SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2006; **36**:971–998
Published online 2 May 2006 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.720

Binary rewriting and call interception for efficient runtime protection against buffer overflows

Kumar Avijit[1], Prateek Gupta[2] and Deepak Gupta[3,*,†]

[1]*Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, U.S.A.*
[2]*Department of Computer Sciences, University of Texas, Austin, U.S.A.*
[3]*Department of Computer Science and Engineering, Indian Institute of Technology, Kanpur 208016 UP, India*

SUMMARY

Buffer overflow vulnerabilities are one of the most commonly and widely exploited security vulnerabilities in programs. Most existing solutions for avoiding buffer overflows are either inadequate, inefficient or incompatible with existing code. In this paper, we present a novel approach for transparent and efficient runtime protection against buffer overflows. The approach is implemented by two tools: Type Information Extractor and Depositor (TIED) and LibsafePlus. TIED is first used on a binary executable or shared library file to extract type information from the debugging information inserted in the file by the compiler and reinsert it in the file as a data structure available at runtime. LibsafePlus is a shared library that is preloaded when the program is run. LibsafePlus intercepts unsafe C library calls such as `strcpy` and uses the type information made available by TIED at runtime to determine whether it would be 'safe' to carry out the operation. With our simple design we are able to protect most applications with a performance overhead of less than 10%. Copyright © 2006 John Wiley & Sons, Ltd.

Received 19 July 2004; Revised 21 July 2005; Accepted 7 September 2005

KEY WORDS: buffer overflow attacks; binary rewriting; dynamic linking; function call interception

INTRODUCTION

Since the advent of the Morris worm in 1988, buffer overflows have constituted a major threat to the security of computer systems. A buffer overflow exploitation is very powerful and is capable of

*Correspondence to: Deepak Gupta, Department of Computer Science and Engineering, Indian Institute of Technology, Kanpur 208016 UP, India.
†E-mail: deepak@cse.iitk.ac.in

Contract/grant sponsor: Prabhu Goel Research Center for Computer and Internet Security

rendering a computer system in total control of the attacker. As reported by the computer emergency response team (CERT), 11 out of 20 most widely exploited vulnerabilities in 2003 have been found to be buffer overflow vulnerabilities [1]. More than half of CERT advisories for 2003 [2] reported buffer overflow vulnerabilities. It is thus a major concern of the computing community to provide a practical and efficient solution to the problem of buffer overflows.

In a buffer overflow attack, the attacker's aim is to gain access to a system by changing the control flow of a program. In most variants of buffer overflow attacks, the program is made to execute code that has been carefully crafted by the attacker. The code can be inserted in the address space of the program using any legitimate form of input. The attacker then corrupts a code pointer in the address space by overflowing a buffer and makes it point to the injected code. When the program later dereferences this code pointer, it jumps to the attacker's code. Variants of the buffer overflow attacks, such as return-into-libc attacks, do not require the attacker to inject new code. In such attacks, execution flow is diverted to an existing function, such as the `system` function of the C library. The attacker simply crafts the arguments to this function and diverts the control flow by overwriting pointers. Such buffer overflows occur mainly due to the lack of bounds checking in C library functions and carelessness on the programmer's part. For example, the use of `strcpy()` in a program without ensuring that the destination buffer is at least as large as the source string is apparently a common practice among many C programmers.

Buffer overflow attacks come in various forms. The simplest and also the most widely exploited form of attack changes the control flow of the program by overflowing some buffer on the stack so that the return address or the saved frame pointer is modified. This is commonly called the 'stack smashing attack' [3]. Other more complex forms of attacks may not change the return address but attempt to change the program control flow by corrupting some other code pointers (such as function pointers, global offset table (GOT) entries, longjmp buffers, etc.) by overflowing a buffer that may be local, global or dynamically allocated. In almost all the cases, buffer overflow attacks are facilitated due to the storage of control information in-band with user data, such as return addresses and frame pointers on the program stack, heap management data structures adjacent to the dynamically allocated chunks, etc. Many common forms of buffer overflow attacks are described in [4].

Due to the huge amount of existing C code that lacks bounds checking, an efficient runtime solution is needed to protect the code from buffer overflows. Other solutions which have developed over the years such as manual/automatic auditing of the code, static analysis of programs, etc., are mostly incomplete as they do not prevent all attacks. A runtime solution is required because certain type of information is not available statically. For example, information about dynamically allocated buffers is available only at runtime. However, most current runtime solutions are unacceptable because they either do not protect against all forms of buffer overflow attacks, break existing code or impose too high an overhead to be successfully used with common applications.

In this paper, we present a simple yet robust solution to guard against buffer overflows on local, global and dynamically allocated variables arising due to the use of unsafe C library functions. The solution is a transparent runtime approach to prevent such attacks, and consists of two tools: Type Information Extractor and Depositor (TIED) and LibsafePlus. LibsafePlus is a dynamically loadable library and is an extension to Libsafe [5]. LibsafePlus contains wrapper functions for unsafe C library functions such as `strcpy`. A wrapper function determines the source and target buffer sizes and performs the required operation only if it would not result in an overflow. To enable runtime size checking we need to have additional type information about all buffers in the program.

DOI: 10.1002/spe

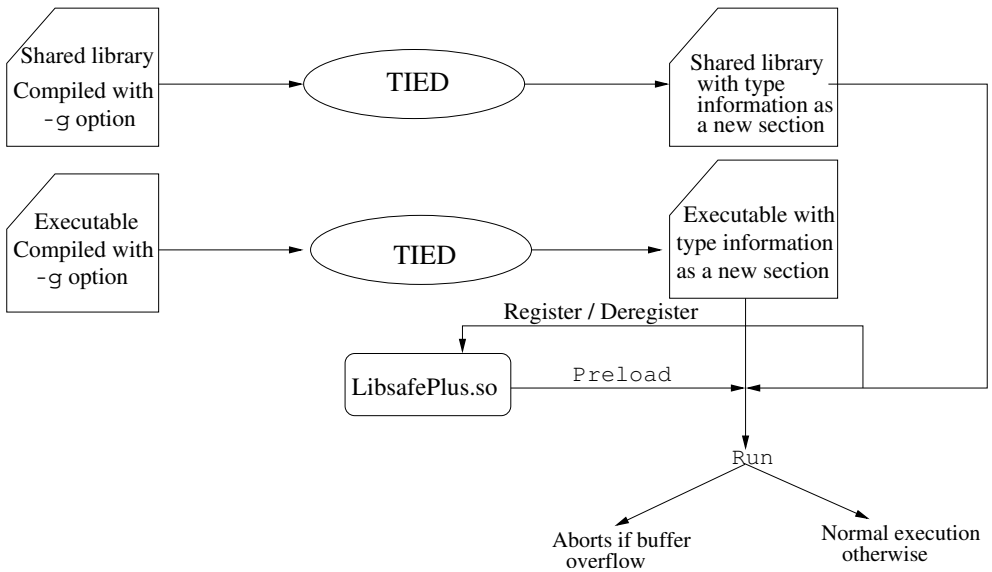


Figure 1. Rewriting of the executable and shared libraries by TIED and runtime range checking by LibsafePlus.

This is done by compiling the target program as well as the shared libraries which are used by the program with the `-g` debugging option. TIED is a binary rewriting tool that can be used for rewriting both executive linkable format (ELF) executables and shared libraries. TIED extracts the debugging information from the program binary and shared libraries, and then augments them with an additional data structure containing the size information for all buffers in the program. This information is utilized by LibsafePlus to range check buffers at runtime. To keep track of the sizes of dynamically allocated buffers, LibsafePlus intercepts calls to the `malloc` family of functions. Our tools thus require neither access to the source code (if it was compiled with the `-g` option) nor any modifications to the compiler, and are completely compatible with existing C code. The tools have been found to be effective against all forms of attacks and impose a low runtime performance overhead of less than 10% for most applications.

APPROACH OVERVIEW

The steps in the protection of a program using TIED and LibsafePlus are shown in Figure 1. The key idea here is to augment the executable with information about the locations and sizes of character buffers.

To this end, the program source and any shared libraries that the program uses should be compiled with the `-g` option that directs the compiler to dump debugging information, including information about sizes, addresses and types of all variables in the program, in the generated binary.

DOI: 10.1002/spe

The next step is to use TIED to rewrite the executable and the libraries with the required information as an additional data structure in the form of a separate section available at runtime. LibsafePlus is implemented as a dynamically loadable library that must be preloaded for a process to be protected. To enable range checking, LibsafePlus provides wrapper functions for unsafe C library functions. Each such wrapper function checks the bounds of the destination buffer before performing the actual operation. For dynamically allocated buffers, LibsafePlus maintains an additional runtime data structure that stores information about the locations and sizes of all dynamically allocated buffers. Because all the dynamically linked libraries are also modified using TIED, LibsafePlus has complete knowledge about sizes of all the buffers in the program image. In contrast to many other approaches which are mainly compiler extensions, LibsafePlus does not require source code access if the program is compiled with the `-g` option and is not statically linked with the C library.

LibsafePlus is implemented as an extension to Libsafe [5]. Libsafe is also a dynamically loadable library which provides wrapper functions for unsafe C library functions such as `strcpy()`. However, Libsafe only protects against stack smashing attacks. Even for stack variables, Libsafe assumes a safe upper bound on the size of a buffer instead of determining its exact size. Therefore, it is possible for the attacker to overwrite variables in the program that are next to the buffer in memory. Unlike Libsafe, our tools enable precise range checking of all local, global and dynamically allocated buffers. They have been tested extensively and have been found to be effective against buffer overruns on all such buffers that occur due to use of unsafe C library functions. Our tools successfully prevented all the 20 different overflow attacks in the testbed developed by Wilander and Kamkar [6] to test tools for their capability to prevent buffer overflow attacks, while the original Libsafe could detect only 6 of the 20 attacks.

