Detecting Memory Errors at Runtime with Source-Level Instrumentation

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

Shuanglong Kan
Ju Qian
Nanjing University of Aeronautics and Astronautics
Nanjing, Jiangsu, China

Jingling Xue
School of Computer Science and Engineering
University of New South Wales
Sydney, NSW, Australia
j.xue@unsw.edu.au

ABSTRACT

The unsafe language features of C, such as low-level control of memory, often lead to memory errors, which can result in silent data corruption, security vulnerabilities, and program crashes. Dynamic analysis tools, which have been widely used for detecting memory errors at runtime, usually perform instrumentation at the IR-level or binary-level. However, their underlying non-source-level instrumentation techniques have three inherent limitations: optimization sensitivity, platform dependence and DO-178C non-compliance. Due to optimization sensitivity, these tools are used to trade either performance for effectiveness by compiling the program at `-O0` or effectiveness for performance by compiling the program at a higher optimization level, say, `-O3`.

In this paper, we overcome these three limitations by proposing a new source-level instrumentation technique and implementing it in a new dynamic analysis tool, called Movec, in a pointer-based instrumentation framework. Validation against a set of 86 microbenchmarks (with ground truth) and a set of 10 MiBench benchmarks shows that Movec outperforms state-of-the-art tools, SoftBound-CETS, Google's AddressSanitizer and Valgrind, in terms of both effectiveness and performance considered together.

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, source-level instrumentation, compiler optimization

ACM Reference Format:
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 '19), July 15–19, 2019, Beijing, China.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3293882.3330581

1 INTRODUCTION

C is widely used for implementing system and embedded software such as safety-critical avionics systems [3]. Unfortunately, its unsafe language features, such as low-level control of memory, often lead to *memory errors*, which can result in silent data corruption, security vulnerabilities, and program crashes. Typical memory errors include *spatial errors* such as buffer overflows and out-of-bounds accesses, and *temporal errors* such as dangling pointer dereferences (i.e., use-after-free). There is a large body of work that uses static analysis to detect memory errors such as buffer overflows and memory leaks [8, 10, 15, 34, 35]. However, verification of memory safety via static analysis is undecidable even for closed, terminating programs [28]. Furthermore, static analysis cannot achieve both precision and efficiency simultaneously due to its approximative nature, thus suffering from either false positives or false negatives.

Therefore, dynamic analysis tools have been widely used for detecting memory errors at runtime due to their precision (with few or no false positives, in practice). Several representative approaches are object-based approaches in terms of shadow spaces [13, 17, 25, 26, 30, 31, 39] or bounds tables [1, 7, 14, 29, 32, 33, 36], guard-based approaches [12, 13, 16, 30, 40], and pointer-based approaches [21, 23, 24, 27, 32, 33, 37, 38, 41]. These existing dynamic tools perform instrumentation at different representations of a program, including IRs (Intermediate Representations) as in CIL [12, 23, 24, 37], GCC's SSA form [14, 29] or LLVM-IR [7, 20–22, 30, 32, 33, 38, 41], object code [13, 16, 40], and binary executables [25, 26, 31], possibly accelerated by hardware [18, 19]. However, we are not aware of any dynamic tools that truly perform instrumentation at the source-level code, i.e., the source code directly written by software developers.

Existing non-source-level instrumentation techniques and their associated tools suffer from three inherent limitations: *optimization sensitivity*, *platform dependence* and *DO-178C non-compliance*.

Optimization Sensitivity. Existing dynamic analysis tools, such as SoftBoundCETS (SoCets) [20–22, 41], Google's AddressSanitizer (ASan) [30], and Valgrind [25, 31], are sensitive to compiler optimizations. Therefore, one must make two opposite tradeoffs when applying these tools to detect memory errors in a program: (1) trading performance for effectiveness by compiling the program at `-O0` or (2) trading effectiveness for performance by compiling the program at a higher optimization level, say, `-O3`.

Let us consider two examples in Figure 1. In Figure 1a, `p` is initialized to point to `i` instead of `a`, possibly by mistake, causing `*(p+5)` to result in a spatial error. In Figure 1b, `*n` accesses an out-of-scope stack variable `x`, resulting in a temporal error.

```c
/*Option -O1 hides error*/
#include <stdio.h>
int main() {
  int a[10] = {0};
  int i = 1, sum = 0, *p = &i;

  *(p+5) = 1;
  /*spatial error*/

  for(i = 0; i < 10; i++)
    sum += a[i];
  printf("sum is %d\n", sum);
  return 0;
}
```

(a) A spatial error

```c
/*Option -O2 hides error*/
#include <stdio.h>
void foo(int **p_addr) {
  int x;
  *p_addr = &x;
}

int main() {
  int *n;
  foo(&n);
  printf("%d\n", *n);
  /*temporal error*/
  return 0;
}
```

(b) A temporal error

Figure 1: Memory errors sensitive to optimizations.

Existing dynamic tools, which are expected to detect these errors, exhibit varying degrees of weakness. SoCets [20–22, 41], which takes a pointer-based approach, can detect both errors at `-O0` (with the compiler optimizations turned off), but not at a higher optimization level, say, `-O3` (when the dereference `*(p+5)` in Figure 1a and the call `foo(&n)` in Figure 1b have been optimized away). Google's ASan [30] and Valgrind [25, 31] fail to detect both errors at any optimization level due to the object-based approach used, but suffer from the optimization sensitivity problem on some other programs (as evaluated later). This is because compiler optimizations may cause the non-equivalence between source and binary codes. When the source code contains an error, such non-equivalence may lead to binary code where the original error is no longer detectable by existing tools.

This is why dynamic tools are usually suggested to run under `-O0` to ensure maximal effectiveness [41], but at the expense of performance (as validated later). Under `-O0`, more memory errors can be found, but rather inefficiently. At a higher optimization level, say, `-O3`, the opposite tradeoff is made, but still unsatisfactorily.

Platform Dependence. Existing dynamic analysis tools are usually platform-dependent. C is a cross-platform language. Thus, C programs are expected to be compiled under different compilers and deployed to different platforms, including embedded systems in the avionics and defense industries. However, non-source-level instrumentation techniques, which operate at the IR-level or binary level, are always integrated into a limited number of platform-dependent host compilers as an extension, a plug-in or a compiler pass.

For example, LLVM-based instrumentation tools, such as SoCets [20–22, 41] and Google's ASan [30], support a wide range of operating systems, e.g., Linux, Solaris, FreeBSD, NetBSD, MacOS X, Cygwin/Win32, and Windows running under a number of ISAs, e.g., x86, AMD64, PowerPC, ARM, and Ultrasparc. However, some OS and ISA combinations, such as MacOS X and Cygwin on AMD64, are not supported. Worse still, there is a lack of support for VxWorks and μC/OS, the operating systems popular in the avionics and defense industries, and MIPS, MicroBlaze and RISC-V, the ISAs popular in embedded systems. Therefore, it is impossible to test or deploy instrumented programs on the platforms not supported by LLVM-based dynamic tools.

DO-178C Non-Compliance. Existing dynamic analysis tools are not usually compliant to DO-178C, the de facto standard for developing avionics software systems [9]. According to this standard, an instrumented program should be inspected and verified to ensure that instrumentation neither alters the functionality of the original program nor introduces new faults or malicious code [6]. Other related certification standards, e.g., Department of Defense Standards (DOD-STD) and Military Standards (MIL-STD) of the United States, have similar requirements. In addition, the structure of a program, once instrumented, should stay relatively unchanged, in order to facilitate compliance checks against low-level requirements, and preserve traceability and verifiability. However, non-source-level instrumentation techniques, which operate at the IR-level or binary level, make such inspection and verification tasks hard to perform.

