PAriCheck: An Efficient Pointer Arithmetic Checker for C Programs

Yves Younan
Katholieke Universiteit Leuven
yvesy@cs.kuleuven.be

Pieter Philippaerts
Katholieke Universiteit Leuven
pieter@cs.kuleuven.be

Lorenzo Cavallaro
Vrije Universiteit Amsterdam
University of California at Santa Barbara
sullivan@cs.ucsb.edu

R. Sekar
Stony Brook University
sekar@seclab.cs.sunysb.edu

Frank Piessens
Katholieke Universiteit Leuven
frank@cs.kuleuven.be

Wouter Joosen
Katholieke Universiteit Leuven
wouter@cs.kuleuven.be

ABSTRACT

Buffer overflows are still a significant problem in programs written in C and C++. In this paper we present a bounds checker, called PAriCheck, that inserts dynamic runtime checks to ensure that attackers are not able to abuse buffer overflow vulnerabilities. The main approach is based on checking pointer arithmetic rather than pointer dereferences when performing bounds checks. The checks are performed by assigning a unique label to each object and ensuring that the label is associated with each memory location that the object inhabits. Whenever pointer arithmetic occurs, the label of the base location is compared to the label of the resulting arithmetic. If the labels differ, an out-of-bounds calculation has occurred. Benchmarks show that PAriCheck has a very low performance overhead compared to similar bounds checkers. This paper demonstrates that using bounds checkers for programs or parts of programs running on high-security production systems is a realistic possibility.

Categories and Subject Descriptors

D.4.6 [**Operating systems**]: Security and Protection

General Terms

Security

Keywords

buffer overflows, bounds checking

ASIACCS'10 April 13–16, 2010, Beijing, China.

1. INTRODUCTION

Security has become an important concern for all computer users. Worms and hackers are a part of every day Internet life. A particularly dangerous type of vulnerability is the buffer overflow. This type of vulnerability can be used to perform a number of attacks. A first type of attack is the code injection attack, where attackers are able to insert code into the program's address space and can subsequently execute it. However, such a vulnerability can also lead to non-control data attacks, where attackers overwrite data to change the behavior of the program. Programs written in C are particularly vulnerable to such attacks. Attackers can use a range of vulnerabilities to inject code. The most well known and most exploited buffer overflow is, of course, the standard buffer overflow: attackers write past the boundaries of a stack-based buffer, overwrite the return address of a function and make it point to their injected code (or they could execute existing code). When the function subsequently returns, the code injected by the attackers is executed [4]. While this type of vulnerability has existed for a long time, it is still an active research topic with both attackers [37] and defenders [2, 3] continually improving their techniques, in terms of effectivity, efficiency and automation.

Buffer overflows also still occur often in real world programs: according to the NIST's National Vulnerability Database [31], 465 buffer overflow vulnerabilities were reported in the first 10 months of 2008, making up 10% of the 4,668 vulnerabilities reported in that period. Of those buffer overflows vulnerabilities, 347 had a high severity rating. As a result, buffer overflows make up 15% of the 2,324 vulnerabilities with a high severity rating reported in those 10 months.

Many countermeasures exist to protect against these types of problems [44, 20]. They range from safe languages [23, 32, 19, 27, 43] that remove the vulnerabilities entirely, to bounds checkers [24, 36, 18, 2] that will perform runtime checks for out-of-bounds accesses, to very lightweight countermeasures that prevent certain memory locations from being overwritten [17, 13] or prevent attackers from finding or executing injected code [40, 7, 6].

While these last techniques are efficient and useful against many attacks, they often suffer from significant drawbacks.

Many of these countermeasures rely on memory locations remaining secret, which is not always the case. Other countermeasures prevent execution of injected code, but attackers may still achieve their objective by simply calling existing code in unintended ways. As a result, attackers have found several ways of bypassing these approaches.

Bounds checkers on the other hand prevent the entire class of buffer overflow vulnerabilities by ensuring that the application cannot write outside the bounds of objects. However, the deployment of bounds checkers has not been forthcoming because, while they can offer a high level of assurance, they suffer from other drawbacks. Most bounds checkers have a significant amount of performance overhead related to the checks that they insert. Others require significant architectural changes that require a specific memory allocation scheme to be used. Yet others rely on static analysis techniques to speed up performance, sometimes at the expense of compatibility and scalability.

This paper presents PAriCheck (Pointer Arithmetic Checker), a new approach for protecting C applications against the exploitation of buffer overflow vulnerabilities by performing efficient dynamic bounds checks without requiring large infrastructural changes (like replacing the memory allocator). To perform our bounds checks, we start from a well-known bounds checking approach developed by Jones and Kelly [24], our primary contribution is the development of a more efficient alternative to the problem of pointer arithmetic checking rather than using the splay-tree based approach used by Jones and Kelly. To achieve compatibility with existing code, the pointer representation is left unchanged and to achieve more efficient lookups no bounds information for objects [1] is stored. Instead we store a unique ID, called a label, for every object and ensure that the label is associated with the entire memory area that is allocated for that object. We then instrument pointer arithmetic to compare the label of the memory location resulting from the arithmetic with the label of the base object. If the labels differ, an out-of-bounds calculation has occurred. According to the C standard [25], this would result in undefined behavior if the out-of-bounds calculation was larger than 1 element out-of-bounds and as a result we could terminate the program. However, as noted by CRED [36], a significant amount of programs that are not vulnerable to buffer overflows will perform out-of-bounds calculations that are larger than 1 element out-of-bounds. To provide support for these programs, we use a technique similar to the one described by CRED [36]: where out-of-bounds calculations are recorded and if subsequent calculation goes in bounds again the intended value is calculated. As a result, PAriCheck is also compatible with these programs.

The rest of this paper is structured as follows: Section 2 briefly recaps buffer overflows and how attackers can abuse this type of vulnerabilities. Section 3 discusses the design of our countermeasure, while Section 4 examines the details of our prototype implementation. Section 5 presents an evaluation of our prototype by benchmarking it both in terms of performance and memory overhead. This section also discusses the security impact of our countermeasure. In Section 6 we compare our bounds checker to other countermeasures. Finally, Section 7 presents our conclusion.

[1]We will use the term object to refer to an array, structure or union allocated on the stack or in global memory, or any area of memory that has been dynamically allocated.

2. BUFFER OVERFLOWS

Buffer overflows are the result of an out-of-bounds write operation on an array. In this section we briefly recap how an attacker could exploit such a buffer overflow. A detailed overview of how a buffer overflow can be exploited, can be found in [4].

Buffers can be allocated on the stack, the heap or in the data/bss section in C. For arrays that are declared in a function body, space is reserved on the stack. Buffers that are allocated dynamically (using the *malloc* function, or some other variant), are put on the heap, while arrays that are global or static are allocated in the data/bss section. The array is manipulated by means of a pointer to the first byte. Bytes within the buffer can be addressed by adding the desired index to this base pointer.

