MEDS: The Memory Error Detection System

Jason D. Hiser, Clark L. Coleman, Michele Co, and Jack W. Davidson

Department of Computer Science
University of Virginia
Charlottesville, Virginia U.S.A.
{hiser,clc5q,mc2zk,jwd}@cs.virginia.edu

Abstract. Memory errors continue to be a major source of software failure. To address this issue, we present MEDS (Memory Error Detection System), a system for detecting memory errors within binary executables. The system can detect buffer overflow, uninitialized data reads, double-free, and deallocated memory access errors and vulnerabilities. It works by using static analysis to prove memory accesses safe. If a memory access cannot be proven safe, MEDS falls back to run-time analysis. The system exceeds previous work with dramatic reductions in false positives, as well as covering all memory segments (stack, static, heap).

1 Introduction

Modern computer software is absolutely essential to today's information and communications infrastructure. Software provides high performance, upgradeable, patchable, and diverse functionality in nearly every computer operated device from desktop and laptop computers, to cell phones and PDAs, and even automotive, marine, and aeronautical environments.

Considering the importance of computer software, it is no surprise that we demand software be reliable. Software must be reliable in two ways: it must be free from bugs which crash the program, and it must be free from vulnerabilities which might let a malicious user attack the system. Current software development practices yield a variety of memory errors: buffer overflow, array out-of-bound, double-free, and uninitialized pointer dereference errors. These errors may be relatively benign and only cause a program to crash. However, in a critical application, there may be significant dangers, such as important data being lost, aeroplanes being off course, or breach of security if the bug is exploited by a malicious attacker.

While there are a variety of memory-safe languages, such as Java and C#, that are available to address this problem, developers are reluctant or unable to use such languages for a variety of reasons. First, there is significant cost associated with maintaining memory safety, and software developers may decide that the system performance goals cannot be met with a memory safe language. Second, many applications existed before the wide availability of such languages, and the cost of moving from a non-safe language to a safe language is prohibitive. Finally, language support might be lacking on the platform in which the system is to be deployed. For example, Java run-time libraries might not exist for an embedded GPS device. Consequently, memory safe languages are not always a viable option for software.

An even worse situation arises when the software's source code is not available. Such a situation could occur because some or all of the software comes from an untrust-

ed third party. For example, a third party library might be used for font rendering, or the entire project might be developed by an untrusted source and used in critical infrastructure. Another example is black-box testing, where the tester must find memory errors with no access to the source code. Thus, a tool to detect memory errors requiring only the binary executable file would be of great use for testing and security purposes.

To meet this need, we have developed MEDS: The Memory-Error Detection System. MEDS operates by using a combination of static and dynamic analysis to achieve a variety of advances to the state-of-the-art in binary-only memory error detection. MEDS provides:

1. Comprehensive protection of all memory segments, including global-static data, heap-allocated data structures, and stack allocations without requiring debug information. Previous techniques required debug information and failed to protect the stack due to excessive false positive rates.
2. Significant reduction of false positive rates. Previous techniques had several categories of false positives which would render those techniques useless for on-line protection, and much less desirable for offline use.
3. Aggressive static and dynamic analysis provide higher levels of protection with run-time performance comparable to previous, less effective techniques.
4. No reliance on source code, object code, or debugging information. Many memory error detectors require source code or source code changes, or object code so that custom allocation and tracking libraries can be linked.
5. An optional profiling step to help separate false positives from actual errors and provide further performance benefits. Previous techniques have no mechanisms for helping a user differentiate real errors from falsely reported errors.

Together, these advances demonstrate that MEDS provides significant improvement to state-of-the-art memory overwriting defences.

The remainder of the paper is organized as follows: Section 2 describes the MEDS system in detail, while Section 3 gives experimental evidence of MEDS' effectiveness. Section 4 discusses related work, and finally Section 5 summarizes our findings.

2 MEDS

2.1 System Overview

MEDS, as shown in Fig. 1, takes a binary program as input. The binary is used for static analysis and to prepare the mmStrata run-time system. MEDS first prepares a run-time system using a binary instrumentation tool we call the Stratafier, so named because it inserts a software dynamic translation system called Strata into an executable image.

After the binary has been "stratafied," MEDS runs a static analysis step to create an annotations file. The annotations file contains information obtained during all types of analysis to facilitate further analysis and performance improvements. Furthermore, the annotation file is the means for communication among all MEDS components.

