AFRL-RI-RS-TR-2009-140
Final Technical Report
May 2009

SOFTWARE EXPLOIT PREVENTION AND REMEDIATION VIA SOFTWARE MEMORY PROTECTION

University of Virginia

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2009-140 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/s/
JAMES L. SIDORAN
Work Unit Manager

/s/
WARREN H. DEBANY, Jr.
Technical Advisor, Information Grid Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
MAY 09	Final	Apr 07 – Dec 08

4. TITLE AND SUBTITLE

SOFTWARE EXPLOIT PREVENTION AND REMEDIATION VIA SOFTWARE MEMORY PROTECTION

5a. CONTRACT NUMBER
FA8750-07-2-0029

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
N/A

6. AUTHOR(S)

Clark L. Coleman, Michele Co, Jack W. Davidson, John C. Knight, and Anh Nguyen-Tuong, Jason D. Hiser

5d. PROJECT NUMBER
NICE

5e. TASK NUMBER
00

5f. WORK UNIT NUMBER
05

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

University of Virginia
159 Engineer's Way
Charlottesville, VA 22904-4257

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFRL/RIGA
525 Brooks Rd.
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
N/A

11. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-RI-RS-TR-2009-140

12. DISTRIBUTION AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Deployed software often contains memory overwriting vulnerabilities which can be exploited by malicious users who provide input that causes critical data to be overwritten in the program (called a memory overwriting exploit). There are a wide variety of such exploits (e.g. buffer overflows, formatting string exploits, etc.). Some defenses have been limited to defeating memory overwrites in heap or stack memory, and most defenses require access to source code. The Software Memory Protection (SMP) project addresses these limitations and shortcomings by supplying a general defense against all known memory overwriting exploits, requiring no source or object code or recompilation of the protected application, with a remediation mechanism that does not rely on crashing the program to defeat attempted exploits. Therefore, SMP: (i) can defend a program binary for which no source code is available, including its linked libraries; (ii) need not be combined with any other defense against memory overwriting; and (iii) does not turn exploits into potential DOS (denial of service) attacks. SMP can be applied to a binary during testing, field deployment, or both.

15. SUBJECT TERMS

Computer security; vulnerabilities; exploits; memory overwriting; buffer overflow; software dynamic translation; static analysis.

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			James L. Sidoran
U	U	U	UU	28	19b. TELEPHONE NUMBER (*Include area code*) N/A

Table of Contents

1 Summary of the Research Project

The intelligence community has determined adversaries willing to invest considerable effort in finding and exploiting software vulnerabilities. In particular, deployed software often contains memory overwriting vulnerabilities which can be exploited by malicious users who provide input that causes critical data to be overwritten in the program. This type of attack is called a *memory overwriting exploit*. There are a wide variety of such exploits (e.g. buffer overflows, format string exploits, double-free exploits, dangling pointer exploits, integer overflow exploits, etc.). Many researchers have attempted to defeat these exploit types one at a time, crafting defenses that are specific to one exploit. Some defenses have also been limited to defeating memory overwrites in heap memory, or stack memory. Most defenses have required access to source code, so that the application can be re-compiled with security checking code inserted. Some defenses are probabilistic and can be defeated by brute force attacks in a short period of time. These limitations have made prior defenses incomplete and difficult or impossible to use in certain environments (e.g. some deployed applications have no source code publicly available for analysis and recompilation). In addition, the only response of most prior work has been to terminate (crash) the program, which can allow an attacker to turn his unsuccessful exploits into a successful denial of service (DOS) attack. Mission critical systems in the intelligence community cannot tolerate repeated crashes.

The Software Memory Protection (SMP) project addresses these limitations and shortcomings in prior work by supplying a general defense against all known memory overwriting exploits, requiring no source or object code or recompilation of the protected application, with a remediation mechanism that does not rely on endlessly crashing the program to defeat attempted exploits. Therefore, SMP: (i) can defend a program binary for which no source code is available, including its linked libraries; (ii) need not be combined with any other defenses against memory overwriting; and (iii) does not turn exploits into potential DOS attacks. SMP can be applied to a binary during testing (to detect memory overwriting errors), during its field deployment (to defeat exploits), or both.