Our Solution. In this paper, we propose a new source-level instrumentation technique and introduce an associated dynamic analysis tool, (the memory safey module of) Movec which also supports parametric runtime verification [5], implemented in a pointer-based instrumentation framework. Movec operates directly on the original source code of a C program by inserting instrumented code fragments written in ANSI C, and generates the instrumented program with the original code structure remaining unchanged. As a result, Movec has overcome the three limitations of the state of the art discussed above. First of all, Movec is no longer sensitive to compiler optimizations, since it inserts the instrumented code before any optimization is applied, making it possible to enjoy the speed advantages of compiler optimizations such as `-O3` without compromising its ability in detecting memory errors. In addition, Movec is platform-independent, since any instrumented C program generated is still a C program, which can be compiled by any C compiler and deployed in any platform. Finally, Movec is compliant to DO-178C [9], as any instrumented program generated can be further inspected and verified by any source-level analysis tool, with source-level debugging information now possible.

By overcoming a number of challenges faced in performing source-level instrumentation for C programs (supporting, e.g., multi-level pointers and arbitrary pointer arithmetic operations), as described in Section 3, we make the following contributions:

- We introduce a new source-level instrumentation technique, which is the first of its kind implemented in a pointer-based framework (to the best of our knowledge).
- We have developed a new tool, Movec, for implementing the proposed source-level instrumentation technique.
- We have coded a set of 86 microbenchmarks (with ground truth) for evaluating dynamic analysis tools [2].
- We have evaluated Movec against three state-of-the-art dynamic analysis tools, SoCets [20–22, 41], Google's ASan [30], and Valgrind [25, 31] in terms of effectiveness (on detecting errors) and performance using our 86 microbenchmarks [2] and a set of 10 MiBench benchmarks [11]. For effectiveness, Movec outperforms the three state-of-the-art tools, for all the optimization levels evaluated, `-O0`, `-O1` and `-O3`. For performance, Movec outperforms SoCets and Valgrind, and is on par with (but less memory-hungry than) ASan when the three existing tools run under `-O0` (the optimization level for maximizing their effectiveness). Overall, Movec, invoked under `-O3`, outperforms all these tools in terms of both effectiveness and performance considered together.

The rest of this paper is organized as follows. Section 2 reviews the pointer-based approach for detecting memory errors. Section

3 discusses some challenges faced by performing pointer-based instrumentation directly at the source level. Section 4 describes our source-level instrumentation in a pointer-based framework. Section 5 evaluates our technique by comparing with the state of the art. Section 6 discusses the related work. Finally, Section 7 concludes.

2 POINTER-BASED INSTRUMENTATION

In general, Movec adopts a pointer-based approach as in SoCets [20–22, 41] and MemSafe [32, 33]. This approach creates and maintains a disjoint *pointer metadata* (pmd) for each pointer variable, and checks the pmd for memory safety when the pointer is dereferenced. The pmd of a pointer variable stores its *address* (not its value) and the *status*, *base* and *bound* of its referent. Note that the address can be used as a unique ID for indexing pmds, the status describes the current storage of its referent, e.g., invalid, stack or heap, and the base and bound record the valid address range of its referent. This approach also creates and maintains a disjoint *function metadata* (fmd) for each function that has pointer parameters or returns a pointer. The fmd of a function stores its *address*, an *array of function pointer metadata* (pointer metadata for the pointer parameters and return value, indexed by their relative positions in the function definition), and the *capacity* (i.e., size) of the array. A *function pointer metadata* (fpmd) also stores status, base and bound, like a pmd, but *excludes pointer address* as the fpmd can be identified by its relative position. In this approach, we maintain the pmd and fmd tables at runtime, which map a pointer variable or a function to its metadata by its address, respectively. In this work, we have implemented the two tables using hash tables.

To facilitate instrumentation, we provide some interfaces for our data structures. For the pmd table, `pmd_tbl_lookup()` retrieves and returns the pmd of a pointer variable by its address, and `pmd_tbl_update_as()` updates the pmd of a pointer variable using the given status, base and bound. To verify the dereference of a pointer variable, `check_dpv()` checks its pmd against the base pointer and size of the memory block to be accessed.

Figure 2 gives the standard instrumentation algorithm for pointer metadata initialization, propagation and checking. When a pointer variable is defined, possibly without any initial value or is initialized to `NULL`, its pmd is created and set to invalid (Figure 2a). When a heap object is explicitly allocated by calling `malloc()`, and the returned address is assigned to a pointer variable, its pmd is updated using the heap status, base and bound of the object (Figure 2b). When a heap object is deallocated by calling `free()` on a pointer variable, its pmd is updated to invalid (Figure 2c). When the address of a variable is assigned to a pointer variable by using the & operator, its pmd is updated using the status, base and bound of the variable (Figure 2d). These pmds are propagated on pointer manipulations such as assignments, stores and loads. In a pointer assignment, the left-hand side (lhs) pointer variable inherits the pmd of the right-hand side (rhs) (Figure 2e). When a pointer is stored to memory, the intermediate memory block inherits the pmd of the pointer variable (Figure 2f). When a pointer is loaded from memory, the pointer variable inherits the pmd of the intermediate memory block (Figure 2g). Whenever a pointer variable is dereferenced to access memory (including pointer stores and loads), spatial and temporal safety checks are performed by invoking function `check_dpv()` before the dereference (Figure 2h).

```
T *p; /* or T *p = NULL */
pmd_tbl_update_as(&p, NULL, NULL, NULL);
```

(a) Pointer Definitions

```
p = (T *)malloc(n);
pmd_tbl_update_as(&p, heap, p, (char*)p+n);
```

(b) Heap Object Allocations

```
free(p);
pmd_tbl_update_as(&p, invalid, NULL, NULL);
```

(c) Heap Object Deallocations

```
p = &obj;
pmd_tbl_update_as(&p, obj_status, &obj, &obj+1);
```

(d) Address Assignments

```
p = p1;  /* p = p1 + i or p = &p1[i] */
pmd1 = pmd_tbl_lookup(&p1);
pmd_tbl_update_as(&p, pmd1->status, pmd1->base, pmd1->bound);
```

(e) Pointer Assignments

```
check_dpv(pmd_tbl_lookup(&ptr), ptr, sizeof(*ptr));
*ptr = p;  /* store */
pmdp = pmd_tbl_lookup(&p);
pmd_tbl_update_as(ptr, pmdp->status, pmdp->base, pmdp->bound);
```

(f) Pointer Stores

```
check_dpv(pmd_tbl_lookup(&ptr), ptr, sizeof(*ptr));
p = *ptr;  /* load */
pmdp = pmd_tbl_lookup(ptr);
pmd_tbl_update_as(&p, pmdp->status, pmdp->base, pmdp->bound);
```

(g) Pointer Loads

```
check_dpv(pmd_tbl_lookup(&p), p, sizeof(*p));
... = *p; /* or *p = ... */
```

(h) Unary Dereferences

Figure 2: Pointer-based instrumentation algorithm.