Finally, the run-time system is ready to detect memory errors. When run, the revised program binary is dynamically instrumented by mmStrata. Memory writes that

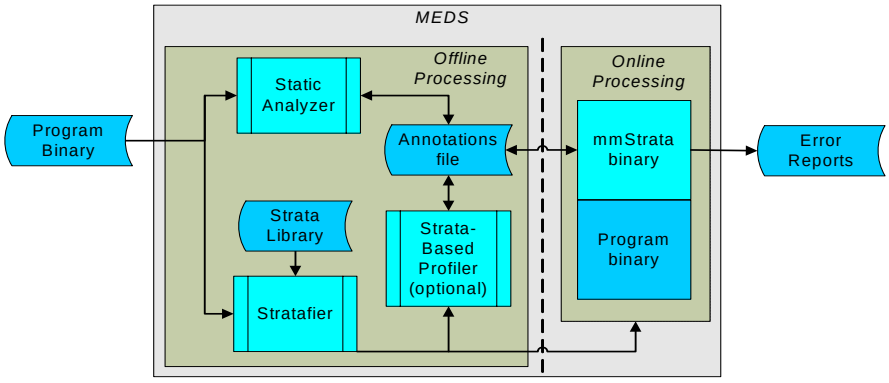

Fig. 1. High-level MEDS overview.

cannot be statically proven safe are checked for safety. Any violations detected result in further annotations (with diagnostic information) which are later reported to the user.

Sections 2.2-2.6 discuss these components in more detail.

2.2 MEDS Type System

To effectively detect memory errors, MEDS stores metadata for every program storage location: each hardware register and memory location is assigned its own metadata. The base system has two types of metadata:

- `n` – a numeric type (i.e. non-pointer) object is held in the storage location.
- $\texttt{p}_{\texttt{obj}}$ – a pointer is stored in the corresponding storage location with referent `obj`. Note that this metadata carries with it the bounds of `obj`, so that dereferencing of this object can be bounds checked. Furthermore, two objects of the same type, with different bounds, receive distinct metadata.

This data is initialized at program start-up, and updated for each program operation so that the metadata is consistent with the value held in each storage location. For example, consider a `mov eax,[0x8100800]` instruction. The metadata associated with storage location `0x8100800` is loaded, and stored into the metadata for register `eax`. For instructions that involve computation, a metadata computation is performed as well. Fig. 2 shows how metadata types are combined to compute a new metadata type. As the figure shows, most operations simply return that the result is numeric. `Add` and `subtract` operations are valid on pointers, and can result in a pointer type. A few operations, namely bitwise `and` and `or` operations, can result in either a pointer or a numeric type when a pointer type is used for input. For these operations, we use a simple heuristic that examines the result value. If the result stays within the referent object, then the result is a pointer, otherwise the result is a numeric.

For efficiency purposes, memory is divided into pointer-sized (4-byte) blocks and one metadata entry is kept for each block. In our implementation, the metadata is a pointer to a bounds-information object. The object contains the metadata type, as well as information about the referent if the type is a pointer. Furthermore, a reference count

+	p	n
p	n	p
n	p	n

-	p	n
p	n	p
n	n	n

&,|	p	n
p	p/n	p/n
n	p/n	n

*, /, %, ^, ~, <<, >>	p	n
p	n	n
n	n	n

Fig. 2. Rules for combining metadata types.

is maintained so we know when it is safe to deallocate the bounds-information in a garbage collected manner. In the dynamic systems, metadata information is kept in a small array for registers, and a splay tree (for fast common case look-ups) for memory. Bounds-information objects are allocated whenever the system detects the allocation of an object by the program, and are lazily deallocated after the corresponding program object is deallocated and the reference count falls to zero.

2.2.1 Challenges

Although the MEDS type system sounds easy to implement, there are a variety of challenges. These challenges are maintaining bounds information, type identification, code identification, and handling non-standard code. This section describes each challenge and gives a high-level view of how it was overcome. Complete details about the solutions are in following sections, as noted.

The first challenge the MEDS system faces is that every object created in the system needs to have a bounds-information object associated with it. For heap objects, this is as simple as having the dynamic system watch calls to allocation and deallocation routines. For static-global and stack objects, allocating the bounds-information object properly is not as easy. The information for static-global variables comes from the static analyzer (see Section 2.4). Stack variables are by definition at variable addresses, and bounds-information objects cannot be allocated at program start-up. Again, the static analyzer helps by analyzing stack frames (again, details in Section 2.4) which the dynamic system uses to create bounds-information objects dynamically. For functions that are unanalyzeable (because they contain dynamic stack allocation via `alloca`, or some other non-standard stack manipulation), the dynamic run-time system creates and updates bounds-information objects for the stack frames (see Section 2.6).

Besides tracking object creation, MEDS also needs to know whether a created object is a numeric type or a pointer to another object to set the metadata for the newly created object. For many objects this is easy, e.g. objects returned from `malloc` never contain a pointer initially. But consider the instruction `mov eax,$0x8108004`. Should `eax` be considered a pointer after this instruction? If so, to what object does it point? In the instruction `lea eax,[esp + 36]`, to which stack frame should `eax` point? If statically allocated memory were to contain the value `0x8108004`, is this value a pointer? Compiler optimizations can produce code in which a pointer is initialized to point outside its referent data object, because accesses through the pointer will always contain an offset to bring the address within the referent object. These are all cases of a general *pointer identification problem*. The static analyzer and the profiler combine forces to resolve questions about any value which might be a pointer (Sections 2.4-2.5).