The structure of the remainder of this report is as follows. Section 2 will introduce the topic of memory overwriting vulnerabilities and exploits, and define the important terms used in the rest of this report. Section 3 will describe the central technical advances of the SMP system, and how SMP protects a program from memory overwriting exploits. Section 4 evaluates SMP according to three objective criteria (false negatives, false positives, and run-time overhead). Section 5 summarizes the project's accomplishments. Section 6 points the way to future development of the research, including technology transfer.

2 Introduction

Memory overwriting exploits have been commonly used by malicious attackers to gain unauthorized control over a program or the computer system running the program. In this section, these exploits and associated terminology will be defined.

A *memory overwriting exploit* is an attack on a running program in which the malicious user supplies input that causes a memory write to a memory location that should not be overwritten through the memory pointer that is being used for that memory write. This exploit takes advantage of a *memory overwriting vulnerability* in the program, which unintentionally permits a pointer to be caused to point outside the bounds of its normal *memory referent*. Examples of memory overwriting exploits include buffer overflows, format string exploits, double free exploits, integer overflow exploits, and integer truncation exploits. In each of these exploits, a pointer is made to point somewhere other than to the memory referent intended by the author of the program. This common feature of all memory overwriting exploits makes it possible to design a general defense against all such exploits, as will be shown in Section 3.

The data memory of a running program can be divided into three categories: *stack memory* is continually allocated and deallocated on the run time stack as the program enters into, and returns from, program functions or subroutines; *heap memory* is dynamically allocated and freed by calls to system functions; and *global static memory* is allocated only once, at the time the program is loaded into memory by the system loader, and is used to store global variables and constants. All three categories of memory can include *critical data*, i.e. data whose values control the security-critical operation of the program, and hence all categories must be defended from overwriting exploits.

Any security monitoring of a running program should be evaluated according to three criteria, each of which should be minimized. Each criterion will be defined here specifically in terms of a memory overwriting defense. A *false negative* occurs when a memory overwriting exploit escapes detection. A *false positive* occurs when a valid memory write is incorrectly identified as a memory overwriting exploit. *Run time overhead* is the measurement of the slowdown of the program's normal operation caused by the security monitoring system.

A security system can use *online processing*, in which a program is analyzed during its normal execution, and/or *offline processing*, in which the program is analyzed separately from its normal execution. Offline processing has the advantage of tolerating higher run time overhead than online processing. However, online processing to some degree is usually required to enforce security properties without incurring excessive false negatives and false positives. A security analysis can be *static analysis*, in which a program is examined without executing it, or *dynamic analysis*, in which the execution of a program is monitored. Note that a dynamic analysis (e.g. profiling) can be performed during offline processing, as well as during online processing.

Most security tools terminate programs that are under attack. Some tools terminate the program gracefully, while others provide defenses that will cause a program crash due to a run time exception. A desirable alternative is to provide *remediation*, which can include one or both of *recovery* (suppressing the memory overwrite and continuing execution) and *repair* (removing the memory overwriting vulnerability). Repeatedly crashing a program uses system resources and permits an attacker to conduct a *denial of service (DOS) attack*, which can make the system unusable due to lack of resources.

The terms *program binary* and *program executable* will be used interchangeably in this report. Both refer to the file produced after all compilation and linking steps have been performed. This file is ready to be loaded and executed by the operating system.

3 Methods, Assumptions, and Procedures

3.1 Assumptions

As SMP was developed as a research project, rather than as a commercial security tool, it was designed and tested in a limited environment that was sufficient to demonstrate success. Program binaries are assumed to be statically linked ELF format x86/Linux binaries. Only the Fedora Core 2.6 operating system and gcc version 3.2.2 compiler were used in testing, although similar compilers and operating systems are probably compatible with SMP. SMP does not currently process programs that use signals or threads. All tools used in SMP have been designed for portability, and these limitations can be removed with reasonable engineering effort in the future.