Listing 1: A C function that is vulnerable to a buffer overflow.

```
void copy(char* src, char* dst) {
  int i = 0;
  char curr = src[0];
  while(curr) {
    dst[i] = curr;
     i++;
    curr = src[i];
  }
}
```

At run-time, no information about the array size is available. Consequently, most C-compilers will generate code that will allow a program to copy data beyond the end of an array. This behavior can be used to overwrite interesting data in the adjacent memory space.

On the stack this is usually the case: it stores the addresses to resume execution at, after a function call has completed its execution. This address is called *the return address*. Manipulating the return address might give the attacker the possibility to execute arbitrary code. In addition to the return address, the contents of other variables on the stack might also be overwritten. This could be used to manipulate the results of previous calculations or checks.

Likewise, the heap also often contains important memory management information right before the allocated buffer. By manipulating this information, an attacker can control the execution path of the processor when the application frees the allocated memory.

Listing 1 shows a straightforward string copy function. Improper use of this function can lead to a buffer overflow, because there is no validation that the destination buffer can actually hold the input string. This function will be used throughout the paper as a running example to explain the workings of the proposed algorithm.

Many current countermeasures try to protect applications by protecting the return addresses and other important management information. However, this isn't enough to thwart buffer overflow attacks. The following subsections show how someone might be able to successfully attack an application, without overwriting the return address. These attacks will also be protected against by PAriCheck.

2.1 Non-control Data Attacks

An example of how an attacker can modify the execution flow, without overwriting important management information, comes from a real bug present in old versions of the UNIX login program. Listing 2 shows a simplified version of the algorithm used to authenticate a user.

Listing 2: A simplified version of the UNIX login function.

```
int login() {
  char username[8];
  char hash_pw[8];
  char pw[8];
  printf("login:");
  gets(username);
  // Put stored hash
  //  in hashed_pw
  lookup(username, hash_pw);
  printf("password:");
  gets(pw);
  if (eq(hash_pw,hash(pw)))
    return OK;
  else
    return INVALID_LOGIN;
}
```

Since it is impossible to specify the maximum number of characters that should be read from the input, using the *gets* function, the attacker might trigger a buffer overflow. By entering a password that is longer than eight characters, the array that stores the hash of the password can be overwritten. Hence, if the attacker enters an eight-character password, and appends this password with its eight-character hash value, the original hash value will be overwritten with the value supplied by the attacker. This will cause the login application to authenticate the attacker.

Attacks like this do not inject code, but they can be used to modify the behavior of an application to the benefit of an attacker. Even though the example presented here occurred decades ago, the problem is still relevant today (see [12]). Furthermore, state of the art countermeasures often do not detect this type of buffer overflows.

2.2 Attacker-controlled Offsets

Changing the execution flow of an application can sometimes be accomplished without executing an overflow of contiguous memory. Listing 3 shows an example of a function that contains such a flaw. In this example, the attacker controls the data of the three parameters. However, unlike in previous examples, no contiguous buffer overflow is used to take over the application.

Listing 3: A function that is vulnerable to an arbitrary code injection attack

```
void (*log_msg)(char *msg)
    = printf;

void change_and_log(int *buffer,
      int offset, int value) {
  buffer[offset] = value;
  log_msg("Buffer␣has␣changed");
}
```

The function in listing 3 modifies the contents of a buffer, and sends a message to the log. Logging is done by calling a function pointer to a function that logs the message. In this example, the log is written to the console (using the *printf* function).

By adjusting the values of the *offset* parameter, the attacker is able to overwrite the *log_msg* function pointer. If the location of *buffer* is known, and the location of the *log_message* pointer is also known, the offset can be calculated as the difference between *buffer* and *log_message*. Then, the attacker-controlled *value* is written to this location. After this assignment, the *log_message* function pointer is called, which transfers control to whatever address the attacker wrote in it in the previous step. This could for instance be an address in *buffer* that contains malicious machine code.

2.3 Integer Errors

Integer errors [10] are not exploitable vulnerabilities by themselves, but exploitation of these errors could lead to a situation where the program becomes vulnerable to a buffer overflow. Two kinds of integer errors that can lead to exploitable vulnerabilities exist: integer overflows and integer signedness errors. An integer overflow occurs when an integer grows larger than the value that it can hold. The ISO C99 standard [25] mandates that unsigned integers that overflow must have a modulo of MAXINT+1 performed on them and the new value must be stored. This can cause a program that does not expect this to fail or become vulnerable: if used in conjunction with memory allocation, too little memory might be allocated causing a possible heap-based buffer overflow.

Integer signedness errors occur as follows: when the programmer defines an integer, it is assumed to be a signed integer, unless explicitly declared unsigned. When the programmer later passes this integer as an argument to a function expecting an unsigned value, an implicit cast will occur. This can lead to a situation where a negative argument passes a maximum size test but is used as a large unsigned value afterwards, possibly causing a stack-based or heap-based overflow if used in conjunction with a copy operation (e.g. *memcpy* expects an unsigned integer as size argument and when passed a negative signed integer, it will assume this is a large unsigned value).

3. COUNTERMEASURE DESIGN

In this section we describe the general approach taken by PAriCheck.

3.1 Approach

The vulnerabilities described in Section 2 are the result of out-of-bounds operations on arrays or via pointers. One approach to protecting against this type of attack is to perform bounds checking: ensuring that no access is allowed out-of-bounds of the memory area that an object inhabits. However, bounds checking in C is a hard problem, due to the pointer manipulation that is freely allowed in C and the lack of bounds information that is available at runtime. Most bounds checkers will protect against attacks on those vulnerabilities by ensuring that a pointer does not go out-of-bounds by storing extra information with the pointer. The bounds checker will then check if the pointer is pointing to the object that it is supposed to be pointing to when it is

dereferenced. PAriCheck is based on the observation in [24] that pointers go out-of-bounds as the result of an arithmetic operation. As a result, the main principle behind our countermeasure is that we only perform out-of-bounds checks on the results of pointer arithmetic. If we ensure that pointers that go out-of-bounds via arithmetic are made invalid, then an out-of-bounds access via these pointers is not possible.

Our countermeasure will not change the way that pointers are represented, so that we stay compatible with existing non-protected code. Unlike other countermeasures, we also do not store bounds information for objects or pointers. This is the main efficiency improvement made by our countermeasure: by not storing bounds information for objects, we do not need complex checks to find out which object a pointer is pointing into, which results in more lightweight lookups and checks.