3 CHALLENGES FACED IN SOURCE-LEVEL POINTER-BASED INSTRUMENTATION

The instrumentation algorithm given in Figure 2 works well at the IR-level, e.g., on the SSA form. However, we must address a number of challenges when moving from the IR-level to the source level:

- **Nested Expressions.** Consider a pointer dereference "`*(--p)`", which contains an inner decrement expression. At the IR-level, this expression is split into two smaller ones, "`--p`" and "`*p`", causing the standard instrumentation algorithm to insert a check in between. At the source level, however, this is impossible, as we cannot insert such a check immediately before or after an inner expression.
- **Pointer Arithmetic Expressions (Assignments).** Consider a pointer assignment "`p = p1+p2[0]`", where `p1` and `p2` are both pointers. At the IR-level, this is replaced by `i = p2[0]` followed by "`p = p1+i`", which can be instrumented by the standard algorithm. At the source level, however, we must infer that `p` should inherit the pmd from `p1` rather than `p2`. In other words, given the pointer arithmetic expression "`p = p1+p2[0]`", we must infer that the object accessed is actually pointed to by `p1` rather than `p2`.
- **Side effects.** Consider pointer assignments "`p = ++p1`" and "`p = f()`". In either case, the standard algorithm would use the address of the rhs expression in the instrumentation code

to lookup its pmd. However, "`&(++p1)`" contains a side effect, so the instrumentation code can incorrectly increment the value of `p1`. On the other hand, "`&(f())`" is incorrect, as the address for a returned (expiring) value cannot be taken.

- **Struct Assignments.** Consider a struct assignment "`s1 = s2`" for a struct containing pointer members. At the IR-level, it is split into a series of member-wise assignments, for which the standard instrumentation algorithm applies. At the source level, however, we must update the pmds of multiple pointer members in one expression, as it may be an inner expression in another statement, as in `s1 = s2 = s3`.
- **Nested Function Calls.** Consider `f1(f2(p))`. At the IR-level, this is split into two separate function calls, where the return value of `f2` is explicitly stored in a new temporary pointer variable, so we can propagate the pmd of the return value into `f1`. At the source level, however, this is impossible, as the return value is an expiring value and we cannot use its address to index, retrieve and update a pmd.
- **Variadic Functions.** Consider "`T func(t1 p1, tn pn, ...)`". This is hard for both the IR-level and source-level, as variadic arguments are directly obtained from the call stack by `va_arg` expressions. We should carefully handle `va_arg` expressions to obtain the pmds of the variadic arguments.

All these challenges will be addressed in our source-level pointer-based instrumentation. At first glance, one may simplify the instrumentation by decomposing complex statements into simpler ones in some SSA form. Unfortunately, the same level of difficulties still exists. Furthermore, such a transformation will inevitably change the structure of the program with temporary variables added, reducing traceability and instrumentation performance.

4 THE MOVEC APPROACH

4.1 The Overall Algorithm

We should first find out all dark corners in C programs, where pointers may be hidden. To start with, definitions of pointer variables produce pointers. However, this is far from complete. Figure 3 shows a more systematic view of the types and storage classes that may contain pointers, where an empty cell denotes an impossible combination. The types containing pointers include pointers, structs containing pointer members, arrays of pointers, arrays of structs containing pointer members. A *variable* of one such type is an l-value stored in memory and has an address. Pointer *constants*, e.g., function names, array names and variable addresses, are pure r-values that do not have memory addresses. The pointers or structs returned by *function calls* are expiring values whose addresses cannot be taken. The C constructs related to these objects containing pointers can be divided into three groups separated by back slashes.

Type	Storage Class		
	variable	constant	func call
pointer	dlr/e/vpa	r/e/a	r/e/a
structure	dlr/ /vpa		r/ /a
pointer array	d/ /		
structure array	d/ /		

Figure 3: Types and storage classes.

- **Variable Definitions and Assignments.** If an object of a given type and storage class can be created using an explicit *definition*, or used in the *lhs* or *rhs* of an assignment, then the first group in its corresponding cell is flagged with d, l or r, respectively.
- **Pointer *Dereferences.*** These include unary dereferences, array subscript expressions and member expressions. If an object of a given type and storage class can be directly dereferenced, then the second group in its corresponding cell is flagged with e.
- **Function Definitions and Calls.** If an object of a given type and storage class can be used as the *return value* or a *parameter* of a function definition, or used as an *argument* of a function call, then the third group in its corresponding cell is flagged with v, p or a, respectively. Note that we also consider conditional expressions as function calls, as a conditional expression takes its three operands as arguments and returns one of the last two operands.

In general, our source-level instrumentation algorithm uses a recursive AST visitor to traverse the entire AST of a source file, instruments exactly the constructs discussed in the above systematic analysis during the traversal, and finally, saves the instrumented code in another source file. The inserted code fragments are used to maintain and manipulate the pmds of the pointers contained in the objects of these types and storage classes. More specifically, our instrumentation pass includes the following three parts:

- Instrument *variable definitions* to initialize their pmds and *variable assignments* to update their pmds, probably using the pmds of the rhs objects. These variables may be pointers, structs containing pointer members, pointer arrays or struct arrays containing pointer members, i.e., those flagged with d or l. These rhs objects may be pointers or structs containing pointer members that may be variables, constants or function calls (including `va_arg` calls), i.e., those flagged with r.
- Instrument *pointer dereferences* to check their pmds, where dereferences may be *unary dereferences* like `*p`, *array subscript expressions* like `p[i]`, and *member expressions* like `p->m` and `(*p).m`. These pointers may be variables, constants or function calls (including `va_arg` calls), i.e., those flagged with e.
- Instrument *function definitions* to initialize the pmds of parameters, compute and store the pmds of return values. Then we generate and insert wrapper function definitions for function definitions or function call expressions to pass the pmds of arguments to the called functions and pass back the pmds of return values to the caller. Finally, we instrument *function call expressions* by renaming and inserting additional arguments to redirect the function calls to their wrapper functions. These return values and parameters are variables that may be pointers or structs containing pointer members, i.e., those flagged with v and p. These arguments may be pointers or structs containing pointer members that may be variables, constants or function calls (including `va_arg` calls), i.e., those flagged with a.

System Overview. Figure 4 gives a systematic overview of our tool implementation. It consists of a parser and a recursive AST visitor, which together transform input C programs into instrumented C programs and generate interface source files of data structures. Note that our instrumentation algorithm is designed to recursively traverse all AST nodes, and every instrumented expression has the same return value as the original one so that it can be embedded in the original outer expression. Thus, our algorithm can handle all the complex nested constructs, as discussed in Section 3.

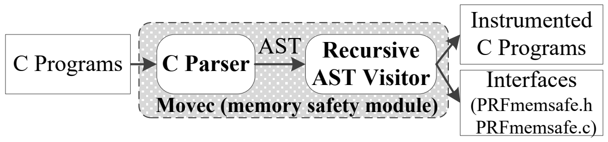
C Programs
C Parser
AST
Recursive AST Visitor
Movec (memory safety module)
Instrumented C Programs
Interfaces (PRFmemsafe.h PRFmemsafe.c)

Figure 4: System overview.