The program binary to be protected is assumed to execute in an environment in which the user interacts with the program through normal input and output operations. It is assumed that the user cannot change memory directly via privileged tools such as debuggers. SMP assumes that any memory write instruction that the program executes could be a memory overwriting vulnerability, unless an SMP tool is able to prove otherwise.

No source code is assumed to be available for SMP's analyses. SMP also does not assume that libraries and object code files for program modules are separately available. No user effort to re-compile or re-link the software is required. The statically linked executable is assumed to have been compiled with optimizations and without preserving debug information. Debug information would be useful to SMP, but it is unrealistic to assume its presence in application program binaries.

SMP is designed to produce zero false negatives and minimal false positives, making it useful for provision of software assurance with regards to memory overwriting.

3.2 Methods Used by SMP

SMP, as shown in Figure 1, takes a binary program as input. The binary is used for static analysis and to prepare the mmStrata run-time system, which monitors the program during its normal execution to detect and prevent memory overwrites. SMP first prepares a run-time system using a binary instrumentation tool we call the Stratafier, so named because it inserts software dynamic translation system called Strata into an executable image.

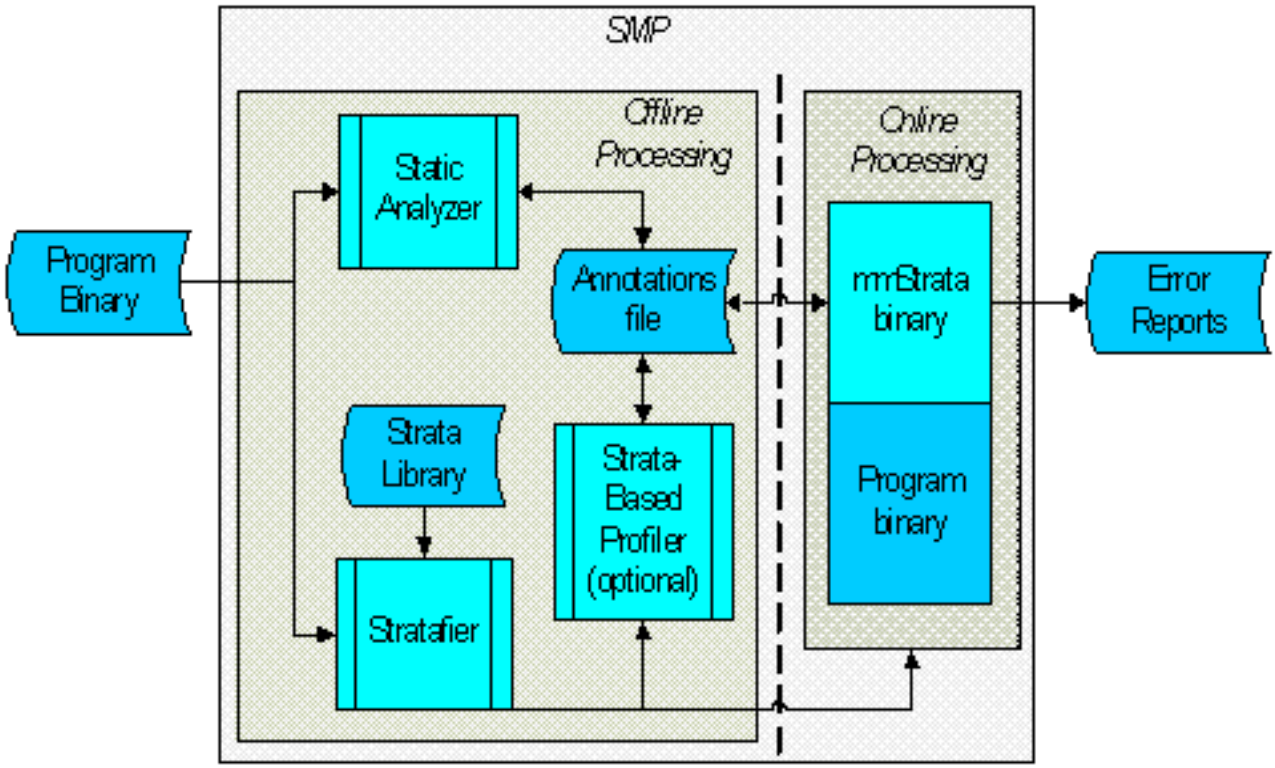

Figure 1: SMP system overview.

