

# LibsafeXP: A Practical and Transparent Tool for Run-time Buffer Overflow Preventions

Zhiqiang Lin, Bing Mao and Li Xie

*Abstract—*

This paper presents a practical tool, LibsafeXP, to protect the software against the most common and severe attack, buffer overflows. As a dynamic shared library and an extension to Libsafe and LibsafePlus, LibsafeXP contains wrapper functions for all the buffer related functions in C Standard Library. These wrapper functions are enforced to check the source and target buffer's size using the following information: global buffer knowledge extracted from the program symbol information, heap buffer knowledge by intercepting memory allocation family functions, and stack buffer bound information by dynamically determined from the frame pointer. Compared with other approaches, LibsafeXP is more transparent to programs: it works on binary mode, and neither requires the source code nor any debugging information. The performance and effectiveness evaluation indicates LibsafeXP could be used to defend against buffer overflow attacks and impose about 10 percent overhead on the protected software.

## I. INTRODUCTION

Buffer overflow is widely recognized as one of the most common vulnerabilities. According to the NIST National Vulnerability Database [1], buffer overflow accounts for approximately from 10 to 20 percent of the total vulnerabilities in the past few years. Buffer overflow is also regarded as one of the most severe vulnerabilities, since normal buffers usually lack bounds checking on the input, attackers can overflow any desired data and inject malicious code with carefully crafted inputs. As a typical example, most of the fast and wide spreading worms, such as Code Red, SQL Slammer and Blaster, use buffer overflow to propagate on the Internet and have caused severe economic losses.

Despite many defensive methods against buffer overflow attacks have been investigated in the past several years, buffer overflows do not hence disappear and people still struggle in defending against these attacks. This is mainly due to the following reasons: (1) Many proposed solutions (e.g., StackGuard [7], CRED [10]) require access to program source code, however, for most commercial software, the source code is not usually available to public. (2) Some of them incur undesirable performance overheads (e.g., Jones & Kelly's bounds checking [13]), thus users are reluctant to apply them to protect their software. (3) Some of them require hardware support (e.g., SmashGuard [8]), which is sometimes inconvenient to use. (4) Some of them require the relevant information such as debugging information (e.g., LibsafePlus [14]), but in many cases these are always unavailable in the released software. Thus, it is yet necessary to provide both *practical* and highly *efficient* solution to prevent buffer overflows.

In this paper, we propose a lightweight tool, LibsafeXP, to

transparently defend against buffer overflow attacks at run-time. Our purpose is simple: just to add bounds checking for all the program dereferencing buffer. Specially, based on different memory regions to which the buffer belongs in the mapped process image, we divide them into three categories: global buffers, dynamically allocated heap buffers and stack buffers. We model each program dereferencing buffer using a pair of its starting address and size, and by leveraging library interposition technique, we enforce a bounds checking for all the program dereferencing buffer in LibsafeXP, which extends Libsafe and LibsafePlus and provides wrapper functions for all the buffer related as well as those dynamic memory allocation family functions in C Standard Library. The global buffer's size and starting address are extracted from the symbol table section of ELF (Executable and Linking Format) [3] executable file (in this paper, we focus on Linux platform, ELF and C programs), and dynamically allocated heap buffer's size and location information are tracked at run-time in the intercepted `malloc` family functions. For the size of stack buffers, we use the frame pointer to act as the upper bound as Libsafe [2] to calculate.

Compared with existing methods, our approach offers several advantages:

- *Practical in application.* Our approach operates on a normally compiled binary program, and appears almost transparent to the protected software. This makes it practical to use for a wide variety of programs, including proprietary and commodity programs for which no source code is available.
- *Effective against buffer overflow attacks.* We extend the test suite developed by Wilander et al. [16] to evaluate the effectiveness of our approach. The result was LibsafeXP successfully prevented all of overflows on global and heap buffers, and most of the stack buffers.
- *Easy to use.* For protections, users only need to set the environment variable and restart the program. Moreover, it can be simply set for the protection of either specific program or all processes on the machine.
- *Low run-time overheads.* As the experiment indicated, our approach imposes about 10 percent run-time overhead on the protected software.