Another major challenge faced by MEDS is how to locate the executable code within a program. The static analyzer solves this problem (see Section 2.4).

Compiler optimizations and non-standard code can cause the type system to erroneously consider some objects to be numeric. Consider the code in Fig. 3 as an example of code commonly created by a combination of strength reduction and induction variable elimination [1]. In the loop preheader, register `ecx` gets assigned the offset from object `a` to object `b`, eliminating complex address arithmetic and multiple induction variable updates. However, the basic type system assigns `ecx` as type numeric. Thus, both the load and store instructions in the loop are seen as references to variable `a`, when clearly one is a reference to variable `b`. To solve this problem, we extend the basic type system with an offset type, $o_{ptr1,ptr2}$. The offset type is created when the difference between two pointers is taken. In most operations, the offset type behaves as numeric, except in the case of adding a pointer and an offset; when $p_{ptr1}+o_{ptr1,ptr2}$ is calculated, the result is p_{ptr2}. Care must be taken when deallocating objects to ensure that all offset types that reference the object are marked as invalid.

```
for(i=0;i<N;i++)        eax=&a             // eax is ptr to a
   a[i]=b[i];           ecx=&b             // ecx is ptr to b
                        ecx=ecx-eax        // what type is ecx?
                  L1:mov ebx,[eax+ecx]     // ptr to a?
                     mov [eax],ebx
                     add eax,4
                     …
```

Fig. 3. Strength-reduction example.

A problematic non-standard coding style is to use *block* operations to work on an object as an aggregate, e.g. using `memcpy` to copy a list node from one location in memory to a second location. Since pointers are almost always on a word boundary, and most block operations occur on word or double-word boundaries, the common case is handled without problem. However, if `memcpy` or a similar user-written routine uses byte-by-byte operations, then the byte load and byte store operations seem inherently to have a numeric type. Consequently, the copy can cause the type system to lose information about pointers in the destination of the byte-by-byte copy. MEDS solves this problem by considering the most significant byte of a pointer to be a pointer regardless of how or where it is stored. Thus, sign extension and truncation of the "pointer," as one byte of the pointer is copied, results in no issues. Likewise, we only set the metadata for a 4-byte memory storage location if the most significant byte is written.

Before moving to an in-depth description of each tool, we first briefly examine Software Dynamic Translation, a mechanism used to dynamically instrument binaries.

2.3 Software Dynamic Translation

Strata is a software dynamic translation (SDT) system designed for high retargetability and low overhead translation. Strata has been used for a variety of applications including system call monitoring, dynamic download of code from a server, and enforcing security policies [2, 3]. This section describes some of the basic features of Strata which

are important to understanding the experiments presented later. For an in-depth discussion of Strata, please refer to previous publications [4, 5, 6].

2.3.1 Strata Overview

Strata operates as a co-routine with the program binary it is translating, as shown in Fig. 4. As the figure shows, each time Strata encounters a new instruction address (i.e., PC), it first checks to see if the address has been translated into the *fragment cache*. The fragment cache is a software instruction cache that stores portions of code that have been translated from the native binary. The fragment cache is made up of *fragments*, which are the basic unit of translation. If Strata finds that a requested PC has not been previously translated, Strata allocates a fragment and begins translation. Once a termination condition is met, Strata emits any *trampolines* that are necessary. Trampolines are pieces of code emitted into the fragment cache to transfer control back to Strata. Most control transfer instructions (CTIs) are initially linked to trampolines (unless the transfer target already exists in the fragment cache). Once a CTI's target instruction becomes available in the fragment cache, the CTI is linked directly to the destination, avoiding future uses of the trampoline. This mechanism is called *fragment linking* and avoids significant overhead associated with returning to Strata after every fragment [4].

Strata's translation process can be overridden to implement a new SDT use. In this paper, we modify Strata's default translation process to insert instrumentation to enforce the MEDS type system in both the profiler driven analysis (Section 2.5) and the online detector (Section 2.6).

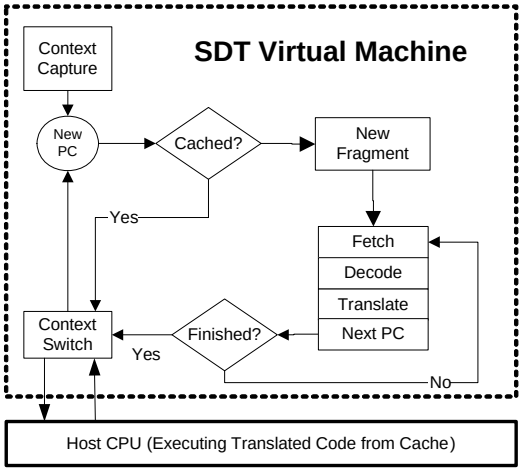

Fig. 4. High-level overview of Strata operation.