4.2 Interfaces of Data Structures

To facilitate instrumentation, we have developed a set of interfaces. Figure 5 shows some interfaces that may be called by instrumentation code. Function `PRFcheck_dpv()` is an example of the checking interfaces, which is used to detect errors in dereferencing pointer variables. In these checking interfaces, the last parameter "..." denotes omitted parameters, including the names of the file and function containing the dereferenced expression, the line and column numbers, and the code text. Note that these interfaces are needed by all instrumented source files. Thus we make their declarations and definitions available in two separate files, which will be automatically included by the instrumented files on demand.

```
/*Get the status value of a pmd.*/
PRFstat PRFpmd_get_stat(pmd);

/*Lookup for the pmd of a pointer variable by address.*/
PRFpmd *PRFpmd_tbl_lookup(ptr_addr);
/*Update the pmd of a pointer var using status, base and bound.*/
PRFpmd *PRFpmd_tbl_update_as(ptr_addr, status, base, bound);
/*Update the pmd of a pointer var and return the last param.*/
void *PRFpmd_tbl_update_as_ret(ptr_addr, status, base, bound,
                               ret);
/*Update the pmd of a pointer var using fpmd.*/
PRFpmd *PRFpmd_tbl_update_fpmd(ptr_addr, fpmd);
/*Update the pmd of a pointer var using the pmd of another one.*/
PRFpmd *PRFpmd_tbl_update_ptr(ptr_addr, ptr_addr2);
/*Update the pmd of a pointer var and return the last param.*/
void *PRFpmd_tbl_update_ptr_ret(ptr_addr, ptr_addr2, ret);

/*Lookup for the i-th fpmd of a function.*/
PRFfmd_pmd *PRFfmd_tbl_lookup_fpmd(func_addr, i);

/*Check dereferences of pointer variables.*/
void *PRFcheck_dpv(pmd, ptr, size, ...) {
  stat = PRFpmd_get_stat(pmd);
  /*Check pointer validity.*/
  if(ptr == NULL) print_error();
  /*Check temporal safety.*/
  if(pmd == NULL || stat == PRFinvalid) print_error();
  /*Check spatial safety.*/
  if(ptr < pmd->base || ptr + size > pmd->bound) print_error();
  return ptr;
}
/*Check dereferences of pointer constants.*/
void *PRFcheck_dpc(base, bound, ptr, size, ...);
/*Check dereferences of function pointer variables.*/
void *PRFcheck_dpfv(pmd, ptr, ...);
/*Check dereferences of function pointer constants.*/
void *PRFcheck_dpfc(base, bound, ptr, ...);
```

Figure 5: Interfaces of the metadata tables.

4.3 Basic Operations for Instrumentation

Now, we describe the implementation of our instrumentation algorithm in details. We first introduce some intensively used basic operations, and then focus on the instrumentation rules of rewriting the language constructs discussed in Section 4.1.

4.3.1 Adding Prefixes and Suffixes. To avoid conflicts with existing identifiers, we always add a prefix to each inserted identifier, denoted by `PRF` in our presentation. Sometimes we also add a suffix `ID`, i.e., a unique number allocated for an expression, to avoid the conflicts between inserted identifiers themselves.

4.3.2 Handling Pointer Arithmetic Expressions. A complex pointer arithmetic expression is usually dominated by one component, which determines the intended referent. For example, `p+i` computes a pointer offsetting the value of `p` by `i` elements. Note that the intended referent of the expression should be the same as `p`, as it is usually used to traverse a memory block. Thus, `p` is said to *dominate* the expression, and it is the kernel pointer expression (kpe) of the expression. A consequence is that the pmd of `p` actually includes the metadata of the expression's referent. Similarly, a kernel struct expression (kse) exists for an expression containing structs, especially in an object-oriented setting. In our algorithm, we often need to get the kpe of an expression `E(kpe)`, or the kse of an expression `E(kse)`, for retrieving the pmd that includes the metadata of `E`'s referent.

Algorithm 1 sketches the basic idea for finding the kernel pointer expression of `E`. It first ensures that `E` is not a type cast from non-pointer to pointer, then uses structural recursions on unary and binary operators, to find the smallest sub-expression of `E` that has the same type as `E` after removing type casts. For example, consider "`p1+p2[0]`", where `p1` and `p2` are both of the pointer type "`int *`". Its kpe is `p1`, as `p2[0]` is of the type `int`.

Algorithm 1 Find the kernel pointer expression of E

```
function GetKpe($E$)
    $E_0 \leftarrow$ IgnoreCasts($E$)
    $t \leftarrow$ GetType($E_0$)
    if $t$ is not pointer then return $E$
    end if
    if $E_0 := op\ E_1$ then     //case of unary operator
        if $E_1$ is of type $t$ then return GetKpe($E_1$)
        end if
        return $E_0$
    end if
    if $E_0 := E_1\ op\ E_2$ then     //case of binary operator
        if $E_1$ is of type $t$ then return GetKpe($E_1$)
        end if
        if $E_2$ is of type $t$ then return GetKpe($E_1$)
        end if
        return $E_0$
    end if
    return $E_0$
end function
```

4.3.3 Getting Address. In our algorithm, we often need to obtain the address of an expression `E` (notably kpe/kse) for retrieving its pmd, where `E` may be a variable or (the return value of) a function call, and its type is restricted to pointer or struct containing pointer members. This procedure works as follows.

- If `E` does not have side effects, and is not a conditional expression, then we return `&E`. For example, if `E` is `p` or `p[i]`, then we return `&p` or `&p[i]`, respectively.
- If `E` has simple side effects (including function calls), or is a conditional expression, then we return its address via

a simple inference, or the address of the variable storing return value. For example, if `E` is `++p`, then we return `&p`. If `E` is `func()`, `va_arg()` or `cond?TE:FE`, then we return `&PRFret_ID`, where `PRFret_ID` is a new variable storing the return value by rewriting `E` (cf. rewriting function calls in Section 4.6).
- If `E` has side effects that cannot be addressed above, then we return the name of the pointer variable storing its address. For example, if `E` is `p[++i]`, then `E` is replaced by "`*(PRFaddr_ID = &(p[++i]))`" with `PRFaddr_ID` returned, where `PRFaddr_ID` is a new pointer variable storing its address.

Algorithm 2 describes how to decide whether an expression `E` has side effects. It uses structural recursions on unary and binary operators to find a sub expression or an operator with side effects, e.g., function call, prefix, postfix and assignment. Note that side effects play an important role in getting addresses. As we will see, an address string may be used in multiple places in an instrumented expression, thus side effects in the string may incorrectly change the program state. The above procedure ensures that the string is free of side effects.

Algorithm 2 Decide whether an expression E has side effects

```
1: function HasSideEffects($E$)
2:     $E \leftarrow$ IgnoreCasts($E$)
3:     if $E$ is a function call then return true
4:     end if
5:     if $E := op\ E_1$ then     //case of unary operator
6:         if $op$ is prefix/postfix ++ or −− then return true
7:         end if
8:         return HasSideEffects($E_1$)
9:     end if
10:    if $E := E_1\ op\ E_2$ then     //case of binary operator
11:        if $op$ is assignment then return true
12:        end if
13:        if HasSideEffects($E_1$) := true then return true
14:        end if
15:        if HasSideEffects($E_2$) := true then return true
16:        end if
17:        return false
18:    end if
19:    return false
20: end function
```

4.3.4 Getting Status, Base and Bound. In our algorithm, we often need to get the status, base and bound of the referent of a pointer constant expression (notably kpe). Note that pointer constants do not have memory addresses for indexing in the pmd table. Thus, we never get their addresses as above. This procedure works according to the type of pointer constant expression as follows.