Our work makes the following contributions. We present an improved *dynamic* and *transparent* approach to the detection and prevention of buffer overflows. In general, it is a highly *practical* and *efficient* solution in the defense against buffer overflow attacks. Besides, we have implemented the prototype, LibsafeXP, and made an empirical evaluation showing the feasibility of our approach.

The rest of this paper is organized as follows. Section 2 presents a technical description of our approach. Section 3 de-

Z. Lin: State Key Lab for Novel Software Tech., Nanjing University, China.  
B. Mao: State Key Lab for Novel Software Tech., Nanjing University, China.  
L. Xie: State Key Lab for Novel Software Tech., Nanjing University, China.

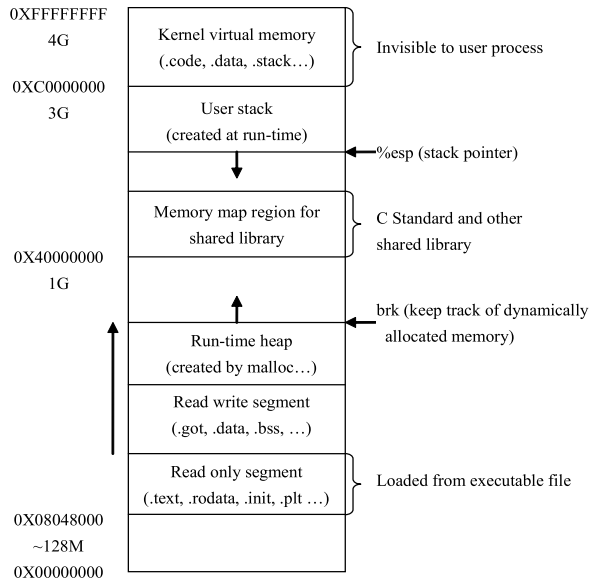


Fig. 1. Memory layout of a typical Unix process.

scribes the design and implementation of our tool, LibsafeXP. Section 4 provides the experimental results on effectiveness and performance evaluation. Then the advantages and limitations is discussed in Section 5, followed by the related work in Section 6. Finally, Section 7 concludes the paper.

## II. APPROACH DESCRIPTION

### A. Memory Layout of UNIX Process

Executable file in UNIX system is often organized as ELF structure, which contains a set of necessary information such as instructions, read-only-data and symbol tables, for program execution. The executable file when run as a process is typically mapped into memory as Figure 1 shows.

Since to only overflow the pure data memory would not directly cause buffer overflow attacks, malicious attackers must overwrite the return address or function pointer to achieve their goals. According to the memory layout of a typical Unix process (Figure 1), we can find these data areas could be in (a) `.stack` for local automatic variables, (b) `.heap` for dynamically allocated variables, (c) `.bss` and `.data` for uninitialized and initialized global or local static variables (in the following text, we will refer to both of them as global variables). Note why we do not take other data memory areas, such as `.got` and `.ctors`, into consideration, the reason is for these areas user's normal functions are not permitted to access them.

Thus, we can divide buffer overflow areas into three categories: (a) stack buffers, (b) heap buffers and (c) global buffers. If we prevent these three types of area from out-of-bounds data

```

1 #include <stdio.h>
2 #include <string.h>
3 char str1[10]; //in .bss
4 char str2[20]="a test string"; //in .data
5 int main() {
6     static char str3[30]; //in .bss
7     static char str4[30]="in .data";
8     strcpy(str1,str2);
9     strcpy(str3,str4);
10    printf("str1=%s\n",str2);
11    return 0;
12 }
    
```

Fig. 2. A simple typical C program.

occurrence, we could hence prevent the buffer overflow attacks.