2.4 Static Analysis

The MEDS Static Analyzer is implemented as a plug-in to the popular IDA Pro disassembler [7]. After IDA Pro completes its disassembly of the program binary, the static analyzer plug-in analyzes the program and produces *informative annotations*.

A preliminary step is to assist IDA Pro in the disassembly. Disassembly of a program binary involves solving the problem of precisely identifying code and data within the binary. This problem is not perfectly solvable at present. The two basic design approaches for disassemblers, *recursive descent* and *linear scan*, have different strengths and weaknesses when analyzing different program binaries. The static analyzer improves upon the recursive descent approach of IDA Pro by using a linear scan disassembler (the GNU Linux tool `objdump`) to get a second opinion on which addresses are code and which are data. If code identified by `objdump` but not identified by IDA Pro can now be successfully analyzed by IDA Pro, as requested by the static analyzer, the code is incorporated into the IDA Pro code database. This augmentation of IDA Pro's abilities improves the analysis coverage of MEDS and prevents false positives that would arise if code sections escaped static analysis and received no annotations.

Informative annotations are provided to identify all functions and static-global data objects by three attributes: name, starting address, and total size. If the executable has been stripped, the names will be dummy names generated by IDA Pro. The static-global data annotations will be used at run time to create bounds-information objects for static-global memory objects, as described in Section 2.2.1. The function annotations are used to identify the location of key library functions that allocate memory (such as `malloc`), so that heap memory referents can be tracked at run time.

The most important informative annotations describe the run time stack. Functions that allocate a local stack frame, or activation record, with space for local variables must have their stack memory objects tracked at run time. The instruction that allocates space for the stack frame (usually by subtracting the stack frame size from the stack pointer) is identified by a series of annotations that enable the run time dynamic analysis to divide the stack region into local variables, saved registers, and a saved return address. The run time dynamic analysis will then be able to check the bounds of stack references so that a memory access cannot overflow from one stack object to another (e.g. from a local variable to the return address). Precise identification of all sub-regions of the run-time stack permits monitoring of stack accesses without incurring false positives or false negatives. This synergy of static and dynamic analysis is a significant advance over comparable prior work in this area, which relied entirely on dynamic analysis and suffered from numerous stack-related false positives, as described in Section 4.

2.5 MEDS Profiler

The optional MEDS profiler is based on Strata, and uses information from the static analysis phase to mimic the MEDS type system. One of the main goals is to reduce false positives. The MEDS profiler does this by answering the question "is this object a pointer, and if so, to what does it point?"

To answer this question, any time an object might be a pointer (i.e. is pointer-sized and has a value that might legitimately make it a pointer), it is assigned a metadata type of `q`. The `q`-type tells the profiler that the decision on whether the object is a pointer has not yet been made, and carries along information about the creation point of the object. When the result of the operation depends on whether the `q`-type should have been a pointer, the profiler lazily evaluates it to either a `p`-type or an `n`-type by determining if there's a corresponding referent for the object's current value. Ultimately, if a `q`-type is

dereferenced in the program, the profiler records which object was dereferenced. If, at the end of execution, the `q`-type always referenced the same object, then the profiler records that the `q`-type should be created as a pointer type during final execution. Otherwise, the profiler fails to prove that the object was a pointer, and safely reverts to the assumption that the created object is a numeric type. In either case, an informative annotation is written into the annotations file to inform the other parts of the system.

If no profiler information is available (for example, if the profiler was not run), MEDS falls back to a simple heuristic that works quite well. The heuristic assumes that a static constant is a pointer if and only if it is within the bounds of some data object. Section 3 gives evidence that this simple heuristic works well for many benchmarks, but that some benchmarks will require stronger static analysis or profiler information.

2.6 mmStrata

The MEDS dynamic monitoring system is called mmStrata (memory-monitor Strata). It makes the final determination of whether a memory access causes a memory error or not. To make this determination, it strictly enforces the MEDS type system. At program start-up, mmStrata uses informative annotations created during static and profile analyses to make decisions about which static entities are pointers and sets metadata appropriately. A bounds-information object is created for each program object at start-up (statically allocated data, as well as bounds-information objects for the program's incoming arguments that exist on the stack), and the program begins to execute.

