# ARCHERR: Runtime Environment Driven Program Safety

Ramkumar Chinchani, Anusha Iyer, Bharat Jayaraman, and Shambhu Upadhyaya

University at Buffalo (SUNY), Buffalo, NY 14260
{rc27, aa44, bharat, shambhu}@cse.buffalo.edu

**Abstract.** Parameters of a program's runtime environment such as the machine architecture and operating system largely determine whether a vulnerability can be exploited. For example, the machine word size is an important factor in an integer overflow attack and likewise the memory layout of a process in a buffer or heap overflow attack. In this paper, we present an analysis of the effects of a runtime environment on a language's data types. Based on this analysis, we have developed Archerr, an automated one-pass source-to-source transformer that derives appropriate architecture dependent runtime safety error checks and inserts them in C source programs. Our approach achieves comprehensive vulnerability coverage against a wide array of program-level exploits including integer overflows/underflows. We demonstrate the efficacy of our technique on versions of C programs with known vulnerabilities such as Sendmail. We have benchmarked our technique and the results show that it is in general less expensive than other well-known runtime techniques, and at the same time requires no extensions to the C programming language. Additional benefits include the ability to gracefully handle arbitrary pointer usage, aliasing, and typecasting.

## 1 Introduction

Several research efforts have been invested in detecting and preventing vulnerabilities such as buffer overflows and heap corruption in programs. Static bounds checking approaches [1] attempt to detect overflows in arrays and strings. However, due to the undecidability of pointer aliasing [2, 3], some pointer approximations have to be used, which result in false positives. Other techniques like CCured [4] have augmented the C programming language type system to support safety properties, allowing programs to be statically checked based on this new stronger type system. While these techniques provide a systematic way to detect invalid memory accesses, the flexibility of the language is reduced. There is also an additional burden on the programmer to familiarize himself with the new dialect. Finally, runtime safety techniques [5] defer the actual process of checking till program execution. However, they are known to cause significant slowdown in program execution time.

In terms of coverage, while these approaches are able to detect and catch common vulnerabilities such as buffer overflows [6], there are other vulnerabilities in

software which manifest in most unexpected ways and have proven very difficult to catch. For example, innocuous-looking errors such as arithmetic overflows and integer misinterpretation have been successfully exploited in ssh [7] and apache [8] daemons. The code snippet in Figure 1 illustrates the possible repercussions of an integer overflow. The function alloc_mem allocates memory of a specified

```
1. char * alloc_mem(unsigned size) {
2.   unsigned default_size = 4096;
3.   unsigned max_size = 0;
4.   char *retval = (char *)NULL;
5.
6.   size = size + 4; /* Add padding */
7.   max_size = (default_size > size) ? default_size : size;
8.   retval = (char *) malloc(sizeof(char) * max_size);
9.   return retval;
10. }
```

**Fig. 1.** A strawman integer overflow vulnerability

size. Now assume that another subroutine calls this function in order to copy a large string. On a 16-bit architecture, if an attacker is able to send a string whose length lies in the interval $[2^{16} - 4,\ 2^{16} - 1]$, then in line 6 when some padding is added, an arithmetic overflow will occur. This results in a smaller amount of memory being allocated in lines 7 and 8 than expected. On architectures with wider register words, strings of a much larger length will produce the same effect. Therefore, the same program behaves differently when compiled and executed on different architectures. Such overflow errors also occur in strongly typed languages like Java [1]. These kinds of errors do not speak well about portability and safety. The vulnerability in Figure 1 could have been prevented if an appropriate check was placed before line 6. But this is not a straightforward procedure since it requires the knowledge of the runtime architecture. A strong indication of the relationship between vulnerabilities and runtime architecture is seen in the information provided by CERT alerts which not only report the program which is vulnerable but also the relevant platforms.

### 1.1 Approach Overview

In this paper, we discuss a comprehensive, architecture-driven approach for statically analyzing and annotating C programs with runtime safety checks.

- **Runtime Environment Dependent Type Analysis**
  Our technique uses runtime environment information to define constraints on the domain of values corresponding to different data types and the operations defined on them. During the course of program execution, variables may assume different values, but from a program safety point of view, only a

[1] However, actual out of bound array accesses in Java do raise the java.lang.ArrayIndexOutOfBoundsException exception.

subset of them must be allowed. Pointer aliasing and dynamic binding prevents us from deriving all the constraints statically; for example, the set of valid addresses that can be accessed through pointer dereferencing changes dynamically with program execution. We achieve what compile-time type safety techniques like CCured can but without extending the programming language. Therefore, our technique is cleaner, but we pay a small price in terms of execution slowdown due to runtime checks.

– **Runtime Safety Checks for Data Types; Not Individual Variables**
Our technique also differs from other runtime bounds checking approaches [5] in terms of the nature of the checks. Protection is achieved by deriving and asserting safety checks for each data type rather than enforcing separate bounds for individual data variables/objects. This allows us to perform the same check on all variables of a given data type. As a result, in spite of pointer aliasing and arbitrary typecasting, the runtime checks incur smaller overheads (∼2-2.5X). We demonstrate that our technique performs comparably to CCured [4] (∼1.5X) and significantly better than Jones and Kelly's bounds checking [9] ($> 30$X) on the same suite of benchmarks. Moreover, the ability to detect vulnerabilities is not compromised as is evident from running Archerr on vulnerable versions of common C programs. We have been able to detect and preempt heap corruption attacks, buffer overflows, null pointer dereferences, and integer overflow attacks.

– **Checks Not Limited to Only Pointers**
Pointers and integers are the primary data types of the C programming language. All other data types are either variations or some user-defined combination of these types. As shown in the earlier illustration, vulnerabilities can be directly attributed to not only pointers, but also integers. Therefore, we derive safety checks for both pointers as well as integer operations. To the best of our knowledge, this is the first technique to systematically handle integer-based attacks.

– **Ability to Handle Typecasting**
A significant positive side-effect of our approach is the ability to handle arbitrary pointer usage, typecasting and complex array definitions. The coarse-grained data type checks on pointer variables implicitly assume that memory that they point to is not immutable. Therefore, typecasting is allowed as long as the variables satisfy the appropriate safety conditions. We explain this with examples in later sections.