After the binary has been Stratafied, SMP 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 SMP components.

Finally, the run-time system is ready to detect memory errors. When run, the revised program binary is dynamically instrumented by mmStrata (the 'memory monitor Strata' run time tool). Memory writes that 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 3.2.1 – 3.2.5 discusses these components in more detail.

3.2.1. **SMP Type System**

To effectively detect memory errors, SMP 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
- $p_{\texttt{obj}}$ - 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. Figure 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.

+	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

Figure 2: Rules for combining metadata types.

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 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.

3.2.1.1. **Type System Challenges**

Although the SMP 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 SMP 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 3.2.3). 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, see Section 3.2.3). For functions that are unanalyzable (because they contain dynamic stack allocation via `alloca`, or some other non-standard stack manipulation), the dynamic run-time system dynamically creates and updates stack frames (see Section 3.2.5).

Besides tracking object creation, SMP 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 3.2.3 – 3.2.4).

Another major challenge faced by SMP is how to locate the executable code within a program. The static analyzer solves this problem (see Section 3.2.3).

Compiler optimizations and non-standard code can cause the type system to erroneously consider some objects to be numeric. Consider the code in Figure 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
                          …
```

Figure 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 that there is no 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. SMP avoids 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.

3.2.2. **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].

3.2.2.1. **Strata Overview**

Strata operates as a co-routine with the program binary it is translating, as shown in Figure 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 SMP type system in both the profiler driven analysis (Section 3.2.4) and the online detector (Section 3.2.5).

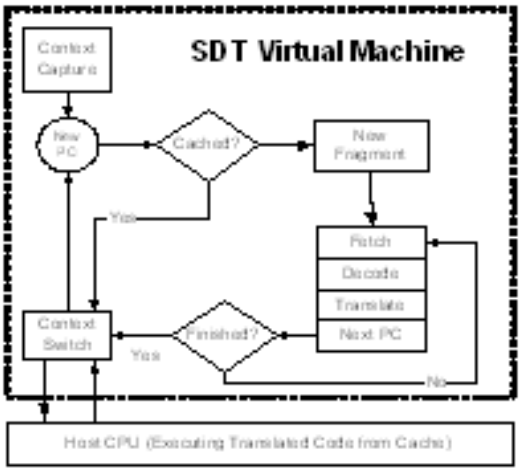

Figure 4: High-level overview of how Strata operates.

3.2.3. **Static Analysis**

The SMP 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 SMP 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 3.2.1.1. 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.

The second type of annotations produced by the static analyzer are *optimizing annotations*. The static analyzer implement a type inference system, in which registers and memory locations are inferred to have pointer or non-pointer types based on their usage in the program binary. If a memory load produces a register value that will always be used as a non-pointer type, then the run time monitor (mmStrata) does not need to instrument that instruction with the lengthy code sequence that would fetch memory referent information from a splay tree that models all memory locations. Instead, the optimizing annotation tells mmStrata to assume the result of the load is a numeric (non-pointer) value, which reduces the instrumentation code emitted for that instruction to a single instrumentation instruction. This optimization produces great time savings.

The type inference system is built on a thorough intraprocedural dataflow analysis framework. This system could be enhanced in several ways in the future to provide even greater optimizations, such as by extending the dataflow analysis and type inference to be interprocedural.

3.2.4. **SMP Profiler**

The optional SMP profiler is based on Strata, and uses information from the static analysis phase to mimic the SMP type system. One of the main goals is to reduce false positives. The SMP 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 seeing 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), SMP 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 4 gives evidence that this simple heuristic works well for many benchmarks, but that some benchmarks will require stronger static analysis or profiler information.

The profiler can also assist in the process of producing optimizing annotations. It records which memory loads always produce a non-pointer (numeric) result. These annotations are read by the static analyzer, which uses them to augment its own type inference. The resulting optimizing annotations are specially encoded to indicate that they are dependent upon a profiler run, which makes them potentially unsafe if the application program behaves differently during a profiler run as compared to a normal execution. Dealing with this potential source of false positives is described in Section 3.4.

