Runtime Detection of Memory Errors with Smart Status

Zhe Chen*
Nanjing University of Aeronautics and Astronautics
Nanjing, Jiangsu, China
zhechen@nuaa.edu.cn

Chong Wang
Junqi Yan
Nanjing University of Aeronautics and Astronautics
Nanjing, Jiangsu, China

Yulei Sui
University of Technology Sydney
Sydney, NSW, Australia
yulei.sui@uts.edu.au

Jingling Xue
University of New South Wales
Sydney, NSW, Australia
j.xue@unsw.edu.au

ABSTRACT

C is a dominant language for implementing system software. Unfortunately, its support for low-level control of memory often leads to memory errors. Dynamic analysis tools, which have been widely used for detecting memory errors at runtime, are not yet satisfactory as they cannot deterministically and completely detect some types of memory errors, e.g., segment confusion errors, sub-object overflows, use-after-frees, and memory leaks.

We propose Smatus, short for *smart status*, a new dynamic analysis approach that supports comprehensive runtime detection of memory errors. The key innovation is to create and maintain a small status node for each memory object. Our approach tracks not only the bounds of each pointer's referent but also the *status* and *reference count* of the referent in its status node, where the status represents the liveness and segment type of the referent. A status node is *smart* as it is automatically destroyed when it becomes useless. To the best of our knowledge, Smatus represents the most comprehensive approach of its kind. In terms of effectiveness (for detecting more kinds of errors), Smatus outperforms state-of-the-art tools, Google's AddressSanitizer, SoftBoundCETS and Valgrind. In terms of performance, Smatus outperforms SoftBoundCETS and Valgrind in terms of both time and memory overheads incurred, and is on par with AddressSanitizer in terms of the time and memory overheads tradeoff (with much lower memory overhead incurred).

CCS CONCEPTS

• **Software and its engineering** → **Dynamic analysis**; **Software testing and debugging**; *Software reliability*; *Software safety*; • **Security and privacy** → *Software and application security*.

KEYWORDS

memory errors, dynamic analysis, testing, error detection

ACM Reference Format:
Zhe Chen, Chong Wang, Junqi Yan, Yulei Sui, and Jingling Xue. 2021. Runtime Detection of Memory Errors with Smart Status. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '21), July 11–17, 2021, Virtual, Denmark.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3460319.3464807

1 INTRODUCTION

C is a dominant language for implementing system software (e.g., operating systems and embedded software such as safety-critical avionics systems [5]) and still one of the most popular programming languages (e.g., with C coming in the first place in the TIOBE ranking of popular programming languages in January 2021). Unfortunately, *memory errors*, which are caused by C's support for low-level control of memory, can result in program crashes and security vulnerabilities [44], and still rank among the most dangerous software errors in recent CVE announcements.

Common types of memory errors include spatial errors and temporal errors [26–28, 40, 41]. *Spatial errors* are bounds violations, e.g., dereferencing null pointers or uninitialized wild pointers, and buffer overflows (including over-reads). *Temporal errors* are accesses to deleted objects, e.g., use-after-free (dereferencing dangling pointers) and double frees. Beyond those, there are other types of memory errors: segment confusion errors and memory leaks. A *segment confusion error* occurs when the pointer of some segment type is used as the pointer of another incompatible segment type [6], e.g., an invalid dereference that uses a function pointer as a data pointer, and vice versa [44], and an invalid free that explicitly frees a non-heap object [10, 11]. A *memory leak* occurs when an object allocated in the heap is no longer accessible but has not been released, causing impaired performance by increasing paging or exhausting memory.

Many dynamic analysis approaches have been proposed to detect memory errors at runtime. They usually maintain *metadata* to track the spatial or temporal information of each memory object or each pointer's referent, e.g., the bounds of the legitimate memory locations that a pointer can access. In general, they do not report false positives, compared to static analysis approaches [14, 16, 42, 43]. Several representative dynamic approaches include pointer-based approaches [12, 17, 21, 23, 27, 29, 30, 33–35, 40, 41, 47, 48, 51], identifier-based approaches [26, 28, 34, 37, 40, 41, 45, 47], object-based approaches using a shadow space [20, 25, 31, 32, 38, 39, 49]

*Zhe Chen is also affiliated with the Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai, China and the State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China.

or bounds table [1, 3, 13, 22, 36, 40, 41, 46], guard-based approaches [19, 20, 24, 38, 50], and quarantine-based approaches [20, 24, 38].

However, these existing dynamic approaches are not satisfactory as they inherently cannot ensure *comprehensive* memory safety. In particular, some approaches can find some kinds of errors but may miss others, or even do not deal with some particular kinds of errors at all. Let us consider four types of errors that cannot be *deterministically* or *completely* detected: segment confusion errors, sub-object overflows, use-after-frees and memory leaks.

```c
void foo(); /* A func */
char *func(char *c)
{ return c; }
int main()
{ void (*p)() = foo;
  char *s=func(p);
  char ch=s[0];/*error*/
  return 0; }
```

(a) Using a function as data.

```c
void (*f(void (*p)()))()
{ return p; }

int main()
{ int a[100];
  void (*p)() = f(a);
  (*p)(); /*error*/
  return 0; }
```

(b) Using data as a function.

Figure 1: Segment confusion errors.

Segment Confusion Errors. No existing approach can detect this class of errors, such as *invalid dereferences* and *invalid frees*, as segment types are not tracked [6]. Figure 1a illustrates an invalid dereference that uses a function pointer as a data pointer and Figure 1b does the reverse. This kind of errors can cause information leakage (by exposing, e.g., the code of foo in Fig. 1a), which can be exploited by the attacker to hijack control-flow by executing malicious code stored in the user memory (e.g., a in Fig. 1b) [44]. Invalid frees may be exploited to call free() on controllable memory locations to modify critical program variables or execute code [10, 11]. Note that a segment confusion error is neither a spatial error nor a temporal error as the accessed memory is indeed in bounds and valid.

Sub-Object Overflows. Existing approaches cannot detect sub-object overflows. For example, Figure 2a illustrates an overflow from a field of a struct (i.e., a sub-object) to another field and Figure 2b illustrates an overflow from an array element (i.e., a sub-object) to another element. Object-based approaches, e.g., Google's AddressSanitizer (ASan) [38] and Valgrind [31, 32, 39], track the bounds of each live memory object, and thus can detect the spatial errors caused by dereferencing a pointer outside the bounds of all live objects. But they cannot detect overflows inside live objects. Even if enhanced with a guard-based approach that inserts some guards around each object, ASan still cannot detect sub-object overflows that occur inside live objects and never access the guards.

Use-After-Frees. Existing approaches can detect use-after-frees only in a probabilistic or partial way. For example, in Figure 2c, dereferencing p2 at Line 7 accesses a heap object that has been deleted via p1, and in Figure 2d, dereferencing p1 at Line 7 accesses the stack object i that has been deleted when f returns. Pointer-based approaches, e.g., CCured [29, 30], MSCC [34, 47], SoftBound [27, 51], MemSafe [40, 41] and Delta Pointers [23], track the bounds of each pointer's referent, but cannot detect use-after-frees alone because the liveness of the referent is not tracked. When enhanced with an identifier-based approach that uses a lock-key scheme to track liveness, SoftboundCETS (SoCets) [26, 28] can catch many temporal errors but yields false negatives due to key reuse [37]. Object-based approaches cannot detect accesses to a freed object after the reuse of the memory. Even if enhanced with a quarantine-based approach that maintains a quarantine to postpone memory deallocations and reuses, ASan still cannot detect errors on accessing deallocated memory objects after they have been removed from the quarantine and then reallocated.

```c
1 typedef struct
2 { int m1;
3   int m2; } st;
4 int main()
5 { st s;
6   int *p = &s.m1;
7   p[1] = 0;
8   /* spatial error */
9   return 0; }
```

(a) Sub-object overflow.

```c
1 typedef struct
2 { char buf[10];
3   int i; } st;
4 int main()
5 { st arr[5];
6   st *p = arr;
7   p[2].buf[20] = 'A';
8   /* spatial error */
9   return 0; }
```

(b) Intra-array overflow.

```c
1 int main()
2 { int *p1;
3   int *p2;
4   p1 = (int*)malloc(8);
5   p2 = p1;
6   free(p1);
7   *p2 = 0;
8   /* temporal error */
9   return 0; }
```

(c) Use-after-free (heap).

```c
1 void f(int **pa)
2 { int i;
3   *pa = &i; /* safe */ }
4 int main()
5 { int *p1;
6   f(&p1);
7   *p1;
8   /* temporal error */
9   return 0; }
```

(d) Use-after-free (stack).

Figure 2: Examples of spatial and temporal errors.

Memory Leaks. Existing approaches only provide partial ability in detecting memory leaks. For example, object-based approaches (e.g., ASan) can detect memory leaks by checking whether there exist live heap objects at the end of an execution. Valgrind can detect memory leaks by remembering the addresses of all live heap objects in a hash table and checking whether the table is empty at the end of an execution. However, such checks can only be performed at the end of an execution. This means, they cannot immediately detect memory leaks on the spot, which is less helpful to debugging.

Challenges. It would be tempting to combine all existing dynamic approaches in a single tool to ensure *comprehensive* memory safety at runtime. Unfortunately, such a simple-minded approach will still be incapable of detecting the above four types of errors in a *deterministic* and *complete* way. Moreover, this combination will impose unnecessary overheads as it must maintain redundant metadata, e.g., the bounds of pointers' referents duplicate the bounds of live objects. Therefore, we need a new approach to overcome these difficulties, but proposing a new approach that outperforms existing tools in both effectiveness and performance is non-trivial.