### B. Overview

As normal ELF executable file always contains the symbol table to describe the program symbol's address, size and other related information for link editing and relocations [3]. It would be very useful if we utilize this information. Consider a simple but typical program as shown in Figure 2, it has global uninitialized buffer `str1`, global initialized buffer `str2`, static uninitialized local buffer `str3`, and static initialized local buffer `str4`. When this program is compiled and linked, the content of its symbol table looks like as Figure 3 shows. The first column is the symbol's starting address, the next is the associated size, then the type, binding attributes and visibility, the fifth column is the section name in which the symbol resides, and the last is the symbol's name. Despite the program function symbol information also exists in the symbol table, we can distinguish it from the variable symbol by symbol's type (program variable is OBJECT type and function is FUNC type [3]).

Due to the symbol's starting address and associated size information clearly described in the ELF symbol table, we can make use of this information for our security purpose. As a result, the first step of our approach is to extract the starting address and size of all the program global variables. For the program listed in Figure 2, it would work (no false positive and false negative) by simply retrieving the program variable symbol's starting address and associated size, and validating them in the buffer related `glibc` functions (e.g., `strcpy`).

|     |          |    |        |        |       |        |
|-----|----------|----|--------|--------|-------|--------|
| ... | 08049714 | 30 | OBJECT | LOCAL  | .bss  | str3.3 |
|     | 080495dc | 30 | OBJECT | LOCAL  | .data | str4.4 |
|     | 08049734 | 10 | OBJECT | GLOBAL | .bss  | str1   |
|     | 080495c8 | 20 | OBJECT | GLOBAL | .data | str2   |
| ... |          |    |        |        |       |        |
|     | 080483b4 | 91 | FUNC   | GLOBAL | .code | main   |
| ... |          |    |        |        |       |        |

Fig. 3. Some part of symbol table of the example program.

However, we should note the symbol information extracted from the ELF executable file is just the symbol's starting address and whole size about the global variable. Other information such as variable type (e.g., the symbol is an integer type, character array type or struct type), and information about local

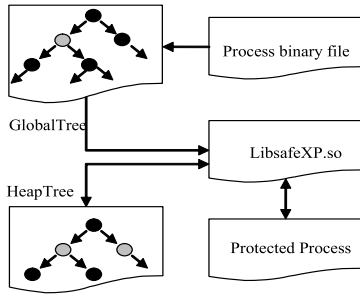


Fig. 4. LibsafeXP overview.

automatic variable is unavailable in ELF symbol table. In addition, the buffer variable may be a member of certain record, and its address and size information also does not exist in the symbol table.

As a result, in our approach we propose not to strictly assure every buffer's access is absolutely accurate within its scope. Instead if we can restrict these access (when dereferencing global, heap and stack buffers) to a relative smaller scope than that of no protection, then it can also prevent most of the buffer overflows. This is the key idea of our approach. As a proof of concept, we have implemented a prototype LibsafeXP, which intercepts all the reference to unsafe `glibc` functions to do the enforcement, and make use of the program relevant symbol information for global buffer overflow prevention, dynamically determined bound information for stack buffer overflow prevention, and dynamically tracked information for heap buffer overflow prevention. In practice, our approach is efficient and powerful.

### III. DESIGN AND IMPLEMENTATION

Our prototype LibsafeXP is a program pre-loaded shared library. Its architecture is illustrated in Figure 4. Further explanations about the design and implementation are provided in this section. We proceed as follows. First, we show how to extract the starting address and size information from program ELF symbol tables, and then we describe in detail how LibsafeXP handles the three kinds of buffer overflow protection.

#### A. Symbol Information Extraction

Since program symbol information exists in ELF binary image, when using our tool for software protections, we will firstly need to identify the corresponding binary file (by looking into the proc file system to determine). After we find out the program binary file, we traverse its program header table to locate the symbol table section, and extract the starting address and size information for program global variable (including the local static one) from the ELF symbol section. This process takes place when loading LibsafeXP.