- Casts from non-pointer expressions, e.g., `(T*)E`. The status of the referent is global, as the user tries to directly access a globally visible address. If the expression is `(T*)0`, then its base and bound are both 0. If it is another cast, e.g., `(T*)(i+1)`, then its base is `(char*)E` and bound is `(char*)E+sizeof(T)`, as this access is usually desired by the user.
- String literals, e.g., `"string"`. The status is global. Its base is `"string"` and bound is `"string"+strlen("string")+1`.
- Function names, e.g., `func` and `&func`. The status is function. Its base is `func` and bound is `(char*)func+1`.
- Array names, e.g., `arr` and `&arr`. The status is the same as `arr`. Its base is `arr` and bound is `(char*)arr+sizeof(arr)`.
- Variable addresses, e.g., `&var`. The status is the same as `var`. Its base is `&var` and bound is `&var+1`.
- Addresses of array elements, e.g., `&arr[M][N]`. The status is the same as `arr`. Its base is `arr` and bound is `(char*)arr+sizeof(arr)`, as it may be used to traverse the array.
- Addresses of struct members, e.g., `&s.m`. The status is the same as `s`. Its base is `&s.m` and bound is `&s.m+1`.
- Addresses of pointed members, e.g., `&(ps->p->m)`. The status is the same as the referent of `p`, but should be obtained by retrieving the pmd of `ps->p` in the pmd table. Its base is `&(ps->p->m)` and bound is `&(ps->p->m)+1`.

4.4 Variable Definitions and Assignments

Variable definitions and assignments have a similar form "`lhs=rhs`", although definitions may additionally include the type of lhs, and may exclude the initial value rhs. Thus, our algorithm consists of modules shared by the two objectives. Our instrumentation rule depends on the type of lhs, i.e., a pointer, struct containing pointer members or array. We discuss these cases as follows.

4.4.1 The lhs is a pointer. Suppose there is a definition of a pointer variable with an initial value "`T p = E(kpe)`" or an assignment "`p = E(kpe)`", where `E(kpe)` contains a kernel pointer expression `kpe`. Then it is rewritten as follows.

(a) If `kpe` is a variable, then it is replaced by:

```
p = (T)PRFpmd_tbl_update_ptr_ret(&p, &kpe, E(kpe))
```

Note that this replacement requires our algorithm to first get the kernel pointer expression `kpe` from `E(kpe)` and then the address of `kpe`, denoted by `&kpe`. These two operations have been presented in Section 4.3, where special cases like side effects have been well considered. Note that the inserted address will be correctly computed at runtime, even if `kpe` has side effects, as we exploit the order of evaluating function arguments, i.e., from right to left.

(b) If `kpe` is a constant, then it is replaced by:

```
p = (T)PRFpmd_tbl_update_as_ret(&p, AS, BASE, BOUND, E(kpe))
```

where `AS`, `BASE` and `BOUND` are the status, base and bound of the referent of `kpe`. Note that this replacement requires our algorithm to first get the kernel pointer expression `kpe` from `E(kpe)` and then the status, base and bound of `kpe`'s referent. These two operations have been presented in Section 4.3.

(c) If `kpe` is a function call, e.g., "`p = E(func(a1, an))`", then it is replaced by "`p = E(PRFfunc(&p, a1, an))`", where `PRFfunc` is a wrapper function which updates the pmd of `p` (cf. Section 4.6, where we use `&p` as the address of the variable storing return value).

4.4.2 The lhs is a struct containing pointer members. Suppose there is a definition of struct variable with an initial value "`struct st obj = E(kse)`" or an assignment "`obj = E(kse)`", where `E(kse)` contains a kernel struct expression `kse`. It is rewritten as follows.

(a) If `kse` is a variable, then it is replaced by:

```
obj = (PRFpmd_tbl_update_ptr(&(&obj)->p1, &(&kse)->p1),
       PRFpmd_tbl_update_ptr(&(&obj)->pn, &(&kse)->pn), E(kse))
```

where we update the pmds of pointer members `p1`, ..., `pn` of `obj` by exploiting the parenthesis operators. If `kse` has side effects, then it is replaced by:

```
obj = (PRFrhs_ID = E(*(PRFaddr_ID = &(kse))),
       PRFpmd_tbl_update_ptr(&(&obj)->p1, &PRFaddr_ID->p1),
       PRFpmd_tbl_update_ptr(&(&obj)->pn, &PRFaddr_ID->pn),
       PRFrhs_ID)
```

where `PRFaddr_ID` is a new pointer variable storing the address of `kse`, and `PRFrhs_ID` is a new variable storing the value of `E(kse)`. We also need to insert their definitions.

(b) If `kse` is a function call, e.g.,"`obj = E(func(a1, an))`", then it is replaced by "`obj = E(PRFfunc(&obj, a1, an))`", where `PRFfunc` is a wrapper function which updates the pmds of the pointer members of `obj` (cf. Section 4.6, where we use `&obj` as the address of the variable storing return value).

(c) If `E` is an init list expression in a definition of struct variable, e.g.,

```
struct st obj = {E1(kpe1), En(kpen)} //member list or
struct st obj = {E1(kse1), En(ksen)} //nested member list
```

then we consider it as the initialization of a list of pointer variables: `obj.p1`, ..., `obj.pn`, or a list of struct variables: `obj.obj1`, ..., `obj.objn`, respectively. Thus its instrumentation is similar to the above cases by aligning these variables with their corresponding initial values. If the init list expression is a nested member list for a nested struct, then we use a recursive algorithm which enumerates all members and instruments them recursively.

4.4.3 *The lhs is an array.*

Suppose there is a definition of pointer array or struct array containing pointer members, with an init list expression:

```
T parr[M][N] = { {E11(kpe11), E1N(kpe1N)},
                 {EM1(kpeM1), EMN(kpeMN)} };
struct st sarr[M][N] = { { {E111(kpe111), E11n(kse11n)},
                           {E1N1(kpe1N1), E1Nn(kse1Nn)} },
                         { {EM11(kpeM11), EM1n(kseM1n)},
                           {EMN1(kpeMN1), EMNn(kseMNn)} } };
```

then we consider it as the initialization of a list of pointer variables: `parr[0][0]`, ..., `parr[M-1][N-1]`, or a list of struct variables: `sarr[0][0]`, ..., `sarr[M-1] [N-1]`, respectively. Thus its instrumentation is similar to the above cases by aligning these variables with their corresponding initial values.

4.5 Dereferences

Suppose there is a unary dereference of pointer "`*E(kpe)`", where `E(kpe)` is of type `T` and contains a kernel pointer expression `kpe`. Our instrumentation rule depends on whether `E(kpe)` is a function pointer. If `E(kpe)` is not a function pointer, then the unary dereference is rewritten as follows.

(a) If `kpe` is a variable, then it is replaced by:

```
*(T)PRFcheck_dpv(PRFpmd_tbl_lookup(&kpe), E(kpe), SIZE, ...)
```

where `SIZE` is the size of `E(kpe)`'s referent type, and "`...`" denotes omitted arguments containing the precise location of `E(kpe)`.

(b) If `kpe` is a constant, then it is replaced by:

```
*(T)PRFcheck_dpc(BASE, BOUND, E(kpe), SIZE, ...)
```

where `BASE` and `BOUND` are the base and bound of the referent of `kpe`.

(c) If `kpe` is a function call, e.g., "`*E(func(a1, an))`", then it is replaced by:

```
*(T)PRFcheck_dpv(PRFpmd_tbl_lookup(&PRFret_ID),
                 E(PRFfunc(&PRFret_ID, a1, an)), SIZE, ...)
```

where `PRFfunc` is a wrapper function which updates the pmd of `PRFret_ID` (cf. Section 4.6, where we use `&PRFret_ID` as the address of the variable storing return value).

If `E(kpe)` is a function pointer, the instrumentation rule is similar to the above case, but we will replace `PRFcheck_dpv` and `PRFcheck_dpc` by `PRFcheck_dpfv` and `PRFcheck_dpfc`, respectively, and remove the argument `SIZE`.