Our Solution. We introduce Smatus, short for *smart status*, a new dynamic analysis approach that supports comprehensive runtime detection of memory errors. The key innovation is to create and maintain a small *status node* for each memory object. This approach tracks not only the bounds of each pointer's referent but also the *status* and *reference count* of the referent in its status node, where the status represents the liveness and segment type (e.g., heap, stack, global, static or function) of the referent. All the pointers pointing to the same object share the same status node in their pointer metadata. A status node is "*smart*" in the sense that it is automatically destroyed when it becomes useless (indicated by its reference count).

Smatus's capability in detecting memory errors is comprehensive. A spatial error is detected if the accessed locations are outside the tracked bounds. A temporal error is detected if the tracked status indicates invalidity. A segment confusion error is detected if the tracked segment type does not match the way the pointer is used. A memory leak is immediately detected on the spot when the tracked reference count of a live heap object becomes zero.

In summary, we make the following contributions:

(1) We propose Smatus, a single memory safety solution that can detect spatial errors, temporal errors, segment confusion errors, and memory leaks. To the best of our knowledge, Smatus represents the most comprehensive approach of its kind.
(2) We have developed an implementation of Smatus. Our tool supports both hash-table- and trie-based metadata spaces, enabling a metadata-related optimization.
(3) We have evaluated Smatus against three state-of-the-art tools, Google's AddressSanitizer (ASan) [38], SoftBound-CETS (SoCets) [26–28, 51] and Valgrind [31, 32, 39], in terms of effectiveness and performance. We use a large set of benchmarks including 1197 benchmarks from the NIST Software Assurance Reference Dataset (SARD) complemented with 124 hand-crafted microbenchmarks (to cover typical memory errors and C constructs that are not included in the SARD), 20 MiBench benchmarks, and 5 SPEC CPU 2017 benchmarks. For effectiveness, Smatus outperforms the three existing tools by detecting more kinds of errors. For performance, Smatus outperforms SoCets and Valgrind in terms of both time and memory overheads incurred, and is on par with ASan in terms of the time and memory overheads tradeoff (with much lower memory overhead incurred).

The rest of this paper is organized as follows. Section 2 introduces the Smatus approach. Section 3 describes possible implementation choices. Section 4 evaluates Smatus. Section 5 discusses the related work. Section 6 concludes the paper.

2 THE SMATUS APPROACH

In this section, we describe our Smatus approach including the metadata structure and the monitoring algorithm used.

2.1 The Pointer Metadata and Monitoring

Smatus creates and maintains a *pointer metadata* (pmd) for each pointer variable. Figure 3 defines the pmd structure. To detect spatial errors at the object and sub-object levels, the pmd stores the *base* and *bound* of a pointer's referent, i.e., the legitimate range that can be accessed by the pointer. Before the pointer is dereferenced or freed, the access is checked against the bounds to ensure spatial safety. For example, in Fig. 2a, when p points to s.m1 (whose address is assumed to be 0x3000) after the assignment at Line 6, the pmd of p stores the bounds [0x3000, 0x3008), as shown in Fig. 4. When p is

```
typedef struct
{
  void *base;
  void *bound;
  SND   *snda;
} PMD;
```

Figure 3: Pmd.

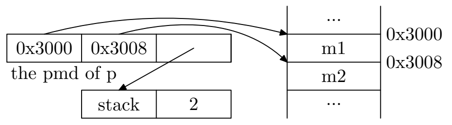


Figure 4: The pmd of p in Fig. 2a.

```
typedef enum {
 function, ...
} status;
typedef struct
{
 status stat;
 size_t count;
} SND;
```

Figure 5: Snd.

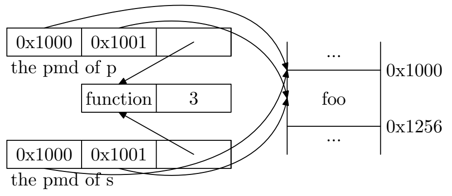


Figure 6: The pmds of p and s in Fig. 1a.

dereferenced at Line 7, the accessed range [3008, 3016) is outside the legitimate range [3000, 3008), indicating a sub-object overflow.

Smatus creates and maintains a small *status node* (snd) for each memory object. Figure 5 defines the snd structure. To detect a segment confusion error, a temporal error or a memory leak, the snd stores the *status* and *reference count* of the object. The status value is one of these possibilities: invalid, heap, stack, global, static and function, i.e., the liveness and segment type of the object. The reference count tracks the number of pointers pointing to the object. To model the points-to relation, the address of the referent's snd is stored in a pointer field of the pmd, say, snda in Figure 3.

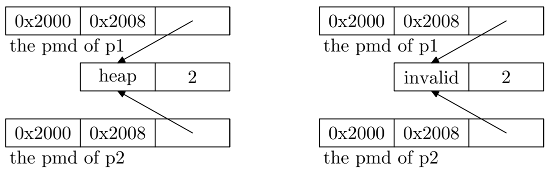
The initial reference count of a heap snd is zero. Pointer assignments are instrumented to update pmds and snds. Deallocating snds is necessary because keeping useless snds can cause heavy memory overhead. An snd can be safely deallocated when its reference count becomes 0 again, i.e., when no pointer points to the corresponding object and, equivalently, no pmd uses this snd. As a result, the instrumented allocators create the snd and then let the monitoring algorithm automatically delete it when the count becomes 0. The "*smart status*" idiom resembles reference-counted *smart pointers* in languages such as C++11: you create the object and then let the system take care of deleting it at the correct time.

Automatically deallocating snds requires a different initial reference count in a non-heap snd. Note that a pointer variable can reference a stack object via the address-of operator, i.e., p=&obj. As a result, when no pointer points to the object, we still need to ensure that the count is greater than 0 to avoid automatic deallocation, as the object may be referenced again via &. Thus, the initial count of stack snds should be 1. Furthermore, to make automatic deallocation possible after the object is out of scope, i.e., when it can no longer be referenced via &, the count should be additionally decremented before the function returns. Similarly, the initial counts of global, static and function snds should also be 1.

When a heap or stack object is manually or automatically allocated, a corresponding heap or stack snd is created, respectively. For example, in Figure 4, a stack snd is created for the struct s at Line 5 and its reference count becomes 2 after Line 6 (recall that the initial count is 1). When the object is explicitly or automatically deallocated, its status is marked `invalid`. When the program starts running, the snds of global, static and function objects are created, but never marked `invalid` as these objects are never deallocated.

To reduce overhead, multiple objects may share the same snd. For example, all stack objects *within a function* share one `stack` snd as they are allocated and deallocated at the same time, i.e., at the beginning and end of the function execution, respectively. Similarly, all global, static or function objects share one `global`, `static` or `function` snd. Note that heap objects cannot share one snd as they are usually allocated and deallocated at different times.

Smatus can detect segment confusion errors as segment types are tracked in snds. For example, in Figure 1a, when `s` points to function `foo` after the assignment at Line 6, the pmds of `p` and `s` are shown in Figure 6. When `s` is dereferenced at Line 7, its referent's segment type `function` does not match the way the pointer is used (as a data pointer), and thus a segment confusion error is detected.



(a) Before deallocation. **(b) After deallocation.**

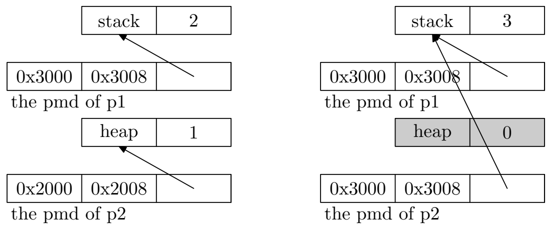
Figure 7: The metadata of `p1` and `p2` in Figure 2c.

Smatus can also detect temporal errors as liveness is tracked in the snd. Note that a pmd contains the address of the referent's snd, instead of the snd itself. Thus, the snd can be shared by multiple pmds. For example, in Figure 2c, when `p2` points to the heap object after the assignment at Line 5, the pmds of `p1` and `p2` are shown in Figure 7a. Note that the reference count also indicates the number of pmds that use the snd. When the heap object is deallocated via `p1` at Line 6, its status is marked `invalid` as shown in Figure 7b. Note that the status value in the pmd of `p2` has also been implicitly updated. As a result, when `p2` is dereferenced for a write at Line 7, its referent's status is invalid, and thus, a use-after-free is detected. Note that we should not deallocate the snd along with the object at Line 6, otherwise the pmds sharing this snd, e.g., the pmd of `p2`, would incorrectly lose their status information. This means that the snd may live past the lifetime of the object. Similarly, the temporal error in Figure 2d can be detected as the status of `p1`'s referent, i.e., `i`, is marked `invalid` when `f` returns. Note that the error detection is deterministic and complete, as the snd is not reused; when the memory is reallocated, a new snd is created for the memory.

Smatus can detect memory leaks on the spot. For example, in Figure 8, when `p2` points to the heap object after the assignment at Line 2, the pmds of `p1` and `p2` are the same as the pmds in Figure 7a. When `p1` points to `i` after the assignment at Line 3, as shown in Figure 9a, the pmd of `p1` switches to the snd of `i`, and thus, the count in the heap snd is decremented. When `p2` also points to `i` at Line 4, as shown in Figure 9b, the pmd of `p2` also switches to the snd of `i`, and thus, the count in the heap snd is decremented to 0. Thus a memory leak is detected, as no pointer points to the heap object. The snd of the heap object (the gray cell) can be safely deallocated at this time.

```
1 p1 = (int*)malloc(8);
2 p2 = p1;
3 int i;  p1 = &i;
4 p2 = &i; /*mem leak*/
```

Figure 8: A memory leak.



(a) After `p1 = &i`. **(b) After `p2 = &i`.**

Figure 9: The metadata of `p1` and `p2` in Figure 8.

2.2 Instrumentation Semantics

Now we formally present the instrumentation details using the semantics of instrumented programs. We assume that the program is in a low level intermediate representation in which data structures have been flattened and all operations are performed on atomic data types. Figure 10 gives the syntax of the C fragment used in the operational semantics. Figure 11 shows the semantic rules.