### 1.2 ARCHERR Implementation

Archerr is implemented as a preprocessor for C source programs. Figure 2 gives an overview of the entire code transformation procedure. The original source program passes through a source transformation stage where annotations are inserted into the code. These annotations serve as hooks through which control can be transferred to safety checking routines (implemented as an external library) during runtime. The process of code transformation requires the runtime

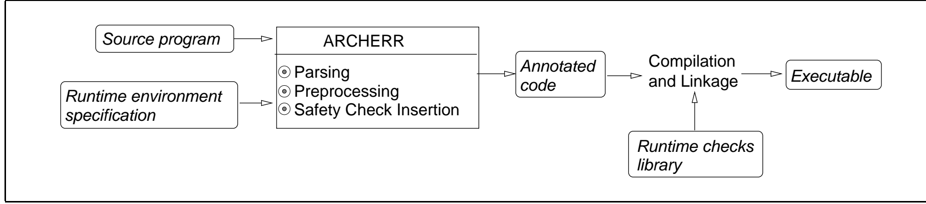


**Fig. 2.** Modifying the compilation process to integrate runtime specific safety properties

environment specification as a parameter, which is discussed further in the next section. The transformed source code when compiled and statically linked with the safety checking routines produces a safer version of the program. The entire process of code transformation is transparent to the programmer.

The rest of the paper is organized as follows. Section 2 presents the methodology that forms the core of our work. Section 3 enumerates a few optimizations to our main technique to reduce the runtime overhead. Section 4 reports the results that we have obtained through security and performance testing. Section 5 discusses the related efforts and puts our work in perspective, and Section 6 presents conclusions and gives suggestions for future work.

## 2 Runtime Environment Dependent Type Analysis

The environment we have chosen for our discussion and demonstration is 32-bit Intel architecture running Linux (kernel 2.4.19) and the object file format is *executable and linking format* (ELF) [10]. This is a popular and open development environment and we believe that our work would be most useful in these settings. We provide a brief background on some aspects of this runtime environment that are relevant to our technique as well as describe how they are specified in Archerr. Unless otherwise specified, any discussion will be in the context of only this runtime environment.

**Machine Word Size.** The ubiquitous Pentium belongs to a 32-bit processor family. The width of a register word is 32 bits and it can hold an absolute binary value in the interval $[0,\ 2^{32}-1]$. This defines the domain for meaningful and consistent integer operations. The machine word size by default is 32-bit in the Runtime Environment Specification (REspec). It may also be specified as a command line argument to Archerr.

**Memory Layout.** When an ELF executable is produced using a compiler, it not only contains the machine code, but also directives to the operating system regarding the memory layout of the various parts of the binary. All this information is contained in the ELF executable header. We can think of the executable as two parts: 1) one containing code and data that directly pertains to the source program, and 2) the other containing control information for purposes like dynamic linking and loading (note: statically linked programs may not have

this part). Memory violations can occur if information outside the first part is accessed or modified. Although the information in an ELF executable is created during link-time, it is still possible to create a memory map of the first part by inserting bookkeeping code at compile-time, which groups together variables of similar nature.

Various data types in a program are affected differently by a given runtime environment. We perform a runtime environment dependent type analysis to establish these relationships.

### 2.1 Analysis of Numerical Types

Let the entire set of signed integers be denoted by $\mathbb{I}$. Let $\mathbb{I}$ be represented by the interval $(-\infty, +\infty)$. For the sake of simplicity, we say that a variable is of type `int` if it assumes values in $\mathbb{I}$ and is closed under the successor (*succ*) and predecessor (*pred*) operations, that is,

$$x : \texttt{int} \Rightarrow x \in \mathbb{I} \tag{1}$$

$$succ(x : \texttt{int}) = x + 1 : \texttt{int} \tag{2}$$

$$pred(x : \texttt{int}) = x - 1 : \texttt{int} \tag{3}$$

The *succ* and *pred* primitives are defined at all points in $\mathbb{I}$. Moreover, the basic integer operations such as addition, subtraction, multiplication and division can all be expressed using these two primitives.

Taking into account the machine architecture, we denote the set of signed integers as $\mathbb{I}_n$ where the subscript $n$ represents the size of a word. For example, the set of signed integers on a 16-bit architecture is $\mathbb{I}_{16}$. Similarly, we can define the machine architecture dependent integer type as $\texttt{int}_n$ where the subscript $n$ denotes the size of a word.

$$x : \texttt{int}_n \Rightarrow x \in \mathbb{I}_n \tag{4}$$

Since the width of a machine word restricts the values that a signed integer can assume on a given architecture, the set $\mathbb{I}_n$ is represented by the interval $[-2^{(n-1)}, 2^{(n-1)} - 1]$. Now, the operations *succ* and *pred* can no longer be applied an arbitrary number of times on a given integer and still yield another valid integer. Therefore, the *succ* and *pred* operations are correctly defined only as:

$$succ(x : \texttt{int}_n) = \begin{cases} x + 1 & \text{for } x \in [-2^{(n-1)},\ 2^{(n-1)} - 2] \\ \textit{undefined} & \text{elsewhere} \end{cases} \tag{5}$$

$$pred(x : \texttt{int}_n) = \begin{cases} x - 1 & \text{for } x \in [-2^{(n-1)} + 1,\ 2^{(n-1)} - 1] \\ \textit{undefined} & \text{elsewhere} \end{cases} \tag{6}$$

Given a program, performing classical type based analysis at the programming language level ensures that operations are correct only at a level described by equation (2) and (3). Clearly, this is not adequate as demonstrated by equations (5) and (6) where $n$ becomes an important factor. We can present arguments on the similar lines for other numerical types such as `unsigned int`, `long int`, etc.

Interpreting floating point types is a little more complicated. However, the IEEE standard [11] requires the width of a single precision value to be 32 bits and a double precision value to be 64 bits. This makes them relatively independent of the machine architecture.

**Ensuring Safety Properties of Numerical Types**
Safety properties of numerical types are ensured by asserting their correctness of operations in terms of set closure. For every operation over numerical types, it is important to assert that the properties (5) and (6) hold. The algorithm to generate assertions for some basic operations on the `int` type are given Table 1.

Let $\diamond$ be an operator such that $\diamond \in \{+, -, \times, \div, \%\}$. Let $a$ : `int`, $b$ : `int` be the two operands of this operator. Let $\mathsf{MAXINT}$ be $2^{n-1} - 1$ and $\mathsf{MININT}$ be $-2^{n-1}$. Then, in order to ensure the safety of the operator $\diamond$, the assertions in Table 1 must be placed before the actual operation. For an insight into

$$
\begin{aligned}
&\text{if } a \geq 0, b \geq 0, \text{ then} \\
&\qquad a + b \Rightarrow \mathit{assert} : a \leq (\mathsf{MAXINT} - b) \\
&\text{if } a \geq 0, b < 0, \text{ then} \\
&\qquad a - b \Rightarrow \mathit{assert} : a \leq (\mathsf{MAXINT} + b) \\
&\text{if } a < 0, b \geq 0, \text{ then} \\
&\qquad a - b \Rightarrow \mathit{assert} : a \geq (\mathsf{MININT} + b) \\
&\text{if } a < 0, b < 0, \text{ then} \\
&\qquad a + b \Rightarrow \mathit{assert} : a \geq (\mathsf{MININT} - b) \\
&\forall\, a,\ b, \\
&\ \ a \times b \Rightarrow \text{assert: } a \geq \lfloor \mathsf{MININT}/b \rfloor \ \wedge\ a \leq \lfloor (\mathsf{MAXINT}/b \rfloor \\
&\ \ a \div b \Rightarrow \text{assert: } b \neq 0 \\
&\ \ a \,\%\, b \Rightarrow \text{assert: } b \neq 0
\end{aligned}
$$