In the following sections, we describe in detail the design of Libsafe and our extensions to it. We first describe the protection mechanism used by Libsafe and then show how LibsafePlus extends the basic protection mechanism to prevent overflows to all local, global and dynamically allocated buffers occurring due to the use of unsafe C library functions.

Runtime range checking by Libsafe

The goal of Libsafe is to prevent corruption of the return addresses and saved frame pointers on the stack in the event of a stack buffer overflow. To ensure that the frame pointers and the return addresses are never overwritten, Libsafe assumes a safe upper bound on the size of stack buffers as it does not possess sufficient information to determine their exact sizes. The underlying principle is that a buffer cannot extend beyond the stack frame within which it is allocated. Thus the maximum size of a buffer is the difference between the starting address of the buffer and the frame pointer for the corresponding stack frame. To determine the frame corresponding to a stack buffer, the topmost stack frame pointer is retrieved and the frame pointers are traversed on the stack until the frame containing the buffer is discovered. Since Libsafe prevents overwrites beyond the stack frame boundaries, it protects the function arguments, in addition to the return address and saved frame pointer, from being overwritten.

Based on the above design, Libsafe is implemented as a dynamically loadable library that provides wrapper functions for unsafe C functions such as `strcpy()`. The purpose of a wrapper function is to determine the size of the destination buffer and check whether the destination buffer is at least as large as the source string. If the check fails, the program is terminated. Otherwise, the wrapper function simply calls the original C library function.

In addition to protecting from buffer overflows, Libsafe also provides a mechanism to prevent format string attacks. Libsafe provides a wrapper function for `_IO_vfprintf`. The wrapper function checks

DOI: 10.1002/spe

whether the arguments corresponding to the `%n` specifiers point to any return address or saved frame pointer. In addition, it also checks if all the arguments lie within a single stack frame. This prevents the attacker from probing the stack beyond the current stack frame using statements such as `printf("%x %x ...")`.

Extended runtime range checking by LibsafePlus

As discussed above, Libsafe determines bounds on the size of stack buffers and prevents overwriting of frame pointers and return addresses. Although it provides transparent runtime protection against buffer overflows it does so only for stack buffers. Also, for stack buffers the attacker is allowed to overwrite everything in the stack frame upto the frame pointer.

Our extension to Libsafe, LibsafePlus, is able to prevent overflows to all local, global and dynamically allocated buffers occurring due to the use of unsafe C library functions. In order to perform precise range checking of global and local buffers, LibsafePlus uses the information about buffer sizes made available to it at runtime by TIED. If the shared libraries used by the executable have also been modified using TIED, LibsafePlus has size information about buffers declared within the libraries too. If the size information for a specific buffer is not available, LibsafePlus falls back to the checks performed by Libsafe—no range checks for global buffers and loose bounds on sizes of local buffers. Note that this means that our approach can still be used successfully, although not as effectively, if neither source nor a binary compiled with the `-g` option is available for a program or some of the shared libraries that it uses.

For range checking dynamically allocated buffers, LibsafePlus intercepts calls to the `malloc` family of functions and thus keeps track of the sizes of various dynamically allocated buffers.

IMPLEMENTATION OF TIED

In this section, we describe how TIED extracts type information from a program binary and how it organizes this information as a data structure to be used by LibsafePlus. We first describe the techniques used for binary rewriting of executables and later show the techniques used to rewrite shared libraries.

Extracting type information

If the `-g` option is used to compile a program, the GNU C compiler adds type information about all variables to the executable in the form of special debugging sections. Debugging with arbitrary record format (DWARF) [7] is the standard format for encoding the symbolic, source-level debugging information. TIED uses the *libdwarf* consumer interface [8] to read the DWARF information present in the executable. For each function, information about all the local buffers is collected in the form of an ⟨offset from frame pointer, size⟩ pair. In the current implementation, TIED extracts information about character arrays only. For global buffers, the starting addresses and sizes are extracted. The members of arrays, structures and unions are also explored to detect any buffers that may lie within them. Figure 2 demonstrates a typical case of buffers within structures. TIED detects all the 40 buffers in this case. Buffers that appear inside a union may overlap with each other. For example, consider the

DOI: 10.1002/spe

```
struct s{
      char a[10];
      char b[5];
};
struct s foo[20];
```

Figure 2. Buffers within a structure.

```
struct my_struct1{
    char a[10];
    void *b;
    char c[10];
};
struct my_struct2{
    void *a;
    char b[16];
};
union my_union{
    struct my_struct1 s1;
    struct my_struct2 s2;
} x;
```

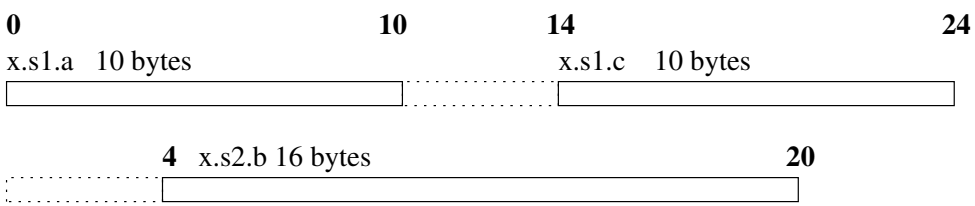


Figure 3. Overlapping buffers inside a union.

variable `x` declared as in Figure 3. Here, the buffer `x.s2.b` partially overlaps with both `x.s1.a` and `x.s1.c`. The problem is to decide whether a string copy of 10 bytes at destination address `((void *)&x + 4)` should be permitted. If it is, it may be used by an attacker to overflow `x.s1.a` and write an arbitrary value to `x.s1.b`. On the other hand, if the operation is not permitted, legitimate writes to `x.s2.b` may be denied. TIED, by default, takes the latter approach in order to prevent all possible buffer overflows. However, it is possible to force TIED to take the former approach by specifying a command line option.

After extracting the type information from the DWARF tables in the executable, TIED first filters it to retain information only about variables that are character arrays. It then constructs data structures to store this information for efficient runtime lookup. These data structures are then dumped back into the executable or the shared library file as a new loadable section. Currently TIED handles executable and shared library files in the ELF format only.

DOI: 10.1002/spe

TIED organizes the type information about character buffers in the form of several tables that are linked with each other through pointers, as shown in Figure 4. The top-level structure is a type information header that contains pointers to and sizes of a global variable table as well as a function table. The global variable table contains the starting addresses and sizes of all global buffers. The function table contains an entry for each function that has one or more character buffers as local variables or arguments[‡]. Each entry in the function table contains the starting and ending code addresses for the function, and the size of and a pointer to the local variable table for the function. The local variable table for a function contains sizes and offsets from the frame pointer for each local variable of the function or argument to the function that is a character array. The global variable table the function table and the local variable tables are all sorted on the addresses or offsets to facilitate fast lookup.

The type information header contains four extra fields in the case of shared libraries: the minimum and maximum virtual addresses occupied by library code and data at runtime; and place holders for two pointers that allow LibsafePlus to maintain a linked list of type information headers of various libraries loaded at runtime.

Rewriting executables

Once the type information about all variables has been organized in the form of tables as shown in Figure 4, TIED creates an empty space in the executable file for dumping the data structure. The space is created by extending the executable file towards lower addresses by a size that is large enough to hold the complete data structure and some other sections, which we shall shortly see, and is a multiple of page size. This is done because the virtual addresses of existing code and data objects cannot be changed in an executable file since the code is usually not position independent. The data structure is then 'serialized' to a byte array and the pointers in it are relocated according to the address where it will be placed in the binary.

As the position of the dumped data structure may vary from file to file, a pointer to the new section is made available as the value of a special symbol `_tied_type_info_structure_` in the dynamic symbol table (`.dynsym` section) of the binary. The name of the symbol itself needs to be added to the string table (`.dynstr` section), and the hash value of the symbol name needs to be added to the hash table (`.hash` section). Thus, in all, TIED enlarges three existing sections: `.dynsym`, `.hash` and `.dynstr`. Since the sizes of the original sections cannot be increased *in situ*, their new copies are placed in the gap created in the executable. Relevant pointers to the three sections from the dynamic (`.dynamic`) section are fixed to point to their new locations. Figure 5 illustrates the changes made to the target binary.

Rewriting shared libraries

The process of rewriting a shared library differs substantially from that of rewriting an executable, primarily because a shared library is relocated at runtime by the dynamic linker (`ld.so`), and its

[‡]An array can be an argument passed by value to a function if the array is part of a structure and the structure is passed by value.

DOI: 10.1002/spe

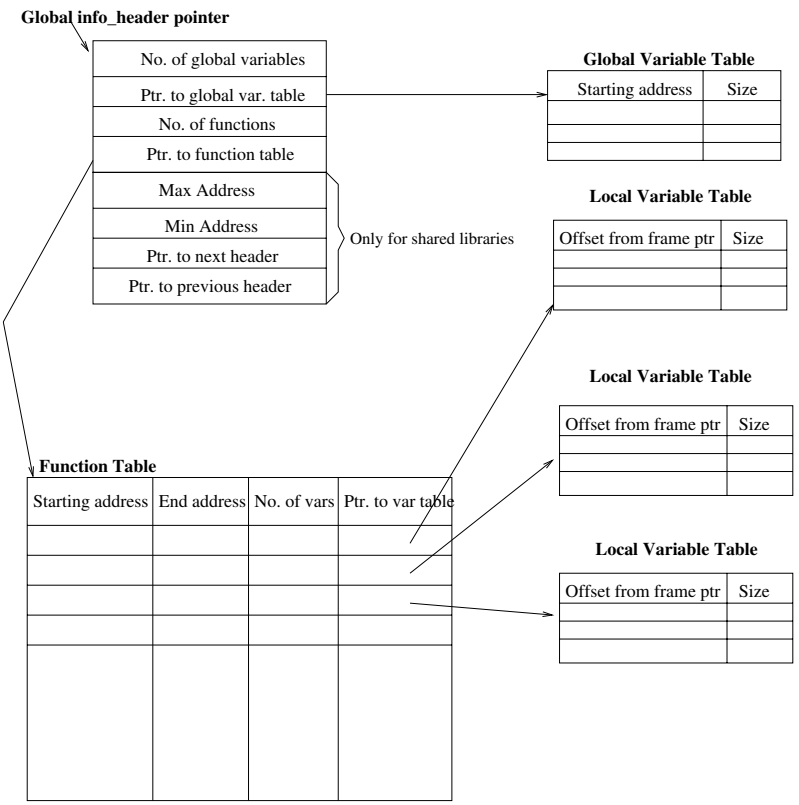


Figure 4. Data structures for storing type information.

actual load address is known only after this relocation. The virtual addresses used within the library file itself start with 0, and thus, unlike executables, the address space of a library cannot be extended towards lower addresses. Therefore, in the case of shared libraries, we insert the type information data structure and other new sections just after the program headers as shown in Figure 6. This necessarily changes the locations of the existing code and data objects in the library. However, since all new data are inserted at contiguous locations, the relative offset between any two existing objects remains the same as before. Any absolute reference to another object from an existing code or data object in the library must necessarily have a corresponding relocation entry in the relocation table. Such references are fixed by adding the size of the new data inserted to the offsets in such relocation entries. Similarly, values for all symbols defined in the dynamic and static symbol tables (sections `.dynsym` and `.symtab`) are updated.