Instead, we store a unique number for the memory area (called a label) that an object inhabits. When the program performs pointer arithmetic, we look up the label of the memory location that we start the operation with and then look up the label of the resulting pointer that results from the arithmetic operation. If the labels match, the pointer is still pointing within the bounds of the object; if they differ, then we have an out of bounds calculation. If we were to use size information the lookup would be more complex, instead of comparing two labels we would have to look up the size information of the memory area that the current pointer is pointing into and then make sure that the arithmetic being performed does not go beyond these bounds. While the last check is fairly cheap, data structures for storing and looking up boundaries of allocated objects are more expensive in terms of computation as well as storage

These labels will take up a particular amount of memory. To be able to use a single label for more than one byte of memory, we make sure that all objects are a multiple of 2^k bytes and we also ensure that the object's base address is aligned to 2^k. We call such a memory area of 2^k bytes a region. An object can inhabit multiple regions and we will need the same label for each region. The region's size is fixed per application and the optimal size will depend on the application's memory use. We dicuss this further in Section 4. When an object is allocated (by entering a function, when the program starts for global/static memory or by explicit dynamic allocation), we assign a label to the regions that the object inhabits. When the object is later deallocated, we do not remove the label as a new label will be assigned when the object's previous regions are reused. If an object inhabits multiple regions, then it will be assigned the same label multiple times. Since all regions that an object inhabits contain the same label, any copy to any location within the object will be considered in-bounds. Since the label is associated with the memory region, a pointer can point to any location within the object: the bounds check will be performed correctly by looking up the label associated with the memory location that the pointer is pointing to.

Listing 4: A C function that calls the copy function from Listing 1

```
int main() {
char src1[8], dst1[100];
char src2[100], dst2[8];
// init all strings with a
// null-terminated string
// of their maximum size
copy(src1,dst1);
copy(src2,dst2);
}
```

Listing 4 contains a small program that calls the vulnerable copy function described in Listing 1. Figure 1 has a graphical representation of what the program looks like when k is set to 5 (i.e., each region is 32 bytes large). The full lines represent the memory area that was originally requested for the object, while the dotted lines represent the regions that the object's inhabits.

Listing 5: Transformed copy and main functions from Listings 1 and 4

```
int main() {
char src1[32], dst1[128];
char src2[128], dst2[32];
labelsspace(&src1,sizeof(src1),1);
labelsspace(&dst1,sizeof(dst1),2);
labelsspace(&src2, sizeof(src2),3);
labelsspace(&dst2,sizeof(dst2),4);
// init all strings with a
// null-terminated string
// of their maximum size
copy(src1,dst1);
copy(src2,dst2);
}

void copy(char *src, char *dst) {
int i = 0; char curr;
char *dsti, *srci;
short int label1, label2;
curr = *src;
while (curr) {
  label1 = getlabel(dst);
  label2 = getlabel(dst+i);
  if (label1 == label2) {
    dsti = dst+i;
    *dsti = curr
  }
  i++;
  label1 = getlabel(src);
  label2 = getlabel(src+i);
  if (label1 == label2) {
    srci = src+i;
    curr = *srci;
  }
}
```

Listing 5 contains an example of how our basic pointer arithmetic checks are performed (Section 4 contains a complete transformation of the copy function) on the program 4. In the loop in copy, we lookup the label of the memory that *dst* is pointing to and then compare it to the label of the memory location that *dst+i* would point to. Similar checks are done for *src*.

When main calls *copy(src1,dst1)*, the loop will stop at *src+7*. Because *src* is null-terminated all label checks will be correct. In the case of *copy(src2,dst2)*, the label of *dst2+32* will be different from the label of *dst2*. This means we have tried to access an out-of-bounds memory location. The next section describes how we handle these situations.

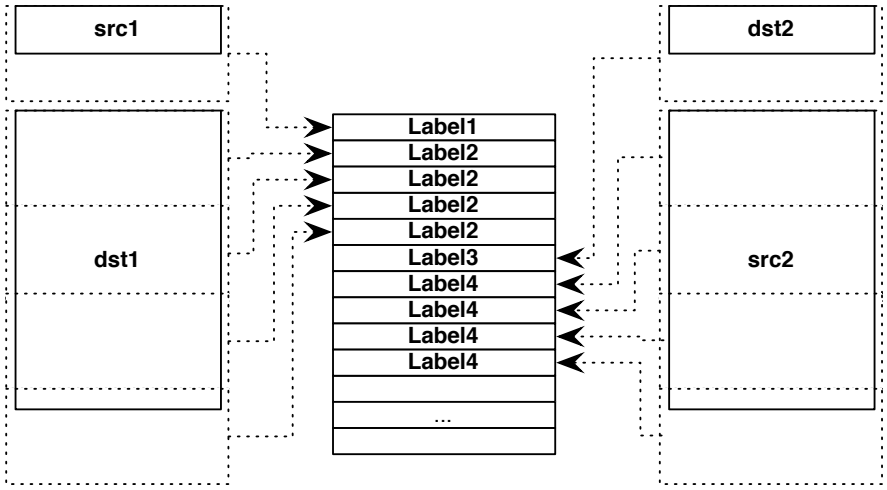

Figure 1: Memory layout of main in PAriCheck

3.2 Handling out-of-bounds pointers

When we detect an out-of-bounds access we use a technique similar to the ones used in [36]. We make the pointer point to an invalid memory location: on 32-bit versions of Linux, all memory above 3GB is considered kernel space. Accessing this memory from a regular program will cause the program to crash. So we set the pointer to point to a unique value above 3GB which we will call an OOB-pointer. We then allocate a block of memory (called the OOB-object) where we store the label of the base pointer, the resulting memory location that would have been the result of our arithmetic (which we call the intended pointer) and the OOB-pointer value. When the program performs an arithmetic operation using an OOB-pointer, we look up the correct OOB-object and then use the intended pointer as the value to perform the arithmetic operation on. Then we perform a bounds check again by comparing the stored label with the label of the result of the arithmetic operation. If the object goes inbounds again, then we set the pointer to the result of the operation, otherwise we create a new OOB-object which contains the updated information and set the pointer to the new OOB-pointer.

As a result, if the pointer is never accessed while out-of-bounds, the program continues to function correctly.

3.3 Uninitialized pointers

Uninitialized pointers are handled by initializing all uninitialized pointers to 0. Location 0 is assigned a unique label (label 1). Because a unique label is assigned to this memory location, any arithmetic performed on these pointers larger than the minimum object size will be detected as an out-of-bounds access. However for this to work, we must ensure that this memory location is never used: in Linux this can be done simply by ensure that */proc/sys/vm/ mmap_min_addr* is not 0 (the default value for kernel 2.6.24 is 65536). In operating systems where mapping page 0 is legitimate behavior, this can be achieved by ensuring that the countermeasure maps this page without any permissions: any access to this page will result in a segmentation violation.

