Such an issue cannot be detected because the source code for f is not available. At a present stage of implementation temporal validity of pointers assigned through integer values and within external functions should be validated by the users of the approach. To draw the users' attention to such issues E-ACSL emits appropriate warnings.

Presently E-ACSL does not support detection of temporal errors in multi-threaded programs. In practice, given that each allocation is identified uniquely (which can be achieved by synchronization), extending the proposed approach to tracking temporal memory errors during runs of multi-threaded programs is straightforward but requires engineering effort.

4 Experimental Results

This section now presents the results of the experimentation with the E-ACSL plug-in. The main objective of this experimentation is to evaluate precision and runtime performance of the present approach with respect to detecting temporal errors in C programs. The objective is addressed using two experiments discussed in the following sections. The platform for all results reported here was 2.30GHz Intel i7 processor with 16Gb RAM, running 64-bit Gentoo Linux.

Temporal Error Detection The first experiment aims to evaluate precision of the present analysis. During this experiment E-ACSL is used to check small programs seeded with temporal errors. The same programs are then analysed using AddressSanitizer [20] (embedded in GCC-5.4.0), Dr. Memory [3] (version 1.11.0) and MemCheck [22] (version 3.10.1) memory debuggers.

The programs used in this experiment belong to a small benchmarking suite called TempLIST. Programs from TempLIST are aimed at evaluating precision of runtime analysers with respect to detection of temporal memory errors. The suite has been created by the authors of this paper for the purpose of assessing temporal error detection capabilities of the present approach. The authors have resorted to creating their own benchmark only because detection of temporal memory errors is not a well-studied area and appropriate code samples either do not exist or are not easy to acquire. The TempLIST benchmark has been made publicly available[2] in hope that the researchers working in related areas may find it useful. Presently TempLIST comprises 15 programs ranging from 20 to 46 lines of code. Each program, containing one or two clearly marked errors explores a scenario leading to a temporal memory safety violation. Code examples derived from TempLIST have been shown in Section 1.

During this experiment E-ACSL has been successful in detecting all 23 temporal errors present in the benchmark used. AddressSanitizer, Dr. Memory and MemCheck have been able to detect only 5 errors of 23. Issues discovered by these tools stem from `malloc-free-malloc` sequences (see discussion in Section 1) that can be detected using quarantine zones. However, once multiple reallocations that exhaust quarantine areas are involved such issues go unnoticed. An example of such an issue is shown in the following snippet.

[2] `http://nikolai.kosmatov.free.fr/TempLIST.zip`

```
1 int *p , *q;
2 p = q = (int*)malloc(sizeof(int));
3 ...
4 do {
5   free(q);
6   ...
7   q = malloc(sizeof(int));
8 } while (...);
9 ...
10 *p = 1;
```

Performance Overheads The second experiment assesses runtime costs of the present temporal analysis. During this experiment E-ACSL was used to monitor programs from SPEC datasets [25] for CPU testing. A series of runs of original and instrumented programs were performed and their runtime measured. A runtime of a program accounts for real time between the program's invocation and its termination. This experimentation calculates performance overhead of executions that check temporal and spatial validity relative to the runtime performance of runs that enable spatial checks only. To account for variance due to external factors, such as test automation process or system I/O, the overhead was calculated using an arithmetic mean over 10 runs of the modified and the original executables. It should be noted that the focus of this experiment is on the cost of temporal analysis rather than of overall E-ACSL performance. Comparative analysis of E-ACSL overhead with respect to the overhead of the state-of-the-art memory debuggers has been recently reported [28].

The programs monitored by E-ACSL were instrumented using the annotations that validate memory safety of pointer or array accesses. The annotations were generated automatically by the RTE plug-in of Frama-C (see Section 3). To take into account all factors contributing to the programs' overhead this experimentation disabled static analysis pass during E-ACSL instrumentations.

This experimentation uses 15 C programs from SPEC CPU 2000 and 2006 datasets ranging from 1,155 to 36,037 lines of code. Remaining C programs were rejected due to current limitations of Frama-C and E-ACSL instrumentation engine independent of the proposed technique. `998.specrand` and `999.specrand` programs were also excluded from this experiment due to their small size and absence of temporal checks. Runs of SPEC programs were performed using inputs provided by the test input dataset of SPEC.

Figure 1 shows runtime overhead of E-ACSL with temporal analysis enabled relative to the normalized execution time of E-ACSL using only spatial analysis. The results of this experimentation show that introduction of temporal analysis results in an average slowdown factor of 42% with the maximal result of 91% in `197.parser` and the minimal result of 17% in `433.milc`. One of the main factors contributing to the runtime overhead is the amount of pointer assignments and dereference operations. This is because each pointer assignment results in a transfer of referent numbers, whereas pointer dereferences trigger additional checks. That is, programs, using mostly integer or array operations are likely to result in lower overhead, whereas programs that mostly manipulate pointers are prone to incurring greater overhead.