DOI: 10.1002/spe

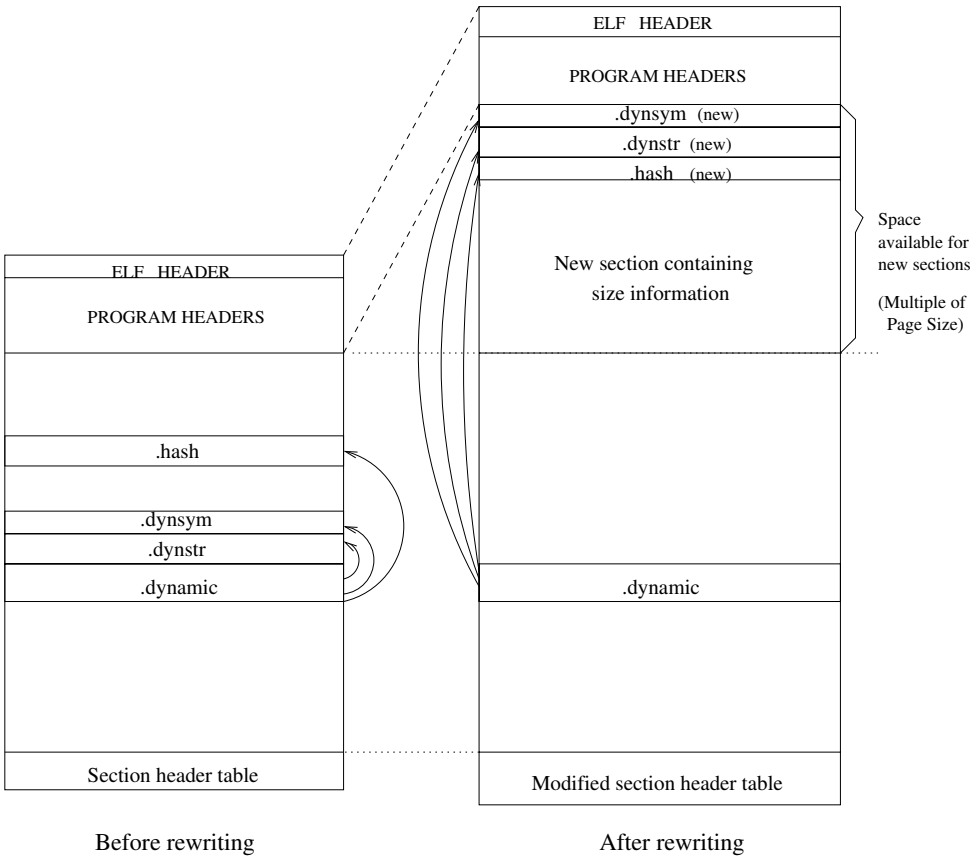


Figure 5. ELF executable before and after rewriting.

Relocating pointers in the data structure

The type information data structure contains two kinds of addresses: addresses of global variables and functions; and pointers that link various tables of the data structure. TIED ensures that all these pointers hold correct values before they are used by LibsafePlus. Since the base address at which the library is loaded is not known before loading, the pointers cannot contain absolute addresses. TIED adds relocation entries of the type `R_386_RELATIVE` to relocate all such pointers. The initial values contained in the pointers are offsets which when added to the base address yield actual addresses after relocation by `ld.so`.

DOI: 10.1002/spe

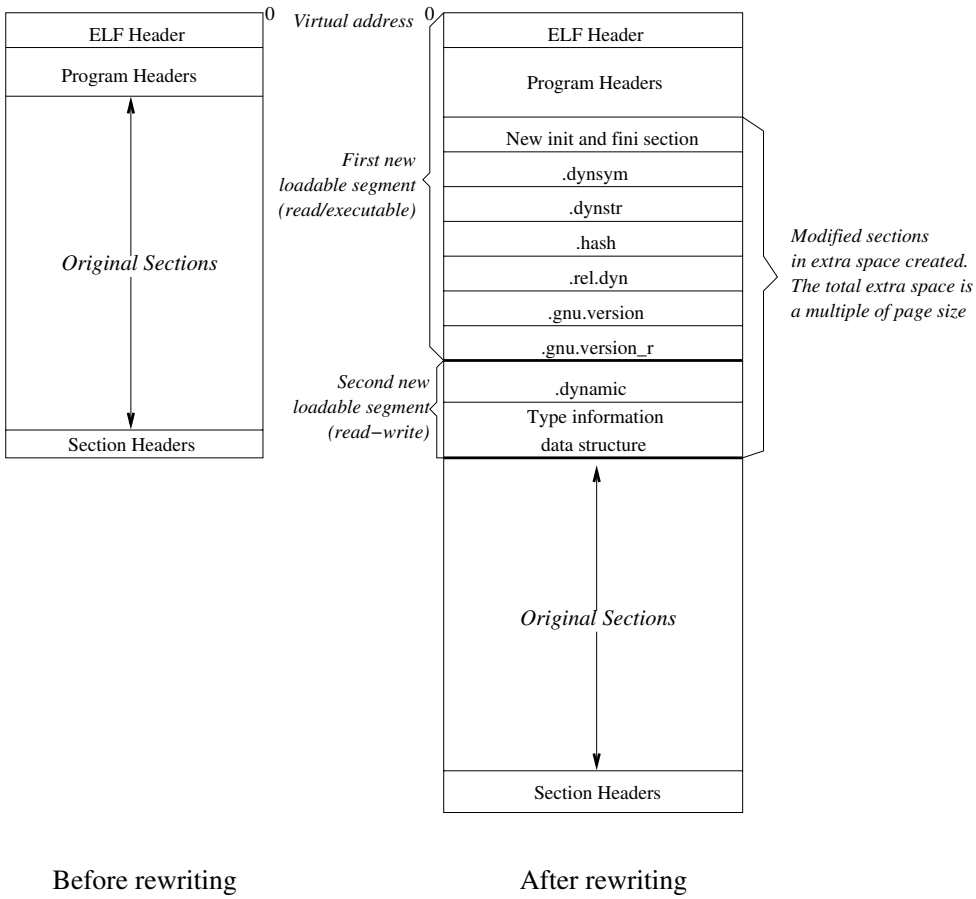


Figure 6. Shared library before and after rewriting by TIED.

Making the type information available to LibsafePlus

At runtime, LibsafePlus must be able to obtain a pointer to the type information header for a shared library when it is loaded. Unlike executables, a new symbol cannot be used to make this address available to LibsafePlus since multiple libraries may be used by a program in general. Also, since libraries can be dynamically loaded and unloaded by the program (using `dlopen` and `dlclose` calls), there must be a mechanism for LibsafePlus to be notified whenever a library is loaded or unloaded. TIED harnesses the initialization and finalization code mechanism for shared libraries to achieve this. Most libraries contain an initialization and a finalization function. The addresses of these functions are specified in the `DT_INIT` and `DT_FINI` entries respectively in the dynamic section of the library.

DOI: 10.1002/spe

To notify LibsafePlus of library loading and unloading events, TIED adds new initialization and finalization functions to the library. The new initialization function first calls the original initialization function of the library and then calls the LibsafePlus library registration function specifying the address of the type information structure as an argument. Similarly the new finalization code first calls the LibsafePlus library deregistration function and then the original finalization function. The `DT_INIT` and `DT_FINI` entries in the dynamic section are changed to point to these new initialization and finalization functions, respectively. The entire code of the new initialization and finalization functions is position independent.

Note that the rewritten library should work correctly even if LibsafePlus is not loaded. Therefore, our initialization and finalization functions use the `dlsym` function to determine whether the LibsafePlus registration and deregistration functions are available before calling these functions.

To call an external function from a library usually requires a procedure linkage table (PLT) entry and a GOT entry. To avoid enlarging these tables, we allocate space for a pointer and add a relocation entry of type `R_386_GLOB_DAT` to set the value of this pointer to the address of the `dlsym` function at the time of dynamic linking. The address of the type information structure is determined using position-independent code.

The entire process of changing the initialization and finalization functions also, in general, requires enlargement of the following existing loadable sections: the dynamic section (`.dynamic`) to add the library `libdl.so`, which defines the `dlsym` function, to the list of libraries required by this library; the dynamic symbol table (`.dynsym`), dynamic string table (`.dynstr`) and the hash table (`.hash`) to add the symbol table entry for the symbol `dlsym`; and the sections related to symbol versioning (`.gnu.version` and `.gnu.version_r`) [9].