3.4 Casting a pointer to an integer and back again

PAriCheck will protect against out-of-bounds violations caused by invalid pointer arithmetic and invalid array index accesses. These are the most common types of bounds violations that occur in a program. However PAriCheck will not protect programs that cast a pointer to an integer, then perform the calculation and later cast the resulting value back to a pointer. This is a limitation in our approach, which is also present in most other bounds checkers [38, 5, 24, 36, 18, 3], that stems from the fact that the arithmetic is no longer performed on a real pointer and as such will not be detected. While this is a limitation in our approach, the vast majority of buffer overflows occur as the result of invalid pointer arithmetic (e.g. an out-of-bounds *strcpy*) or invalid array index accesses.

3.5 Overflows in structures

A limitation in our approach, one that is also present in most other bounds checkers, is the fact that we do not protect pointers in a structure from other information in that structure. This limitation is a result of the C standard which specifically allows code to access the entire memory of a structure from a pointer to an element in the structure (e.g. to allow for instructions like memset(&strct,0,**sizeof**(strct));). This makes it impossible to check for these kinds of buffer overflows while remaining compatible with the C standard.

3.6 Bounds checking arrays

In C there is a difference between array types and pointer types. If an array is declared as array[size], then the use of array[index] means we are accessing an array via its index. While the array type can be used interchangeably with a pointer type in C, the two are handled differently by the compiler. Array types have a size associated with them during compilation, which means we can look up the array's size when accessing an array type via an index. As a result, bounds checking array types is a trivial operation: we can simply insert a check to make sure that the index used to access an element in the array is smaller than the size of

the array. This type of check is trivial and does not occur a significant amount of overhead and this is also the approach used for checking arrays in our countermeasure. If a pointer is used to access elements of the array, then our regular approach for dealing with pointers that is discussed in 3.1 is used.

Variable length arrays are translated into calls to *alloca* by the framework we use to implement our transformation, which means that pointer arithmetic will be performed and thus will be checked. However if they were handled as real variable length arrays, our checker would just add the variable denoting the size in the check instead of the compile-time size.

3.7 Unprotected code

Code that is linked to code that is protected by PAriCheck will work without issues. Any objects allocated in this code will of course not have been labeled and will thus not enjoy protection. Given that these objects have not been labeled, that also means they do not have to be aligned. When a label lookup is performed for these objects the label will simply be 0 as will the result of the arithmetic, unless the result of the arithmetic points into a protected object. If, when running protected code, an unprotected object overflows into a protected object, the resulting out-of-bounds access will also be detected because the label of the unprotected object (label 0) will differ from the label of the protected object.

4. PROTOTYPE IMPLEMENTATION

Our implementation has many facets. The most important code transformations that result in the actual checks being inserted whenever pointer arithmetic is performed, were done using CIL [33]. Ensuring that all stack-based objects are our minimum size and allocated on our alignment boundary, can be done by encapsulating them inside a struct. However, for expedience, our prototype does some changes to GCC to allow this. Finally we modified the malloc, realloc and calloc functions to ensure that objects are aligned correctly, are a multiple of our minimum size, and are assigned a label.

We place the labels in a sequential area of memory that is big enough to hold labels for all the regions.[2] In our prototype implementation, labels are 2 bytes large. This is a trade-off: an application which creates more than 2^{16} objects will create labels which are no longer unique and thus may allow an out-of -ounds object to point to an object with the same label. The implementation can easily be adapted to use long instead of short integers, allowing the program to create 2^{32} objects. This would, however, double the amount of memory that the labels take up in memory.

Because a label for every byte would waste too much memory, we have a default region size of 32 bytes. However, the region size can be chosen depending on the application. Our default region size ensures that we only need 1 label per 32 bytes. This means that the labels will take up only 6% of memory per object, while a label for every byte would have unacceptable overhead. It also means that every object will take up 32 bytes. For example programs with 16 byte regions will have a 12% label overhead, but if most objects are close to 16 bytes in size and we would have an internal fragmentation of 100% if we used a region size of 32 bytes. So for a program with many smaller or larger chunks it becomes more interesting to set the region size closer to the size of the most used objects (if this is known by the person compiling the application).

To find the correct entry of a memory location in the label area, we shift the address right by $log_2(regionsize)$ bits (i.e., by 5 bits for our default region size of 32 bytes), and then use the resulting value as an offset into the table. This allows for fast lookups of labels.

Listing 6 is a sample of the automatic transformation[3] of the code in Listing 1. This transformation demonstrates how our bounds checker performs its checks:

Listing 6: Transformed sample program

```
void copy(char *src, char *dst) {
  int i = 0; char curr;
  char *mem5; char *mem6;  char *mem7;
  char *dsti; char *srci;
  short int label1; short int label2;
  char *oobdst; char *oobsrc;

  mem5 = src + 0;
  curr = *mem5;
  while (curr) {
    if (dst < 3221225472) {
      dsti = dst + i;
      label1 = getlabel(dst);
    } else {
      oobdst = getoob(dst);
      label1 = getooblabel(dst);
      dsti = oobdst + i;
    }
    label2 = getlabel(dsti);
    if (label1 != label2)
      dsti = setoob(label1, dsti);
    mem6 = dsti;
    *mem6 = curr;
    i ++;
    if (src < 3221225472) {
      srco = src + i;
      label1 = getlabel(src);
    } else {
      oobsrc = getoob(src);
      label1 = getooblabel(src);
      srci = oobsrc + i;
    }
    label2 = getlabel(srci);
    if (label1 != label2)
      srci = setoob(label1, srci);
    mem7 = srci;
    curr = *mem7;
  }
}
```

The pointer arithmetic has been transformed as follows: our countermeasure checks if *dst*'s address is below 3GB. If it is, the label for *dst* is looked up and the arithmetic *dst* + *i* is performed. Then the label for the buffer is looked up. If

[2]This area of memory is virtual memory, which means that only pages where we actually store a label will use physical memory.

[3]Some minor editiorial changes were made: variable names have been changed to make the code more readable, some excessive curly brackets were removed, etc.

dst is above 3GB, then it went out-of-bounds earlier and we must look up the out-of-bounds value and look up the original label. We then perform the arithmetic *out_of_bounds_dst + i*. The resulting arithmetic and labels of either part of the if statement will be stored in *dsti* and *label1*, respectively. We then look up the label of this newly calculated address and compare it to the label in *label1*. If the labels match, the arithmetic was in bounds, if they do not, the arithmetic went out-of-bounds and we assign an out-of-bounds value to *dsti*, which will contain a unique ID (3GB + unique number), the labeling information and the original arithmetic. Afterwards, *dsti* is used as the result of our arithmetic in the operation we must perform. If the arithmetic would go out-of-bounds in this case then the **mem_6 = curr* statement will try to dereference an invalid memory location, causing the program to crash. Similar checks are performed for *src*.