During execution, mmStrata watches for newly created objects. For example, calls to dynamic memory allocation routines are considered to create new objects. Likewise, when mmStrata reaches a program point that static analysis has determined creates a new stack frame, mmStrata creates new bounds-information objects for the stack frame. However, some functions fail to have a stack frame clearly identified, and the dynamic system must still create bounds-information objects to protect the non-analyzed stack frame. To create these objects, the dynamic system watches call instructions (or other control flow instructions that cross function boundaries). If the call instruction targets a function that failed the static analysis of the stack frame, then the dynamic system creates a bounds-information object to represent a new, empty stack frame. While within this function, changes in the stack pointer cause a change in bounds to the bounds-information object. For example, if a dynamic stack allocation (perhaps from `alloca`) extends the stack by 700 bytes, the bounds contained within the bounds-information object are extended. Another example is if outgoing arguments for a function call are pushed, the bounds within the bounds-information object are extended, then when the call returns, those arguments are removed from the stack, and the bounds shrink. In one function we analyzed, the stack frame was created by moving 0 into the `ecx` register, then pushing `ecx` 128 times. Consequently, we believe this mechanism for monitoring non-standard stack frames is a key part of providing complete protection.

MEDS deals with stack deallocations the same way. If the stack frame is being watched dynamically, the bounds within the bounds-information object shrink. If the bounds shrink to the point at which the object has negative size, we assume that the stack frame is no longer needed and mark the stack frame as invalid, and mark the bounds-information object ready for deallocation when the reference count falls to zero.

With the information provided by the static analysis phase, the profiler and the MEDS type system, mmStrata can detect a variety of memory errors. For example, mmStrata detects double-free errors by monitoring calls to allocation and deallocation routines. Out of bounds pointer writes are detected by examining the metadata associated with the pointer to determine if the write is in-bounds. Writes to stale objects are detected by examining the valid bit of the bounds-information object. MEDS clearly provides a general defence that can detect most memory errors within a binary executable. The next section discusses limitations where memory errors might be missed.

2.7 Limitations

MEDS currently has several limitations. These limitations are active areas of research.

The first limitation is how signals are handled. The base Strata system watches asynchronous signals correctly, efficiently, and transparently. The mmStrata extensions, however, need to be enhanced. In particular, new bounds-information objects need to be created for the stack area when a signal is caught. We do not expect this to be a challenging extension, but our system currently does not handle signals.

Like signal handling, system calls to `mmap` are a challenge. The `mmap` system call offers a wide variety of ways in which memory can be allocated, including mapping multiple address ranges to the same physical memory. MEDS only handles basic `mmap` calls that simply allocate memory. Again, we do not see this as a challenging extension.

Our system currently only supports statically linked code. Dynamic linking makes communication through the annotation file more difficult, as absolute addresses cannot be used. Instead, MEDS would have to use a path to a dynamically linked library and an offset within the library. The run-time system would have to watch dynamically loaded libraries and adjust its data structures appropriately. For libraries that are chosen based on dynamic input from the user, the system needs the ability to run the static analysis phase online. These same solutions would be necessary for a system that uses self-modifying code or generates code on the fly (such as a JIT). Our system currently implements none of these solutions.

Our system also does not currently handle kernel-level threading. To handle threading efficiently would be significant work, but not impossible. One solution would involving using a locking mechanism to protect all metadata updates. This could be prohibitively expensive, however, and a scheme that moves exclusive metadata access between threads would likely provide better performance. The best mechanism for threading support is an ongoing area of research.

A subtle, yet important limitation is that our system only protects variables at the allocation level. For example, consider a call to `malloc` that allocates the structure shown in Fig. 5. MEDS will provide protection so that no pointer to the object can be dereferenced outside the object. However, the `user_input` buffer may still over- or underflow and cause the program to misbehave. We believe that modern methods for detecting variable types will allow us to solve this problem, but our current implementation of this mechanism is incomplete and not reported here. A related problem is how to deal with third party allocators (such as an arena allocator) built on top of system allocation routines (such as `malloc`, `sbrk`, or `mmap`). If the application uses a third party allocator, MEDS may only provide a very coarse-grained protection for the super-

```
struct foo {
    int rootID;
    char user_input[100];
    int (*function_pointer)();
}
```

Fig. 5. Allocation-level granularity example.

objects allocated with the underlying allocation routines. How to solve these problems is an area of ongoing research.

2.8 MEDS Summary

MEDS, the Memory Error Detection System, is a system to detect memory errors in binary executables. No source or object code is required to use the system. The system operates by first running a static analysis pass, then optionally a profiling pass. Finally, a software dynamic translation-based system called mmStrata is used to detect memory errors. Each of these systems work by applying the MEDS type system, composed in its most basic form of numeric and pointer types. Simple extensions allow for the offset type to reduce false positives, and the q-type in the profiler for further reduce false positives. Section 3 gives compelling evidence to the efficacy of this approach.

3 Experimental Results

We separate our evaluation of MEDS into two broad categories. First is a security evaluation in which we use a selection of real benchmarks to test for true and false overwriting detections, as well as any errors that the MEDS system may have missed (Section 3.2). The second category evaluates the performance of the MEDS system on a variety of benchmarks (Section 3.3). Before that, however, we briefly describe the experimental setup (Section 3.1).