Securing the TIED data structures

In addition to registering the library with LibsafePlus, the new initialization function also makes the pages containing the TIED structures read-only. The TIED data structures are initially located in a writable segment in order to allow dynamic relocation. Once the linker relocates the structures, the initialization function of the library calls `mprotect` system call to make the concerned pages read-only. This is done to prevent malicious modifications to these data structures. Note that a similar issue does not arise while rewriting executables since the TIED data structures in executables are in a read-only section as they do not need relocation.

As shown in Figure 4, the top-level table in TIED structures contains pointers to maintain a linked list of structures. These pointers are updated whenever a shared library is closed and it deregisters itself from LibsafePlus. In order to perform these updates, LibsafePlus drops the protection level of the page containing the pointers using `mprotect` and restores the level after altering the pointers.

Putting it all together

TIED modifies a number of existing sections in the process described above. For all sections that are increased in size, a new copy is made. Sections such as `.rel.plt` are modified by TIED *in situ* because they do not change in size. Two new program segments are created to hold all the new sections. The first segment is read-only and holds the new copies of the sections `.dynsym`, `.dynstr`, `.hash`, `.rel.dyn`, `.gnu.version` and `.gnu.version_r`. The second program segment is a

DOI: 10.1002/spe

read–write segment and contains the type information data structure and the new `.dynamic` section. The `.dynamic` section is modified to add an extra entry of the type `DT_NEEDED` that makes the library dependent on `libdl.so`. This is needed because the initializer function makes use of the `dlsym` function from `libdl.so`. The dynamic section needs to be in a read–write segment because the dynamic linker needs to fix absolute addresses in it. The data structure too needs to be in a read–write segment because it is relocated at load time. Figure 6 shows the changes done to a shared library by TIED while adding type information data structure.

LibsafePlus IMPLEMENTATION

LibsafePlus provides wrapper functions for unsafe C library functions such as `strcpy` and `memcpy` and performs bounds checks on the buffer being copied to. For bound verification of local and global buffers, LibsafePlus utilizes type information from the program binary and the shared libraries. For dynamically allocated buffers, it keeps track of the sizes and locations of buffers by intercepting calls to the `malloc` family of functions as described later. LibsafePlus is completely thread safe and can be transparently used for multi-threaded applications as well.

Determining sizes of heap buffers

By binary rewriting, all the buffers whose sizes are known at compile time can be protected from overflow. To capture the sizes of all dynamically allocated buffers, LibsafePlus intercepts all calls to the `malloc` family of functions, viz. `malloc`, `calloc`, `realloc` and `free`. In addition to calling the actual C library function, the wrapper function records the starting address and the size of the chunk of memory allocated. The number of elements, `nmem`, in the buffer is also recorded. The value `nmem` is equal to 1 except for buffers allocated using `calloc(nmemb, size)`, in which case it is equal to `nmemb`. LibsafePlus uses `nmem` to enforce a more rigorous size check[§]. For example, for the code below, an overflow will be detected if the tighter check is enforced. The loose heap check will, however, permit such usage:

```
char *buf = (char *)calloc( 5, 10 );
strcpy(buf, "A long string");
```

A red–black tree [10] is used to maintain the size information about dynamically allocated buffers. The tree contains a node for each buffer allocated using `malloc`, `calloc` or `realloc`. On freeing a memory area using `free`, the corresponding node is removed. Memory allocation for nodes in the red–black tree is done by a fast, custom memory allocator that directly uses `mmap` to allocate memory.

[§] A few programs have been found to fail when the rigorous check is applied. LibsafePlus, therefore, provides the strict check as an option that can be turned on using an environment variable.

DOI: 10.1002/spe

Table I. Unsafe C library functions intercepted by LibsafePlus. Formal parameters in bold are vulnerable to buffer overflows. Functions marked with a † are vulnerable to format string exploits.

strcpy (char ***dest**, const char *src)	strncpy (char ***dest**, const char *src, size_t len)
stpcpy (char ***dest**, const char *src)	wcscpy (wchar_t * **dest**, const wchar_t * src)
wcpcpy (wchar_t ***dest**, const wchar_t *src)	memcpy (void ***dest**, const void *src, size_t n)
strcat (char ***dest**, const char *src)	strncat (char ***dest**, const char *src, size_t n)
wcscat (wchar_t ***dest**, const wchar_t *src)	gets (char ***s**)
realpath (char ***path**, char resolved_path[])	sprintf (char ***str**, const char *format, ...)
vsprintf (char ***str**, const char *format, va_list ap)	getwd (char ***buf**)
vfprintf† (FILE *fp, const char *format, va_list ap)	
snprintf (char ***str**, size_t size, const char *format, ...)	
_IO_vfprintf† (FILE *fp, const char *format, va_list ap)	
vsnprintf (char ***str**, size_t size, const char *format, va_list ap)	
_IO_vfscanf (_IO_FILE *s, const char *format, _IO_va_list **argptr**, int *errp)	

Intercepting unsafe functions and bounds verification

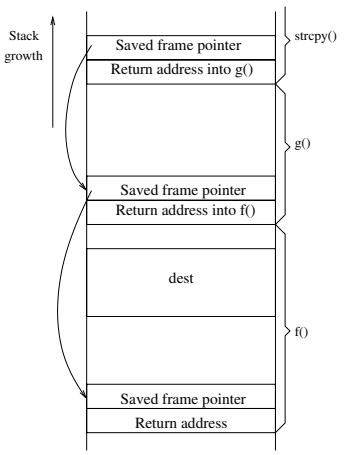
As outlined earlier, LibsafePlus works by intercepting unsafe C library functions such as `strcpy`. Table I lists all such functions intercepted by LibsafePlus.

The wrapper functions attempt to determine the size of destination buffer. If the size of the source string is less than that of the destination buffer, an actual C library function such as `memcpy` or `strncpy` is used to perform the requested operation. An overflow is declared when the size of the source string is more than that of the destination buffer, in which case the program is killed. If the size of the buffer cannot be determined, for instance if TIED was not used to augment the binary/shared libraries and the buffer is either global or local, the default protection offered by Libsafe is provided.

To determine the size of the destination buffer, it is first checked whether the destination buffer is on the stack, simply by checking if the buffer address is greater than the current stack pointer value. If found on stack, the stack frame encapsulating the buffer is found by tracing the frame pointers.

For example, consider the buffer `dest` shown in Figure 7. The buffer `dest` may either be a local variable of the function `f`, as in case (a), or may be an argument to the function `g`, as in case (b). LibsafePlus first assumes that `dest` is a local buffer defined in the function corresponding to its encapsulating stack frame and attempts to determine its size. In this example, LibsafePlus attempts to locate the local variable table of the function `f`. Note that the return address present in the stack frame of `g` is an address within the code of the function `f`. This return address is used to locate the entry for `f` in the function tables. First, LibsafePlus determines the correct type information data structure where the function `f` must be searched. For this purpose, the return address is compared with the minimum and maximum addresses of shared libraries, which are retrieved from the linked list of type information data structures. If `f` has been defined in one of the shared libraries, the corresponding type information data structure is obtained and the function `f` is searched for in its function table. Otherwise, the function table in the type information data structure corresponding to the executable,

DOI: 10.1002/spe




```
f(){
    char dest[] = "ab";
    ...
    g(dest);
    ...
}

g(char *a){
   ...
   strcpy(a,"yz");
   ...
}
```

(a)

```
typedef struct{
   char dest[3];
} S;

f(){
   S s = {"ab"};
   g(s);
   ...
}

g(S s){
   strcpy(s.dest,"yz");
   ...
}
```

(b)

Figure 7. Determining size of a stack buffer.

which is obtained by retrieving the value of the symbol `_tied_type_info_structure`, is used to search for the function `f` using the return address. Once the function table entry corresponding to `f` has been located, a search is made for the buffer `dest` in the local variable table of `f` using the offset of `dest` from the frame pointer of the stack frame of `f`. If the above search does not yield the size of `dest`, LibsafePlus assumes that `dest` is an argument to the function owning the stack frame just above the encapsulating frame. In this example, the return address into `g`, which is retrieved from the stack frame of `strcpy`, is used to locate the function entry for `g` using a similar procedure as outlined above. A search for the buffer `dest` is then made in the local variable table of `g` using the offset of `dest` from the frame pointer in the stack frame of `g`.

DOI: 10.1002/spe

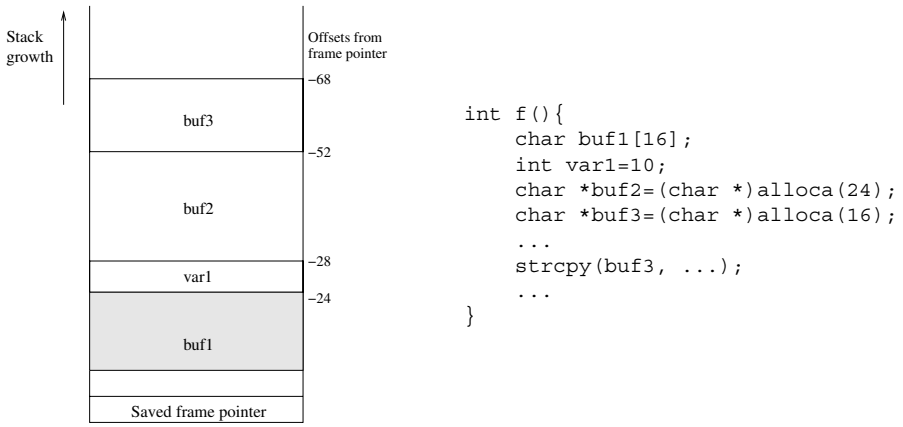


Figure 8. Estimating the size of a stack buffer that is not present in the local variable table. The shaded region represents the buffer (`buf1`) whose size has been recorded in the local variable table. The buffer `buf3` is therefore estimated to extend up to just before `buf1`. Thus, its size is estimated to be $(-24) - (-68) = 44$ bytes.