**Table 1.** Assertions for basic numerical operators

the validity of these assertions, consider the first assertion. When $a, b$ are both nonnegative, there is no danger of an underflow under the operator $+$. All that remains is to check that $a+b \leq MAXINT$. This can be safely done by asserting $a \leq (MAXINT - b)$. Consider the bitwise operators $\{\&, |, \hat{}, <<, >>\}$. Among these, only the shift operators, $<<$ and $>>$ are unsafe. They are equivalent to multiplication and division by 2 respectively, and similar checks in Table 1 apply. The assignment operator $=$ does not cause any changes in values if it is applied to variables of identical types. Therefore, type correctness properties will ensure the right use of the assignment operator.

## 2.2 Analysis of Pointer Types

We now extend the architecture-dependent analysis to pointer types. A pointer variable has two aspects - a memory location and an association to a type. We

will use the notation $p : q(\tau)$ to refer to a pointer variable $p$ that points to a memory location $q$ which holds a value of type $\tau$. For example, a pointer of type `char **p` is represented as $p : q($`char *`$)$ and it implies `p` is a pointer to a variable of type `char *`. The possible values for a pointer variable, like numerical types, are dependent on a given architecture and runtime environment. They are governed by the address format and the address space used. While a pointer variable may be assigned any arbitrary address, a *safe* assignment requires that the pointer variable be assigned a *valid* address. We evolve some theoretical machinery before any further discussion.

For the convenience of representation of sets, we use interval ranges. The set $\{a,\, a+1,\, a+2,\, \ldots,\, b\}$ is denoted as $[a, b]$. The insertion and deletion of ranges are defined in terms of set union and set difference as follows, where $S$ is a given set and $[a, b]$ is a range that is inserted or deleted.

$$append(S, [a, b]) : S = S \cup [a, b] \tag{7}$$

$$remove(S, [a, b]) : S = S - [a, b] \tag{8}$$

Let $\mathbb{P}$ be the set of all valid addresses. The elements of $\mathbb{P}$ are represented as interval ranges $[a_i, b_i]$ and $\mathbb{P} = \bigcup_i [a_i, b_i]$. Then a pointer $p$ is said to be *safe* if the following is true.

$$p : q(\tau) \Rightarrow \exists\, [a_i, b_i] \in \mathbb{P} \text{ s.t. } p \in [a_i, b_i] \;\wedge\; p + |\tau| \in [a_i, b_i] \tag{9}$$

where $|\tau|$ denotes the size allocated to data types of type $\tau$ and $p$ used alone on the right hand side represents an address. Let us denote the function or primitive that enforces this property as *validate(p)*.

Pointer arithmetic is generally allowed in a language supporting pointers and we can define this arithmetic in terms of the successor and predecessor operations but unlike integer arithmetic these are interpreted slightly differently. An operation on a pointer variable containing an address yields another address. Note that these operations defined by (10) and (11) are safe only if they too obey (9).

$$succ(p : q(\tau)) \Rightarrow p + |\tau| \tag{10}$$

$$pred(p : q(\tau)) \Rightarrow p - |\tau| \tag{11}$$

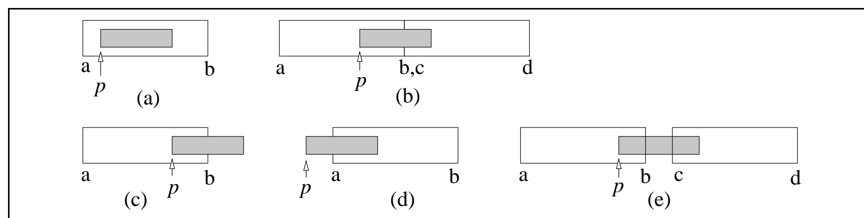


**Fig. 3.** (a), (b) Safe pointer assignments whereas (c), (d), (e) unsafe pointer assignments

Fig. 3 (a) and (b) show safe pointer assignments. The outer rectangle represents a valid address range and the shaded inner rectangle represents some variable or data object that pointer $p$ points to. If the entire object lies completely inside a valid address range, then the assignment is safe. Fig. 3 (b) shows an object that straddles two valid address ranges but as these ranges are contiguous, they collapse into one and no part of the object lies in an invalid region. However, when one of the address ranges becomes invalid, then any references to this object become invalid. In Fig. 3 (c), (d) and (e), the pointer $p$ points to data objects or variables which are not completely inside a valid address range. This is possible because C language allows arbitrary type casting and memory referencing as illustrated by the example in Figure 4. In line 11, an unsafe pointer assignment allows a member of the structure to be accessed that actually lies outside the memory region allocated in line 7. This is an example of the scenario represented by Fig. 3 (c).

```
 1. struct st { char ch; int i; };
 2.
 3. int main() {
 4.   char *p;
 5.   struct st *pst;
 6.
 7.   p = (char *) malloc(sizeof(char));
 8.   pst = (struct st *) p;
 9.   pst->i = 10;
10.
11.   return 0;
12. }
```

**Fig. 4.** An unsafe pointer assignment

The key to enforcing pointer safety within the Archerr framework involves bookkeeping of the various objects allocated/deallocated during the lifetime of a process' execution. This is accomplished through the premise that they must reside in one of the following regions and are accessible through the program itself.

**Valid Address Map For Data Segment.** The data segment holds all initialized and uninitialized global and static variables. The data segment is initialized based on the information present in the object file, which contains more than just these variables. Therefore, not all memory in the data segment should be accessible from the program. From a program's perspective, the memory that corresponds to only the global and static variables should be visible. However, virtual addresses and bounds for these sections are known only at link-time when all symbols and names have been resolved. But it is still possible to construct this address map at compile-time using a simple workaround. Let $var_i$ be some variable declared in a C program as either a `static`, `extern` or a global variable. Then the address range which bounds this variable is given by $[\&var_i, \&var_i + |\tau_i|]$, where & represents the 'address-of' operator and $|\tau_i|$ is the size allocated for this $i$th variable which is of type $\tau_i$. Some examples are the first three declarations and their corresponding ranges below:

```
int i;              [&i, &i + sizeof(i)];
char *str1;         [&str1, &str1 + sizeof(str1)];
char str2[10];      [&str2, &str2 + sizeof(str2)];
char *str = ''abc''; [&str, &str + sizeof(str)] ∪
                      [str, str + strlen(str)]
```

String literals which are used to initialize pointer variables of type `char *` are allocated in the data segment. They, however, do not have an l-value and have to be handled differently as shown in the last declaration above. The accessible portion of the data segment, denoted by $\mathbb{D}$ is the union of all such memory ranges, i.e., $\mathbb{D} = \bigcup_i [\&var_i, \&var_i + |\tau_i|]$. Since the data segment persists unaltered until the termination of the program execution, $\mathbb{D}$ needs to be constructed only once and no further modifications are required. $\mathbb{D}$ is constructed at the program's entry point, typically the first executable statement of *main()*.