The instrumentation rule of array subscript expressions and member expressions is similar to unary dereferences. For example, the member expression `E(kpe)->mi` is equivalent to the unary dereference expression "`(*(ti*)&(E(kpe) ->mi))`".

4.6 Function Definitions and Calls

A function may be called in two forms:

- Call by name, e.g., "`ret = func(a1(kpe1), an(kpen))`", where `ret` may be absent if we do not store the return value, `func` is a function name, `a1(kpe1)` and `an(kpen)` are arguments that contain kernel pointer expressions `kpe1` and `kpen`, respectively.
- Call by function pointer, e.g., "`ret = E(fp)(a1(kpe1), an(kpen))`", where `E(fp)` contains a kernel pointer expression `fp`.

Our instrumentation rule generally includes three parts: (1) rewrite the definition of the called function, (2) insert the definition of the wrapper function, and (3) rewrite the function call expression. The rule depends on the above two call forms and whether the called function is variadic, described below.

4.6.1 *Call non-variadic functions by name.*

Suppose there is a non-variadic function with *n* parameters "`T func(t1 p1, tn pn)`", where `T`, `t1` and `tn` may be pointers or structs containing pointer members. For example, let us consider the case of pointers. We first rewrite the function definition as follows.

- Add code fragments to initialize the pmds of parameters by retrieving the fmd table, at the beginning of the function definition.
- Replace each return statement to store the pmd of return value in the fmd table, and jump to the end of the code block.
- For each loop, we add a new variable to record whether a break or continue statement was executed in the loop. Then we replace each break and continue statement in the loop by statements that set this variable and jump to the end of its code block.
- At the end of each code block, we add statements that remove the pmds of the pointer variables defined in this block and jump to an appropriate statement depending on whether a return, break or continue statement was executed.
- At the end of the function, we add statements that remove the pmds of the pointer variables defined in this function including its parameters, and set the status of stack variables to invalid.

Then we insert a wrapper function definition `PRFfunc`, which takes additional parameters to pass the pmds of arguments from the caller to `func`, and pass back the pmd of return value to the caller. The wrapper function first stores the pmd parameters to the fmd table, then calls the original function, and finally extracts the pmd of return value from the fmd table.

Finally, we rewrite the function call expression by renaming and inserting additional arguments to redirect the function call to the wrapper function. Our instrumentation rule depends on the storage classes of `kpe1` and `kpen`, i.e., variables, constants or function calls. For example, if `kpe1` and `kpen` are variables, then it is replaced by:

```
ret = PRFfunc(&ret, PRFpmd_tbl_lookup(&kpe1),
        PRFpmd_tbl_lookup(&kpen), a1(kpe1), an(kpen));
```

where `&ret` takes back the pmd of return value via the pmd table. Note that, if `ret` is absent, we use `&PRFret_ID` as the address of the variable storing return value, instead of `&ret`. If `kpe1` and `kpen` are function calls, then it is replaced by:

```
ret = PRFfunc(&ret, PRFpmd_tbl_lookup(&PRFret_ID1),
        PRFpmd_tbl_lookup(&PRFret_IDn), a1(kpe1), an(kpen));
```

Note that function calls `kpe1` and `kpen` are nested function calls in `func`. As our algorithm recursively traverses all AST nodes, thus `kpei` will be further rewritten according to the instrumentation rule of function calls, by using `&PRFret_IDi` as the address of the variable storing return value. For example, consider nested function call `ret = f1(f2(p))`, it is replaced by:

```
ret = PRFf1(&ret, PRFpmd_tbl_lookup(&PRFret_ID1),
              PRFf2(&PRFret_ID1, PRFpmd_tbl_lookup(&p), p));
```

4.6.2 Call variadic functions by name. Suppose there is a variadic function with *n* fixed parameters "`T func(t1 p1, tn pn, ...)`". The function is called in the form "`ret = func(a1(kpe1), an(kpen), va1(vkpe1), vam(vkpem))`". We only need to modify the above rule slightly: (1) When we rewrite the function definition, insert a new variable `PRFptr_cnt` to count the number of pointers before variadic parameters. (2) We insert a wrapper function definition `PRFfunc_ID` *for each function call to the variadic function* before the definition of the caller function. Its function body is similar to the non-variadic case. (3) We rewrite the function call expression as the non-variadic case, but should additionally provide the pmds of variadic arguments.

A special case in rewriting variadic function definitions is `va_arg` expressions in the form "`ret = va_arg(ap, vti)`", replaced by:

```
ret = (PRFpmd_tbl_update_fpmd(&ret,
        PRFfmd_tbl_lookup_fpmd(cur_func, PRFptr_cnt++)),
      va_arg(ap, vti))
```

where `cur_func` is the current variadic function that calls `va_arg`. Note that we cannot provide wrapper functions for `va_arg` expressions, because `va_arg` can only take effects in the body of the current variadic function. Thus, its instrumentation is largely different from other function calls.

To instrument function calls via function pointers, we only need to modify the above rule slightly.

5 EVALUATION

We evaluate Movec against three state-of-the-art dynamic analysis tools, SoCets [20–22, 41], Google's ASan [30], and Valgrind [25, 31] in terms of effectiveness (on detecting errors) and performance (in terms of execution time and memory consumption) using a set of 86 microbenchmarks (with ground truth about the memory errors present) [2] and a set of 10 representative MiBench benchmarks [11]. We use Valgrind 3.11.0 from the latest Ubuntu repository, the latest version of SoCets for LLVM-3.4, and ASan for LLVM-6.0.

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.

5.1 Effectiveness

The first experiment aims at evaluating the effectiveness of our approach in detecting various memory errors. However, we found that existing benchmarks do not cover many typical memory errors. Thus, we have decided to develop a set of microbenchmarks that are designed to include all known types of memory errors. Currently, our benchmark suite consists of 86 programs, most of which are written based on the typical errors reported in the literature [2]. There are 79 unsafe programs seeded with various memory errors and 7 safe programs, where each unsafe program contains one clearly marked error. Note that the benchmarks cover various memory errors (spatial and temporal, segment errors [4], memory leaks, read/write/library errors), triggered by a variety of language constructs (pointers, structs, pointer arrays, struct arrays, global/static/stack/heap variables, constants, and function calls), and are classified into two categories: `all-mem-err` and `c-syntax`.

We compare Movec and the three state-of-the-art tools in terms of their effectiveness using these microbenchmarks. Under `-O0` (with all compiler optimizations turned off), Movec is successful in detecting all the 79 errors in our benchmark suite, while SoCets, ASan and Valgrind can detect only 59, 67 and 15 errors, respectively (with SoCets crashing in compiling 6 programs). For the 7 safe programs, SoCets is the only one reporting false positives for all 7 safe programs (1 per program).

We have also considered two higher optimization levels, `-O1` and `-O3`. Movec can still detect all the 79 errors under both optimization levels, showing that Movec is insensitive to compiler optimizations. However, this is not the case for the three state-of-the-art tools. Under `-O1`, the numbers of errors detected by SoCets, ASan and Valgrind have dramatically dropped to 3, 6 and 7, respectively. Under `-O3`, these numbers have dropped further to 1, 4 and 6, respectively. The more aggressive the optimization level is, the less effective these tools are. These results show that Movec significantly outperforms the state-of-the-art tools in their bug-finding effectiveness, regardless of the optimization level used.