There are some cases when the information about the size of a stack buffer is not available in the local variable tables, for example when `alloca` is used to dynamically allocate buffers or when variable-length arrays (supported by gcc) are used. `Alloca` dynamically creates space for the buffer in the current stack frame and the space is automatically freed when the function returns. In such a case, LibsafePlus attempts to estimate the size of the destination buffer using the sizes of buffers available in the local variable tables. As shown in Figure 8, LibsafePlus estimates the extent of the destination buffer to be up to the largest address in its stack frame, such that it does not overlap with any other buffer in the local variable table.

If the buffer is not on the stack, it is checked to see whether it is on the heap by comparing its address with the minimum heap address. The minimum heap address is recorded by the `malloc` and `calloc` wrappers and is the address of the chunk allocated by the first call to `malloc` or `calloc`. The buffer is assumed to be on the heap if its address is greater than the minimum heap address. In this case, its size is determined by searching in the red–black tree.

Finally, if the buffer is neither on the stack nor on the heap, it must be a global variable defined either in the executable or in one of the shared libraries. In this case, again, the type information structure containing information about the buffer is located by comparing the address of the buffer with the minimum and maximum addresses of shared libraries. The buffer is then searched for in the appropriate global variable table. If none of the above checks yield the size of the buffer, the intended operation of the wrapper is performed. If the size of the destination buffer is available, the size of the contents of the source buffer is determined. The contents are copied only if the destination buffer is large enough to hold them. The program is killed otherwise. While killing the program does make the application vulnerable to denial-of-service attacks, letting the execution continue with a possibly malformed buffer

DOI: 10.1002/spe

is a more dangerous alternative. LibsafePlus, therefore, chooses to terminate the program whenever an overflow is detected.

Handling `memcpy`

Due to the fact that `memcpy` requires the programmer to specify the size of the data to be copied, its use seldom permits an opportunity for buffer overflow. However, the GNU C compiler transforms certain `strcpy` calls to `memcpy` calls. For this reason, the original Libsafe intercepted `memcpy` as well. However, this optimization is used only if the source of `strcpy` is a constant string. Such a situation is clearly not likely to lead to buffer overflow exploitation since the attacker cannot control the data copied to the destination buffer. Therefore, we believe that the interception of `memcpy` is not necessary in most cases. Note also that intercepting `memcpy` usually has a significant performance penalty since many programs use it quite heavily and the library implementation of `memcpy` is usually a very efficient, hand-optimized one.

LibsafePlus uses an environment variable to decide whether `memcpy` should be intercepted or not. If it is not to be intercepted, the wrapper function for it simply calls the C library `memcpy` without any checks. When `memcpy` *is* intercepted, only very loose checks are performed on the destination buffer. This is because `memcpy` is often used to copy the contents of one contiguous block of memory, such as an array or a structure, to another. This array or structure may contain within itself several character buffers. While the copying of the whole array is completely legitimate, the wrapper for `memcpy` may consider it to be an attempt to overflow a constituent character buffer. To prevent such false positives, only the loose checks of Libsafe are performed for stack buffers, i.e. a stack buffer is assumed to extend until just before the saved frame pointer in the enclosing stack frame. This check is reasonable because the stack buffer cannot extend beyond the stack frame and no 'sensible' use of `memcpy` should copy across stack frames. No size checks are carried out for global buffers because it cannot be ascertained from the type information data structure whether the location being copied to is a single buffer or a sequence of possibly non-adjacent buffers. For heap buffers being copied to by `memcpy`, the loose heap check, as described earlier, is always used.

PERFORMANCE

We have tested LibsafePlus for its ability to detect buffer overflows as well as for the overhead incurred due to its use. To test the protection ability of LibsafePlus, we used the test suite developed by Wilander and Kamkar [6]. This test suite implements 20 techniques to overflow buffers located on stack, `.data` or `.bss` sections. The test suite uses `memcpy` to overflow the target buffer. Since LibsafePlus implements loose checks for buffers being copied to by `memcpy`, we modified the test suite to use `strcpy` for overflowing the target buffer. The test suite executable was then modified using TIED. TIED detected all the global and local buffers declared in the test suite program. LibsafePlus was then preloaded while running the binary. All tests were successfully terminated by LibsafePlus when an overflow was attempted.

For testing performance overhead incurred due to LibsafePlus, we first measured overhead at a function call level. Next, the overall performance of 11 representative applications was measured. In the following subsections, we describe these tests and their results.

DOI: 10.1002/spe

Microbenchmarks

In this section, we present a comparison of the execution times of library functions such as `malloc()`, `memcpy()` etc. for the following three cases:

- the test was run without any protection;
- the program was protected with Libsafe;
- the program was protected with LibsafePlus.

The tests were conducted on a 1.5 GHz Pentium M machine with 512 MB of RAM running Linux 2.4.22.

We present here the performance results for two most commonly used string handling functions: `memcpy` and `strcpy`. To measure the overhead of finding sizes of global and local buffers using the new section in the executable, we performed the following experiment. The test program contained 100 global buffers and 100 functions. Each function had three local buffers. The time required by a single `memcpy()` into global and local buffers was measured for varying number of bytes copied. As shown in Figure 9, we found a constant overhead of 0.31 μs for `memcpy()` to global buffers. The percentage overhead decreases from about 50% for `memcpy` to buffers of 64 bytes to about 5% for `memcpy` to buffers as large as 1024 bytes. For local buffers, the overhead due to LibsafePlus is 0.5 μs per call to `memcpy()` as shown in Figure 10. This is basically the overhead for locating the stack frame corresponding to the buffer and is the same as the overhead due to Libsafe. To measure the overhead of finding the size of a heap variable from the red–black tree, the test program first allocated 1000 heap buffers. It then allocated another heap buffer and measured the time taken for one `memcpy()` to this buffer. This represents the worst-case performance as the buffer being copied to is the right-most child in the red–black tree. As shown in Figure 11, the overhead due to LibsafePlus is 1.5 μs per call to `memcpy()`.

We also measured the performance of LibsafePlus for calls to `strcpy()`. The testbed was similar to the one described earlier for `memcpy()`. Figure 12 shows the time taken by one `strcpy()` to a global buffer. The overhead drops from 0.9 μs for buffers of size 32 bytes to 0 μs for buffers of about 400 bytes. This is because the wrapper function for `strcpy()` in LibsafePlus uses `memcpy()` for copying, which is six to eight times faster than `strcpy()` for large buffer sizes. Figures 13 and 14 show similar results for `strcpy()` to local and heap buffers, respectively.

Next, we measured the overhead due to LibsafePlus in dynamic memory allocation. The insertion and deletion of nodes in the red–black tree is the primary constituent of this overhead. We measured the time required by a pair of `malloc()` and `free()` calls. The number of buffers already present in the red–black tree at the time of allocating the buffer was varied from 2^5 to 2^{23}. As shown in Figure 15, the time taken by LibsafePlus for the `malloc()`, `free()` pair grows almost logarithmically with the number of buffers already present in the red–black tree. This is expected because of the $O(\log(N))$ time operations of insertion and deletion of nodes in a red–black tree.

Macrobenchmarks

Next, we measured the performance overhead due to LibsafePlus using a number of applications that involve substantial dynamic memory allocation and operations such as `strcpy()` to buffers. In all, a total of 11 applications were used to evaluate the overhead of LibsafePlus and Libsafe.

DOI: 10.1002/spe

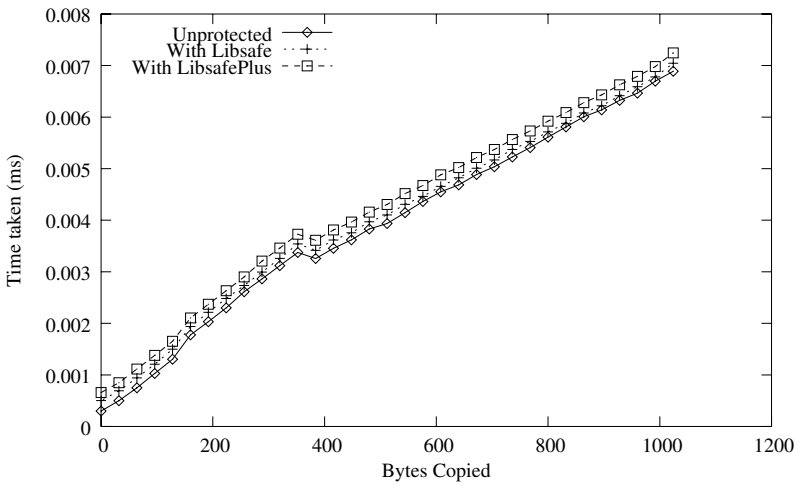


Figure 9. `memcpy()` to a global buffer.

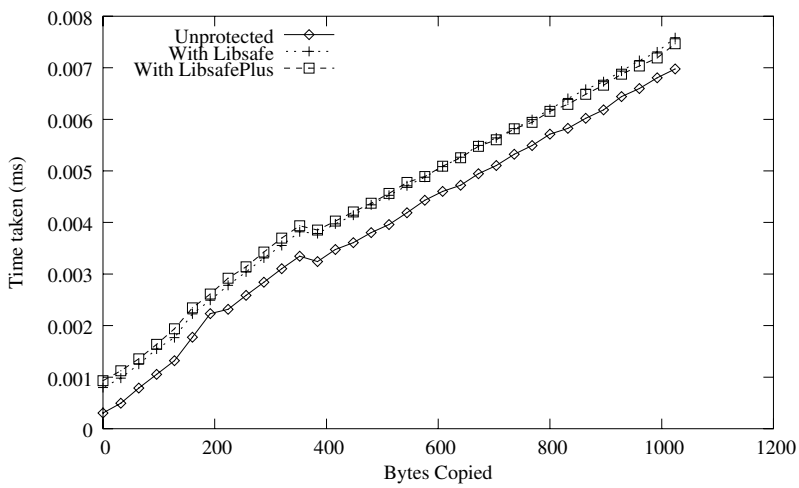


Figure 10. `memcpy()` to a local buffer.