Atomic Types	a	::=	`int` | $r*$
Referent Types	r	::=	a | `struct id` | `void ()`
Functions	f	::=	`fid() {` $\cdots$ `;` `vid`$_i$: a_i; $\cdots$; c `}`
Global Variables	g	::=	`nil` | `vid` : a ; g
LHS Expressions	lhs	::=	`vid` | $*lhs$
RHS Expressions	rhs	::=	`fid` | lhs | $\&lhs$ | $rhs + rhs$
		|	$(r*)$`malloc`(rhs)
Commands	c	::=	$lhs = rhs$ | $rhs()$ | `free`(rhs)

Figure 10: The syntax of the C fragment.

We denote a pmd by a tuple (b, e, sa), which includes the base b and bound e of its referent and the address sa of its snd. We denote an snd by a pair (s, c), which includes the status value s and reference count c of its referent. For a given program, all its function (global) objects share the unique snd address `f_sa` (`g_sa`) with its status value being `function` (`global`).

The operational semantics relies on an environment $E = (M, G, H, K, P, S)$, where the memory $M : Adr \mapsto Val$ is a mapping from addresses to values, the global storage $G : Var \mapsto Adr \times AType$ is a mapping from global variable identifiers to their addresses and types, the heap block table H is a mapping from base addresses to bounds, the stack K is a list of frames, the pmd table $P : Adr \mapsto Pmd$ is a mapping from addresses of pointer variables to pmds, the snd table $S : Adr \mapsto Snd$ is a mapping from addresses to snds. We assume that E_n always consists of M_n, ..., S_n. The operation $M[l \Downarrow v]$ stores value v to address l and results in a new mapping, while $K[\Downarrow v]$ pushes value v to the top frame. Let F be the function table that maps function pointers to function frames and commands.

Evaluating an lhs expression by $(E, lhs) \Rightarrow_l loc_{(sa)} : a$ yields its address loc, the address sa of its snd and its type a without changing the environment. Evaluating a rhs expression by $(E, rhs) \Rightarrow_r$

$\boxed{(E, lhs) \Rightarrow_l loc_{(sa)} : a}$ Rules for LHS:

[GVar] $\dfrac{G(\texttt{vid}) = (l, a)}{(E, \texttt{vid}) \Rightarrow_l l_{(\texttt{g\_sa})} : a}$ [SVar] $\dfrac{K(\texttt{vid}) = (l, a, sa)}{(E, \texttt{vid}) \Rightarrow_l l_{(sa)} : a}$

[Deref] $\dfrac{\begin{array}{l}(E, lhs) \Rightarrow_l l_{(.)} : a* \quad M(l) = l' \quad P(l) = (b, e, sa)\\ (l' \neq 0) \wedge (b \leq l') \wedge (l' + \texttt{sizeof}(a) \leq e)\\ (sa \neq 0) \wedge (S(sa).s \neq \texttt{invalid}) \wedge (S(sa).s \neq \texttt{function})\end{array}}{(E, *lhs) \Rightarrow_l l'_{(sa)} : a}$

$\boxed{(E, rhs) \Rightarrow_r (E', val_{(b,e,sa)} : a)}$ Rules for RHS:

[Fun] $\dfrac{F(\texttt{fid}) = (fr, c)}{(E, \texttt{fid}) \Rightarrow_r (E, \texttt{fid}_{(\texttt{fid},\texttt{fid+1},\texttt{f\_sa})} : \texttt{void (*)()})}$

[Lhs] $\dfrac{(E, lhs) \Rightarrow_l l_{(.)} : a \quad M(l) = val \quad P(l) = (b, e, sa)}{(E, lhs) \Rightarrow_r (E, val_{(b,e,sa)} : a)}$

[Ref] $\dfrac{(E, lhs) \Rightarrow_l l_{(sa)} : a \quad \texttt{sizeof}(a) = n}{(E, \&lhs) \Rightarrow_r (E, l_{(l,l+n,sa)} : a*)}$

[AddPtr] $\dfrac{\begin{array}{l}(E, rhs_1) \Rightarrow_r (E_1, l_{(b,e,sa)} : r*) \quad \texttt{sizeof}(r) = m\\ (E_1, rhs_2) \Rightarrow_r (E_2, n_{(.)} : \texttt{int})\end{array}}{(E, rhs_1 + rhs_2) \Rightarrow_r (E_2, (l + n * m)_{(b,e,sa)} : r*)}$

[Alloc] $\dfrac{\begin{array}{l}(E, rhs) \Rightarrow_r (E_1, n_{(.)} : \texttt{int}) \quad \texttt{sizeOf}(r) > 0\\ \texttt{umalloc}(E_1, n) = ((M_2, G_1, H_2, K_1, P_1, S_1), l)\\ \texttt{getFreshSnd}(S_1) = sa\\ E_3 := (M_2, G_1, H_2, K_1, P_1, S_1[sa \Downarrow (\texttt{heap}, 0)])\end{array}}{(E, (r*)\texttt{malloc}(rhs)) \Rightarrow_r (E_3, l_{(l,l+n,sa)} : r*)}$

$\boxed{(E, c) \Rightarrow_c (E', R)}$ Rules for commands:

[AssignPtr]

$$\dfrac{\begin{array}{l}(E, rhs) \Rightarrow_r (E_1, v_{(b,e,sa)} : r*) \quad (E_1, lhs) \Rightarrow_l l_{(.)} : r*\\ sa_l := P_1(l).sa \quad (sa_l = 0) \vee (S_1(sa_l).s \neq \texttt{heap}) \vee (S_1(sa_l).c > 1)\\ S_2 := \begin{cases} S_1, \text{ if } sa = sa_l\\ S_1[sa.c \Downarrow sa.c + 1], \text{ if } P_1(l) = \texttt{None} \vee S_1(sa_l) = \texttt{None}\\ S_1[sa.c \Downarrow sa.c + 1, sa_l.c \Downarrow sa_l.c - 1], \text{ if } S_1(sa_l).c > 1\\ S_1[sa.c \Downarrow sa.c + 1, sa_l \Downarrow \texttt{None}], \text{ if } S_1(sa_l).c = 1\end{cases}\\ E_3 := (M_1[l \Downarrow v], G_1, H_1, K_1, P_1[l \Downarrow (b, e, sa)], S_2)\end{array}}{(E, lhs = rhs) \Rightarrow_c (E_3, \texttt{OK})}$$

[Call]

$$\dfrac{\begin{array}{l}(E, rhs) \Rightarrow_r (E_1, l_{(b,e,sa)} : a)\\ (l \neq 0) \wedge (l = b) \wedge (sa \neq 0) \wedge (S_1(sa).s = \texttt{function})\\ F(l) = (fr, c) \quad \texttt{cframe}(E_1, fr) = (M_2, G_2, H_2, K_2, P_1, S_1)\\ \texttt{getFreshSnd}(S_1) = sa_f\\ E_3 := (M_2, G_2, H_2, K_2[\Downarrow sa_f], P_1, S_1[sa_f \Downarrow (\texttt{stack}, 1)])\\ (E_3, c) \Rightarrow_c (E_4, R) \quad \texttt{dframe}(E_4) = (M_5, G_5, H_5, K_5, P_4, S_4)\\ E_6 := (M_5, G_5, H_5, K_5, P_4, S_4[sa_f \Downarrow (\texttt{invalid}, S_4(sa_f).c - 1)])\end{array}}{(E, rhs()) \Rightarrow_c (E_6, R)}$$

[Free]

$$\dfrac{\begin{array}{l}(E, rhs) \Rightarrow_r (E_1, l_{(b,e,sa)} : r*)\\ (l \neq 0) \wedge (l = b) \wedge (sa \neq 0) \wedge (S_1(sa).s = \texttt{heap})\\ \texttt{ufree}(E_1, l) = (M_2, G_1, H_2, K_1, P_1, S_1)\\ S_2 := S_1[\forall n : b \leq n < e.\ P_1(n).sa.c \Downarrow P_1(n).sa.c - 1]\\ P_2 := P_1[\forall n : b \leq n < e.\ n \Downarrow \texttt{None}]\\ E_3 := (M_2, G_1, H_2, K_1, P_2, S_2[sa.s \Downarrow \texttt{invalid}])\end{array}}{(E, \texttt{free}(rhs)) \Rightarrow_c (E_3, \texttt{OK})}$$

Figure 11: Semantics of instrumented programs.

$(E', val_{(b,e,sa)} : a)$ yields its value val, the pmd (b, e, sa) of its referent and its type a, and changes the environment E to E'. Evaluating a command by $(E, c) \Rightarrow_c (E', R)$ yields a result R (which must be OK if successful) and changes the environment E to E'.

Rule [Alloc] allocates a block at address l in the heap using the original allocator `umalloc`, creates a new snd at address sa using `getFreshSnd`, and sets its status value to the segment type `heap` and its reference count to zero. Rule [Call] allocates local objects in frame fr on the stack using the frame constructor `cframe`, creates a new snd at address sa_f, pushes sa_f to the top frame, and sets its status value to the segment type `stack` and its reference count to 1. Then, the commands c in the function are executed, and subsequently, the frame is destroyed using `dframe`. Finally the snd's status is set to `invalid` and its count is decremented.

Rule [AssignPtr] assigns the rhs pointer to the lhs pointer variable. If the lhs variable has a pmd, then the pmd is checked to ensure that the reference count is greater than 1 if its status is heap, i.e., no memory leak occurs (as some other pointers point to the old referent). Then the reference count of the new referent is incremented (as a reference to the new referent has been added). If the lhs variable has an old referent, then the reference count of the old referent is decremented (as a reference to the old referent has been removed). If its reference count is 1, then its snd is removed as no pointer points to it afterwards. Finally the value and pmd of the lhs variable is updated using those of the rhs pointer.