Movec is insensitive to compiler optimizations since its source-level instrumentation happens before any optimization is applied. In contrast, the state-of-the-art tools are sensitive to compiler optimizations due to the IR-level or binary-level instrumentation used, since some buggy statements have been optimized away. This leads to a dilemma for the existing dynamic analysis tools. The compiler designers usually advise the user to consider `-O1` or higher in order to boost their program performance, but this will significantly reduce their bug-finding capability. Therefore, in order to ensure maximal effectiveness, the user has to turn off all compiler optimizations, but this will often reduce program performance unduly. In this regard, Movec is therefore significant as its effectiveness is no longer affected by the optimization level used.

5.2 Performance

The second experiment aims at evaluating the runtime instrumentation overhead incurred by our approach. We have used MiBench

Table 1: Performance of SoCets, ASan, Valgrind and Movec on MiBench

Programs	Original -O3		SoCets -O0				ASan -O0				Valgrind -O0				Movec -O0			
	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.
adpcm (s)	0.032	1556	0.096	1756	3.00	1.13	0.077	6354	2.41	4.08	0.479	25732	14.97	16.54	0.142	1564	4.44	1.01
adpcm (l)	0.373	1480	1.398	1752	3.75	1.18	0.806	6528	2.16	4.41	5.238	25736	14.04	17.39	2.403	1564	6.44	1.06
basicmath (s)	0.018	1480	0.023	2356	1.28	1.59	0.033	6884	1.83	4.65	0.468	27564	26.00	18.62	0.041	1568	2.28	1.06
basicmath (l)	0.266	1484	0.308	2420	1.16	1.63	0.432	6948	1.62	4.68	7.841	27864	29.48	18.78	0.459	1468	1.73	0.99
bitcount (s)	0.014	1484	0.043	1776	3.07	1.20	0.038	6576	2.71	4.43	0.328	26056	23.43	17.56	0.096	1628	6.86	1.10
bitcount (l)	0.085	1548	0.493	1896	5.80	1.22	0.303	6504	3.56	4.20	3.025	26044	35.59	16.82	1.311	1564	15.42	1.01
CRC32 (s)	0.018	1488	0.038	1748	2.11	1.17	0.033	6560	1.83	4.41	0.344	25832	19.11	17.36	0.119	1560	6.61	1.05
CRC32 (l)	0.198	1548	0.496	1812	2.51	1.17	0.275	6480	1.39	4.19	4.332	25828	21.88	16.68	2.154	1632	10.88	1.05
FFT (s)	0.023	1564	0.038	2388	1.65	1.53	0.052	7160	2.26	4.58	0.651	26560	28.30	16.98	0.083	1556	3.61	0.99
FFT (l)	0.112	1620	0.195	2760	1.74	1.70	0.198	7620	1.77	4.70	2.398	26856	21.41	16.58	0.504	1552	4.50	0.96
patricia (s)	0.024	2016	0.045	9644	1.88	4.78	0.099	9060	4.13	4.49	0.688	27664	28.67	13.72	0.113	3972	4.71	1.97
patricia (l)	0.097	7680	0.205	45828	2.11	5.97	0.239	16396	2.46	2.13	3.287	34768	33.89	4.53	0.637	18308	6.57	2.38
qsort (s)	0.012	2076	4.122	8104	343.50	3.90	0.025	9048	2.08	4.36	0.518	27728	43.17	13.36	0.102	2224	8.50	1.07
qsort (l)	0.057	3024	56.537	7828	991.88	2.59	0.096	9368	1.68	3.10	1.301	28968	22.82	9.58	0.419	3336	7.35	1.10
sha (s)	0.011	1568	0.023	1876	2.09	1.20	0.025	6588	2.27	4.20	0.495	41508	45.00	26.47	0.058	1628	5.27	1.04
sha (l)	0.035	1564	0.148	1884	4.23	1.20	0.089	6540	2.54	4.18	0.964	41508	27.54	26.54	0.519	1564	14.83	1.00
stringsearch (s)	0.005	1560	0.007	1820	1.40	1.17	0.017	6768	3.40	4.34	0.119	25776	23.80	16.52	0.007	1548	1.40	0.99
stringsearch (l)	0.009	1636	0.011	2208	1.22	1.35	0.019	6656	2.11	4.07	0.142	25884	15.78	15.82	0.018	1556	2.00	0.95
susan (s)	0.013	1484	0.077	2516	5.92	1.70	0.205	8060	15.77	5.43	0.594	26516	45.69	17.87	0.211	1476	16.23	0.99
susan (l)	0.069	2100	1.119	3248	16.22	1.55	0.422	9536	6.12	4.54	2.416	27588	35.01	13.14	3.042	2280	44.09	1.09
AVERAGE	0.074	1998			69.83	1.95			3.21	4.26			27.78	16.54			8.69	1.14
Over Movec-O0					8.04	1.70			0.37	3.73			3.20	14.47				
Over Movec-O3					**22.59**	**1.70**			**1.04**	**3.71**			**8.99**	**14.42**				

[11], which is a free and commercially representative embedded benchmark suite, covering a variety of application areas such as automotive, consumer, network, office, security and telecommunication. We have selected a set of 10 representative programs, where `patricia`, `qsort` and `susan` contain a number of memory leaks and the remaining seven are memory safe.

We run each tool on a program and its instrumented version ten times using the supplied small (s) and large (l) input datasets, and collect its execution time and memory consumption using the arithmetic average over the ten runs. The execution time of a program accounts for the real 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. The execution time and memory consumption of a program are reported using GNU's `time`, measured in seconds and kilobytes respectively. Furthermore, we also measure the runtime overhead in terms of the time ratio (T.R.) and memory ratio (M.R.) of an instrumented program, i.e., the execution time (memory consumption) of an instrumented program over that of the original program. For example, the time ratio 3.75 means that an instrumented program is 3.75x slower than the original one, indicating a time overhead of 275%.

Let us first examine the results obtained by all the tools under `-O0`. For effectiveness, Movec has found all the errors present in the benchmarks without any false positive or false negative. However, SoCets and Valgrind fail to detect any memory leak while ASan fails to detect memory leaks in `qsort`. Once again, Movec outperforms these existing tools in terms of their effectiveness. The performance results are given in Table 1. Movec exhibit runtime overheads ranging from 1.40x to 44.09x, with an average of 8.69x and memory overheads ranging from 0.95x to 2.38x with an average of 1.14x. In contrast, SoCets, ASan and Valgrind suffer from runtime overheads at 69.83x, 3.21x and 27.78x, respectively, much larger memory overheads at 1.95x, 4.26x and 16.54x, respectively. Relative to Movec, SoCets is 8.04x slower and consumes 1.70x more memory, ASan is faster at 0.37x but consumes 3.73x more memory, and Valgrind is 3.20x slower and consumes 14.47x more memory. These results show that Movec outperforms SoCets and Valgrind in terms of both execution time and memory consumption considered individually, and is comparable with ASan in terms of both execution time and memory consumption considered together.

For a number of programs, SoCets is faster than Movec. Currently, Movec is purely dynamic while SoCets takes advantage of static analysis (including dominator-based redundant check elimination [21, 41] and dataflow analysis [20, 22]) to remove unnecessary and redundant runtime checks. While being faster than Movec for a number of programs, SoCets is extremely slow for `qsort` (a slowdown of 343x), because a large number of checks on pointer manipulations cannot be removed by its static analysis. This means that SoCets should have shown higher overhead if its static analysis is turned off. Conversely, Movec is expected to close the performance gap once it also applied static analysis similarly.