3.1 Experimental Setup

Both exploit testing and benchmark timings were performed. All test programs were evaluated on an Opteron 148 CPU, running Linux Fedora Core 6, using the gcc 3.2.2 compiler w/static linking, and -O3 -fomit-frame-pointer optimization flags. While MEDS is designed to enable detection of read or write memory errors, only memory overwrites were monitored in the configuration tested.

The benchmark programs evaluated are shown in Fig. 6. The applications in the evaluation included standard benchmark suites with no expected vulnerabilities such as the SPEC CPU2000 benchmark suite [8]. Applications with known or seeded vulnerabilities, including the Apache web server, many of the relevant cases in the SAMATE static analysis test suite, the Wilander buffer overflow suite, and the BASS vulnerability suite, were also included in the evaluation [9, 10, 11]. In addition, commonly used applications and test benchmarks were included in the evaluation such as the binutils-2.18 utility suite and the vpo [12] regression test suite, which includes benchmarks such as fm-part (VLSI placement program), matrix multiply, 8-queens solver, sieve of Eratosthenes, wc, Whetstone, Dhrystone, a travelling salesperson problem solver, etc.

Benchmark Suite	Description
SPEC CPU 2000	ammp, art, bzip2, crafty, equake, gap, gcc, gzip, mcf, mesa, parser, perlbmk, twolf, vortex, vpr
Wilander buffer overflow suite	buffer overflows on the stack, heap, and BSS
Benchmarks for Architectural Security Systems (BASS)	buffer overflows: 01_overflow_fp, 02_overflow_variable, 04_overflow_shellcode_injection
SAMATE Reference Dataset	test cases related to memory overwriting
Apache	web server with manually seeded vulnerability
binutils	nm, objdump, readelf, size, strings
VPO compiler test suite	ackerman, arraymerge, banner, bubblesort, cal, cb, dhrystone, fm-part, grep, hello, iir, matmult, od, puzzle, queens, quicksort, shellsort, sieve, strip, subpuzzle, wc, whetstone
nasm	netwide x86 assembler

Fig. 6. Benchmarks evaluated.

In addition, several of the vpo test suite benchmarks were seeded and tested with four categories of vulnerabilities: buffer overflows, array out of bounds accesses, double free/dangling pointer references, and uninitialized pointer dereferences.

3.2 Error Detection Evaluation

The benchmarks listed above were run to evaluate detection of memory overwriting.

3.2.1 False Positives

A warning generated by the MEDS system is considered a false positive if it is determined that a memory overwrite or underwrite has been detected by the system, but no overwrite or underwrite actually occurred.

As seen in Fig. 7, for SPEC CPU2000, some false positives occurred when the profiling pass was not performed. For the set of applications with known or seeded vulnerabilities, mmStrata produced warnings only when memory overwriting was attempted, i.e. it generated no false positive reports. For the other applications tested, only apache and queens produced false positives, which were eliminated by profiling. No other tests yielded false positives, even without the profiling pass, as shown in Fig. 8.

For all benchmarks which produced false positive reports prior to profiling, the false positive was eliminated by the profiling pass, due to profiler assistance in solving the problems described in Section 2.2.1. The profiling pass did not generate any new false positive reports, because the profiling algorithm is conservative.

Several benchmarks from SPEC CPU 2000 (`gap`, `parser`, `vortex`, `vpr`) contained code which was generated as the result of the combination of strength reduction and induction variable elimination. After the introduction of the offset type to the shadow type system, false positives due to encountering this type of code were eliminated.

3.2.2 False Negatives

False negatives are recorded when the MEDS system does not generate a warning report when a memory overwrite should be detected. To evaluate false negatives, we verified

Benchmark	Pre-Profile False Positives?	Post-Profile False Positives?	Required Offset Type Extension?	MEDS Slowdown (ref input)
ammp	No	No	No	24.4
art	No	No	No	9.5
bzip2	Yes	No	No	44.8
crafty	Yes	No	No	35.4
equake	Yes	No	No	20.4
gap	Yes	No	Yes	64.9
gcc	No	No	No	61.9
gzip	Yes	No	No	34.9
mcf	No	No	No	15.0
mesa	No	No	No	34.4
parser	Yes	No	Yes	39.7
perlbmk	Yes	No	No	51.0
twolf	Yes	No	No	34.8
vortex	No	No	Yes	52.0
vpr	No	No	Yes	35.9
Geo. mean				32.6

Fig. 7. False positive and performance results for SPEC CPU 2000.