If the lhs variable is a global or stack variable, Rules [GVar] and [SVar] compute its address and snd. If the rhs pointer is a function pointer, Rule [Fun] computes its value and pmd. If the rhs pointer is a global or stack variable, Rule [Lhs] reads its value and pmd from the memory and pmd table, respectively. If the rhs pointer is the address of an object or sub-object, Rule [Ref] computes its value and pmd. If the rhs pointer is a pointer arithmetic, i.e., the sum of a pointer and an integer, Rule [AddPtr] computes its value as usual and its pmd inherits the pmd of the pointer.

Rule [Deref] dereferences a pointer, by checking the accessed locations $[l', l' + \texttt{sizeof}(a))$ against its pmd (b, e, sa) to ensure that they are within the bounds $[b, e)$ (spatial safety), the status is not `invalid` (temporal safety), and the segment type is not `function` (segment safety). Note that Rule [Call] requires the segment type to be exactly `function` as the pointer is used as a function.

Rule [Free] first ensures that the freed object is in the heap, then deallocates the object using the original deallocator `ufree`. After that, the reference counts of the pmds indexed by the addresses between the base and bound are decremented and then these pmds are removed, where these pmds are exactly the pmds of the object's pointer members if they exist. Finally, the snd of the freed object is marked as `invalid`. Note that this implicitly changes the status value in the pmds of all the pointer variables pointing to this object, as they share the same snd.

2.3 The Function Metadata and Monitoring

When pointers are passed across the function calls as arguments and return values, their pmds travel with them. Smatus creates and maintains a *function metadata* (fmd) for each function that has pointer parameters or returns a pointer. The fmd stores an *array of the function's pointer metadata* (fpmds) and the *capacity* of the array. Note that the array contains the pointer metadata of its arguments and return value, indexed by their relative positions in the function definition. Like a pmd, each fpmd also stores the base, bound and snd address of the referent. For example, in Figure 2d, function `f` has a pointer parameter `pa`. Before `f` is called at Line 6, its fmd is created. The fmd contains one fpmd, which is updated using the pmd of the pointer constant `&p1`, with the bounds [`&p1`, `&p1+1`). When the argument `&p1` is assigned to the parameter `pa` at Line 1, the pmd of `pa` is updated using the fpmd. When `pa` is dereferenced for a write at Line 3, the accessed range [`&p1`, `&p1+1`) is compared with the legitimate range tracked in its pmd. In this case, no error is detected as all the checks are passed successfully.

The rules of our monitoring algorithm for function calls are:

(1) Before a function is called, its fmd is created and its fpmds are updated using the pmds of arguments.
(2) When arguments are assigned to parameters after the function execution starts, the pmds of the parameters are updated using the corresponding fpmds. After that, the pointer parameters are treated like local pointer variables in the stack.
(3) Before the function returns, its fpmds are updated using the pmds of return value (as a return value may be a structure containing multiple pointers).
(4) After the function returns, the pmds of the variable storing the return value are updated using the corresponding fpmds.

2.4 Library Function Calls

The above instrumentation can be directly applied to library functions to generate instrumented libraries so as to pass pmds. However, if a library is provided in the pre-compiled form, we are not able to rewrite its function definitions. Thus, we do not pass the pmds of arguments and return value across library function calls via the fmd. Instead, we provide wrappers for library functions.

A wrapper function summarizes the behavior of the original function on its arguments and return value, so as to check the pmds of the arguments for memory safety before calling the original function and update the pmds of the variable storing the return value after the call. For example, the wrapper of `malloc` accepts the pmd of the pointer variable storing the return value as an argument so as to update the pmd of the variable after calling `malloc`. The wrapper of `free` accepts the pmd of the pointer argument as a new argument so as to check whether the pointer is the base of a valid heap object before calling `free`, and then update the status of the pmd to invalid after the call.

3 DESIGN CHOICES

Source Code Transformation. Instrumentation can be performed on different representations of a program [9]. For example, ASan and SoCets perform instrumentation at the IR-level while Valgrind uses binary-level instrumentation. Unfortunately, these techniques are optimization sensitive [8], i.e., many errors caught under the compiler optimization level `-O0` cannot be detected under a higher level such as `-O1`, `-O2` and `-O3` (validated in Section 4.1). Thus, Smatus uses Clang-based source transformation, i.e., rewrites the code written by the programmer, to ensure effectiveness at high optimization levels.

Wrapper Functions for the C Library. Smatus provides wrappers for the most frequently used library functions in the standard C library. It also automatically generates wrappers for other library functions, which do not detect memory errors but assign large bounds and the global snd to the pmds of the returned pointers to avoid false positives. To check the memory accesses in these library functions, it also accepts user-defined wrappers.

Disjoint Metadata. The pmd can be inlined to the pointer variable to form a fat pointer [12, 21, 29, 30, 33], or stored in a metadata space disjoint from pointer variables. Compared with fat pointers, disjoint metadata space causes higher overhead due to the lookup expense but does not change the memory layout and thus provides better compatibility. Thus we use a disjoint metadata space. It consists of a pmd table and a fmd table, which map a pointer variable or a function to its metadata structure by its address, respectively.

Hash Table and Trie. The pmd and fmd tables can be implemented using hash tables or tries. Smatus supports both for evaluating their impact to performance. To enable explicit table lookups, the pmd includes the pointer variable's *address* (not its value), which uniquely identifies the pmd. When hash tables are used, the hash function converts the address into the slot index of the pmd:

```
index = hash(address, capacity)
```

When two-level tries are used, the address is converted into the index in the primary trie using the lower 26-48th bits and the index in the secondary trie using the lower 4-25th bits:

```
  primaryIndex = (address >> 25) & 0x7FFFFF
secondaryIndex = (address >>  3) & 0x3FFFFF
```

The pmd table only maintains the pmds of pointer variables. That is, pointer constants (e.g., function names, array names and variable addresses) do not need pmds, because they do not have addresses for indexing and their referents' bounds and status are known at compile-time. Note that the fpmd does not include the address of a pointer argument as the fpmd can be identified by its relative position in the parameter list and this saves memory.

Using pmd Variables. One optimization is to store pmds in stack variables instead of the pmd table to reduce time overhead by reducing table lookups. For each pointer `p`, a pmd variable definition named `pmd_p` is inserted. For each struct `s` containing pointers, a pmd array definition named `pmd_s[K]` is inserted, where `K` is the number of pointers in the struct. For each array `a[M][N]` containing pointers, a pmd array definition named `pmd_a[M][N][K]` is inserted, where `K` is the number of pointers in the element.

Note that this optimization only applies to pointers in explicitly defined variables. This means, if a pointer is dynamically allocated in the heap at runtime, its pmd must be stored in the pmd table as the pointer is not named. Furthermore, if the address of the variable escapes from the function, e.g., being stored in a pointer or passed as an argument, then its pmd should still be stored in the pmd table as the pointer may be used outside the function where the pmd variable is not accessible. Thus we have implemented a simple static analysis to decide whether the address of a variable can escape from the function. If yes, no pmd variable is inserted for this variable.

4 EXPERIMENTAL EVALUATION

We have implemented the Smatus approach in the memory safety module of Movec [7, 8]. The artifact is available at https://github.com/drzchen/movec. More information about Movec is available at https://drzchen.github.io/projects/movec.

Table 1: Effectiveness of four tools on SARD and memsafe with all compiler optimizations turned off. In Column "Programs", T (number of programs) = B (number of bad programs) + G (number of good programs). In each of the remaining columns, E and N denote the numbers of the programs with and without errors detected, respectively. The number inside each pair of parentheses denotes the number of good programs. Note that SARD-89-2 is originally part of SARD-89, but identified separately because its programs need inputs. Thus, we have written extra scripts to feed the inputs and run these programs.

Suite	Programs			ASan		SoCets		Valgrind		Movec	
Name	T	B	G	E(G)	N(G)	E(G)	N(G)	E(G)	N(G)	E(G)	N(G)
SARD-81	5	5	0	4(0)	1(0)	4(0)	1(0)	0(0)	5(0)	5(0)	0(0)
SARD-88	28	14	14	23(9)	5(5)	14(2)	14(12)	16(9)	12(5)	23(9)	5(5)
SARD-89	1152	864	288	686(1)	466(287)	829(0)	323(288)	149(1)	1003(287)	862(1)	290(287)
SARD-89-2	12	9	3	6(0)	6(3)	6(0)	6(3)	0(0)	12(3)	9(0)	3(3)
memsafe	124	105	19	88(0)	36(19)	88(-7)	36(12)	25(0)	99(19)	105(0)	19(19)
Total	1321	997	324	807(10)	514(314)	934(2)	380(315)	190(10)	1131(314)	1004(10)	317(314)

In this section, we use a large set of benchmarks to evaluate Movec (i.e., Smatus) in terms of effectiveness (for detecting various memory errors) and performance (for the time and memory overheads incurred).

The first set of 1197 benchmarks comes from three benchmark suites (IDs 81, 88 and 89) of the NIST Software Assurance Reference Dataset (SARD). A benchmark is called a *good program* if it is marked as safe by using a file name with the suffix `OK` and, otherwise, a *bad program*. These benchmarks do not cover many typical memory errors and C constructs. Thus, we have developed a new `memsafe` benchmark suite to cover various typical memory errors and C constructs that are not included in the SARD [4]. This suite contains 124 microbenchmarks, of which 105 are unsafe programs seeded with one clearly marked error and 19 are safe.

The second set of benchmarks consists of real applications including 20 MiBench benchmarks [18] and 5 pure-C SPEC CPU 2017 benchmarks. We use MiBench because C is widely used in developing embedded systems and MiBench is a free and commercially representative embedded benchmark suite and covers many domains such as automotive, consumer, network, office, security and telecommunication. SPEC has also been widely used in evaluating performance of computing systems.