The performance difference between Movec and ASan is primarily due to the two different instrumentation algorithms used. Movec adopts a pointer-based approach, while ASan takes an object-based approach by implementing the metadata space with a large shadow space with often unused memory. Therefore, ASan is generally faster than Movec, but at the expense of consuming much more memory (for almost all the programs evaluated) . Unlike pointer-based approaches, however, object-based approaches do not provide complete spatial safety, since sub-object overflows (e.g., overflows of accesses to arrays inside structs) are missed [13, 17, 25, 26, 30, 31, 39].

5.3 Effectiveness and Performance Combined

Movec stands out as the clear winner when both effectiveness and performance are considered together. We have evaluated Movec under `-O3` in terms of the same 10 MiBench benchmarks used before. Under `-O3` (just like `-O0` discussed in Section 5.2), Movec continues to find all the memory errors in `patricia`, `qsort` and `susan`, and exhibits runtime overheads ranging from 1.20x to 9.16x with an average of 3.09x and memory overheads ranging from

0.95x to 2.38x with an average of 1.15x. Recall that Movec can run under `-O3` without any effectiveness loss, whereas SoCets, ASan and Valgrind must run under `-O0` to ensure their maximal effectiveness. As shown in the last row of Table 1, relative to Movec under `-O3`, SoCets, ASan and Valgrind all have larger runtime overheads at 22.59x, 1.04x and 8.99x, respectively, and larger memory overheads at 1.70x, 3.71x and 14.42x, respectively,

These results show that Movec significantly outperforms the state-of-the-art tools in both execution time and memory consumption, if the user wants to ensure maximal effectiveness.

5.4 Accessibility

Compared to existing dynamic analysis tools, Movec is easier to use and more user-friendly, as it reports more detailed source-level error information for debugging purposes, e.g., error types and precise locations in source code, including names of the files and functions containing errors, line and column numbers, and even the code texts of the accesses causing errors, e.g., dereferenced pointer names. Figure 6 shows the reports produced by Movec, SoCets, ASan and Valgrind, when a temporal error is detected in `2_ph_vh.c`, which manipulates linked lists. The report from Movec looks like that from a compiler, while the reports from the other tools mainly print backward traces, which are less understandable. Thus, Movec can help programmers locate the root causes of memory error more quickly. This feature largely benefits from the proposed source-level instrumentation, which makes all original source information visible to our tool, e.g., identifiers and source locations. In contrast, the capability of other tools is partly limited by the loss of the original source information at the IR-level or binary level.

```
2_ph_vh.c: In function 'main':
2_ph_vh.c:46:11: error: dereferenced pointer 'head->next->value
      ' (val = 0x1ad7350, size = 4) points to an invalid object
      (original block is [0x1ad7350, 0x1ad7360)). [temporal
      error]
1 error generated.
```

(a) Report from Movec

```
[TLDC] Key mismatch key = 8, *lock=7ffb139de010, next_ptr=7
      ffb139de008
Softboundcets: Memory safety violation detected
Backtrace:
./a.out[0x406d85]
(...omitted backtrace...)
Aborted (core dumped)
```

(b) Report from SoCets

```
==5241==ERROR: AddressSanitizer: heap-use-after-free on address
       0x602000000030 at pc 0x000000515d7f bp 0x7ffd3a603790 sp
      0x7ffd3a603788
READ of size 4 at 0x602000000030 thread T0
(...omitted backtrace...)
SUMMARY: AddressSanitizer: heap-use-after-free (/home/a.out+0
      x515d7e) in main
==5241==ABORTING
```

(c) Report from ASan

```
==3769== Invalid read of size 4
==3769==    at 0x4006E2: main (in /home/a.out)
==3769== Address 0x5204090 is 0 bytes inside a block of size 16
       free'd
(...omitted backtrace...)
==3769== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0
       from 0)
```

(d) Report from Valgrind

Figure 6: Reports on memory errors.

6 RELATED WORK

IR-level instrumentation is the most popular way for implementing dynamic analysis tools. It inserts code fragments into the IR code of a program to generate instrumented IR code or a binary executable. The most frequently used IRs include CIL (C Intermediate Language) [12, 23, 24, 37], GCC's SSA (Single Static Assignment) form [14, 29], LLVM-IR [7, 20–22, 30, 32, 33, 38, 41] and some others [1, 27]. The major advantage is that the instrumentation algorithm only needs to handle simple statements. Indeed, the original source code has been significantly simplified by the time the tools run, and the simplification greatly reduces the variety of programs to be handled. For example, CIL consists of a simplified subset of the C programming language, while SSA contains only simple operations. Thus, they can be easily manipulated in instrumentation.

In contrast, one significant limitation of our source-level instrumentation is the requirement of devoting more engineering efforts in its implementation to handle the variety of programs, as discussed in Section 3.

However, IR-level instrumentation has several limitations, as it is optimization-sensitive, platform-dependent, and non-compliant to DO-178C. Besides, it only reports limited error information, due to the loss of original source information at the IR-level.

Binary-level instrumentation directly inserts code fragments into the object code or binary executable of a program, where binaries may be instrumented statically or dynamically. For example, Purify performs instrumentation on object code [13], while Valgrind performs dynamic binary instrumentation [25, 26, 31]. It can also be implemented as a Linux library that can be statically or dynamically linked with the application, to interpose system calls and the memory function family [16, 40]. The major advantage is that it can handle programs even when source code is unavailable.

In contrast, one limitation of our source-level instrumentation, as well as IR-level, is the requirement of source (or IR) code.

However, binary-level instrumentation also share all the limitations of IR-level instrumentation. Besides, binary-level instrumentation usually incurs heavy runtime overhead, because the inserted code fragments cannot be optimized after instrumentation, as compilation has been finished by the time the tools run.

7 CONCLUSION

We have introduced a new source-level instrumentation technique for implementing a pointer-based approach for detecting memory errors. Our approach is optimization-insensitive, generates platform-independent source code, which can be further inspected and verified by following DO-178C. Our tool Movec outperforms state-of-the-art tools in terms of both effectiveness and performance considered together, and reports more detailed error information.

More information about Movec and the download link are available at https://drzchen.github.io/projects/movec.

ACKNOWLEDGMENTS

This work is supported by the Joint Research Funds of National Natural Science Foundation of China and Civil Aviation Administration of China (No. U1533130), and the Key Laboratory of Safety-Critical Software, Ministry of Industry and Information Technology (No. XCA18164-02).

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] Zhe Chen. 2019. Movec-MSBench: A Memory Safety Benchmark Suite, Version 1.0.0. https://github.com/drzchen/movec-msbench

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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)*, Armin Biere and Roderick Bloem (Eds.), Vol. 8559. Springer, 491–507.

[9] RTCA DO-178C. December 2011. *Software Considerations in Airborne Systems and Equipment Certification*. RTCA, Inc.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] Lian Li, Cristina Cifuentes, and Nathan Keynes. 2010. Practical and effective symbolic analysis for buffer overflow detection. In *Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2010*. ACM, 317–326.

[16] 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.

[17] 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.

[18] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2012. Watchdog: Hardware for safe and secure manual memory management and full memory safety. In *39th International Symposium on Computer Architecture (ISCA 2012)*. IEEE Computer Society, 189–200.

[19] Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. 2014. WatchdogLite: Hardware-Accelerated Compiler-Based Pointer Checking. In *12th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2014*. ACM, 175–184.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] Grigore Rosu, Wolfram Schulte, and Traian-Florin Serbanuta. 2009. Runtime Verification of C Memory Safety. In *Proceedings of the 9th International Workshop on Runtime Verification, RV 2009 (Lecture Notes in Computer Science)*, Vol. 5779. Springer, 132–151.

[29] 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.

[30] 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.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.