**Valid Address Map For Heap Segment.** Memory can also be allocated and deallocated dynamically during program execution via the *malloc()* and *free()* family of library calls. This memory resides in the heap segment, denoted by $\mathbb{H}$. Let *malloc()*, for example, allocate a chunk of memory, referenced by a pointer $p$, of type $\tau$ and size $n$. The corresponding address range is represented by $[p, p+n\times|\tau|]$. Then, $\mathbb{H} = \bigcup_i [p, p+n\times|\tau|]$. An additional level of verification is required for deallocation, i.e. the *free()* library function, to prevent heap corruption. The only addresses that can be passed as an argument to *free()* are those that were obtained as a return value of the *malloc()* family of calls.

**Valid Address Map For Stack Segment.** The stack segment contains memory allocated to not only the automatic and local variables, but also other information such as the caller's return address. This makes it one of the primary targets for an attacker since control flow can be diverted to a malicious payload by overflowing a buffer. We take advantage of the fact that variables on the stack are allocated within well-defined frames. Let $\mathbb{S}$ denote the set of valid addresses allocated on the stack. The stack frames on the Intel architecture are defined by the contents of the registers EBP and ESP at the function entry points. The address range corresponding to a stack frame is [ESP, EBP]. This range is added to $\mathbb{S}$ at a function's entry point and it is removed at each function exit point. Note even before the program begins execution, some information is already on the stack, i.e., *argc*, *argv* and *env* as described in [12]. The corresponding address ranges remain valid until the program terminates and they should be added only once to $\mathbb{S}$ at the program entry point. For our implementation, we used inline assembly to capture the values of the EBP and ESP registers. We observed that *gcc* (version 2.95.4) allocates all the memory required in increments of 16 bytes at function entry point whether there are nested blocks or not, hence making it relatively easy to obtain the bounds of a stack frame. Other compilers may implement this differently. For this reason, the above method gives a slightly coarse-grained bound on the stack frame. A tighter bound would be provided by the union of memory ranges occupied by variables on the stack just as in the case

of the data segment. But this construction could cause significant computational overheads.

| | Unannotated version | | Annotated version |
|---|---|---|---|
| 1. | `struct st {` | 1. | `struct st {` |
| 2. | `char ch;` | 2. | `char ch;` |
| 3. | `int i;` | 3. | `int i;` |
| 4. | `};` | 4. | `};` |
| 5. | | 5. | |
| 6. | `struct st glbl;` | 6. | `struct st glbl;` |
| 7. | | 7. | |
| 8. | `int foo(int arg)` | 8. | `int foo (int arg)` |
| 9. | `{` | 9. | `{` |
| 10. | `static int sta;` | 10. | `static int sta;` |
| 11. | ··· | 11. | *append*($\mathbb{P}$, *current_stack_frame*) |
| 12. | | 12. | *append*($\mathbb{P}$, [&`sta`, &`sta` + `sizeof(sta)`]) |
| 13. | `return 0;` | 13. | ··· |
| 14. | `}` | 14. | *remove*($\mathbb{P}$, *current_stack_frame*) |
| 15. | | 15. | `return 0;` |
| 16. | | 16. | `}` |
| 17. | | 17. | |
| 18. | `int main(int argc, char *argv[])` | 18. | `int main(int argc, char *argv[])` |
| 19. | `{` | 19. | `{` |
| 20. | `char *buf;` | 20. | `char *buf;` |
| 21. | | 21. | *append*($\mathbb{P}$, `address ranges of argc, argv`) |
| 22. | `buf = (char *)` `malloc(sizeof(char)*32);` | 22. | *append*($\mathbb{P}$, *current_stack_frame*) |
| 23. | ··· | 23. | *append*($\mathbb{P}$, [&`glbl`, &`glbl + sizeof(glbl)`]) |
| 24. | `foo(32);` | 24. | |
| 25. | ··· | 25. | `buf = (char *)malloc(sizeof(char)*32);` |
| 26. | `free(buf);` | 26. | |
| 27. | `return 0;` | 27. | `if (buf) {` |
| 28. | `}` | 28. | *append*($\mathbb{P}$, [`buf`, `buf` + `sizeof(char)*32`]) |
| 29. | | 29. | $\mathbb{H} = \mathbb{H} \cup$ `buf` |
| 30. | | 30. | `}` |
| 31. | | 31. | ··· |
| 32. | | 32. | `foo(32);` |
| 33. | | 33. | ··· |
| 34. | | 34. | `if (buf` $\notin \mathbb{H}$`) then raise exception` |
| 35. | | 35. | *remove*($\mathbb{P}$, [`buf`, `buf` + `sizeof(char)*32`]) |
| 36. | | 36. | $\mathbb{H} = \mathbb{H} -$ `buf` |
| 37. | | 37. | `free(buf);` |
| 38. | | 38. | *remove*($\mathbb{P}$, *current_stack_frame*) |
| 39. | | 39. | `return 0;` |
| 40. | | 40. | `}` |

**Fig. 5.** The construction of the map of valid address ranges $\mathbb{P}$ in a program.

The bookkeeping operations that were described construct a map of valid address ranges, i.e., $\mathbb{P} = \mathbb{D} \cup \mathbb{H} \cup \mathbb{S}$. Figure 5 gives a simple example to show how $\mathbb{P}$ is constructed in a C program. More examples are provided in the appendix. Initially, $\mathbb{P}$ is an empty set and is populated and maintained as the program executes. In our implementation, address ranges belonging to the stack, data and heap are maintained in three separate linked lists. During address range search, these lists are indexed by the higher order bits of an address, for example, a stack address starts with 0xbfff—-. Also, in order to exploit spatial and temporal locality of programs, we perform insertions and subsequent deletions at the head of the lists. We also maintain a LRU cache of range entries to speed up address range lookups. In spite of these careful optimizations, the worst case is still $O(n)$ in the number of address ranges. However, production level programs are rarely written in the manner which achieves this worst-case and our empirical observations have shown the common case to be $O(1)$. All the memory corresponding to the bookkeeping operations of Archerr have to be kept private and not be a part of $\mathbb{P}$ or else the set of valid addresses may reach an inconsistent or incorrect state defeating the very purpose of this technique. This is accomplished through the pointer checks which also assert that pointer dereferences never access this private memory.