3.2.5. **mmStrata**

The SMP 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 SMP 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.

SMP 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 SMP 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. SMP provides a general defence that can detect all known memory errors within a binary executable.

3.3 **Granularity Issues**

SMP provides a comprehensive defense against memory overwrites by identifying pointer referents and detecting when a pointer goes outside the bounds of its referent. The SMP defense, by default, is applied at memory allocation granularity, i.e. a pointer referent has bounds determined at the time of its allocation. For example, if a data item of size 100 bytes is allocated on the heap, the mmStrata monitor establishes a 100 byte memory referent upon return from the heap allocation system function. A pointer that is initialized to point within this 100 byte location will be constrained to only perform writes within those 100 bytes.

It could be the case that a heterogeneous data structure was allocated within that 100 byte region, with multiple data fields within the data structure. In that case, the defense provided by SMP would be even better if it could make memory referents that are subsets of the 100 bytes. For example, if the 100 byte data structure includes a 64 byte buffer and numerous smaller fields, it is desirable to prevent a buffer overflow from the 64 byte buffer from overwriting any of the smaller fields.

When operated in the default coarse-grained mode, SMP would only prevent a buffer overflow from leaving the bounds of the 100 byte parent data structure, and would not prevent fine-grained overflows within the parent data structure. The same is true in the rarer case in which a heterogeneous data structure is allocated in global static data, and in the more common case in which a buffer exists within a stack frame alongside other local variables.

To improve the granularity of its defenses, SMP uses information about memory access patterns gathered during the optional profiler run, which is then used by the static analyzer to infer the presence of buffers in the midst of memory allocation units (e.g. stack frames, heap allocations, global variables). The inference is based on noting which memory accesses were indexed or scaled accesses (the kind of instructions used to access buffers), versus direct (non-indexed, non-scaled) accesses. Instructions that write to buffers within larger memory allocations then receive a special annotation from the static analyzer, directing mmStrata to limit the bounds of the write to a subset of the parent data referent bounds. Note that the absence of a profiling run implies that SMP will run in coarse-grained mode.

3.4 **Adaptive Feedback**

Because SMP utilizes both offline and online processing, it is able to explore an innovative area of program analysis: adaptive feedback. Observations made by mmStrata at run time can be communicated via the annotations file to future runs of the static analyzer and/or profiler. This information can be used for optimization of run time overhead, reductions in false positives, or in searching for the best recovery policy.

For example, the SMP profiler records which memory loads always produce a pointer value and which loads always produce a non-pointer (numeric) value. This information causes the static analyzer to emit profiler-dependent optimizing annotations. If an instruction always loaded a numeric value during the profiling run, but then sometimes loaded a pointer value during normal execution, mmStrata will think that a numeric value is being illegally dereferenced as if it were a pointer (because it has been told by an optimizing annotation that the value is numeric). This erroneous optimizing annotation will cause a false positive for this instruction. Because only the profiler-dependent optimizing annotations are unsafe, mmStrata can record in the annotations file that profiler-dependent optimizing annotations should be ignored on succeeding runs of the program, reducing false positives back to zero. This mechanism permits aggressive optimization, with only occasional (i.e. a minority of benchmarks) reduction in aggressiveness dictated through adaptive feedback.

A second example of adaptive feedback is the selection of a recovery policy. After detecting an overwrite mmStrata can attempt to recover using the specified recovery policy (see next section for details). If that recovery attempt produces more detected overwriting errors, then mmStrata can record in the annotations file that a different policy should be tried during the next execution of the program. As the number of recovery policies increases with future research, this mechanism will be automated and used to find the optimal recovery policy, perhaps even on a per-subroutine basis within the application.