5. EVALUATION

Performing extra checks comes at a cost: both in terms of performance and in terms of memory overhead. To evaluate how high the performance overhead of our bounds checker is, several benchmarks were run. All benchmarks were performed on a single machine: an Intel Core 2 Duo 3.16 Ghz with 8GB of RAM running Ubuntu 08.04 with kernel 2.6.24-24-server. All benchmarks were compiled with using gcc-4.3.0-20070803 with the O2 parameter and were linked to dlmalloc 2.7.2 (the original one for the baseline benchmark and our modified version for our bounds checker)[4]. The use of the O2 parameter means that standard performance optimizations are applied to both the countermeasure and the rest of programs.

We performed two sets of benchmarks: the SPEC CPU2000 integer benchmarks and the Olden benchmarks. The SPEC CPU2000 integer benchmarks provide us with a measurement of very CPU intensive programs, while the Olden benchmarks allow us to measure applications that are very memory intensive. The SPEC CPU2000 benchmarks were run, as is, with the default parameters which allowed for a reportable run. For the Olden benchmarks, a number of options were chosen that would let the program run for a significant amount of time and would let it use a significant amount of memory. The first column of Table 1 contains the program names and, for the Olden benchmarks, the parameters that we used to achieve these results. For editing purposes we abbreviated the amounts: in the table a K is used to denote 1000 and M is used to denote 1000000. As mentioned in Section 4, we use a default object alignment and object size of 32 bytes for our countermeasure.

To perform our transformations we use the dosimpleMem option of the CIL infrastructure. This option will simplify all memory operations and makes our transformation easier. We then apply our bounds checker. However using this option has a slight impact on performance. Since we want to measure the overhead of only our transformations rather than other transformations we supply benchmarks for programs compiled regularly with GCC, compiled with CIL with dosimpleMem and finally compiled with CIL with our bounds checker.

[4]This memory allocator was chosen because it is easy to modify and because it is used as the basis for the default memory allocator in Linux.

In Section 5.1 we discuss our performance overhead, while section 5.2 discusses the memory overhead associated with PAriCheck.

5.1 Performance overhead

Table 1 contains the performance measurements for our countermeasure for both the SPEC CPU2000 integer benchmarks and the Olden benchmarks. Columns 2 to 5 in Table 1 contain the runtime in seconds of our benchmarks. Column 2 is simply the original program compiled with the default gcc-4.3.0-20070803. While the third column is the benchmark run with CIL with the option dosimplemem and the default memory allocator. As can be seen, in most cases the performance difference between CIL and GCC is negligible. Column 4 is essentially the same benchmark as in Column 3, except CIL was linked with a memory allocator which has 32 bytes as a minimum chunk size. By providing both measurements we can show the impact of using our modified memory allocator and the impact of the checks we added. Column 5 contains the runtime of the benchmark compiled with our extra checks. Since we aim at measuring the overhead of our transformation rather than the impact of various CIL transformations, we compare the CIL columns with PAriCheck. Column 6 contains the relative overhead of Column 3 over Column 5, i.e. the sixth column in Table 1 is the relative performance overhead of the runtime of PAriCheck compared to the original program compiled with CIL and the default memory allocator. Column seven contains the same as column 6, but over Columns 4 and 5, i.e. PariCheck compared to the original program compiled with CIL and the memory allocator with 32-byte size.

There are some surprising results, when running these benchmarks: aligning *tsp* to 32-bytes improves its performance significantly. This is an important reason for providing measurements of CIL with a memory allocator with the default alignment (8 bytes) and the 32-byte alignment used in our countermeasure. By providing both measurements we can clearly assess the impact of our transformation even if we would use a modified memory allocator in normal cases. Another significant difference is in the memory overhead for applications which use many small chunks, like *twolf*, where the relative memory overhead because of the new chunk size is significant.

The average overhead of our countermeasure is 49% for the SPEC CPU2000 integer benchmarks and 4.5% (equal alignment) to 8% (when using default alignment) for the Olden benchmarks. This makes our approach the fastest way of performing bounds checks on C applications. The closest other approach, baggy bounds checking [3] has an overhead of 60% on the SPEC CPU2000 benchmarks and 5.5% on the Olden benchmarks, where some of the speed improvement relies on using a faster memory allocator.

However, some overheads, like the overhead of *vpr*, are relatively high, although better (or comparable in the case of baggy bounds checking) than existing bounds checkers that do not rely on static analysis for optimizations. The main reason for the high overheads of *vpr* is the use of out-of-bounds pointers in many calculations, which means we have to store and look up out bounds pointers.

For a production-level implementation of our approach, these overheads can be further improved upon by performing compile-time optimizations on our checks: removing redundant checks (e.g. if an arithmetic operation like src[i]

Table 1: Performance benchmarks: runtime in seconds

SPEC CPU2000 Integer benchmarks						
Program	GCC	CIL	CIL32	PAC	CIL/PAC	CIL32/PAC
gzip	89	90,3	90,3	195	115,95%	115,95%
vpr	68,3	68,2	68	230	237,24%	238,24%
mcf	37,9	37,1	37,8	48,1	29,65%	27,25%
crafty	46,1	48,5	48,5	65,5	35,05%	35,05%
parser	958	946	946	1030	8,88%	8,88%
gap	44,9	45,3	45,4	126	178,15%	177,53%
bzip2	70,6	70,7	70,7	173	144,7%	144,7%
twolf	101	104	108	237	127,88%	119,44%
Average	177	176,3	176,8	263,1	49,23%	48,81%
Olden Benchmarks						
Program	GCC	CIL	CIL32	PAC	CIL/PAC	CIL32/PAC
bh (410K,32)	76,1	76,9	78,2	80,2	4,29%	2,56%
bisort (16M,200M)	21,5	21,6	23	24,1	11,57%	4,78%
em3d (5K,3.5K,2K,100)	66	66	66	67,7	2,58%	2,58%
health (10,90,20)	12,5	12,5	19,6	21,4	71,2%	9,18%
mst (10K)	21,8	22,2	22	24,6	10,81%	11,82%
treedadd (26,1)	3,5	3,5	5,3	5,5	57,14%	3,77%
tsp (10M)	24,3	24,5	20,7	21,4	-12,65%	3,38%
voniroi (4M)	14,5	14,5	15,3	16,6	14,48%	8,5%
Average	30	30,2	31,3	32,7	8,28%	4,47%

has been checked and we can be sure that if neither i nor src change then the following check can be removed), moving checks outside of loops, etc. [11].

5.2 Memory overhead

Table 2 contains the memory usage in megabytes of the programs compiled with CIL with the default memory allocator, compiled with CIL with the memory allocator that aligns to 32-bytes and compiled with our bounds checker in columns 2, 3 and 4 respectively. Because the memory overhead of compiling with GCC and with CIL is the same, we have omitted the GCC column. The fifth column shows the memory overhead of using our countermeasure compared to the memory allocator with default alignment and column 6 shows the overhead of our countermeasure when using a memory allocator with the same alignment. These overheads were measured by adding a call to *getchar()* at the end of the program and then examining the *VmHWM* entry in $/proc/ < pid > /status$. This entry contains the peak resident set size which is the maximum amount of RAM the program has used during its lifetime. Since our tests were run with swap turned off this is equal to the actual maximum memory usage used by our program.