**Handling Function Aliasing Through Function Pointers**

The stack segment is generally not marked non-executable [2] because some important operating system mechanisms (such as signal delivery) and software (such as xfree86 4.x) require that code be executed off of the stack. This makes it difficult to specify safety properties for function pointers since they need not point only to the text segment. Currently, Archerr handles function aliasing through function pointers by raising warnings during preprocessing if function names are encountered which are neither previously defined not already recognized by Archerr such as the malloc family of system calls. We extend this checking to runtime by maintaining a list of known function addresses and raising warnings when function pointers are assigned values not belonging to this list.

**Handling String Library Functions**

Most string functions operate on character pointers of type `char *` either through assignment or pointer increments. For example, *strlen()* function takes a variable of type `char *` and increments it till it finds the end of string marker, i.e., the NULL character. We have specified earlier that if $p$ is assigned a valid address, then the operations that increment or decrement are safe only when properties (10) and (11) hold. It is useful to define primitives that determine the amount by which $p$ can be safely incremented or decremented. Since these primitives are essentially delta operators, they are denoted as $\Delta^+$ and $\Delta^-$. Let $p : q(\tau)$ be a pointer, then

$$\Delta^+(p) = \begin{cases} \lfloor (b-p)/|\tau| \rfloor & \text{if } \exists[a,\, b] \text{ and } a \leq p \leq b \\ 0 & \text{otherwise} \end{cases} \tag{12}$$

[2] Kernel patches are available that make the stack non-executable, but it breaks some applications and in any case, attacks exist that can bypass it.

$$\Delta^{-}(p) = \begin{cases} \lfloor (p-a)/|\tau| \rfloor & \text{if } \exists [a,\, b] \text{ and } a \le p \le b \\ 0 & \text{otherwise} \end{cases} \tag{13}$$

Since *strlen()* merely attempts to find the end of string marker, i.e., the NULL character, we consider this function to be safe in terms of memory accesses. Other string functions potentially modify memory locations and the goal is to prevent any invalid accesses. Assertions are inserted before statements in the code to ensure safety properties. When an assertion fails, the program raises an exception and terminates gracefully. If the requirement is continued execution, then there is yet another approach. We can simply modify these statements to produce a safer version. Table 2 enumerates a few of such conversions. The min operator has its usual meaning of finding the argument with the smallest value. Note that all the arguments should first be valid addresses and hence, property (9) should be asserted before these annotated statements.

| Original statement | Safer annotated statement |
|---|---|
| strlen(str) | strlen(str) |
| strcpy(dest, src) | strncpy(dest, src, min($\Delta^+$(dest), $\Delta^+$(src))) |
| strncpy(dest, src, n) | strncpy(dest, src, min(n, $\Delta^+$(dest), $\Delta^+$(src))) |
| gets(dest) | fgets(dest, $\Delta^+$(dest), stdin) |
| fgets(dest, n, fp) | fgets(dest, min(n, $\Delta^+$(dest))) |
| strcat(dest,src) | strncat(dest, src, min($\Delta^+$(dest + strlen(dest)), $\Delta^+$(src))) |
| strncat(dest, src, n) | strncat(dest, src, min(n , $\Delta^+$(dest + strlen(dest), $\Delta^+$(src), strlen(src)))) |

**Table 2.** Modified string library functions to support safety properties

### 2.3 User Defined Types

User defined data types are constructed using the basic data types. The operations on these new data types can be translated to equivalent operations of the constituent basic data types and safety properties may be individually asserted on these members. For example, consider a user defined structure containing a pointer variable as in Figure 6. One can assign an instance of this structure to another and this implicitly results in the addresses contained by the pointer variables to be copied. As long as a member variable is validated before use, unsafe behavior will not occur. Similarly, union types are simply an instance of implicit typecasting. Since we validate data types and not objects, it is straightforward to handle union types.

### 2.4 Typecasting

`void *` is the generic pointer type in C and typecasting is done to retrieve the actual object. This type of conversion is commonly seen in C programs. In our approach, since the only enforced constraint is that a program's memory accesses remain confined to the map of valid addresses, it results in a minimal set of safety

```
 1. typedef struct {
 2.   char *str;
 3.   int i;
 4. } USERDEF;
 5.
 6. int main() {
 7.   USERDEF a, b;
 8.
 9.   a.str = malloc(sizeof(char)*2);
10.   a.i = 0;
11.   b = a;
12.   strcpy(b.str, ``a''); /*safe*/
13.
14.   return 0;
15. }
```

**Fig. 6.** Handling user defined types

```
 1. typedef struct { int a; int b; } LG;
 2. typedef struct { int a; } SM;
 3.
 4. int main() {
 5.  SM *a = (SM *) malloc(sizeof(SM));
 6.  SM *b = (SM *) malloc(sizeof(SM));
 7.   LG *a_alias;
 8.
 9.   a_alias = a;
10.   a_alias->a = b->a;
11.
12.   return 0;
13.}
```

**Fig. 7.** Undisciplined typecasting

properties. Data within these confines can now be interpreted as required and therefore, there is no loss of flexibility even in the presence of pointer arithmetic and typecasting. However, there are instances where typecasting could introduce false positives in Archerr's framework. Consider the unannotated example in Figure 7. Before line 10, `a_alias` would be validated as of type `LG *` using rule (9). The code is valid, yet the check would signal a violation. This is due to the fact that our technique relies in small part on the static types of pointers at the preprocessing stage. This style of programming although not unsafe, is undisciplined. Archerr inadvertently discourages this style and therefore, false positives in such cases are acceptable.

## 3 Optimizations

An overly aggressive approach to code annotation introduces a large number of redundant checks causing an unnecessary overhead during execution. Fortunately, many aspects of our technique are amenable to optimization, reducing the slowdown significantly. Currently, some of these optimizations are done in an ad-hoc manner. Also, this is a partial list and research in this direction is a part of our ongoing and future work.

**Arithmetic Operations.** Our observation was that correctness of most of the arithmetic operations could be verified statically and therefore, checks do not have to be inserted for these operations. Typically, checks have to be inserted only for operations on integers whose values are determined at runtime and operations involving dereferenced integer pointers. Otherwise Archerr validates all statically resolvable operations during the preprocessing stage, significantly reducing the number of inserted runtime checks.

**Pointer Dereferencing.** Property (9) implies that when a pointer variable is encountered in a program either in simple assignment or dereference, it must be validated. This results in a large number of checks, many of which are redundant. Mere address assignments to pointer variables have no direct security

implications. The only locations where checks are mandatory are those where pointer dereferencing occurs through the $*$ and $->$ operators. This reduces the number of checks and improves the execution performance without loss of safety. However, a downside to this optimization is that the point where an exception is raised due to an invalid memory access need not be the actual source of the problem; it could be several statements earlier.