We compare Movec with three state-of-the-art tools, Google's ASan, SoCets and Valgrind. Valgrind [31, 32, 39] is probably the most widely used tool for detecting memory errors. ASan [38] significantly outperforms Valgrind and Dr. Memory [2] in runtime overheads, and has been integrated into popular compilers such as GCC and Clang. SoCets [26–28, 51] significantly outperforms Jones and Kelly's BCC [22] and Valgrind in runtime overheads, and has comparable performance as MSCC [47]. As discussed in Section 1, proposing a new approach that outperforms these tools in both effectiveness and performance is rather challenging.

All experiments were conducted on a computer equipped with a 2.3GHz Intel Core i5-6200U CPU and a 4GB DDR3 RAM, running on a 64-bit Ubuntu 16.04 with Linux kernel 4.4.0. We used ASan integrated with LLVM-6.0, the latest version of SoCets integrated with LLVM-3.4, and Valgrind 3.11.0 from the latest Ubuntu repository.

4.1 Effectiveness

For the first set of benchmarks, we first compile the instrumented programs with all compiler optimizations turned off, i.e., using `-O0`. As shown in Table 1, Movec successfully detects the errors in 1004 programs. In particular, Movec reports not only all the labeled errors in bad programs but also more errors beyond the labeled ones for some bad programs. For example, Movec detects the labeled error and 8 extra spatial errors in the benchmark ID 283 of SARD-88, including out-of-bounds accesses in array subscripts and library functions such as `printf`, `strcpy`, `strlen` and `strrchr`. Movec reports the errors in 10 good programs that actually contain errors. For example, Movec detects extra memory leaks in some good benchmarks (IDs 290, 292, 294, 296 of SARD-88 etc.). Movec does not report errors in only three bad programs (IDs 163, 164, 165 of SARD-89), and neither do the other three tools, as the seeded error is not reachable at runtime (these programs abort on an assertion failure). In contrast, ASan, SoCets and Valgrind are able to detect errors in only 807, 934 and 190 programs, respectively, i.e., they are ineffective for the remaining bad programs. SoCets is the only one that reports false positives for 7 safe programs in `memsafe`. To summarize, Movec is the only tool that identifies all the erroneous programs.

We then compile the instrumented programs with compiler optimizations turned on, i.e., using `-O1`, `-O2` or `-O3`. Due to its support for source code transformation, Movec is *optimization-insensitive*, as the number of detected errors does not fluctuate across the optimization flags [8, 9]. For example, Movec still identifies 1004 erroneous programs under `-O3` just as under `-O0`. In contrast, the existing tools do not enjoy this property. Indeed, ASan, SoCets and Valgrind identify only 56, 26 and 31 erroneous programs under `-O3`, respectively, which are much less than what are found under `-O0`. This is because they perform instrumentation on IR or binary code (instead of source code), which is optimization sensitive.

For the second set of benchmarks, Table 2 shows the results on the programs with the errors detected under `-O0` (default) or `-O3`. It is clear that Movec can detect more memory errors. For example, Movec is the unique tool that detects spatial errors in `lame`, `x264` and `xz` and memory leaks in `ispell`, `lame`, `qsort` and `rsynth`. Again, SoCets reports false positives for `rsynth`, `tiff`, `x264` and `imagick`. Valgrind can detect uninitialized values that are not memory errors. To summarize, Movec again outperforms the other tools in terms of effectiveness.

Table 2: Effectiveness of four tools on MiBench and SPEC. ✓ denotes that no error is reported, × denotes that false positives are reported, and ? denotes that the instrumentation fails. For detected errors, SP = spatial errors, e.g., buffer overflows and invalid reads; ML = memory leaks; UT = errors of unknown type; UV = uninitialized values.

Programs	ASan	SoCets	Valgr.	MOVEC
blowfish	SP	SP	(UV)	SP
ispell	✓	✓	✓	ML
jpeg	SP	UT	✓	SP
jpeg-O3	SP	?	✓	SP
lame	✓	✓	✓	SP+ML
lame-O3	✓	?	✓	SP+ML
patricia	ML	✓	ML	ML
qsort	✓	✓	✓	ML
rijndael	SP	SP	✓	SP
rijndael-O3	✓	SP	✓	SP
rsynth	SP	×	SP	SP+ML
rsynth-O3	✓	×	SP	SP+ML
susan	ML	✓	ML	ML
tiff	✓	×	✓	✓
tiff-O3	✓	?	✓	✓
typeset	ML	UT	ML	SP+ML
typeset-O3	ML	?	ML	SP+ML
x264	✓	×	✓	SP
imagick	ML	×	ML	ML
imagick-O3	ML	?	ML	ML
nab	ML	✓	ML	ML
nab-O3	ML	×	ML	ML
xz	ML	✓	ML	SP+ML

When compiler optimizations such as `-O3` are used, MOVEC retains its effectiveness, but ASan, SoCets and Valgrind are not effective as before. For example, ASan fails to detect errors in `rijndael` and `rsynth` under `-O3` but succeeds under `-O0`. SoCets fails to instrument `jpeg`, `lame`, `tiff`, `typeset` and `imagick` but succeeds under `-O0`, and reports false positives for `nab` only under `-O3`.

Let us systematically analyze and summarize the effectiveness of these four tools evaluated. Table 3 shows the results in two parts, where Y/N/P denote Yes/No/Partial, respectively.

- Its first half shows whether each tool can detect particular types of memory errors, where dashed lines separate the spatial errors, temporal errors, segment confusion errors, memory leaks and the errors that occur during the library function execution.

  ASan cannot detect uninitialized pointers, global object and sub-object overflows, long-jump overflows (that access a legitimate location by using a long-distance pointer arithmetic), and stack use-after-frees. For example, if an uninitialized pointer accidentally points to allocated memory, ASan does not detect any error as it does not track the bounds per pointer. ASan cannot detect the overflows on global objects and sub-objects as it does not insert guards around these objects. ASan cannot detect long-jump overflows as such an overflow may jump over the guards inserted around memory blocks and fall into allocated memory. ASan cannot detect the errors of using a stack object after its automatic deallocation, because the stack frame is reused by some subsequent calls, and thus, ASan does not invalidate the shadow space of stack objects after the function returns.

Table 3: Effectiveness summarized for four tools.

Errors/Syntax	ASan	SoCets	Valgr.	MOVEC
null ptr	Y	Y	Y	Y
uninitialized ptr	N	Y	Y	Y
manufactured ptr	Y	N	N	Y
global overflow	N	Y	N	Y
static overflow	Y	Y	N	Y
stack overflow	Y	Y	N	Y
heap overflow	Y	Y	Y	Y
sub-object overflow	N	N	N	Y
long-jump overflow	N	Y	N	Y
stack use-after-free	N	Y	N	Y
heap use-after-free	Y	Y	Y	Y
double free	Y	Y	Y	Y
invalid dereference	N	N	N	Y
invalid free	P	P	P	Y
memory leak	P	N	P	Y
library func	Y	N	Y	Y
func ptr assignment	Y	N	Y	Y
func ptr dereference	N	N	N	Y
variadic func decl	Y	N	Y	Y
va_arg call	Y	N	Y	Y
va_arg dereference	Y	N	Y	Y
va_arg subscript	Y	N	Y	Y
va_arg member	Y	N	Y	Y
const data ptr	Y	N	N	Y
stack data ptr	N	Y	N	Y
global data ptr	N	Y	N	Y
global ptr array	N	N	N	Y
global struct array	N	N	N	Y

SoCets does not support manufactured pointers, e.g., it reports a false positive for the safe program in Fig. 12a where pointer `p` is assigned with a designated address value and then dereferenced, as it incorrectly computes the bounds of the constant data pointer. SoCets cannot detect sub-object overflows (e.g., Fig. 2a) as it only computes the bounds of the outer object.

Valgrind cannot detect manufactured pointers, non-heap object and sub-object overflows, long-jump overflows, and stack use-after-frees. All these weaknesses are essentially attributed to the disadvantages of the object-based approach used.

Neither of ASan, SoCets and Valgrind can detect segment confusion errors, e.g., the invalid dereferences in Figure 1, since neither tracks the segment type of a memory object. These existing tools can report invalid frees, but they neither locate the errors nor provide any precise error information, which can be less helpful to the developers when compared with MOVEC.

Finally, ASan and Valgrind miss some memory leaks in MiBench as they check for leaked objects only at the end of an execution, whereas MOVEC checks for leaks during the execution to enable immediate leak detection, which is more helpful to debugging. SoCets is not designed for detecting memory leaks.

- The second half of Table 3 shows how each tool supports diverse C constructs, which are easy to be handled incorrectly. ASan, SoCets and Valgrind cannot handle many constructs, even if they operate on the *simplified* IR or binary code of a C program.

  ASan and Valgrind provide a similar degree of support for C constructs, as they use a similar object-based approach, where

Table 4: Performance of ASan, SoCets, Valgrind and Movec on the MiBench and SPEC benchmarks. We measure overheads by calculating the time ratios (T.R.) and the memory ratios (M.R.) of the instrumented programs, i.e., execution time (in seconds) and memory consumption (in kilobytes) of the instrumented programs relative to those of the original programs. For example, the T.R. 1.78 means that the instrumented program is 1.78x slower than the original one, indicating a time overhead of 78%.