Benchmark Suite	Pre-profiler False Positives?	Post-profiler False Positives?	Required Offset Type Extension?
Wilander	No	No	No
BASS	No	No	No
SAMATE	No	No	No
Apache	Yes	No	No
binutils	No	No	No
VPO compiler test suite	queens: Yes Others: No	No	No
nasm	No	No	No

Fig. 8. False positive results for assorted benchmarks.

that all the memory overwriting occurrences in our benchmarks were detected by our system. We also tested several vpo regression tests seeded with the following four categories of vulnerabilities: buffer overflows, array out of bounds accesses, double free/ dangling pointer references, and uninitialized pointer dereferences. Our tool detected every instance of the seeded vulnerabilities. No false negatives for coarse-grained memory overwrites were generated for our test applications. Some fine-grained false negatives occurred, but these are outside the scope of this paper.

3.3 Performance

Fig. 7 shows the performance of the MEDS system normalized to native execution for a variety of SPEC CPU2000 benchmarks. The best performing benchmark is `179.art` at only 9.5 times slower than native speed. The worst performing is `254.gap` at 65 times slower than native speed, while the geometric mean of the benchmarks is about 33 times slower than native execution. We realize that this level of run-time overhead is too high for many application domains. However we believe that it is very suitable for off-line testing and debugging. Furthermore, it may be useful in secure environments for programs that do not have high throughput requirements, such as I/O bound applications, interactive applications or lightly loaded server programs.

As our implementation has only had modest tuning effort, we are encouraged that MEDS performs as well as past techniques, even though it is a more comprehensive system with a more in-depth type system. The closest related work, *Annelid* based on Valgrind, reported a geometric mean slowdown of 36.7 times (for the SPEC benchmarks they report), without protecting the stack, but with the additional overhead of protecting memory reads [13]. For the same benchmarks, MEDS shows 32.6 times slowdown. Continued overhead reduction is an area of ongoing research.

4 Related Work

Over time, a wide variety of memory overwriting exploits have been invented, and a corresponding variety of software defenses have been developed. Some defenses are specific to particular subsets of all memory overwriting exploits, such as stack smashing, format string, code injection, or buffer overflow exploits [14, 15, 16, 17, 18, 19, 20]. Many memory overwriting defenses require source code or pre-linkage object code, unlike MEDS, making their use infeasible in many computing environments [20, 19, 17, 21, 22]. Rewriting software in a memory-safe language (e.g., Java, C#) would prevent memory overwriting exploits, but would require source code and great time expenditure. Some defenses are probabilistic, using randomization, and therefore subject to being defeated by brute force attacks [23]. Many defenses are designed only to protect control data, i.e. code addresses used in control flow, such as return addresses and function pointers [20, 24]. However, security-critical data can include non-control data [25]. MEDS protects against all memory overwrites, whether the target of the overwrite is control data or not, and regardless of whether the attack vector is a buffer overflow, format string exploit, integer overflow of a pointer, double-free, etc.

The most comparable prior work is the Annelid tool, which was based upon the Valgrind SDT [13]. Annelid detects out of bounds reads and writes to global-static and heap memory objects. Lacking a profiler and static analyzer, it incurred too many false positives for stack objects, and the stack portion of Annelid was disabled before completion. Annelid also encountered the problems with false positives discussed in Section 2.2.1. The pointer identification problem was left unsolved, causing some false positives. The difference between pointers problem was also left unsolved, although the authors proposed that a pointer offset type (the MEDS solution) could be implemented in the future. Annelid segments (equivalent to MEDS bounds-information objects) have an unsafe cleanup mechanism. The only sound solution proposed by the authors was a slow run-time garbage collection mechanism that would have increased overhead. Fi-

nally, Annelid makes use of some (not usually available) debug information in the executable, unlike MEDS. It appears that Annelid is not being maintained or used.

5 Summary

This paper has described MEDS, the Memory Error Detection System. MEDS detects common memory errors, such as buffer overflows, within binary executable programs: no source or object code is required. MEDS starts with a static analysis phase which analyzes functions and objects in the program binary. The static analyzer writes informative annotations into a file, which are read by an optional profiling step. The profiling step helps avoid several classes of false positives unsolved by previous work. Finally, MEDS instruments the program via software dynamic translation to detect memory errors. We show how extending the type system to include the offset-type eliminates a variety of common false positives. Performance of the system is 33 times slower than native execution for our SPEC CPU2000 benchmarks. This performance is suitable for off-line uses and I/O intensive or low throughput applications.