**Loops.** When the primary purpose of a loop is to initialize or access data using an index, then the safety checks instead of being placed inside the loop can be placed outside. The following example shows an unoptimized piece of code followed by its optimized version. Note that Archerr currently does not implement this optimization, however it is on our list of future work.

**Unoptimized version**

```
1. char *buffer;
2.
3. buffer = (char *)
     malloc(sizeof(char)*100);
4.
5. for (i = 0; i < 100; i++) {
6.    validate(&buffer[i])
7.    buffer[i] = (char)0;
8. }
```

**Optimized version**

```
1. char *buffer;
2.
3. buffer = (char *)
     malloc(sizeof(char)*100);
4.
5. validate(buffer) ∧
     assert(Δ+(buffer) < 100)
6. for (i = 0; i < 100; i++) {
7.    buffer[i] = (char)0;
8. }
```

## 4 Experiments and Results

We have evaluated Archerr both for vulnerability coverage and overheads due to runtime checks. Our test machine was a 2.2GHz Pentium 4 PC with 1024MB of RAM.

### 4.1 Security Testing

Our technique is a passive runtime checking approach that can detect vulnerabilities only when an exploit causes a safety violation in some execution path. Therefore, our security tests involve programs with known vulnerabilities and publicly available exploit code. The SecurityFocus website is an excellent repository of such vulnerabilities and corresponding exploit code.

**sendmail-8.11.6.** sendmail is a widely used mail server program. This particular version of sendmail shipped with RedHat Linux 7.1 has a buffer overflow vulnerability [13]. We installed this distribution and tested the exploit code, which successfully obtained a root shell. We rebuilt the sources after running it through Archerr and ran the same exploit. The attack was detected as soon as a pointer variable dereferenced an address outside the valid map.

**GNU indent-2.2.9.** indent is a C program beautifier. This particular version of indent has been reported to have a heap overflow vulnerability [14], which can be

exploited using a malicious C program. Supposedly, it copies data from a file to a 1000 byte long buffer without sufficient boundary checking. The exploit code is able to construct this malicious file, which we used to launch the attack. At first, we caught a potential null pointer dereferencing bug. Once the corresponding check was disabled, we were able to catch the actual heap overflow.

**man 1.5.1.** This program is reported to have a format string vulnerability [15]. The exploit code is supposed to give an attacker a root shell through a format string argument to vsprintf(), which the attacker can control. However, when we tried to replicate the same conditions for the exploit code to work, we got a segmentation fault through the vsprintf() call. Nevertheless, this showed the existence of a format string bug. Archerr's annotations could not detect this attack mainly because vsprintf() is a part of an external library that has not been annotated. This attack could have been caught if Archerr annotations were applied to vsprintf() code. This is one of the limitations of Archerr that it cannot protect a source program if it has not been processed.

**pine 4.56.** pine is a popular mail client. Version 4.56 and earlier are susceptible to an integer overflow attack [16]. No exploit code was publicly available for this vulnerability. However, a mail posting [17] provided directions to construct an attack. The problem can be traced to addition operations on a user-controlled signed integer variable, which cause it to become negative. In the following code, it is used as an index into a buffer, resulting in this vulnerability. Compiling this version was very error-prone due to the non-standard build process. However, once it was setup and annotations were inserted, the attack was detected before an integer overflow could occur.

### 4.2 Performance Testing

We chose the Scimark 2 [18] benchmark for the purposes of evaluating the effect of Archerr's annotations on execution time. This benchmark uses the following kernels: 1) Fast Fourier Transform (FFT) - this kernel exercises complex arithmetic, shuffling, non-constant memory references and trigonometric functions, 2) Jacobi Successive Over-relaxation (SOR) - this algorithm exercises basic memory patterns, 3) Monte Carlo integration (MC) approximates the value of PI, by computing certain integrals, 4) Sparse Matrix Multiply (SMM) - this kernel exercises indirection addressing and non-regular memory references, 5) Dense LU Matrix Factorization (LU) - this kernel exercises linear algebra kernels (BLAS) and dense matrix operations.

In order to understand the overheads imposed by Archerr, we ran the original benchmark without any source code annotations. Then we compiled only the pointer checks. Subsequently, we enabled the full functionality of Archerr. The overheads imposed are reported in Table 3. The overheads incurred by pointer checks alone are in the range of 1.2-3.8X. Additional checks on integer arithmetic causes an overall slowdown of 1.23-4.4X. The composite score of this benchmark suite gives an average slowdown of 2.3X with pointer checks and 2.5X with both pointer and integer checks. The overhead is noticeable but not excessive. We also ran the Scimark2 suite against two well-known language security techniques -

CCured [4], a type-based approach, and Jones & Kelly's bounds-checking [9], a primarily runtime checking approach, for the purposes of comparison. CCuring the Scimark2 benchmarks caused them to run 1.5X slower on an average, and Jones and Kelly's technique incurs a composite performance hit of > 30X. The significant gains observed over Jones & Kelly's technique is mainly because our technique provides type-based coarse-grained safety as opposed to strict object-centric safety. Our technique is also only 1X more expensive than CCured. We noticed a peculiarity in our results that running the source code through some of these techniques seemed to improve the performance. Such anomalies have also been reported in [19] and are attributed to the compiler's internal optimizations.

| Kernel | Original | Ptr Checks | Slow down | Ptr and Int Checks | Slow down | CCured | Slow down | J&K | Slow down |
|---|---|---|---|---|---|---|---|---|---|
| FFT | 89.36 | 31.80 | 2.81X | 24.42 | 3.66X | 60.23 | 1.49X | 5.22 | 17.12X |
| SOR | 178.42 | 149.93 | 1.19X | 145.07 | 1.23X | 295.55 | -1.66X | 7.80 | 22.87X |
| MC | 30.93 | 19.70 | 1.57X | 16.72 | 1.85X | 32.90 | -1.06X | 3.60 | 8.59X |
| SMM | 271.93 | 70.27 | 3.87X | 61.80 | 4.40X | 103.70 | 2.62X | 3.76 | 72.32X |
| LU | 341.33 | 121.90 | 2.80X | 116.10 | 2.94X | 130.78 | 2.61X | 6.86 | 49.76X |
| Composite score | 182.39 | 78.28 | 2.33X | 72.38 | 2.52X | 124.63 | 1.46X | 5.45 | 33.47X |

**Table 3.** MFlops: original vs. only pointer checks vs. pointer and arithmetic checks, and comparison with CCured and Jones & Kelly's technique

### 4.3 Impact of Source Code Size and Runtime Image

The number of checks inserted is directly dependent on the number of pointer dereferences and integer operations that could not be resolved statically. We have seen a source code bloat in the order of 1.5-2.5X. Increase in the runtime image can be attributed to the statically linked library and the in-memory housekeeping information regarding address ranges. Our empirical results show that the overall impact on the runtime image is nomimal (1.2-1.4X). This is due to the following reasons. The Archerr library code is partly written in assembly and partly in C, and the resulting library image is small. Address ranges are implemented as very small data structures. Furthermore, our coarse-grained approach allows multiple address ranges to be collapsed to a single address range when memory on heap or new stack frames are allocated. Although addition or deletion of address ranges could potentially result in $O(n)$ behavior in the worst case, the memory consumption is not significant in the general case.