Programs	Original -O3		ASan				SoCets				Valgrind				Movec -O3 (trie, pmdvar)			
	time	mem	time	mem	T.R.	M.R.	time	mem	T.R.	M.R.	time	mem	T.R.	M.R.	time	mem	T.R.	M.R.
CRC32	0.18	756	0.32	6492	1.78	8.59	0.56	1732	3.11	2.29	4.94	41508	27.44	54.90	0.51	1492	2.83	1.97
FFT	0.09	1340	0.15	7608	1.67	5.68	0.18	2676	2.00	2.00	3.17	42612	35.22	31.80	0.14	2744	1.56	2.05
adpcm	0.37	756	0.81	6496	2.19	8.59	1.44	1708	3.89	2.26	6.13	41508	16.57	54.90	0.83	1484	2.24	1.96
basicmath	0.27	1008	0.43	6884	1.59	6.83	0.30	2376	1.11	2.36	8.34	42620	30.89	42.28	0.29	2304	1.07	2.29
bitcount	0.07	756	0.29	6596	4.14	8.72	0.48	1936	6.86	2.56	3.58	41516	51.14	54.92	0.74	1628	10.57	2.15
blowfish	0.20	772	Abort on SEGV				1.68	1860	8.40	2.41	13.46	41528	67.30	53.79	1.18	1432	5.90	1.85
dijkstra	0.03	872	0.19	11284	6.33	12.94	0.38	3840	12.67	4.40	1.11	47672	37.00	54.67	0.28	1792	9.33	2.06
gsm	0.20	812	0.95	7068	4.75	8.70	2.94	2248	14.70	2.77	4.94	41568	24.70	51.19	2.14	1744	10.70	2.15
ispell	0.00	2136	0.01	7752	2.50	3.63	0.01	3004	1.25	1.41	0.44	41584	110.00	19.47	0.01	2528	1.25	1.18
jpeg	0.01	2076	0.08	8112	8.00	3.91	0.24	3776	24.00	1.82	1.44	41572	144.00	20.03	0.24	3216	24.00	1.55
lame	0.13	3232	0.73	9676	5.62	2.99	3.16	4988	24.31	1.54	9.18	42860	70.62	13.26	2.92	4200	22.46	1.30
patricia	0.08	7684	0.25	16340	3.13	2.13	0.20	45836	2.50	5.97	4.07	71968	50.88	9.37	0.33	55760	4.13	7.26
qsort	0.05	3024	0.09	9420	1.80	3.12	57.59	7760	1151.80	2.57	1.61	42604	32.20	14.09	0.11	3984	2.20	1.32
rijndael	0.05	824	0.51	6744	10.20	8.18	1.23	2008	24.60	2.44	4.13	41536	82.60	50.41	0.88	1544	17.60	1.87
rsynth	0.18	2872	0.52	15616	2.89	5.44	Abort on a false positive				6.64	45208	36.89	15.74	5.11	3480	28.39	1.21
sha	0.02	1352	0.08	6644	4.00	4.91	0.14	1964	7.00	1.45	0.99	41512	49.50	30.70	0.15	1528	7.50	1.13
strsearch	0.01	840	0.02	6636	3.40	7.90	0.01	2184	2.00	2.60	0.49	41536	98.00	49.45	0.01	1836	1.40	2.19
susan	0.04	2100	0.31	8616	7.75	4.10	1.33	3192	33.25	1.52	3.53	42656	88.25	20.31	0.61	3296	15.25	1.57
tiff	0.01	2048	0.01	2076	1.00	1.01	Abort on a false positive				1.92	42584	160.00	20.79	0.01	1972	1.00	0.96
typeset	0.08	10248	0.61	33196	7.63	3.24	Abort on a detected error				5.64	58168	70.50	5.68	0.38	35652	4.75	3.48
AVERAGE					4.23	5.82			77.85	2.49			64.19	33.39			8.71	2.08
over Movec					0.21	2.83			3.86	1.21			3.18	16.24				
over Movec-O3					0.49	2.81			8.94	1.20			7.37	16.09				
GEOMEAN					3.48	4.99			8.55	2.31			53.84	27.61			5.12	1.85
over Movec					0.35	2.72			0.85	1.26			5.36	15.06				
over Movec-O3					0.68	2.70			1.67	1.25			10.51	14.94				
lbm	3.03	420232	11.46	477784	3.78	1.14	32.87	420204	10.85	1.00	104.72	561984	34.56	1.34	35.09	420424	11.58	1.00
x264	51.34	158852	338.16	196752	6.59	1.24	Abort on a false positive				2973.80	242124	57.92	1.52	1484.13	174276	28.91	1.10
imagick	0.02	6868	0.16	21880	8.00	3.19	Abort on a false positive				1.71	47564	85.50	6.93	0.22	11160	11.00	1.62
nab	1.84	3128	4.31	15380	2.34	4.92	12.59	8232	6.84	2.63	63.57	45660	34.55	14.60	9.30	4920	5.05	1.57
xz	12.26	567328	45.72	653728	3.73	1.15	184.22	644832	15.03	1.14	524.30	756524	42.77	1.33	229.72	568716	18.74	1.00
AVERAGE					4.89	2.33			10.91	1.59			51.06	5.14			15.06	1.26
over Movec					0.14	1.79			0.31	1.22			1.45	3.95				
over Movec-O3					0.32	1.85			0.72	1.26			3.39	4.08				
GEOMEAN					4.45	1.91			10.37	1.44			47.93	3.07			12.84	1.23
over Movec					0.15	1.51			0.34	1.14			1.57	2.44				
over Movec-O3					0.35	1.55			0.81	1.17			3.73	2.50				

```
int main() {
  int *p, i;
  unsigned long addr =
    (unsigned long)&i;
  p = (int *)addr;
  i = *p; /*safe*/
  return 0;
}
```

(a) Manufactured Pointer.

```
1 void foo(); /*A func*/
2 int main() {
3   int *p1 = (int *)foo;
4   int *p2 = p1 + 1;
5   void (*p3)() =
6        (void (*)())p2;
7   (*p3)(); /*error*/
8   return 0; }
```

(b) Function Pointer Deref.

Figure 12: Programs that ASan/SoCets/Valgr. cannot handle.

pointer manipulations are irrelevant. For example, in Fig. 12b, all pointers refer to function foo, but p3 is not equal to the base address of foo. Its dereference at Line 7 causes the program to invoke foo three times on our platform, resulting in a spatial error. However, ASan and Valgrind cannot detect this error because the function is stored in an allocated memory. They do not support stack/global data pointers and global pointer/struct arrays well, because these pointers (or in global arrays) are usually initialized to the addresses of stack or global objects but they cannot detect stack use-after-frees and overflows on global objects.

SoCets does not support function pointer assignments and dereferences, variadic function declarations, and va_arg calls, constant data pointers, and global pointer/struct arrays. For example, SoCets does not detect the error of dereferencing a non-base function pointer in Fig. 12b because it does not instrument the dereference of a function pointer due to a representation that is different from that for the dereference of a data pointer. In Fig. 12a, the false positive also shows that SoCets does not support the pointers initialized with constant data pointers.

Smatus is more effective because it provides more comprehensive detection of memory errors. In contrast, ASan, SoCets and Valgrind provide partial memory safety only as they are inherently limited by the capabilities of their monitoring algorithms.

4.2 Performance

The performance evaluation uses the MiBench and SPEC benchmarks. Movec may detect multiple errors in a run without terminating program execution, resulting in a complete measurement for the time and memory overheads incurred. Unfortunately, ASan, SoCets and Valgrind force a program to abort on the first detected error, making these measurements incomplete. Thus we have developed a safer version of the benchmarks for performance evaluation by manually correcting some errors detected in Section 4.1, making these measurements complete. Note that we are not able to correct all errors as we are unware of how to correct the errors strongly related to program-specific allocators and algorithms.

We ran the original programs and the instrumented programs of each tool five times using the large inputs from MiBench and test inputs from SPEC, and collected their execution times and memory consumption using the arithmetic average of the five runs. The execution time of a program accounts for the elapsed time between the program's invocation and its termination, while its memory consumption indicates maximum resident set size (RSS) of the process during its lifetime. Execution time and memory consumption were reported using GNU `time`.

Table 4 shows the performance data. We compiled the instrumented programs with all optimizations turned off (under `-O0`), since ASan, SoCets and Valgrind may miss errors when optimizations are used (Section 4.1). We also used Movec under the most aggressive optimization level `-O3`, since Movec is optimization-insensitive. For MiBench, the execution time of Movec is 8.71x relative to the original run on average, while its memory consumption is 2.08x. In contrast, the time ratios of ASan, SoCets and Valgrind are 4.23x, 77.85x and 64.19x, respectively, while their memory ratios are 5.82x, 2.49x and 33.39x, respectively. When compared with Movec under `-O0`, ASan spends only 0.21x less time but 2.83x more memory, SoCets is 3.86x slower and consumes 1.21x more memory, and Valgrind is 3.18x slower and consumes 16.24x more memory. When compared with Movec under `-O3`, ASan spends 0.49x less time, SoCets and Valgrind are 8.94x and 7.37x slower, respectively. Therefore, Movec outperforms SoCets and Valgrind in both execution time and memory consumption, and is on par with ASan because of their respective strengths in time and memory. If we use the metric of geometric mean (GEOMEAN) instead of arithmetic average, the results are similar. For SPEC, we can draw a similar conclusion except that the performance of SoCets is unclear due to abortion on false positives.

Table 5: Performance impact of various design choices of Movec on the MiBench and SPEC benchmarks. All time and memory ratios are average values on the programs.

Suites	Choices	Hash table		Trie	
		T.R.	M.R.	T.R.	M.R.
MiBench	under -O3	11.39	1.80	10.02	2.08
	+ pmd var	8.43	1.83	8.71	2.08
	+ no check			4.43	1.82
SPEC	under -O3	21.28	1.19	17.40	1.27
	+ pmd var	14.85	1.19	15.06	1.26
	+ no check			5.77	1.18

We have also evaluated the performance impact of various design choices of Movec (Table 5). Hash-table- and trie-based metadata spaces show similar performance in general, as the latter spends 12-18% less time when pmd variables are not used but around 10% more memory. When pmd variables are used, the time overhead is reduced by 26-30% and 13% for the hash table-based and trie-based metadata spaces, respectively. We have also found that much time is spent in checking the validity of memory accesses, as the time overhead can be greatly reduced when no check is performed (only metadata is propagated) at runtime.