3.5 **Remediation**

The default action, when SMP detects a memory overwrite, is to output an error message. In product form, if used as an online security tool, SMP would then terminate the program by default. Termination is not forced in the research version of SMP so that complete evaluation can be made of how many overwrites are detected in a single run of the program (including true overwrites and false positives). Non-termination would also be the default when SMP is used as an offline testing tool.

By setting an environment variable, one of two recovery policies can be chosen in place of termination. The first policy discards the write instruction causing the overwrite, logs a warning message, and continues execution with the next instruction. The second policy also discards the overwrite and logs a warning message, but then sets the outgoing registers to default values based on their type (pointer or non-pointer) as determined by the static analyzer and then forces an immediate return from the overwriting function.

Both recovery policies have performed well in synthetic benchmarks. Further evaluation using real exploits is desirable. Time has not permitted the implementation of further recovery policies, although several have been conceived and vetted in design discussions.

4 Results and Discussion

We separate our evaluation of SMP 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 (Section 4.2)The second category evaluates the run time overhead of the SMP system on a variety of benchmarks (Section 4.3). Before that, however, we briefly describe the experimental setup (Section 4.1).

4.1 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 SMP 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 Table 1. 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 [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.

Table 1: Benchmarks evaluated.

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 Apache	test cases related to memory overwriting 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

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

4.2 **Error Detection Evaluation**

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

4.2.1. **False Positives**

A warning generated by the SMP 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.

Pre-profiler false positives

As seen in Table 2, for SPEC CPU2000, some false positives were detected 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 Table 3.

Post-profiler false positives

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 3.2.1.1. The profiling pass did not generate any new false positive reports, because the profiling algorithm is conservative.

False positives after offset-type analysis.

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.

4.2.2. **False Negatives**

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

4.3 **Performance**

Table 2 shows the run time overhead performance of the SMP 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.

Table 2: False positive and performance results for SPEC CPU 2000

Benchmark	Pre-Profiler False Positives?	Post-Profiler False Positives?	Required Offset Type Extension?	SMP 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
geometric mean				32.6

Table 3: False positive results for assorted benchmarks.

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

As we have only had time to implement a subset of our ideas for reducing run time overhead, we are encouraged that SMP 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, SMP shows 32.6 times slowdown. Continued overhead reduction is an area of ongoing research.

4.4 **Fine-grained Evaluation**

When run in fine-grained mode, SMP delivers the same run time overhead performance as for coarse-grained mode (within 0.5 times additional slowdown; both performance numbers round to 33 times slowdown). The additional granularity should cause SMP to detect fine-grained exploits that would not be detected in coarse-grained mode, and this enhancement in coverage is confirmed with synthetic fine-grained test cases. However, no fine-grained exploits can be found in the security literature, and none are present in the test suites used. It is possible that fine-grained exploits will become more common in the future if all coarse-grained exploits are defeated, making further granularity enhancement a priority for future work. Constructing fine-grained exploits for existing applications is quite difficult, such that the use of SMP only in coarse-grained mode increases the difficulty for the attacker by at least an order of magnitude.