These results give us an average overhead of 9.5% when using the same type of memory allocator for SPEC CPU2000 benchmarks and 5.19% for the Olden benchmarks. The overhead is about the same when using the default alignment for the SPEC CPU2000 benchmarks, but is higher, 22% when using the default alignment for the Olden benchmarks. This is due to the fact that some applications in the Olden benchmarks allocate many objects which have a smaller alignment than our default alignment, resulting in a waste of memory, especially for the applications *bisort*, *health* and *tsp*. By changing our default object size for these applications to the same as the original memory allocator (8 bytes), the memory overhead drops by 20% for each application, while the performance overheads also reduce to 9.7% and 28.8% in the cases of *bisort* and *health*. In the case of *tsp*, where the overhead was negative due to alignment, it did not improve, but went up to 13%. This means that, from a memory point of view, it is advantageous for a developer protecting his application with PAriCheck to use region sizes close to the most often used allocation unit.

6. RELATED WORK

Many countermeasures have been designed to protect against code injection attacks. In Section 6.1 we compare our approach to other bounds checkers in detail. Then, in Section 6.2, we briefly highlight the differences between our approach and other approaches that protect programs against attacks on memory error vulnerabilities.

6.1 Bounds checkers

RTCC [38] is a modification of the Portable C Compiler that adds run-time array subscript and pointer bounds checking. To implement this, pointers are represented by three times their normal size, containing the current value of the pointer and the memory addresses of its lower and upper bounds. However, by changing the pointer representation code can no longer be linked to existing code and the compiler can no longer implicitly cast an integer to a pointer.

Safe C [5] is a bounds checking countermeasure for C. It defines a kind of safe pointer that contains the following attributes: value, pointer base, size, storage class (heap, local, global) and capability (forever, never). The value attribute is the actual pointer, the base and size attributes are used for spatial check while the storage class and capability attributes are used for temporal checks. As with RTCC the pointer representation is changed, resulting in an incompatibility with existing code.

Table 2: Memory benchmarks: memory usage in megabyte

SPEC CPU2000 Integer benchmarks					
Program	CIL	CIL32	PAC	CIL/PAC	CIL32/PAC
gzip	180,8	180,8	192,2	6,31%	6,31%
vpr	20,7	20,9	22,5	8,7%	7,66%
mcf	77,4	77,4	101,4	31,01%	31,01%
crafty	2,5	2,5	3	20%	20%
parser	25,5	25,5	27,5	7,84%	7,84%
gap	193	193	205,8	6,63%	6,63%
bzip2	185,1	185,1	196,8	6,32%	6,32%
twolf	4	5,3	6,8	70%	28,3%
Average	86,1	86,3	94,5	9,76%	9,5%
Olden Benchmarks					
Program	CIL	CIL32	PAC	CIL/PAC	CIL32/PAC
bh (410K,32)	70,5	81,4	86,4	22,55%	6,14%
bisort (16M,200M)	128,4	256,4	272,4	112,15%	6,24%
em3d (5K,3.5K,2K,100)	535,1	535,4	569,2	6,37%	6,31%
health (10,90,20)	587,6	987,6	1049	78,57%	6,25%
mst (10K)	1250	1250,5	1329	6,32%	6,25%
treedadd (26,1)	1024	1024,4	1088	6,25%	6,25%
tsp (10M)	640,5	1024,5	1089	69,95%	6,25%
voniroi (4M)	2051	2147	2204	7,44%	2,64%
Average	785,9	913,4	960,8	22,25%	5,19%

Jones and Kelly [24] developed a bounds checking technique that was used as the basis for the rest of the work described in this section. This bounds checker does not change the way pointers are represented in a program. This allowed bounds checked programs to be linked with existing compiled code. This suffers from very high performance overhead (up to 11x slower than the original program).

CRED [36] improves on the work described in [24] by providing support for arrays going out-of-bounds. Our handling of out-of-bounds pointers is based on the work developed in this paper. They also further improve on the technique by providing a few optimizations, one specific optimization which increases speed is that only character pointers and character arrays are checked. Checking this reduced set of pointers improves performance, however this approach still suffers from a high overhead (up to 11x when protecting all arrays) or does not protect programs against overflows involving non-character arrays or pointers (up to 2x when only protecting strings). Our approach protects all array types and all dynamically allocated memory at an overhead that is faster than the overheads reported by CRED when only protecting strings.

Dhurjati and Adve [18] discuss an efficient bounds checker that has a significantly lower overhead than previous bounds checkers but which is still twice as high compared to our overhead. Their technique rests on the use of automatic pool allocation for a memory allocation, while our approach does not require such a significant change of allocator. To be able to use the pool allocation, global static pointer analysis is required, which introduces scalability and modularity issues. They further improve on efficiency by applying compile-time optimizations to improve their performance. These techniques could also be applied to our bounds checker. We decided to focus on demonstrating that a purely dynamic approach can be made more efficient without changing the underlying architecture significantly or even applying compile-time optimizations.

Clause et al. [14] developed a dynamic tainter that checks for both spatial and temporal errors for dynamically allocated memory. It works by assigning taint marks to objects and assigning the same taint mark to pointers to these objects. The taint marks for pointers are then propagated and transformed through the program whenever an operation (such as arithmetic) on a pointer occurs. When the pointer is dereferenced, the taint mark for the pointer is compared to the taint mark of the object. If the taint marks differ then a memory error has occurred. The approach discussed by Clause at al. works on binaries rather than source code, but requires hardware assistance to be able to efficiently check and propagate the taint marks.

Baggy bounds checking [3] was developed concurrently to and independently from our approach. It is similar in that it pads objects to the next significant byte to achieve faster bounds look ups. This approach is 10% slower than our approach on the SPEC CPU2000 benchmarks, with comparable results for the Olden benchmarks. A significant drawback of the approach is that half the address space cannot be used because out of bounds pointers have their first bit set, while in bounds pointers have their first bit clear. This means that in 32-bit applications, only 2GB of memory can be used, where as in our approach 3GB can be used because Linux reserves 1GB for kernel memory. If this reserved area were smaller, our approach would still work, we would simply need to map an area of virtual memory to which we could point out-of-bounds pointers that would result in the programming crash if dereferenced. For most applications, an area of virtual memory of 16 pages (64 Kb) that are mapped without permissions would suffice.