Acknowledgements. This work was supported in part by the National Science Foundation under awards CNS-0551560 and CNS-0716446. This material is based on research sponsored by Air Force Research Laboratory under agreement number FA8750-07-2-0029. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory or the U.S. Government.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley, Reading, MA, USA (1986)
2. Kumar, N., Misurda, J., Childers, B.R., Soffa, M.L.: Instrumentation in software dynamic translators for self-managed systems. In: Proceedings of the 1st ACM SIGSOFT Workshop on Self-managed Systems, pp. 90–94. ACM, New York, NY, USA (2004)
3. Zhou, S., Childers, B.R., Soffa, M.L.: Planning for code buffer management in distributed virtual execution environments. In: VEE '05: Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments, pp. 100–109. ACM, New York, NY, USA (2005)
4. Scott, K., Davidson, J.: Strata: A software dynamic translation infrastructure. In: IEEE Workshop on Binary Translation. IEEE (2001)
5. Scott, K., Kumar, N., Childers, B., Davidson, J.W., Soffa, M.L.: Overhead reduction techniques for software dynamic translation. In: Proceedings of the 18th International Parallel and Distributed Processing Symposium, pp. 200. IEEE (2004)
6. Scott, K., Kumar, N., Velusamy, S., Childers, B., Davidson, J.W., Soffa, M.L.: Retargetable and reconfigurable software dynamic translation. In: CGO '03: Proceedings of the International Symposium on Code Generation and Optimization, pp. 36–47. IEEE, Washington, DC, USA (2003)
7. Eagle, C.: The IDA Pro Book. No Starch Press, San Francisco, CA, USA (2008)

8. Hening, J.L.: SPEC CPU2000: Measuring CPU performance in the new millennium. IEEE Computer 7, 28–35 (2000)
9. Black, P.E.: Software assurance metrics and tool evaluation. In: Proceedings of the 2005 International Conference on Software Engineering Research and Practice. (2005)
10. Poe, J., Li, T.: Bass: A benchmark suite for evaluating architectural security systems. SIGARCH Computer Architecture News, pp. 26–33. ACM, New York, NY, USA (2006)
11. Wilander, J., Kamkar, M.: A comparison of publicly available tools for dynamic buffer overflow prevention. In: Proceedings of the Network and Distributed System Security Symposium, pp. 149–162. Internet Society (2003)
12. Benitez, M.E., Davidson, J.W.: The advantages of machine-dependent global optimization. In: Proceedings of the 1994 Conference on Programming Languages and Systems Architectures, pp. 105–124. ACM, New York, NY, USA (1994)
13. Nethercote, N., Fitzhardinge, J.: Bounds checking entire programs without recompiling. In: Informal Proceedings of the Second Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management (SPACE 2004). (2004)
14. Barrantes, E.G., Ackley, D.H., Forrest, S., Stefanovic D.: Randomized instruction set emulation. ACM Transactions on Information Systems Security 8, 3–40 (2005)
15. Baratloo, A., Singh, N., Tsai, T.: Transparent run-time defense against stack smashing attacks. In: Proceedings of the USENIX Annual Technical Conference, pp. 251–262. USENIX (2000)
16. Liang, Z., Sekar, R., DuVarney, D.C.: Automatic synthesis of filters to discard buffer overflow attacks: A step towards self-healing systems. In: Usenix 2005 Annual Technical Conference, pp. 375–378. USENIX (2005)
17. Ruwase, O., Lam, M.: A practical dynamic buffer overflow detector. In: Proceedings of the Network and Distributed System Security (NDSS) Symposium, pp 159–169. (2004)
18. Kc, G.S., Keromytis, A.D., Prevelakis, V.: Countering code-injection attacks with instruction-set randomization. In: CCS '03: Proceedings of the 10th ACM conference on Computer and communications security, pp. 272–280. ACM, New York, NY, USA (2003)
19. Cowan, C., Barringer, M., Beattie, S., Kroah-Hartman, G., Frantzen, M., Lokier, J.: FormatGuard: Automatic protection from printf format string vulnerabilities. In: Proceedings of 10th USENIX Security Symposium, pp. 191–200. (2001)
20. Cowan, C., Pu, C., Maier, D., Hinton, H., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q.: Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In: Proceedings of the 7th USENIX Security Symposium. pp. 26–29. USENIX (1998)
21. Necula, G.C., McPeak, S., Weimer, W.: Ccured: Type-safe retrofitting of legacy code. In: POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 128–139. ACM, New York, NY, USA (2002)
22. Akritidis, P., Cadar, C., Raiciu, C., Costa, M., Castro, M.: Preventing memory error exploits with wit. In: IEEE Symposium on Security and Privacy, pp. 263–277. IEEE (2008)
23. Bhatkar, S., DuVarney, D.C., Sekar, R.: Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In: Proceedings of 12th USENIX Security Symposium, pp. 105–120. USENIX (2003)
24. Kiriansky, V., Bruening, D., Amarasinghe, S.: Secure execution via program shepherding. In: Proceedings of the 11th USENIX Security Symposium, pp. 191–206. USENIX (2002)
25. Chen, S., Xu, J., Sezer, E.C., Gauriar, P., Iyer, R.K.: Non-control-data attacks are realistic threats. In: Proceedings of the 14th Usenix Security Symposium. pp. 177–192. USENIX (2005)