The performance difference between Movec and ASan is influenced by their different monitoring algorithms. First, Movec uses a pointer-based approach, whereas ASan uses an object-based approach and implements metadata space using a large shadow space of the memory used by a program including some unused memory (trading memory for time). Thus ASan spends less time than Movec, but much more memory for all programs. ASan inherently cannot detect some memory errors, although it incurs lower time overhead. Second, the lower memory consumption of Movec is also attributed to *smart status*, because a freed object can be immediately deallocated and reused by the operating system, while we only need to preserve its snd until no pointer points to it. In contrast, ASan uses a quarantine-based approach to detect heap use-after-frees, thus it has to maintain a self-managed quarantine to collect freed objects, which consumes additional memory.

The performance difference between Movec and SoCets is influenced by the following factors. First, the current implementation of Movec uses a *purely dynamic technique* that monitors each potential error, whereas SoCets uses static analysis to remove unnecessary and redundant runtime checks. For example, SoCets uses dominator-based redundant check elimination [27, 51] and dataflow analysis [26, 28], which have removed many checks and largely reduced time overheads. However, SoCets is extremely slow on `qsort`, because there is a large number of checks on pointer manipulations that cannot be removed by its static analysis. The time overheads of Movec may be greatly reduced if it is equipped with a similar analysis (Table 5 gives an evidence). SoCets does not track segment confusion errors, memory leaks and library functions, again reducing its overheads. Second, the lower memory consumption of Movec is also attributed to *smart status*, because we do not need to permanently store the snd of a deallocated object as it is automatically deallocated when no pointer points to it. In contrast, SoCets uses an identifier-based approach to detect temporal errors, thus it has to maintain a self-managed pool of freed locks for reuse, which consumes additional memory.

Smatus is more efficient because its new data structure and algorithm consume less memory due to immediate deallocation of freed objects and automatic deallocation of their smart status. In contrast, the performance of existing tools is inherently limited by the capabilities of their monitoring algorithms.

5 RELATED WORK

In this section, we compare Smatus with related work and summarize its limitations at the end of this section.

Pointer-based Approaches. These appropaches track the bounds of each pointer's referent in a fat pointer [12, 21, 29, 30, 33], a low-fat pointer [15, 23] or a disjoint metadata space [17, 27, 34, 35, 40, 41, 47, 48, 51]. Note that these approaches are the only way to enforce complete spatial safety [44], and consequently, have been widely used in CCured [29, 30], MSCC [34, 47], SoCets [27, 51], MemSafe [40, 41], Delta Pointers [23] and many safe dialects of C/C++ such as Cyclone [21], Ironclad C++ [12], and Microsoft's Checked C [17, 35].

One main disadvantage is that they cannot detect temporal errors alone, e.g., use-after-frees and double frees, because the liveness of a referent is not tracked. Smatus achieves temporal safety by using smart status. Another disadvantage is that the tool/user must provide wrapper functions for all pointer-relevant library functions that cannot be instrumented from scratch (e.g., the source code is not available) because libraries do not provide the bounds of a pointer's referent. Smatus faces the same problem but our solution is to provide wrappers for the standard C library and automatically generate wrappers for other library functions (Section 3).

Identifier-based Approaches. These approaches generate a unique identifier number for each object and use a lock-key scheme to track its liveness and can thus catch many temporal errors. They are used in ARM's Memory Tagging Extension [37] and also often used as an enhancement of pointer-based approaches in, e.g., MSCC [34, 47], SoCets [26, 28], and MemSafe [40, 41].

One main disadvantage is that their error detection is probabilistic (i.e., yields false negatives) due to key reuse (i.e., the equality of two keys) [37]. Even combined with a pointer-based approach, they cannot detect segment confusion errors and memory leaks. Moreover, this combination causes additional overheads as they have to maintain a list of unallocated/deallocated locks for reuse.

Smatus is a better alternative. On the one hand, it can detect not only temporal errors without false negatives but also segment confusion errors and memory leaks. On the other hand, smart status nodes are automatically deleted when becoming useless, making the reuse problem irrelevant and also reducing overheads.

Reference Counting and Smart Pointers. These well-established techniques are used in languages like C++. Smart status resembles reference-counted smart pointers. The difference is that smart pointers are used to automatically delete heap objects at the correct time, whereas smart status is used to detect memory errors. Smart pointers have a well-known limitation: they do not work on circular structures. Similarly, Smatus cannot detect the memory leak involving a circular structure that is not pointed by any outside pointers. However, if one element in the circle is explicitly freed, then Smatus can detect the leaks caused by the remaining elements. Other tools like ASan and SoCets cannot detect this kind of memory leaks either. Valgrind can detect it by remembering the addresses of all live heap objects in a hash table, resulting in large overheads.

Object-based Approaches. These approaches track the bounds of a live memory object in a shadow space [20, 25, 31, 32, 38, 39, 49] or a bounds table [1, 3, 13, 22, 36, 40, 41, 46] and can thus detect the spatial and temporal errors caused by dereferencing a pointer outside the bounds of all live objects, such as invalid pointers (e.g., null or uninitialized pointers), buffer overflows to unallocated memory and use-after-frees. As a representative, ASan [38] maintains a *shadow space* of the memory used by a program (one byte for every 8 bytes of data) to specify whether the first k bytes have been allocated (addressable). Every operation that allocates or frees a memory object is instrumented to update the shadow space appropriately. When a pointer is dereferenced or freed, the shadow space of the accessed location is checked to ensure that it belongs to a valid object. Valgrind [31, 32, 39] uses a similar approach, but performs instrumentation on binary code at runtime, instead of on IR, which brings better compatibility but incurs higher overheads.

One main disadvantage is that their error detection is partial, i.e., yields false negatives for both spatial and temporal safety. For example, they cannot detect overflows across or inside live objects and accesses to a freed object after the reuse of the memory. Smatus achieves more complete spatial and temporal safety.

Guard-based Approaches. These approaches insert guards around an object [19, 20, 24, 38, 50] to detect buffer overflows across live objects. For example, ASan [38] inserts *poisoned redzones* around each object where the bytes in redzones are marked as unallocated in the shadow space, and thus accessing redzones indicates overflows.

One main disadvantage is that their error detection is still partial. For example, they cannot detect sub-object overflows (that occur inside live objects and never access the guards) and long-jump overflows (that access another live object by jumping over guards).

Quarantine-based Approaches. These approaches maintain a *quarantine* to collect freed objects and make `free` a no-op to postpone memory deallocations and reuses [20, 24, 38] and can thus detect heap use-after-frees. As one main limitation, they cannot detect accessing deallocated memory after it has been removed from the quarantine (due to, e.g., it being full) and then reallocated [37]. Moreover, the quarantine itself also increases memory overhead.

Conservative Garbage Collectors. These have been used in many high level programming languages to avoid dangling pointers (by making `free` a no-op) and memory leaks. They have also been used by some C dialects, e.g., Cyclone [21] and Ironclad C++ [12]. One main disadvantage is that they can only prevent errors but cannot detect errors in the program. Elsewhere [20, 24, 33], garbage collectors have been used to detect memory leaks by searching the heap for allocated objects that no longer seem to be referenced. However, this can yield false negatives due to the conservatism (garbage objects may be accidentally "referenced" by an integer).

Limitations of Smatus. Let us summarize the limitations discussed in the previous sections.

- Smatus needs wrapper functions for all pointer-relevant library functions that cannot be instrumented from scratch (e.g., the source code is not available). The user may write user-defined wrappers if the tool-provided or automatically generated wrappers are not satisfactory.
- Smatus cannot detect the memory leak involving a circular structure that is not pointed by any outside pointers. The programmer is responsible for breaking the circle by explicitly freeing one element in the circle.
- Smatus incurs higher time overheads than ASan. Although we believe its current time overheads are tolerable for testing, incorporating a static analysis into a future version of Movec may greatly reduce the time overheads.

6 CONCLUSION

We have proposed a new dynamic analysis approach, Smatus, to achieve comprehensive runtime detection of memory errors. Smart status is at the core of this approach. We believe that having a single solution that can detect spatial errors, temporal errors, segment confusion errors and memory leaks alone is, in principle, a useful contribution. Experiments have shown that SMATUS enjoys stronger bug-finding ability with moderate and acceptable overheads.

ACKNOWLEDGMENTS

This work was supported by the Joint Research Funds of National Natural Science Foundation of China and Civil Aviation Administration of China (No. U1533130), the Fundamental Research Funds for AI (No. NZ2020019), the Open Project of Shanghai Key Laboratory of Trustworthy Computing, and the Open Project of State Key Laboratory for Novel Software Technology (No. KFKT2020B10).

REFERENCES

[1] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. 2009. Baggy Bounds Checking: An Efficient and Backwards-Compatible Defense against Out-of-Bounds Errors. In *Proceedings of the 18th USENIX Security Symposium*. USENIX Association, 51–66.

[2] Derek Bruening and Qin Zhao. 2011. Practical memory checking with Dr. Memory. In *Proceedings of the 9th International Symposium on Code Generation and Optimization, CGO 2011*. IEEE Computer Society, 213–223.

[3] Nathan Burow, Derrick McKee, Scott A. Carr, and Mathias Payer. 2018. CUP: Comprehensive User-Space Protection for C/C++. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security, AsiaCCS 2018*. ACM, 381–392.

[4] Zhe Chen. 2021. *Movec-MSBench: A Memory Safety Benchmark Suite, Version 2.0.0.* https://github.com/drzchen/movec-msbench

[5] Zhe Chen, Yi Gu, Zhiqiu Huang, Jun Zheng, Chang Liu, and Ziyi Liu. 2015. Model Checking Aircraft Controller Software: A Case Study. *Software-Practice & Experience* 45, 7 (2015), 989–1017.

[6] Zhe Chen, Chuanqi Tao, Zhiyi Zhang, and Zhibin Yang. 2018. Beyond spatial and temporal memory safety. In *Proceedings of the 40th International Conference on Software Engineering (ICSE 2018), Companion Volume*. ACM, 189–190.