SoftBound [30] is a bounds checker, that stores the base and bounds for each pointer in the program in a separate

memory location. Whenever a pointer is dereferenced, the base and bounds are looked up to ensure that the pointer is in bounds. As such, it does not need to check pointer arithmetic, because out-of-bounds accesses will be detected when the pointer is dereferenced. It has an average overhead of 67% for the SPEC CPU2000 benchmarks, which is about 17% higher than our approach. However, it does have support for preventing overflows within structures if requested via a command line parameter.

Fail-safe C [34] is a compiler that implements a memory safe version of the ANSI C standard. It does this using a number of techniques: fat pointers and integers (because pointers can be cast to integers and back again) for bounds checking, keeping track of runtime type information, garbage collection to prevent dangling pointers, etc. The overhead of this approach is, however, significant. The programs in the ByteMARK benchmark were slowed down by two to four times on average.

6.2 Alternative approaches

Many alternative approaches exist that try and protect against buffer overflow attacks. In this section we will briefly discuss the most important types of countermeasures. A more extensive discussion can be found in [44, 20].

6.2.1 Safe languages

Safe languages are languages where it is generally not possible for any known code injection vulnerability to exist as the language constructs prevent them from occurring. A number of safe languages are available that will prevent these kinds of implementation vulnerabilities entirely. There are safe languages [23, 22, 32, 29, 19, 27] that remain as close to C or C++ as possible, these are generally referred to as safe dialects of C. While some safe languages [15, 43] try to stay more compatible with existing C programs, use of these languages may not always be practical for existing applications.

6.2.2 Probabilistic countermeasures

Many countermeasures make use of randomness when protecting against attacks. Canary-based countermeasures [17, 21, 28, 35] use a secret random number that is stored before an important memory location: if the random number has changed after some operations have been performed, then an attack has been detected. Memory-obfuscation countermeasures [16, 8] encrypt (usually with XOR) important memory locations or other information using random numbers. Memory layout randomizers [40, 7, 42, 9] randomize the layout of memory: by loading the stack and heap at random addresses and by placing random gaps between objects. Instruction set randomizers [6, 26] encrypt the instructions while in memory and will decrypt them before execution.

While these approaches are often efficient, they rely on keeping memory locations secret. However, programs that contain buffer overflows could also contain "buffer overreads" (e.g. a string which is copied via *strncpy* but not explicitly null-terminated could leak information) or other vulnerabilities like format string vulnerabilities, which allow attackers to print out memory locations. Such memory leaking vulnerabilities could allow attackers to bypass this type of countermeasure. Another drawback of these countermeasures is that, while they can be effective against remote attackers, they are easier to bypass locally, because attackers could attempt brute force attacks on the secrets[39].

6.2.3 Separation and replication of information

Countermeasures that rely on separation or replication of information will try to replicate valuable control-flow information [41, 13] or will separate this information from regular data [45, 46]. This makes it harder for an attacker to overwrite this information using an overflow. Some countermeasures will simply copy the return address from the stack to a separate stack and will compare it to or replace the return addresses on the regular stack before returning from a function. These countermeasures are easily bypassed using indirect pointer overwriting where an attacker overwrites a different memory location instead of the return address by using a pointer on the stack. More advanced techniques try to separate all control-flow data (like return addresses and pointers) from regular data, making it harder for an attacker to use an overflow to overwrite this type of data [46].

While these techniques can efficiently protect against buffer overflows that try to overwrite control-flow information, they do not protect against attacks where an attacker controls an integer that is used as an offset from a pointer, nor do they protect against non-control-data attacks.

6.2.4 Runtime enforcement of static analysis results

In this section we describe two countermeasures that provide runtime enforcement of results of static analysis.

Control-flow integrity [1] determines a program's control flow graph beforehand and ensures that the program adheres to it. It does this by assigning a unique ID to each possible control flow destination of a control flow transfer. Before transferring control flow to such a destination, the ID of the destination is compared to the expected ID, and if they are equal, the program proceeds as normal. This approach, while strong and in the same efficiency range as our approach, does not protect against non-control data attacks.

WIT [2] discusses a very efficient technique to check whether instructions write to valid memory location. Their technique is based on static analysis that does a points-to analysis of the application. This analysis is then used to assign colors to memory locations and instructions. Each instruction has the same color as the objects it writes to. Then runtime checks are added to ensure that these colors are the same. This prevents instructions from writing to memory that they cannot normally write to. This technique depends on a static points-to analysis, which can result in false negatives where an instruction is determined to be safe when it is not or it can assign an instruction or object a color that allows an unsafe instruction access to the object. While our technique is similar in that we check labels (i.e. colors), ours is purely dynamic which means we always have full information on the memory that an object inhabits and can assign labels correctly, whereas static alias analysis could confuse objects, allowing instructions access to multiple objects. The technique is also different in that it is applied to instructions that write to memory, which will not prevent out-of-bounds reads.

7. CONCLUSION

The countermeasure described in this paper is an efficient dynamic bounds checker that can be used in high-security production systems with little effort and without requiring major architectural changes. PAriCheck protects against buffer overflow attacks that aim to overwrite control data or non-control data, by ensuring that the results of pointer arithmetic always points within the bounds of the base object. This is done by assigning a unique label for each object and associating this label with the entire memory area that the object inhabits. We then compare the label of the base address of the arithmetic operation with the label of the resulting arithmetic. If they differ, an overflow has occurred. PAriCheck also retains full compatibility with existing C applications, even ones that do not comply with the C standard and calculate out-of-bounds values. In our benchmarks, PAriCheck has an average runtime overhead of 49% for the SPEC CPU2000 benchmarks, which is a very low overhead for this type of countermeasure. If we further combine our approach with static analysis techniques, including loop optimization and preventing redundant checks, performance overhead can be reduced even more.

Acknowledgements

This research is partially funded by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, and by the Research Fund K.U.Leuven. Sekar's work was supported in part by ONR grant N000140710928 and NSF grants CNS-0627687 and CNS-0831298. Cavallaro's work was supported in part by an NSF grant CNS-0627687.

8. REFERENCES

[1] Martin Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM Conference on Computer and Communications Security*, pages 340–353, Alexandria, VA, November 2005.

[2] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing memory error exploits with WIT. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy*, Oakland, CA, May 2008.

[3] Periklis Akritidis, Manuel Costa, Miguel Castro, and Steven Hand. Baggy bounds checking: An efficient and backwards-compatible defense against out-of-bounds errors. In *Proceedings of the 18th USENIX Security Symposium*, Montreal, QC, August 2009.

[4] Aleph1. Smashing the stack for fun and profit. *Phrack*, 49, 1996.

[5] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi. Efficient detection of all pointer and array access errors. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 290–301, Orlando, FL, June 1994.

[6] Elena Gabriela Barrantes, David H. Ackley, Stephanie Forrest, Trek S. Palmer, Darko Stefanović, and Dino Dai Zovi. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, pages 281–289, Washington, D.C., October 2003.

[7] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium*, pages 105–120, Washington, D.C., August 2003.