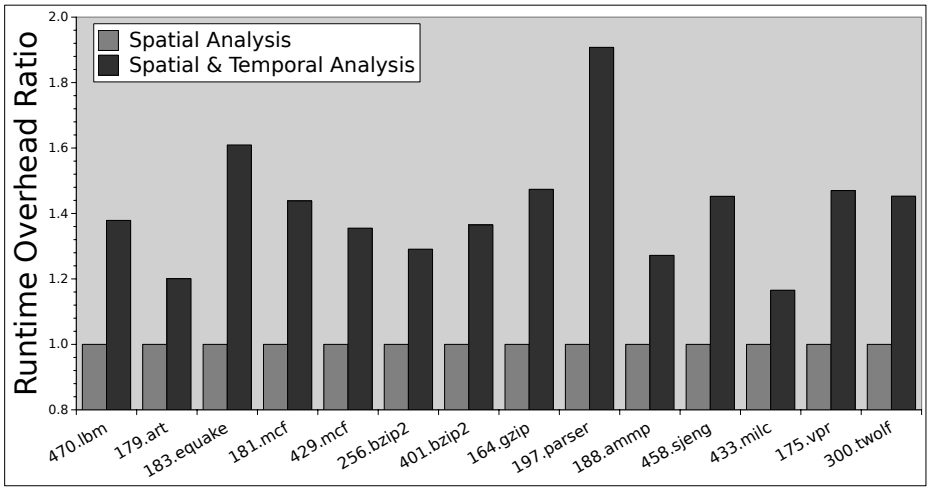

Fig. 1. Runtime overhead of SPEC programs

The overhead of temporal analysis with respect to the runtimes of unobserved programs ranges from approximately 16 times in `470.lbm` to approximately 67 times in `433.milc` and averages to 35 times. Note that these results have been collected using purely dynamic technique that monitored each potential temporal violation. Using static analysis to remove unnecessary checks typically leads to a better performance. For instance, with static analysis enabled the runtime overhead of `256.bzip` of over 40 times drops to only 14.44 times, and 47.31 times overhead of `458.sjeng` is reduced to approximately 27 times.

In summary, even though the overhead of temporal analysis is high, it is likely to be within an acceptable range to be used with testing or debugging. In [28] spatial analysis of E-ACSL has been shown to have overhead comparable to such popular tools as Dr. Memory or MemCheck; an average overhead increase of 42% that has an added benefit of temporal error detection should not present an issue in practice.

Threats to Validity The first issue that may have affected the validity of the presented results is the choice of programs. The experiment evaluating precision of temporal analysis uses programs from a small benchmarking suite developed specifically for evaluating the present approach. While authors have tried to develop representative examples of temporal errors it is possible that they have overlooked certain types of issues that might have affected the results. Further, during the experiment evaluating runtime overhead of the proposed technique programs from SPEC CPU benchmarking suites were used. Even though SPEC are well suited for estimating runtime overhead of memory monitoring tools, different programs or input values may result in different overheads. Additionally, because of technical issues with Frama-C and E-ACSL instrumentation engine several SPEC CPU programs were excluded from the experiment. Using those programs may have also affected the averaged overhead results. The final issue

corresponds to the platform used. The experiments were performed on a machine with a 64-bit architecture where a width of a pointer is 8 bytes. This setup allows to use 1:1 compression ratio for tracking referent and origin numbers. 32-bit architectures require twice as much memory to track temporal information. As such, in 32-bit architectures overhead results may differ to the overhead incurred in 64-bit machines.

5 Related Work

One popular way to detect memory errors at runtime is called memory debugging. This approach tracks memory allocated by a program (typically via shadow mappings) and checks that accessed memory locations have been properly allocated and/or initialized. Rational Purify [7] uses compile-time instrumentations to add instructions tracking the memory state of an executing program directly into object files. MemCheck [22], a memory debugger built atop the Valgrind instrumentation platform [16,17], uses dynamic binary instrumentation (DBI) to instrument a program at runtime and track heap memory with bit-level precision. SGCheck [23] is a similar effort targeting stack and global memory. Another notable DBI-based memory debugger is Dr. Memory [3]. AddressSanitizer [20] is a popular memory debugger targeting out-of-bounds accesses and use-after-free violations. AddressSanitizer [20] is known for its low runtime overheads due to a compact shadow state encoding that tracks 8-byte sequences by 3 bits. AddressSanitizer, initially built on top of a Clang compiler, has now been ported to GCC replacing mudflap [6]. MemorySanitizer [26] and ThreadSanitizer [21] are similar tools aiming detection of initialization errors and data races.