In our implementation, as every buffer's access is enforced to check the relevant size, we use a red-black tree [4] (Figure 5 shows its definition) to store the extracted location and size in-

formation of global buffers in order to promote the performance (we can also use a hash table to store this information). We call this tree variable, which contains the starting address and size of all program global variables, as *GT* (Global Tree). In addition, since the dynamic allocated heap buffer has similar properties, i.e., starting address and size information, to the global buffer, we use another red-black tree variable called *HT* (Heap Tree, the same definition as *GT*) to store the newly allocated buffer (this is demonstrated in Figure 4). Both *GT* and *HT* are sorted on the keyword of *addr*. Some wrapper functions such as `strcpy` in LibsafeXP will refer *GT* or *HT* to calculate the access range of global or heap buffers. The difference between *GT* and *HT* is that *GT* will remain constant during program execution, whereas *HT* will be dynamically changed when heap variables are created and deleted.

```
struct t_tree_node{
    enum {red, black} colour;
    char *addr;
    int size;
    struct t_tree_node *left,
                      *right,
                      *parent;
}
```

Fig. 5. Node definition of our red-black tree.

#### B. Buffer Overflow Prevention in LibsafeXP

Our idea on buffer overflow prevention is via range checking for all the three kinds (global, local and heap) of program dereferencing buffers. To check the source length of any buffer is easy since we can use `strlen` to achieve, thus, the main work of our approach is to get the access range of the destination buffer.

In our approach, for any dereferencing destination buffer  $\alpha$ , its limited maximum access range  $f(\alpha)$  is calculated by

$$f(\alpha) = \begin{cases} EBP - \alpha & \text{if } \alpha \in Stack \\ T(\alpha).size - (\alpha - T(\alpha).addr) & \text{if } \alpha \in GT \cup HT \\ 0 & \text{if } \alpha \notin Stack \cup GT \cup HT \end{cases} \quad (1)$$

where  $EBP$  is the pointer to the stack frame in which  $\alpha$  resides, and  $T(\alpha)$  is the most nearest node that could contain address  $\alpha$  in our red-black tree.

After we calculate the access range for the destination buffer, we can hence prevent buffer overflows if the destination buffer is not enough to hold the source data. In the following subsections, we describe how to implement this feature in our LibsafeXP.

##### B.1 Bounds checking on global buffers

As stated, we use the `LD_PRELOAD` feature provided by Linux/Unix to load LibsafeXP. Thus, after users launch the software protected by our tool, the *GT* will subsequently contain its associated program global symbols' starting address and size

```

...
struct test {
int a[600];
char buf[20];
}A;
char p[20];
char str2[]="hello world\n";
...
int foo(){
    ...
    strcpy(p+4,"buffers");
    strcpy(A.buf,str2);
    ...
}
    
```

Fig. 6. Some piece of example code.

information, which will be referred in the buffer related wrapper functions for buffer overflow determination.

Suppose a wrapper function such as `strcpy(dest, src)` is called. LibsafeXP will firstly check whether or not the `dest` is a stack variable, because the stack variable is much more frequently dereferenced than global or heap variable. If `dest` is not in *Stack*, then it determines which tree (*GT* or *HT*) the `dest` may be in (we use a flag to track the largest address of program global variable during the process of symbol information extraction, and consequently we can easily distinguish it as a global or heap buffer).

After identifying the tree it belongs to, we call the searching operation on *GT* or *HT* to find out the most nearest node of the dereferencing buffer `dest` ( $T(dest)$ ). Note the remained length of the found node is the `dest`'s legal access range ( $f(dest)$ ). If it is found, then the relevant access range is used to detect buffer overflow by comparing  $f(dest)$  with `strlen(src)`, and subsequently it either continues if the operation is permitted ( $f(dest) \geq \text{strlen}(src)$ ) or reports buffer overflow detected and aborts the program. Otherwise, i.e., the `dest`'s address is not found in the two trees ( $f(dest) = 0$ ), there is only one reason for global buffers (for heap buffers, we discuss it below): the `dest`'s address is an illegal access address which is caused from a programming error, or an attack to smash some addresses that normal program is not allowed to access (e.g., `.got` area). In this circumstance, we also report this error and abort the running program.