[8] Sandeep Bhatkar and R. Sekar. Data space randomization. In *Proceedings of the 5th Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, volume 5137 of *Lecture Notes in Computer Science*, Paris, France, July 2008.

[9] Sandeep Bhatkar, R. Sekar, and Daniel C. DuVarney. Efficient techniques for comprehensive protection from memory error exploits. In *14th USENIX Security Symposium*, Baltimore, MD, August 2005.

[10] blexim. Basic integer overflows. *Phrack*, 60, December 2002.

[11] Ratislav Bodik, Rajiv Gupta, and Vivek Sarkar. ABCD: Eliminating array-bounds checks on demand. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 321–333, Vancouver, BC, June 2000.

[12] Shuo Chen, Jun Xu, Emre C. Sezer, Prachi Gauriar, and Ravishankar K. Iyer. Non-control-data attacks are realistic threats. In *Proceedings of the 14th USENIX Security Symposium*, Baltimore, MD, August 2005.

[13] T. Chiueh and Fu-Hau Hsu. RAD: A compile-time solution to buffer overflow attacks. In *Proceedings of the 21st International Conference on Distributed Computing Systems*, pages 409–420, Phoenix, AZ, April 2001.

[14] James Clause, Ioannis Doudalis, Alessandro Orso, and Milos Prvulovic. Effective memory protection using dynamic tainting. In *Proceedings of the 22nd IEEE and ACM International Conference on Automated Software Engineering (ASE 2007)*, pages 284–292, Atlanta, GA, November 2007.

[15] Jeremy Condit, Matthew Harren, Scott McPeak, George C. Necula, and Westley Weimer. CCured in the real world. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 232–244, San Diego, CA, 2003.

[16] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointGuard: protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*, pages 91–104, Washington, D.C., August 2003.

[17] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, pages 63–78, San Antonio, TX, January 1998.

[18] Dinakar Dhurjati and Vikram Adve. Backwards-compatible array bounds checking for C with very low overhead. In *Proceeding of the 28th international conference on Software engineering*, pages 162–171, Shanghai, China, 2006.

[19] Dinakar Dhurjati, Sumant Kowshik, Vikram Adve, and Chris Lattner. Memory safety without runtime checks or garbage collection. In *Proceedings of the 2003 Conference on Language, Compiler, and Tool*

Support for Embedded Systems, pages 69–80, San Diego, CA, June 2003.

[20] Úlfar Erlingsson, Yves Younan, and Frank Piessens. Low-level software security by example. In *Handbook of Information and Communication Security*. Springer, 2010.

[21] Hiroaki Etoh and Kunikazu Yoda. Protecting from stack-smashing attacks. Technical report, IBM Research Tokyo, June 2000.

[22] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 282–293, Berlin, Germany, June 2002.

[23] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, pages 275–288, Monterey, CA, June 2002.

[24] Richard W. M. Jones and Paul H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proceedings of the 3rd International Workshop on Automatic Debugging*, pages 13–26, Linköping, Sweden, 1997.

[25] JTC 1/SC 22/WG 14. ISO/IEC 9899:1999: Programming languages – C. Technical report, International Organization for Standards, 1999.

[26] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security*, pages 272–280, Washington, D.C., October 2003.

[27] Sumant Kowshik, Dinakar Dhurjati, and Vikram Adve. Ensuring code safety without runtime checks for real-time control systems. In *Proceedings of the International Conference on Compilers Architecture and Synthesis for Embedded Systems*, pages 288–297, Grenoble, France, October 2002.

[28] Andreas Krennmair. ContraPolice: a libc extension for protecting applications from heap-smashing attacks, November 2003.

[29] James R. Larus, Thomas Ball, Manuvir Das, Robert DeLine, Manuel Fähndrich, Jon Pincus, Sriram K. Rajamani, and Ramanathan Venkatapathy. Righting software. *IEEE Software*, 21(3):92–100, May 2004.

[30] Santosh Nagarakatte, Jianzhou Zhao, Milo M. K. Martin, and Steve Zdancewic. Softbound: Highly compatible and complete spatial memory safety for c. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 245–258, Dublin, Ireland, June 2009.

[31] National Institute of Standards and Technology. National vulnerability database statistics. `http://nvd.nist.gov/statistics.cfm`.

[32] George Necula, Scott McPeak, and Westley Weimer. CCured: Type-safe retrofitting of legacy code. In *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 128–139, Portland, OR, January 2002.

[33] George C. Necula, Scott McPeak, S. P. Rahul, and Westley Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proceedings of the Conference on Compiler Construction (CC'02)*, volume 2304 of *Lecture Notes in Computer Science*, pages 213–228, Grenoble, France, March 2002.

[34] Yutaka Oiwa. Implementation of the memory-safe full ansi-c compiler. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 259–269, Dublin, Ireland, June 2009.

[35] William Robertson, Christopher Kruegel, Darren Mutz, and Frederik Valeur. Run-time detection of heap-based overflows. In *Proceedings of the 17th Large Installation Systems Administrators Conference*, pages 51–60, San Diego, CA, October 2003.

[36] Olatunji Ruwase and Monica S. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, San Diego, CA, February 2004.

[37] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561, Washington, D.C.,, October 2007.

[38] Joseph L. Steffen. Adding run-time checking to the portable C compiler. *Software: Practice and Experience*, 22(4):305–316, April 1992.

[39] Raoul Strackx, Yves Younan, Pieter Philippaerts, Frank Piessens, Sven Lachmund, and Thomas Walter. Breaking the memory secrecy assumption. In *Proceedings of the Second European Workshop on System Security*, Nuremburg, Germany, 2009.

[40] The PaX Team. Documentation for the PaX project.

[41] Vendicator. Documentation for stackshield.

[42] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Transparent runtime randomization for security. In *22nd International Symposium on Reliable Distributed Systems (SRDS'03)*, pages 260–269, Florence, Italy, October 2003. IEEE Press.

[43] Wei Xu, Daniel C. DuVarney, and R. Sekar. An Efficient and Backwards-Compatible Transformation to Ensure Memory Safety of C Programs. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 117–126, Newport Beach, CA, October 2004.

[44] Yves Younan, Wouter Joosen, and Frank Piessens. Code injection in C and C++ : A survey of vulnerabilities and countermeasures. Technical Report CW386, Departement Computerwetenschappen, Katholieke Universiteit Leuven, July 2004.

[45] Yves Younan, Wouter Joosen, and Frank Piessens. Efficient protection against heap-based buffer overflows without resorting to magic. In *Proceedings of the International Conference on Information and Communication Security*, Raleigh, NC, December 2006.

[46] Yves Younan, Davide Pozza, Frank Piessens, and Wouter Joosen. Extended protection against stack smashing attacks without performance loss. In *Proceedings of the Twenty-Second Annual Computer Security Applications Conference*, Miami, FL, December 2006.