While effective for detection of spatial violations memory debuggers have a limited capacity for detecting temporal errors. Their detection is based upon custom allocation strategies that try to allocate new memory blocks at new addresses as much as possible making stale pointers refer to unallocated portions. Such an approach, however, is unsound, making stale pointers refer to allocated memory eventually possible. Such an approach is also not suitable for stack allocations typically handled by a compiler. The present approach is different in that it tracks temporal metadata allowing to identify temporal memory errors without the need for a custom allocation strategy or modification to compile or runtime environments.

An orthogonal way of tracking memory is by using *fat pointers*, a technique that extends pointers to carry metadata about their pointees. Safe C [2] detects temporal memory errors by associating memory blocks and pointers with "capabilities" that identify allocations uniquely. A temporal error is detected if a capability of dereferenced pointer is the same as that of the memory block it points to. The use of fat pointers has been explored by various researchers [11,15,14].

Detecting temporal errors using Safe C capabilities is similar to origin and referent numbers used by the present technique. One key difference, however, is the way to track memory. Using fat pointers leads to changes in pointer layout that often break legacy programs. Further, Safe C requires "well-behaved" code in which pointer values are not created from or manipulated as non-pointer values. The present technique has no such limitation.

Another way to detect memory errors is by using backwards compatible techniques [12,19,5,1] that use in-memory databases to associate pointer addresses with bounds metadata of their pointees. Even though these techniques solve issues associated with the use of fat pointers by keeping the pointer layout intact, their primary concern is detection of spatial violations.

A notable effort to ensuring safety of C programs is MemSafe [24] that uses a combination of pointer and object metadata. MemSafe maps each allocated object to a unique identifier and each pointer to an identifier of the object the pointer refers to. Pointer identifiers are used to compute object metadata and check spatial validity of pointers before their dereference. MemSafe detects temporal errors by adding statements that forcibly assign invalid values to stale pointers. Additionally, since each allocated object is unique, a stale pointer can be identified by matching the identifier of its referent against object metadata.

One limitation of MemSafe the use of pointer metadata to determine properties of allocated objects. On the contrary, the present approach computes metadata at an address level. This facilitates detection of overflows into allocated areas (unsupported by MemSafe) and ensures that referent numbers are updated correctly. Another key benefit of the present technique is management of referent numbers using shadow memory. This ensures correct updates of referent numbers even during bulk copy of memory areas, where several objects containing pointers can be copied at once (e.g., via `memcpy`). Since MemSafe utilizes pointer metadata it requires approximations to handle such issues.

6 Conclusion

This paper presented a dynamic technique to detection of temporal memory errors in C programs. This approach consumes a C program P and instruments it with statements that track memory allocated by a program at runtime and assertions that verify validity of pointer dereferences. A run of an instrumented program P' that attempts to dereference an invalid pointer (either spatially or temporally) is aborted and the discovered vulnerability is reported. During its execution P' allocates and associates memory objects with origin numbers that uniquely identify memory allocations. Program pointers in turn are associated with referent numbers – origin numbers of the objects they should point to. Dereferencing a pointer to allocated memory whose referent number mismatches the origin number of the object it points to detects a temporal error.

The present technique has been implemented in E-ACSL, a runtime verification tool built on top of the Frama-C source code analysis platform. Experimentation with E-ACSL using TempLIST benchmarks has shown that the present approach is capable of reliable and systematic detection of temporal memory errors missed by such state-of-the-art memory debuggers as AddressSanitizer, Dr. Memory and MemCheck. Further experimentation with SPEC CPU benchmarks has shown that the presented technique incurred an average runtime overhead of 42% compared to monitoring only spatial memory errors. Such results suggest that the present technique is suitable for use during testing and debugging of real programs where an increased analysis cost is amortized by the value of the analysis capable of detecting a wider class of errors.