False positives remain at zero in fine-grained mode for all test cases except the 176.gcc benchmark in SPEC CPU2000. For this benchmark, twenty-two instructions generate a special warning message indicating that the instruction passes coarse-grained tests but fails fine-grained tests. Inspection of these instructions reveals that the false positives are caused by differences between the profiling run and normal executions. A final SMP tool could accommodate the possibility of rare false positives by permitting granularity selection on a per-application basis. For the overwhelming majority of applications, fine-grained mode could be run with no false positives. For the remaining few applications, SMP could be used to provide protection in coarse-grained mode.

4.5 **Comparison to Prior Work**

Over time, a wide variety of memory overwriting exploits has 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 SMP, 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]. SMP 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 3.2.1.1. 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 SMP solution) could be implemented in the future. Annelid segments (equivalent to SMP 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. Annelid only attempts coarse-grained defenses and has no mechanism for extending the granularity in any way. The Annelid authors note that a synergy between static and dynamic analyses would be fruitful, but no effort along those lines was attempted. Finally, Annelid makes use of some (not usually available) debug information in the executable, unlike SMP. It appears that Annelid is not being maintained or used.

5 Conclusions

The SMP system has been a research success in many respects:

1. The prior state of the art in protection of a program binary from memory overwriting has been significantly improved. Protection is provided for all memory regions without incurring significant false positives and with zero coarse-grained false negatives.
2. SMP provides a general defense, whereas much prior work focused on defeating a single class of memory overwriting defenses. It would be difficult to compose these prior defenses into a comprehensive defense, as some of them require particular compiler, compiler libraries, etc., that are incompatible with each other.
3. The research has shown methods to extend the granularity of the defenses without significant increases in false positives (no increase for all test cases except one). Prior work did not attempt to provide any fine granularity in memory overwriting defenses.
4. The SMP system demonstrates a synergy between offline and online processing, and particularly a synergy between static and dynamic analyses. Prior researchers had speculated that their work could be improved by such a synergy, but had not actually implemented both static and dynamic tools in a single security monitoring system.
5. The SMP system uses an innovative method of adaptive feedback to reduce run time overhead while minimizing false positives. Because prior work did not utilize a combination of offline and online analyses, this innovation was not present in any prior work.
6. SMP has begun to incorporate recovery schemes that prevent unsuccessful exploits from being turned into denial of service attacks.
7. Most prior work in memory overwriting prevention was not usable for most applications, because source code was required, along with changes to build procedures, recompilation, re-linking, etc. SMP provides a push-button defense that can be invoked with a single shell script for most program binaries, or with two shell scripts and a profiling run for the remaining binaries.

These successes address significant security needs in the intelligence community. The positive results and innovations point the way for further research and eventual technology transfer, as discussed in the next section.

6 Recommendations

The SMP system can be improved through further research and technology transfer. The promising research areas include:

- Static analysis, dynamic profiling, and adaptive feedback techniques to greatly reduce run time overhead. This performance enhancement would make SMP more useful for online security monitoring, as well as for offline testing.
- Exploration of new recovery policies across a wide range of test cases to make SMP more suitable for online protection of mission critical software.
- Elimination of environmental limitations on SMP's domain (e.g. programs that use signals, threads, or which are not statically linked) to broaden SMP's applicability.
- Porting SMP to platforms other than x86/Linux.

The latter two steps would be part of a technology transfer effort that would produce a valuable security tool, after the first two research steps have concluded successfully and have broadened and deepened the usefulness of SMP to the intelligence community.

7 References

[1] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. Reading, MA, USA: Addison-Wesley, 1986.