DOI: 10.1002/spe

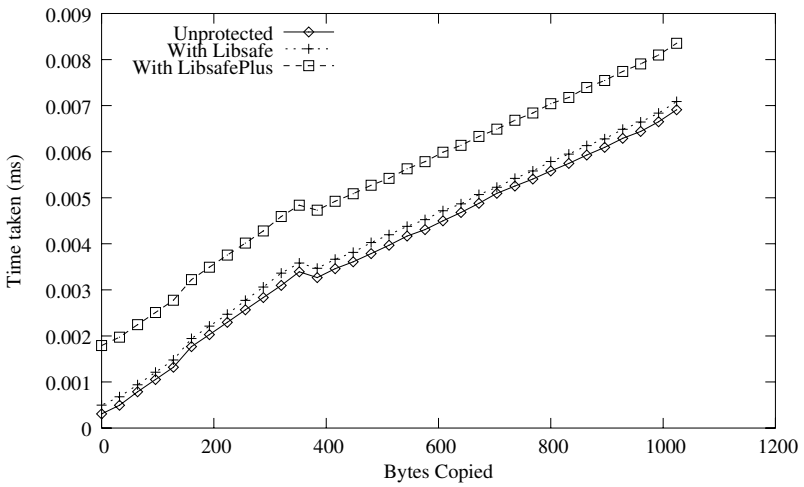


Figure 11. `memcpy()` to a heap buffer.

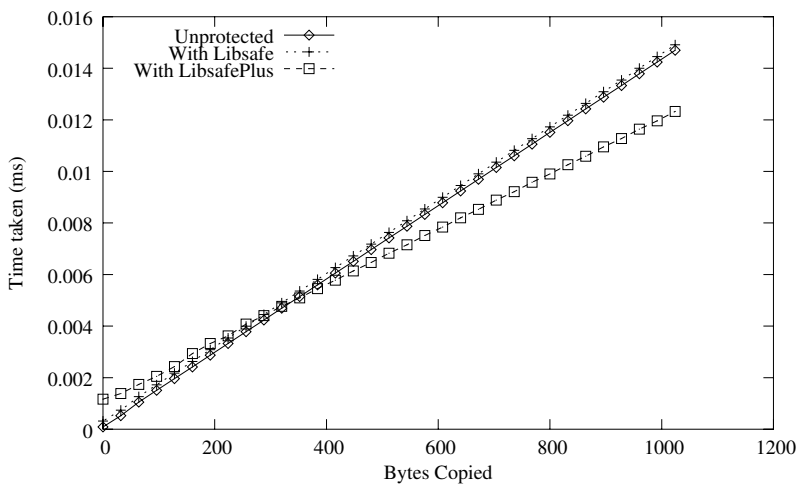


Figure 12. `strcpy()` to a global buffer.

DOI: 10.1002/spe

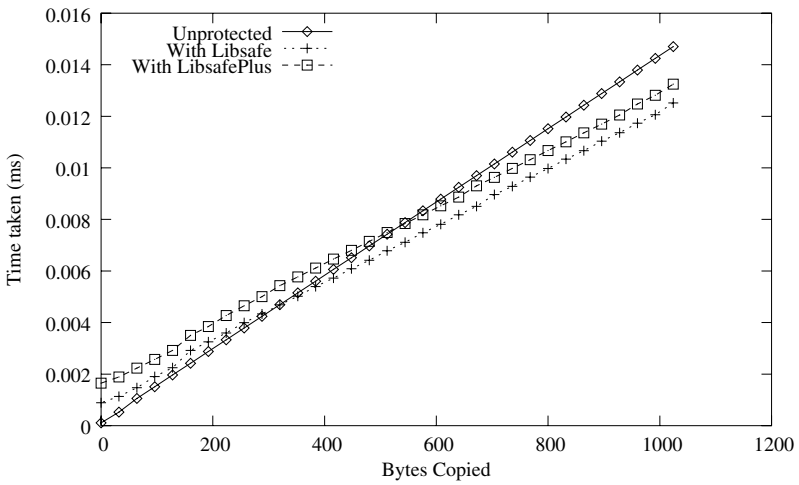


Figure 13. `strcpy()` to a local buffer.

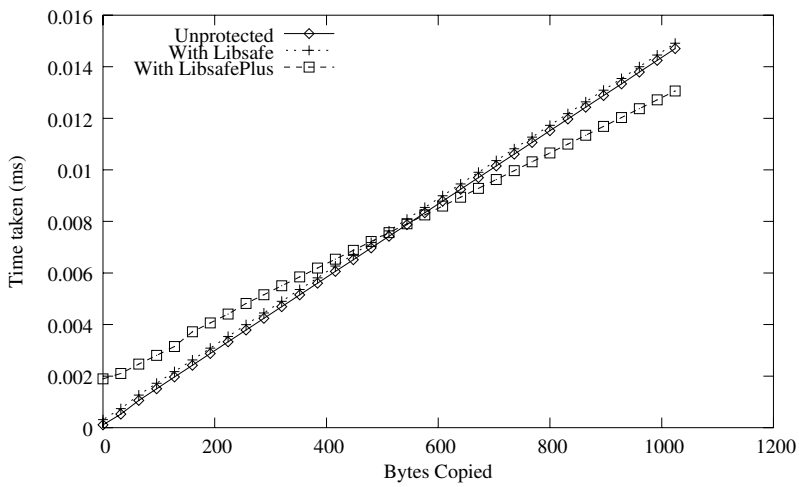


Figure 14. `strcpy()` to a heap buffer.

DOI: 10.1002/spe

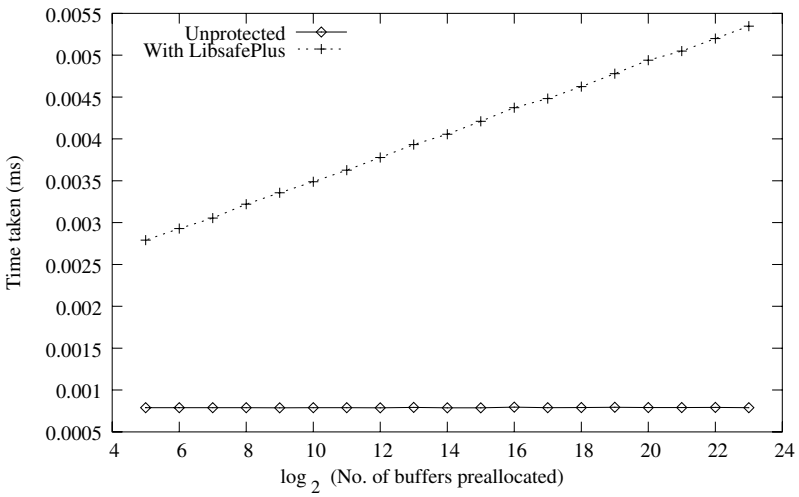


Figure 15. Performance overhead for `malloc()`, `free()` pair.

Two kinds of measurements were taken with LibsafePlus preloaded: with the environment variable `LIBSAFEPLUS_NO_MEMCPY` set, in which case no size checks are done for buffers being copied to using `memcpy`; and with the above variable unset, in which case loose checks are done for `memcpy` as described earlier. For the cases of LibsafePlus, all libraries used by the program were also compiled with the `-g` flag and modified with TIED.

Table II describes the performance metric used in each case. The performance overheads are shown in Figure 16. The graph shows normalized metric values with respect to the case when no library was preloaded. The overhead due to LibsafePlus was found to be less than 10% for all cases except for Bison, Enscript and Grep. In case of Enscript, Grep and Bison, the slowdown observed is due to a huge number of dynamic memory allocations and string operations on heap buffers.

Limitations of our approach

The current version of TIED is implemented for ELF binaries with debugging information in the DWARF format. The approach, however, is not limited by these formats. TIED can be upgraded to handle binaries in any standard format. Any debugging information format, such as the one used in COFF (Common Object File Format), that provides information about location and sizes of local and global variables declared in the program can also be handled. However, since the availability of debugging information is central to our approach, it does not handle commercial software that lack such information.

There are certain other cases that our approach is unable to handle. LibsafePlus can only guard against buffer overflows due to injudicious use of unsafe C library functions and not those due to other kinds of errors in the program itself. However, in many programs buffer overflows occur due to

DOI: 10.1002/spe

Table II. Description of application benchmarks.