References

1. Akritidis, P., Costa, M., Castro, M., Hand, S.: Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In: Proceedings of the USENIX Security Symposium. pp. 51–66. USENIX Association (August 2009)
2. Austin, T.M., Breach, S.E., Sohi, G.S.: Efficient detection of all pointer and array access errors. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. pp. 290–301. ACM (June 1994)
3. Bruening, D., Zhao, Q.: Practical memory checking with Dr. Memory. In: Proceedings of the Annual IEEE/ACM International Symposium on Code Generation and Optimization. pp. 213–223. CGO '11, IEEE Computer Society, Washington, DC, USA (2011)
4. Delahaye, M., Kosmatov, N., Signoles, J.: Common specification language for static and dynamic analysis of C programs. In: Proceedings of the ACM Symposium on Applied Computing. pp. 1230–1235. ACM (March 2013)
5. Dhurjati, D., Adve, V.S.: Backwards-compatible array bounds checking for C with very low overhead. In: Proceedings of the International Conference on Software Engineering. pp. 162–171. ACM (May 2006)
6. Eigler, F.C.: Mudflap: pointer use checking for C/C++. In: Proceedings of the GCC Developers Summit. pp. 57–70 (May 2003)
7. Hastings, R., Joyce, B.: Purify: Fast detection of memory leaks and access errors. In: Proceedings of the Winter USENIX Conference. pp. 125–136 (January 1992)
8. Herrmann, P., Signoles, J.: Frama-C's annotation generator plug-in. CEA LIST, Software Safety Laboratory, Saclay, F-91191 (2016), `https://frama-c.com/rte.html`
9. Jakobsson, A., Kosmatov, N., Signoles, J.: Fast as a shadow, expressive as a tree: Optimized memory monitoring for C. Science of Computer Programming 132, Part 2, 226 – 246 (2016), special Issue on Software Verification and Testing (SAC-SVT'15)
10. Jakobsson, A., Kosmatov, N., Signoles, J.: Rester statique pour devenir plus rapide, plus précis et plus mince. In: Baelde, D., Alglave, J. (eds.) Journes Francophones des Langages Applicatifs. Le Val d'Ajol, France (January 2015)
11. Jim, T., Morrisett, J.G., Grossman, D., Hicks, M.W., Cheney, J., Wang, Y.: Cyclone: A safe dialect of C. In: Proceedings of the General Track: 2002 USENIX Annual Technical Conference. pp. 275–288. USENIX (June 2002)
12. Jones, R.W.M., Kelly, P.H.J.: Backwards-compatible bounds checking for arrays and pointers in C programs. In: Proceedings of the International Workshop on Automatic Debugging. pp. 13–26. Linköping University Electronic Press (September 1997)
13. Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: A software analysis perspective. Formal Aspects of Computing 27(3), 573–609 (2015)
14. Kwon, A., Dhawan, U., Smith, J.M., Jr., T.F.K., DeHon, A.: Low-fat pointers: compact encoding and efficient gate-level implementation of fat pointers for spatial safety and capability-based security. In: Proceedings of the ACM SIGSAC Conference on Computer and Communications Security. pp. 721–732. ACM (November 2013)
15. Necula, G.C., Condit, J., Harren, M., McPeak, S., Weimer, W.: CCured: type-safe retrofitting of legacy software. ACM Transactions on Programming Languages and Systems 27(3), 477–526 (2005)

16. Nethercote, N., Seward, J.: Valgrind: A program supervision framework. Electronic Notes in Theoretical Computer Science 89(2), 44–66 (2003)
17. Nethercote, N., Seward, J.: Valgrind: A framework for heavyweight dynamic binary instrumentation. SIGPLAN Notices 42(6), 89–100 (June 2007)
18. Potapenko, A.: AddressSanitizerUseAfterReturn (2015), https://github.com/google/sanitizers/wiki/AddressSanitizerUseAfterReturn
19. Ruwase, O., Lam, M.S.: A practical dynamic buffer overflow detector. In: Proceedings of the Network and Distributed System Security Symposium. The Internet Society (December 2004)
20. Serebryany, K., Bruening, D., Potapenko, A., Vyukov, D.: AddressSanitizer: A fast address sanity checker. In: Proceedings of the USENIX Annual Technical Conference. pp. 309–319. USENIX Association (June 2012)
21. Serebryany, K., Potapenko, A., Iskhodzhanov, T., Vyukov, D.: Dynamic race detection with LLVM compiler — compile-time instrumentation for ThreadSanitizer. In: Proceedings of the International Conference on Runtime Verification. Lecture Notes in Computer Science, vol. 7186, pp. 110–114. Springer (September 2011)
22. Seward, J., Nethercote, N.: Using Valgrind to detect undefined value errors with bit-precision. In: Proceedings of the USENIX Annual Technical Conference. pp. 17–30. USENIX (2005)
23. SGCheck: an experimental stack and global array overrun detector. http://valgrind.org/docs/manual/sg-manual.html
24. Simpson, M.S., Barua, R.: MemSafe: ensuring the spatial and temporal memory safety of C at runtime. Software: Practice and Experience 43(1), 93–128 (2013)
25. Standard Performance Evaluation Corporation: SPEC CPU (2006), http://www.spec.org/benchmarks.html
26. Stepanov, E., Serebryany, K.: MemorySanitizer: fast detector of uninitialized memory use in C++. In: Proceedings of the Annual IEEE/ACM International Symposium on Code Generation and Optimization. pp. 46–55. IEEE Computer Society (February 2015)
27. Stepanov, E.: AddressSanitizerUseAfterScope (2016), https://github.com/google/sanitizers/wiki/AddressSanitizerUseAfterScope
28. Vorobyov, K., Signoles, J., Kosmatov, N.: Shadow state encoding for efficient monitoring of block-level properties. In: Proceedings of the International Symposium on Memory Management. pp. 47–58. ACM (June 2017)