## 5 Comparison With Related Work

Several important research efforts have attempted to empower the programmer with techniques and tools to produce safe code. We briefly describe these approaches and compare them with the techniques proposed in this paper.

### 5.1 Bounds Checking

Several commercial and open-source tools [9, 20, 21] exist that perform memory access and bounds checking. Jones et al. [9] describe a technique that checks both pointer use and pointer arithmetic, and this work is the closest to our approach in this regard. Safety is addressed through source code annotations by ascertaining that pointer operations are only permitted to take place within an object and not across objects. However, there are some drawbacks that we improve upon. Most markedly, our technique is significantly more efficient than their technique. Also, their technique only addresses pointer usage, while we present a more holistic approach. By inserting dead space around each object, they can detect violations at runtime. This rigid check structure results in inability to handle complex data structures such as multi-dimensional arrays, arrays within arrays, arbitrary typecasting, and finally, excessive runtime overhead. We are able to address these aspects effectively in our approach, mainly due to the fact that we relax the strict object-centric view. Finally, they have implemented their technique as a GNU C compiler extension, while we make no modifications to the C compiler.

Purify [20] is a widely-used tool, which processes binary code to insert runtime memory access checks. However, Purify is used mainly for debugging purposes because of its inefficiency. Also, it will some times permit stack corruption by allowing a program to access past stack-allocated arrays. Archerr clearly guards against this misuse by maintaining maps of delimited stack frames. Wagner et al. [1] proposed a proactive static analysis technique that formulates the buffer overruns as a integer range analysis problem by specifying integer bounds on arrays and strings, and solving the constraint system. Since it is a compile-time technique that actively pursues violations, it can detect many of them even before compilation. In contrast, our technique is a runtime detection technique; hence, it is passive and violations are detected only when the corresponding execution path is active. However, [1] handles pointer typecasting and aliasing in an ad-hoc manner and this approximation raises several false positives.

### 5.2 Stack Protection

StackGuard [22], StackShield [23] and ProPolice [24] are effective runtime techniques that prevent stack overrun. StackGuard [22] is a compiler extension that adds a random padding called the *canary* around the return address on the stack. These bytes are checked at each function entry and exit. StackShield [23] moves the return address to a location that cannot be overflowed. Manipulating the stack frame affects compatibility and can break applications such as debuggers, which probe the stack frame for function call information. Protection in ProPolice [24] is accomplished by inserting checks during compilation and reordering variables to prevent pointer corruption. All these techniques use architecture-specific information about stack layout in a clever manner. Focussing mainly on stack protection, although very efficient, can allow attacks [25] to bypass these techniques.

### 5.3 Type Based Safety

Type theory provides a formal framework to express and verify certain correctness properties of programs. Statically analyzing programs based on type correctness can detect many common programmer errors. Some of these correctness notions have been extended to express safety properties. A few research efforts have targeted the C programming language's flexibility of pointers and have augmented the type system to produce safer dialects. Cyclone [26] introduces a notion of region based type theory into C. All variables apart from their type, have an additional attribute called a *region*, which is determined based on where the variable resides, i.e., stack or heap. Analysis with the help of this concept allows the detection of invalid region based memory accesses in programs. CCured [4] describes an augmented type system for C to prevent invalid memory accesses due to pointers. Based on their usage, pointers are segregated as safe and unsafe. Programs can be statically analyzed and while most pointer usage can be checked statically, some of them require runtime checks. There are cases where a program will stop even when it is safe and manual intervention is necessary. In the paper [4], they report a execution slowdown of up to 150% on their benchmarks. We have observed similar slowdown on our benchmarks and our technique is comparable to CCured in terms of overhead. The caveat to these approaches is that safety is not achieved transparently and the programmer is burdened by a possibly steep learning curve in both cases. Some of the concepts in this paper, such as static analysis of basic types, region and pointer-based analysis are similar. One advantage to our approach as well as to some of the other techniques described in Section 5.1 is that source code need not be rewritten or ported and programmers are minimally affected. Again, we cover pointers and more.

### 5.4 Other Relevant Techniques

From an attacker's point of view, chances of successfully overflowing a buffer are largely dependent on a good approximation of its address. Based on this premise, pointers and location of objects can be randomized in order to render attacks ineffective. Unlike the bounds-checking techniques, the following [19, 27] incur much less overhead but possibly may be bypassable by attackers as discussed in [19]. The PAX project [28] incorporates the technique of ASLR (Address Space Layout Randomization). ASLR randomizes the location of key memory spaces, such as the stack and the heap. Sekar et al. [27] work to randomize the absolute locations of all code and data segments as well as the relative distance between memory objects. Cowan et al. in [19] describe the a similar, yet more fine-grained tool PointGuard that obfuscates all memory addresses via encryption using a key that is generated at run-time. Orchestrating a buffer-overflow attack in this scenario requires knowledge of the secret key. Similar to many of the above tools, these approaches are focused solely on ensuring safe pointer manipulation.

Libsafe [29] is a binary level technique that provides protection against buffer overflows by intercepting calls to "unsafe" string functions and performing a bounds check on the arguments. We address the problem of unsafe functions by inserting sanity checks on the arguments in the source code prior to compilation.

## 6 Conclusion and Future Work

In this paper, we have described a novel approach for statically analyzing and annotating code in order to ensure safe execution in a particular runtime environment. Access checks that are placed in the code ascertain that the code is performing numerical computations and accessing memory in a safe manner. Since there is no notion of well-defined and immutable objects per se, memory within the valid address ranges can be typecast arbitrarily. This largely retains the flexibility of the language. If code annotations are performed in an exhaustive way, then the slowdown is not negligible but very useful for debugging purposes. On the other hand, production systems require efficient execution and in such a scenario, optimizations can reduce the number of annotations. By making the runtime environment specification as a separate parameter, it is possible to generate annotated versions of the same source code that is safe for each instance of the specified runtime environment, subsequently, providing tighter security.