Application	What was measured
Apache-2.0.48	Response time while requesting a large file from the Web server
Sendmail-8.12.10	Connection rate achieved while sending a large message
Bison-1.875	Time to parse a large grammar file and generate C code
Enscript-1.6.1	Time to convert a large text file to postscript
Hypermail-2.1.8	Time to process a large mailbox file
OpenSSH-3.7.1	Time to transfer a large set of files using the loopback interface
OpenSSL-0.9.7	Time to sign and verify using RSA
Gnupg-1.2.3	Time to encrypt and decrypt a large file
Grep-2.5	Time to perform a search for palindromes using back references on a large file
CCrypt	Time to decrypt a large file encrypted using CCrypt
Tar	Time to compress and bundle a large set of files

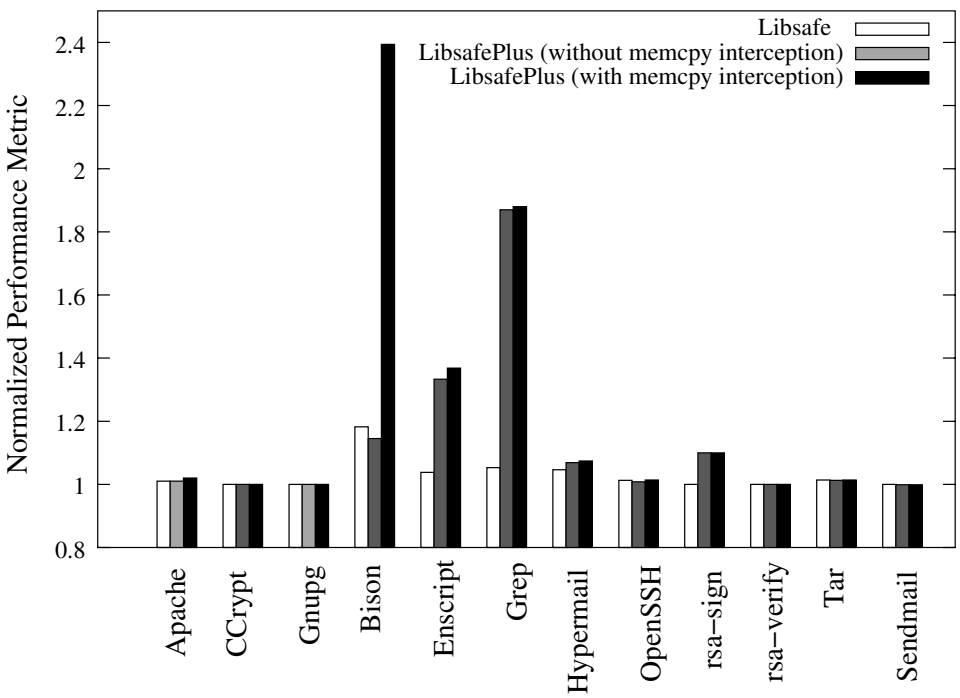


Figure 16. Macro performance overheads.

DOI: 10.1002/spe

improper use of C library functions rather than erroneous pointer arithmetic done by the programmer. Moreover, guarding against erroneous pointer arithmetic implies protecting every pointer instruction that would incur a high performance overhead.

The current implementation of TIED dumps size information about character buffers only. Thus overflows to other kinds of buffers, such as integer buffers, cannot be detected by LibsafePlus. Only character arrays have been handled since other kinds of buffers are seldom passed as parameters to string handling functions such as `strcpy`. This, however, is not a major limitation since TIED can easily be modified to extract information about all kinds of arrays.

Also, LibsafePlus cannot precisely determine sizes of buffers that are dynamically allocated on the stack. LibsafePlus is only able to estimate a safe bound on the size of such a buffer. Variable length local arrays pose a similar problem.

Since LibsafePlus uses `mmap` for allocating nodes for the red–black tree, programs that use `mmap` for requesting memory at specified virtual addresses may not work with LibsafePlus.

RELATED WORK

In this section, we review the related work in the area of protection against buffer overflow attacks.

Kernel-based techniques

The common feature used by the majority of buffer overflow attacks is the ability to execute code located on the stack. Solar Designer has developed a Linux patch that makes the stack non-executable [11], precisely to counteract the stack smashing attacks. The solution has some serious weaknesses. First, nested functions or trampoline functions, which are used by LISP interpreters, many Objective C compilers (including gcc) and most common implementations of signal handlers in Unix, require the stack to be executable. Second, the attacker does not require the code to be stored on a stack buffer for the exploit to work. Methods to bypass the non-executable stack defense have been explored by Wojtczuk [12].

PaX [13] is another kernel patch that aims to protect the heap as well as the stack. The idea behind PaX is to mark the data pages non-executable by overloading the supervisor/user bit on pages and enabling the page fault handler to distinguish the page faults due to attempts to execute data pages. PaX also imposes a significant performance overhead due to additional work done by the page fault handler for each page fault. Although protecting the heap offers some additional protection, it still does not guarantee complete protection from all forms of attacks. For example, return-into-libc attacks are still possible.

Static analysis based techniques

Static analysis approaches to handling buffer overflows attempt to analyze the program source and determine if the program execution can result in a buffer overflow.

Wagner *et al.* [14] formulated the detection of buffer overruns as an integer range analysis problem. The approach models C strings as a pair of integer ranges (allocated size and length) and vulnerable C library functions are modeled in terms of their operations on the integer ranges.

DOI: 10.1002/spe

Thus, the problem reduces to an integer-range tracking problem. The described tool checks, for each string buffer, whether its inferred length is at least as large as the allocated length. The tool is impractical to use since it produces a large number of false positives, due to lack of precision, as well as some false negatives.

The annotation-based static code checker [15] exploits the information provided in programs in the form of semantic comments. The approach extends the LCLint static checker [16] by introducing new annotations which allow the declaration of a set of preconditions and postconditions for functions. The tool does not detect all buffer overflow vulnerabilities and often generates spurious warnings.

CSSV [17] is another tool for statically detecting string manipulation errors. The tool handles large programs by analyzing each procedure separately and requires *procedure contracts* to be defined by the programmer. A procedure contract defines a set of preconditions, postconditions and side-effects of the procedure. The tool is impractical to use for existing large programs since it requires the declaration of procedure contracts by the programmer. Like other static techniques, the tool can produce false alarms. The pointer bug tracking tool [18] uses a hybrid pointer alias analysis that chooses between precise but inefficient path-sensitive analysis for simple access paths and an efficient but imprecise analysis for other references. The tool builds the definition-use chains to determine if an input by the user can potentially overflow a buffer. False positives, although very low in number, have nevertheless been shown to exist.

A very different approach used to prevent a large variety of exploits due to memory-related programming errors is that of address obfuscation [19]. Address obfuscation randomly modifies the absolute addresses of code and data in the program. This prevents return-into-libc-like attacks that are based on the knowledge of the actual addresses of functions. It also randomizes the relative distances between data items by variable and routine reordering and by introducing random padding between memory regions such as stack frames and dynamically allocated memory. This prevents the attacks that rely on specific ordering of data items such as those that overflow a buffer to overwrite a crucial data item stored after it. While address obfuscation itself cannot detect an exploit, it makes crafting an exploit more difficult. Thus, address obfuscation cannot be used as an active defence against memory exploits. Also, some of the obfuscation transformations either require changing the dynamic linker or relinking the program. The paddings used for obfuscating are also statically fixed at the time of transformation, thereby necessitating frequent 'reobfuscation'. Dynamically changing the paddings at runtime may not be feasible.

Runtime techniques

StackGuard [20] is an extension to the GNU C compiler that protects against stack smashing attacks. StackGuard enhances the code produced by the compiler so that it detects changes to the return address by placing a *canary* word on the stack above the return address and checking the value of the canary before the function returns. The *canary* is a sequence of bytes which could be fixed or random. The approach assumes that the return address is unaltered if and only if the canary word is unaltered. StackGuard imposes a significant runtime overhead and requires access to the source code. Techniques to bypass StackGuard protection are described in [21] and [22].

StackShield [23] is another tool, also implemented as a compiler extension, that protects the return address. The basic idea here is to save return addresses in an alternate non-overflowable memory space.

DOI: 10.1002/spe

The resulting effect is that return addresses on the stack are not used; instead the saved return addresses are used to return from functions. As with StackGuard, the source code needs to be recompiled for protection. A detailed description of StackShield protection and techniques to bypass it are available in [21] and [22]. A similar compile-time approach for buffer overflow prevention is described in [24]. In this approach, the return address stack is flanked by read-only sections on either side. The approach also handles complications due to use of `longjmp`. If the top of the return address stack does not match the return address on the function call stack, the discrepancy is attributed to a `longjmp` and the return address stack is popped off until the first match.

Propolice [25] is another compiler extension that aims to protect the saved frame pointer and the return address by placing a random canary on the stack above the saved frame pointer. In addition, Propolice protects local variables and function arguments by creating a local copy of arguments and rearranging the local variables on the stack so that all local buffers are stored at higher addresses than the local variables and pointers. As with StackGuard and StackShield, Propolice requires the recompilation of the source code. Although Propolice protects against stack smashing attacks, it is vulnerable to other forms of attacks.

Libverify [5] uses a similar idea as StackShield to prevent a function from returning to a forged return address. However, Libverify is implemented as a dynamically linked library. Upon initialization, Libverify instruments the program code in memory such that the return address verification code in the library will be called for each function call. A copy of each function defined in the executable is placed on the heap. The original function is modified to jump to a wrapper entry function on entry, which pushes the return address on a separate stack and jumps to the copy of the original function. The return instructions in the copied function are modified to jump to a wrapper exit function that verifies the return address before returning. To handle absolute jumps in the function code, the original function code is replaced by trap instructions and a trap handler is provided to lead the control to the corresponding instruction in the copied version of the function.

The memory-access error-detection technique [26] extends the notion of pointers in C to hold additional attributes such as the location, size and scope of the pointer. This extended pointer representation is called the *safe pointer* representation. The additional attributes are used to perform range access checking when dereferencing a pointer or while doing pointer arithmetic. The approach fails to work with legacy C code as it changes the underlying pointer representation.

The backwards compatible bounds checking technique by Jones and Kelly [27] is a compiler extension that employs the notion of *referent objects*. The referent object for a pointer is the object to which it points. The approach works by maintaining a global table of all referent objects which maintains information about their size, location, etc. Furthermore, a separate data structure is maintained for heap buffers by modifying `malloc()` and `free()` functions. Range checking is done at the time of dereferencing a pointer or while performing pointer arithmetic. The technique breaks existing code and involves a high performance overhead for applications which are pointer and array intensive since every pointer or array access has to be checked at runtime.

The C Range Error Detector (CRED) [28] is an extension of Jones and Kelly's approach. CRED extends the idea of referent objects and allows the use of a previously stored out-of-bounds address to compute an in-bounds address. This is done by storing all the information about out-of-bounds addresses in an additional data structure on the heap. The approach fails if an out-of-bounds address is passed to an external library or if an out-of-bounds address is cast to an integer and subsequently cast back to a pointer. As for Jones and Kelly's technique, the tool imposes a high performance overhead for pointer/array intensive programs since every access to a pointer has to be checked.