[7] Zhe Chen, Zhemin Wang, Yunlong Zhu, Hongwei Xi, and Zhibin Yang. 2016. Parametric Runtime Verification of C Programs. In *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2016) (Lecture Notes in Computer Science, Vol. 9636)*. Springer, 299–315.

[8] Zhe Chen, Junqi Yan, Shuanglong Kan, Ju Qian, and Jingling Xue. 2019. Detecting Memory Errors at Runtime with Source-Level Instrumentation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*. ACM, 341–351.

[9] Zhe Chen, Junqi Yan, Wenming Li, Ju Qian, and Zhiqiu Huang. 2018. Runtime verification of memory safety via source transformation. In *Proceedings of the 40th International Conference on Software Engineering (ICSE 2018), Companion Volume*. ACM, 264–265.

[10] The MITRE Corporation. 2009-05-08. *CWE-762: Mismatched Memory Management Routines.* https://cwe.mitre.org/data/definitions/762.html

[11] The MITRE Corporation. 2020-02-24. *CWE-590: Free of Memory not on the Heap.* https://cwe.mitre.org/data/definitions/590.html

[12] Christian DeLozier, Richard A. Eisenberg, Santosh Nagarakatte, Peter-Michael Osera, Milo M. K. Martin, and Steve Zdancewic. 2013. Ironclad C++: a library-augmented type-safe subset of c++. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013*. ACM, 287–304.

[13] Dinakar Dhurjati and Vikram S. Adve. 2006. Backwards-compatible array bounds checking for C with very low overhead. In *Proceedings of the 28th International Conference on Software Engineering (ICSE 2006)*. ACM, 162–171.

[14] Thomas Dillig, Isil Dillig, and Swarat Chaudhuri. 2014. Optimal Guard Synthesis for Memory Safety. In *Proceedings of the 26th International Conference on Computer Aided Verification, CAV 2014 (Lecture Notes in Computer Science, Vol. 8559)*, Armin Biere and Roderick Bloem (Eds.). Springer, 491–507.

[15] Gregory J. Duck and Roland H. C. Yap. 2016. Heap bounds protection with low fat pointers. In *Proceedings of the 25th International Conference on Compiler Construction, CC 2016*. ACM, 132–142.

[16] I. A. Dudina and A. A. Belevantsev. 2017. Using static symbolic execution to detect buffer overflows. *Programming and Computer Software* 43, 5 (2017), 277–288.

[17] A. S. Elliott, A. Ruef, M. Hicks, and D. Tarditi. 2018. Checked C: Making C Safe by Extension. In *Proceedings of the 2018 IEEE Cybersecurity Development Conference, SecDev'18*. IEEE, 53–60.

[18] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. 2001. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE 4th Annual Workshop on Workload Characterization*. IEEE, 3–14.

[19] Niranjan Hasabnis, Ashish Misra, and R. Sekar. 2012. Light-weight bounds checking. In *10th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2012*. ACM, 135–144.

[20] Reed Hastings and Bob Joyce. 1992. Purify: Fast detection of memory leaks and access errors. In *In Proceedings of the Winter 1992 USENIX Conference*. 125–138.

[21] Trevor Jim, J. Gregory Morrisett, Dan Grossman, Michael W. Hicks, James Cheney, and Yanling Wang. 2002. Cyclone: A Safe Dialect of C. In *Proceedings of the 2002 USENIX Annual Technical Conference*. USENIX, 275–288.

[22] Richard W. M. Jones and Paul H. J. Kelly. 1997. Backwards-Compatible Bounds Checking for Arrays and Pointers in C Programs. In *Proceedings of the 3rd International Workshop on Automated Debugging, AADEBUG 1997*. 13–26.

[23] Taddeus Kroes, Koen Koning, Erik van der Kouwe, Herbert Bos, and Cristiano Giuffrida. 2018. Delta pointers: buffer overflow checks without the checks. In *Proceedings of the 13th EuroSys Conference, EuroSys 2018*. ACM, 22:1–22:14.

[24] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. 2016. DoubleTake: fast and precise error detection via evidence-based dynamic analysis. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016*, Laura K. Dillon, Willem Visser, and Laurie Williams (Eds.). ACM, 911–922.

[25] Alexey Loginov, Suan Hsi Yong, Susan Horwitz, and Thomas W. Reps. 2001. Debugging via Run-Time Type Checking. In *Proceedings of the 4th International Conference on Fundamental Approaches to Software Engineering, FASE 2001 (Lecture Notes in Computer Science, Vol. 2029)*. Springer, 217–232.

[26] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2015. Everything You Want to Know About Pointer-Based Checking. In *1st Summit on Advances in Programming Languages, SNAPL 2015 (LIPIcs, Vol. 32)*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 190–208.

[27] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2009. SoftBound: highly compatible and complete spatial memory safety for C. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009*. ACM, 245–258.

[28] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. 2010. CETS: compiler enforced temporal safety for C. In *Proceedings of the 9th International Symposium on Memory Management, ISMM 2010*. ACM, 31–40.

[29] George C. Necula, Jeremy Condit, Matthew Harren, Scott McPeak, and Westley Weimer. 2005. CCured: type-safe retrofitting of legacy software. *ACM Trans. Program. Lang. Syst.* 27, 3 (2005), 477–526.

[30] George C. Necula, Scott McPeak, and Westley Weimer. 2002. CCured: type-safe retrofitting of legacy code. In *Proceedings of the 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2002)*, John Launchbury and John C. Mitchell (Eds.). ACM, 128–139.

[31] Nicholas Nethercote and Julian Seward. 2007. How to shadow every byte of memory used by a program. In *Proceedings of the 3rd International Conference on Virtual Execution Environments, VEE 2007*. ACM, 65–74.

[32] Nicholas Nethercote and Julian Seward. 2007. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, PLDI 2007*. ACM, 89–100.

[33] Yutaka Oiwa. 2009. Implementation of the memory-safe full ANSI-C compiler. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009*. ACM, 259–269.

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

[35] Andrew Ruef, Leonidas Lampropoulos, Ian Sweet, David Tarditi, and Michael Hicks. 2019. Achieving Safety Incrementally with Checked C. In *Proceedings of the 8th International Conference on Principles of Security and Trust, POST 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019 (Lecture Notes in Computer Science, Vol. 11426)*. Springer, 76–98.

[36] Olatunji Ruwase and Monica S. Lam. 2004. A Practical Dynamic Buffer Overflow Detector. In *Proceedings of the Network and Distributed System Security Symposium, NDSS 2004*. The Internet Society, 159–169.

[37] Kostya Serebryany. 2019. ARM Memory Tagging Extension and How It Improves C/C++ Memory Safety. *The Usenix Magazine* 44, 2 (2019), 12–16.

[38] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference, Boston, MA, USA*. USENIX Association, 309–318.

[39] Julian Seward and Nicholas Nethercote. 2005. Using Valgrind to Detect Undefined Value Errors with Bit-Precision. In *Proceedings of the 2005 USENIX Annual Technical Conference*. USENIX, 17–30.

[40] Matthew S. Simpson and Rajeev Barua. 2010. MemSafe: Ensuring the Spatial and Temporal Memory Safety of C at Runtime. In *10th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2010*. IEEE Computer Society, 199–208.

[41] Matthew S. Simpson and Rajeev Barua. 2013. MemSafe: ensuring the spatial and temporal memory safety of C at runtime. *Software - Practice and Experience* 43, 1 (2013), 93–128.

[42] Yulei Sui, Ding Ye, and Jingling Xue. 2012. Static memory leak detection using full-sparse value-flow analysis. In *International Symposium on Software Testing and Analysis, ISSTA 2012*. ACM, 254–264.

[43] Yulei Sui, Ding Ye, and Jingling Xue. 2014. Detecting Memory Leaks Statically with Full-Sparse Value-Flow Analysis. *IEEE Trans. Software Eng.* 40, 2 (2014), 107–122.

[44] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *2013 IEEE Symposium on Security and Privacy, SP 2013*. IEEE Computer Society, 48–62.

[45] Kostyantyn Vorobyov, Nikolai Kosmatov, Julien Signoles, and Arvid Jakobsson. 2017. Runtime Detection of Temporal Memory Errors. In *Proceedings of the 17th International Conference on Runtime Verification, RV 2017 (Lecture Notes in Computer Science, Vol. 10548)*. Springer, 294–311.

[46] Kostyantyn Vorobyov, Julien Signoles, and Nikolai Kosmatov. 2017. Shadow state encoding for efficient monitoring of block-level properties. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on Memory Management, ISMM 2017*. ACM, 47–58.

[47] Wei Xu, Daniel C. DuVarney, and R. Sekar. 2004. An efficient and backwards-compatible transformation to ensure memory safety of C programs. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2004)*. ACM, 117–126.

[48] Ding Ye, Yu Su, Yulei Sui, and Jingling Xue. 2014. WPBOUND: Enforcing Spatial Memory Safety Efficiently at Runtime with Weakest Preconditions. In *25th IEEE International Symposium on Software Reliability Engineering, ISSRE 2014*. IEEE Computer Society, 88–99.

[49] Suan Hsi Yong and Susan Horwitz. 2003. Protecting C programs from attacks via invalid pointer dereferences. In *Proceedings of the 11th ACM SIGSOFT Symposium on Foundations of Software Engineering and the 9th European Software Engineering Conference, ESEC/FSE 2003*, Jukka Paakki and Paola Inverardi (Eds.). ACM, 307–316.

[50] Qiang Zeng, Dinghao Wu, and Peng Liu. 2011. Cruiser: concurrent heap buffer overflow monitoring using lock-free data structures. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011*. ACM, 367–377.

[51] Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2012. Formalizing the LLVM intermediate representation for verified program transformations. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012*. ACM, 427–440.