Although our technique provides coverage against a wide variety of vulnerabilities, the coverage is limited to the source code that our technique processes. Therefore, vulnerabilities in external libraries cannot be handled unless the libraries are also annotated. Like other code annotation techniques, there is little protection against runtime process image tampering and manipulation using programs such as *gdb*. This implies that protection against exploits is limited to those which do not require direct access to the process image. Although this technique is practical with small impacts on performance, it does not solve all the problems by itself. Therefore, our recommendation is to use this tool in tandem with more proactive approaches.

This paper lays the basic groundwork for a useful and efficient runtime detection technique to prevent exploits. We are investigating further improvements to the technique. In its present form, this technique is reactive and can only be used to prevent exploits. But there is scope to provide detection capabilities at the preprocessing stage itself. For example, pseudo interval ranges can be created even during compilation stage to represent the data segment and the stack frames, and then program execution can be simulated partially to detect possible violations. At this point, it is only a speculation and we are looking at issues regarding this possibility. The runtime environment specified in Section 2 has been the focus of this work. However, we are also looking at other combinations that can serve as a viable runtime environment. For example, a runtime specification for x86/win32/pe would certainly be very useful. As for the development of Archerr, our immediate goals are to implement further optimizations as well as mature the tool to handle large distributions without much manual interaction. We are planning on releasing an alpha version of Archerr very shortly. The long-term goal is to develop a robust, customizable tool that supports a myriad of runtime environments.

## References


1. Wagner, D., Foster, J.S., Brewer, E.A., Aiken, A.: A First Step towards Automated Detection of Buffer Overrun Vulnerabilities. In: Network and Distributed System Security Symposium, San Diego, CA (2000) 3–17
2. Landi, W.: Undecidability of Static Analysis. ACM Letters on Programming Languages and Systems **1** (1992) 323–337
3. Ramalingam, G.: The Undecidability of Aliasing. ACM Transactions on Programming Languages and Systems **16** (1994) 1467–1471
4. Necula, G.C., McPeak, S., Weimer, W.: CCured: Type-safe Retrofitting of Legacy Code. In: Symposium on Principles of Programming Languages. (2002) 128–139
5. Jones, R.W.M., Kelly, P.H.J.: Backwards-Compatible Bounds Checking for Arrays and Pointers in C Programs. In: Automated and Algorithmic Debugging. (1997) 13–26
6. One, A.: Smashing the Stack for Fun and Profit. Phrack 49, Vol. 7, Issue 49 (1996)
7. Bianco, D.J.: An Integer Overflow Attack Against SSH Version 1 Attack Detectors. In: SANS Cyber Defense Initiatives. (2001)
8. Cohen, C.F.: CERT Advisory CA-2002-17 Apache Web Server Chunk Handling Vulnerability (2002)
9. Jones, R., Kelly, P.: (Bounds Checking for C) http://www-ala.doc.ic.ac.uk/ phjk/BoundsChecking.html.
10. TIS Committee: Tool Interface Standard (TIS), Executable and Linking Format (ELF) Specification, Version 1.2 (1995)
11. Standard for Binary Floating Point Arithmetic. ANSI/IEEE Standard 754-1985 (1985)
12. Boldyshev, K.: Startup State of a Linux/i386 ELF Binary. An article hosted on http://linuxassembly.org (2000) http://linuxassembly.org/articles/startup.html.
13. Bugtraq ID 7230: Sendmail Address Prescan Memory Corruption Vulnerability (2003) http://www.securityfocus.com/bid/7230.
14. Bugtaq ID 9297: GNU Indent Local Heap Overflow Vulnerability (2003) http://www.securityfocus.com/bid/9297/info/.
15. Bugtraq ID 7812: Man Catalog File Format String Vulnerability (2003) http://www.securityfocus.com/bid/7812.
16. Bugtraq ID 8589: Pine rfc2231_get_param() Remote Integer Overflow Vulnerability (2003) http://www.securityfocus.com/bid/8589.
17. Posting on Bugtraq Mailing List: (2003) http://archives.neohapsis.com/archives/bugtraq/2003-09/0181.html.
18. Scimark 2.0: (2003) http://math.nist.gov/scimark2/index.html.
19. Cowan, C., Beattie, S., Johansen, J., Wagle, P.: PointGuard: Protecting Pointers from Buffer Overflow Vulnerabilties. In: Proceedings of the 12th USENIX Security Symposium, Washington, D.C. (2003)
20. (Rational PurifyPlus) http://www-306.ibm.com/software/awdtools/purifyplus/.
21. (NuMega BoundsChecker) http://www.numega.com/products/aed/vc_more.shtml.
22. 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: 7th USENIX Security Symposium, San Antonio, TX (1998)
23. Vendicator: (StackShield: A "Stack Smashing" Technique Protection Tool for Linux) http://www.angelfire.com/sk/stackshield/.

24. Etoh, H.: (GCC Extension for Protecting Applications from Stack-smashing Attacks) http://www.trl.ibm.co.jp/projects/security/ssp6.
25. Bulba, Kil3r: Bypassing StackGuard and StackShield. (Phrack Magazine, Volume 0xa Issue 0x38)
26. Jim, T., Morrisett, G., Grossman, D., Hicks, M., Cheney, J., Wang, Y.: Cyclone: A Safe Dialect of C. In: USENIX Annual Technical Conference, Monterey, CA (2002)
27. Bhatkar, S., DuVarney, D.C., Sekar, R.: Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits. In: Proceedings of the 12th USENIX Security Symposium, Washington, D.C. (2003)
28. PAX Project: (2003) "http://pax.grsecurity.net/docs/aslr.txt".
29. Bartaloo, A., Singh, N., Tsai, T.: Transparent Run-Time Defense Againsts Stack Smashing Attacks. In: 2000 USENIX Annual Technical Conference, San Diego, CA (2000)

# APPENDIX

The following examples illustrate a few scenarios and how our technique handles them. Some of these examples are borrowed from [5, 26] for the purpose of comparison.

**Dead Return Values**

```
char * foo() {
    char buffer[32];
    return buffer;
}

int main() {
    char *p;
    p = foo();
    validate(p);
    strncpy(p, ``abc'',
        min(Δ+(p), strlen(``abc''), 3));
```

**Multiple function exit points**

```
int foo(void) {
    ℙ = ℙ ∪ current_stack_frame
    if (cond) {
      ℙ = ℙ − current_stack_frame
      return -1;
    }
    ℙ = ℙ − current_stack_frame
    return 0;
}
```

**Nested blocks**

```
int foo() {
    char *p = 0xBAD;
    append(ℙ, current_stack_frame)
    { char *p;
      validate(p);
      *p = 'a';
    }
    p = 0xBAD2;
    validate(p);
    *p = 'a';
```

**Goto**

```
int z;
{ int y = 0xBAD; goto L; }
{ int *y = &z;
L: validate(y); *y = 3;
}
```