DOI: 10.1002/spe

The type-assisted dynamic-array bounds-checking technique [29] is also a compiler extension that works by augmenting the executable with additional information consisting of the address, size and type of local buffers, pointers passed as parameters to functions and static buffers. An additional data structure is maintained for heap buffers. Range checking is actually performed by modified C library functions which utilize this information to guarantee that overflows do not occur. As for other compiler based techniques, the solution is not portable and requires access to the source code of the program. It can be seen that this approach is the closest to ours. However, the main advantage of our approach is that it does not require compiler modifications and can work with the output of any compiler that can produce debugging information in the DWARF format.

PointGuard [30] is a pointer protection technique that encrypts pointers when they are stored in memory and decrypts them when they are loaded into CPU registers. PointGuard is implemented as a compiler extension that modifies the intermediate syntax tree to introduce code for encryption and decryption. Encryption provides for confidentiality only, hence PointGuard gives no integrity guarantees. Although, PointGuard imposes an almost zero performance overhead for most applications, it protects only code pointers (function pointers and longjmp buffers) and data pointers, and offers no protection for other program objects. Also, protection of mixed-mode code using PointGuard requires programmer intervention.

Program shepherding [31] is another technique aimed at preventing exploits by enforcing security policies based on control flow information. It allows the definition of execution privileges based on code origins (unmodified image from disk, dynamically generated but unmodified, modified etc.). Control flow transfers can be restricted based on the source and target of the calls and the type of call (direct versus indirect). Furthermore, it uses uncircumventable sandboxing to implement the above checks on program execution. The major disadvantage of this technique is that it can impose very high overheads in some cases, especially for multi-threaded applications.

Several other techniques based on binary file rewriting have also been described in the literature. Install-time vaccination of Windows executables and dynamically linked libraries (DLLs) [32] is a technique that rewrites the binary file at the time of installation to prevent functions from returning with a forged return address. The underlying idea is the same as Libverify except that the program binary on the disk rather than its image in memory is modified. First, all the function entry and exit points are discovered. The instructions at the entry and exit points are modified to jump to entry and exit wrapper functions, respectively. The entry wrapper function records the return address on a private stack and jumps back to the original function. The wrapper for function exit verifies the return address. The implementation uses a commercial disassembler (IDAPro) for disassembly of PE binaries. An alternative approach for PE binary disassembly has been discussed in [33].

One of the major drawbacks of all existing runtime techniques (except program shepherding) is that they require changes to the compiler. Most of these techniques have not been adopted by any of the mainstream compilers so far[¶]. In contrast, our approach does not require any compiler modifications and can be used with any existing compiler. We feel that this may lead to more widespread adoption of this technique in practice.

[¶]One of the exceptions is the Microsoft C compiler that provides an option for StackGuard-like protection.

DOI: 10.1002/spe

CONCLUSION

In this paper, we have presented TIED and LibsafePlus. These are simple, robust and portable tools that can together guard against buffer overflow attacks on local, global and heap variables arising from the use of unsafe C library functions. Our approach is a transparent runtime solution to the problem of preventing buffer overflows that is completely compatible with existing code and does not require source code access. Experiments show that our approach imposes an acceptably low overhead due to the runtime checks in most cases.

TIED and LibsafePlus are available in the public domain and can be downloaded from http://www.security.iitk.ac.in/projects/Tied-Libsafeplus.

ACKNOWLEDGEMENTS

Partial support for this work from the Prabhu Goel Research Center for Computer and Internet Security at IIT Kanpur is gratefully acknowledged. We would also like to thank the referees for their helpful comments on the initial draft of this paper.

REFERENCES

1. CERT/CC. Vulnerability notes by metric. http://www.kb.cert.org/vuls/bymetric?open&start=1&count=20 [November 2005].
2. CERT/CC. Cert advisories 2003. http://www.cert.org/advisories/#2003 [November 2005].
3. AlephOne. Smashing the stack for fun and profit. *Phrack* 1996; **7**(49).
4. Cowan C, Wagle P, Pu C, Beattie S, Walpole J. Buffer overflows: Attacks and defenses for the vulnerability of the decade. *Proceedings of the DARPA Information Survivability Conference and Expo (DISCEX)*, Hilton Head Island, SC, January 2000. IEEE Computer Society Press: Los Alamitos, CA, 2000.
5. Baratloo A, Singh N, Tsai T. Transparent run-time defense against stack smashing attacks. *Proceedings of the USENIX Annual Technical Conference*, San Diego, CA, 2000. USENIX Association: Berkeley, CA, 2000.
6. Wilander J, Kamkar M. A comparison of publicly available tools for dynamic buffer overflow prevention. *Proceedings of the 10th Network and Distributed System Security Symposium*, San Diego, CA, February 2003; 149–162.
7. TIS Committee. DWARF debugging information format specification version 2.0, May 1995.
8. UNIX International Programming Languages Special Interest Group. A consumer library interface to DWARF, August 2002.
9. Symbol Versioning. http://www.linuxbase.org/spec/book/ELF-generic/ELF-generic/symversion.html [November 2005].
10. Cormen TH, Stein C, Rivest RL, Leiserson CE. *Introduction to Algorithms* (2nd edn). MIT Press: Cambridge, MA, 2002.
11. Solar Designer. Non-executable user stack, 2000. http://www.openwall.com/linux/ [November 2005].
12. Wojtczuk R. Defeating solar designer non-executable stack patch http://www.insecure.org/sploits/non-executable.stack.problems.html [November 2005].
13. The PaX Team. PaX. http://pax.grsecurity.net [November 2005].
14. Wagner D, Foster JS, Brewer EA, Aiken A. A first step towards automated detection of buffer overrun vulnerabilities. *Proceedings of the Network and Distributed System Security Symposium*, San Diego, CA, February 2000. Internet Society, 2000; 3–17.
15. Larochelle D, Evans D. Statically detecting likely buffer overflow vulnerabilities. *Proceedings of the 10th USENIX Security Symposium*, Washington, DC, August 2001. USENIX Association: Berkeley, CA, 2001; 177–190.
16. Evans D, Guttag J, Horning J, Tan YM. LCLint: A tool for using specifications to check code. *Proceedings of the ACM SIGSOFT '94 Symposium on the Foundations of Software Engineering*, New Orleans, LA, 1994. ACM Press: New York, 1994; 87–96.
17. Dor N, Rodeh M, Sagiv M. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, San Diego, CA, June 2003. ACM Press: New York, 2003; 155–167.

DOI: 10.1002/spe

18. Livshits VB, Lam MS. Tracking pointers with path and context sensitivity for bug detection in C programs. *ESEC/FSE-11: Proceedings of the 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, New York, NY, 2003. ACM Press: New York, 2003; 317–326.
19. Bhatkar S, DuVarney DC, Sekar R. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. *Proceedings of the 12th USENIX Security Symposium*, Washington, DC, August 2003. USENIX Association: Berkeley, CA, 2003; 105–120.
20. Cowan C, Pu C, Maier D, Walpole J, Bakke P, Beattie S, Grier A, Wagle P, Zhang Q, Hinton H. Stackguard: Automatic adaptive detection and prevention of buffer overflow attacks. *Proceedings of the 7th USENIX Security Conference*, San Antonio, TX, January 1998. USENIX Association: Berkeley, CA, 1998; 63–78.
21. Richarte G. Four different tricks to bypass stackshield and stackguard protection. http://downloads.securityfocus.com/library/StackGuard.pdf [November 2005].
22. Bulba and Kil3r. Bypassing stackguard and stackshield. *Phrack* 2000; **10**(56).
23. Stackshield—a stack smashing technique protection tool for Linux. http://www.angelfire.com/sk/stackshield [November 2005].
24. Hsu FH. Rad: A compile-time solution to buffer overflow attacks. *ICDCS '01: Proceedings of the 21st International Conference on Distributed Computing Systems*, Washington, DC, 2001. IEEE Computer Society Press: Los Alamitos, CA, 2001; 409–417.
25. Etoh H, Yoda K. Propolice—improved stack smashing attack detection. *IPSJ SIGNotes Computer Security (CSEC)* 2001: **14**(25).
26. Austin TM, Breach SE, Sohi GS. Efficient detection of all pointer and array access errors. *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, Orlando, FL, 1994. ACM Press: New York, 1994; 290–301.
27. Jones RWM, Kelly PHJ. Backwards-compatible bounds checking for arrays and pointers in C programs. *Proceedings of the International Workshop on Automated and Algorithmic Debugging*, 1997; 13–26.
28. Ruwase O, Lam MS. A practical dynamic buffer overflow detector. *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, San Diego, CA, 2004. Internet Society, 2004; 159–169.
29. Lhee KS, Chapin SJ. Type-assisted dynamic buffer overflow detection. *Proceedings of the USENIX Security Symposium*, San Francisco, CA, 2002. USENIX Association: Berkeley, CA, 2002; 81–88.
30. Cowan C, Beattie S, Johansen J, Wagle P. Pointguard: Protecting pointers from buffer overflow vulnerabilities. *Proceedings of the USENIX Security Symposium*, Washington, DC, 2003. USENIX Association: Berkeley, CA, 2003; 91–104.
31. Kiriansky V, Bruening D, Amarasinghe S. Secure execution via program shepherding. *Proceedings of the 11th USENIX Security Symposium*, San Francisco, CA, August 2002. USENIX Association: Berkeley, CA, 2002; 191–206.
32. Nebenzahl D, Wool A. Install-time vaccination of Windows executables to defend against stack smashing attacks. *Proceedings of the 19th IFIP International Information Security Conference*, Toulouse, France, August 2004. IFIP, 2004; 225–240.
33. Prasad M, Chiueh TC. A binary rewriting defense against stack-based buffer overflow attacks. *Proceedings of the USENIX Annual Technical Conference*, San Antonio, TX, June 2003. USENIX Association: Berkeley, CA, 2003; 211–224.

DOI: 10.1002/spe