[2] N. Kumar, J. Misurda, B. R. Childers, and M. L. Soffa, "Instrumentation in software dynamic translators for self-managed systems," in *Proceedings of the 1st ACM SIGSOFT Workshop on Self-managed Systems*, (New York, NY, USA), pp. 90–94, ACM Press, 2004.

[3] S. Zhou, B. R. Childers, and M. L. Soffa, "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*, (New York, NY, USA), pp. 100–109, ACM Press, 2005.

[4] K. Scott and J. Davidson, "Strata: A software dynamic translation infrastructure," in *IEEE Workshop on Binary Translation*, September 2001.

[5] K. Scott, N. Kumar, B. Childers, J. W. Davidson, and M. L. Soffa, "Overhead reduction techniques for software dynamic translation," in *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, p. 200, IEEE Computer Society, 2004.

[6] K. Scott, N. Kumar, S. Velusamy, B. Childers, J. W. Davidson, and M. L. Soffa, "Retargetable and reconfigurable software dynamic translation," in *CGO '03: Proceedings of the International Symposium on Code Generation and Optimization*, (Washington, DC, USA), pp. 36–47, IEEE Computer Society, 2003.

[7] C. Eagle, *The IDA Pro Book*. San Francisco, CA, USA: No Starch Press, 2008.

[8] J. L. Hening, "SPEC CPU2000: Measuring CPU performance in the new millennium," *IEEE Computer*, vol. 33, pp. 28–35, July 2000.

[9] P. E. Black, "Software assurance metrics and tool evaluation," in *Proceedings of the 2005 International Conference on Software Engineering Research and Practice*, June 2005.

[10] J. Poe and T. Li, "Bass: A benchmark suite for evaluating architectural security systems," *SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 26–33, 2006.

[11] J. Wilander and M. Kamkar, "A comparison of publicly available tools for dynamic buffer overflow prevention," in *Proceedings of the Network and Distributed System Security Symposium*, 2003.

[12] M. E. Benitez and J. W. Davidson, "The advantages of machine-dependent global optimization," in *Proceedings of the 1994 Conference on Programming Languages and Systems Architectures*, pp. 105–124, March 1994.

[13] N. Nethercote and J. Fitzhardinge, "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] E. G. Barrantes, D. H. Ackley, S. Forrest, and D. Stefanović, "Randomized instruction set emulation," *ACM Transactions on Information Systems Security*, vol. 8, no. 1, pp. 3–40, 2005.

[15] A. Baratloo, N. Singh, and T. Tsai, "Transparent run-time defense against stack smashing attacks," in *Proceedings of the USENIX Annual Technical Conference*, June 2000.

[16] Z. Liang, R. Sekar, and D. C. DuVarney, "Automatic synthesis of filters to discard buffer overflow attacks: A step towards self-healing systems," in *Usenix 2005 Annual Technical Conference*, pp. 375–378, 2005.

[17] O. Ruwase and M. Lam, "A practical dynamic buffer overflow detector," in *Proceedings of the Network and Distributed System Security (NDSS) Symposium*, pp. 159–169, February 2004.

[18] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," in *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, (New York, NY, USA), pp. 272–280, ACM Press, 2003.

[19] C. Cowan, M. Barringer, S. Beattie, G. Kroah-Hartman, M. Frantzen, and J. Lokier, "FormatGuard: Automatic protection from printf format string vulnerabilities," in *Proceedings of 10th Usenix Security Symposium*, pp. 191–200, August 2001.

[20] C. Cowan, C. Pu, D. Maier, H. Hinton, P. Bakke, S. Beattie, A. Grier, P. Wagle, , and Q. Zhang, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Proceedings of the 1998 USENIX Security Symposium*, January 1998.

[21] G. C. Necula, S. McPeak, and W. Weimer, "Ccured: Type-safe retrofitting of legacy code," in *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, (New York, NY, USA), pp. 128–139, ACM, 2002.

[22] P. Akritidis, C. Cadar, C. Raiciu, M. Costa, and M. Castro, "Preventing memory error exploits with wit," in *IEEE Symposium on Security and Privacy*, pp. 263–277, May 2008.

[23] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Address obfuscation: An efficient approach to combat a broad range of memory error exploits," in *Proceedings of 12th Usenix Security Symposium*, pp. 105–120, August 2003.

[24] V. Kiriansky, D. Bruening, and S. Amarasinghe, "Secure execution via program shepherding," in *11th USENIX Security Symposium*, August 2002.

[25] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats," in *14th Usenix Security Symposium*, pp. 177–192, August 2005.

8 List of Abbreviations and Acronyms

BASS: Benchmark suite for evaluating Architectural Security Systems

CPU: Central Processing Unit

CTI: Control Transfer Instruction

DOS: Denial Of Service

ELF: Executable and Linkable Format

gcc: GNU compiler collection

IDA: Interactive DisAssembler

mmStrata: Memory Monitor Strata tool

PC: Program Counter register

SAMATE: Software Assurance Metrics And Tool Evaluation

SDT: Software Dynamic Translation

SMP: Software Memory Protection

SPEC: Standard Performance Evaluation Corporation

VPO: Virginia Portable Optimizer

x86: Intel CPU family and instruction set