Look at some pieces of code shown in Figure 6 to better understand how it works. When a buffer related function, e.g., `strcpy(p+4, buffers)` is called in function `foo` ("`p + 4`" is a common pointer arithmetic operation used frequently in C programs), the following steps are used to check its legal access range. Firstly, it checks that "`p + 4`" is not in *Stack*. Then it determines "`p + 4`" is in *GT*, so it searches *GT* to find the nearest node of "`p + 4`" which is the node storing the address of symbol `p` (i.e.,  $T(p+4).addr = p$ ). Next it gets the accurate access range of  $f(p+4) = T(p+4).size - ((p+4) - T(p+4).addr) = 20 - ((p+4) - p) = 16$  bytes, and hence permits the operation ( $16 > (\text{strlen}(\text{buffers})=7)$ ). When calling `strcpy(A.buf, str2)` in function `foo`, it also determines `A.buf` in *GT* and finds its nearest node is the node

storing the address of symbol `A` (i.e.,  $T(A.buf).addr = A$ ). And subsequently it gets `A.buf`'s access range  $f(A.buf) = T(A.buf).size - ((A.buf) - T(A.buf).addr) = 620 - (A.buf - A) = 620 - ((A + 600) - A) = 20$ , which is also in legal access range ( $20 > (\text{strlen}(\text{hello world})=11)$ ).

It may seem amazing that we can correctly get the access range even for a record type member variable. In fact, we cannot get the accurate access range of any legal global variable, but the maximum limited access range in our approach. Therefore we get the `A.a`'s access range  $f(A.a) = 620$  bytes if the intercepted function refers this variable.

For some cases, if the symbol information is not available in the program binary image (e.g, for anti-software reverse engineering purpose), for a given global variable `dest`, we would calculate its  $f(dest) = 0$  ( $dest \notin GT \cup HT \cup Stack$ ) and hence lead to false positives for program legal global buffers. As a result, we add another flag to notify our detection engine if symbol information has been stripped.

## B.2 Bounds checking on heap buffers

Heap variables are often allocated by memory allocation family functions, such as `malloc`. If we intercept these function calls, we could hence track the allocated buffer's location and size easily.

In our implementation, once these `malloc` family functions are called, we add the newly allocated symbol's starting address and associated size into our *HT*. And later, we use the same buffer overflow prevention method described above to determine the legal access range so as to defend against heap buffer overflows.

As mentioned, heap buffers can be freed dynamically as program executions. Thus, when the heap buffer is freed, we need to delete its corresponding node in *HT*. This is achieved in the intercepted `free` function. We should note for a heap variable `dest`, if we calculate its  $f(dest) = 0$ , then we could conclude `dest` may be a double freed variable (it has been previously deleted from *HT*) or an attack address located in heap area.

Note `glibc` itself after 2.3.5 has added several mandatory integrity assertions in `unlink`, `free`, `realloc` and other places to detect heap overflows, so it would seem unnecessary to add bounds checking for all the heap variables. However, the integrity assertions can be bypassed in some cases (e.g., the attacks discovered by Phantasmagoria for bypassing `glibc` integrity check [5]). Therefore, we believe it is still necessary for us to track all the heap buffer's size to detect and prevent heap overflows.

## B.3 Bounds checking on stack buffers

Stack variables are often the function local variables and parameters. Buffer overflows usually take place in stack region, because the security critical control flow data such as return address and function pointers are there and overwriting these data could cause program to change its behavior.

We are aware that the function local variable's symbol information is not available in the program symbol tables. When the wrapper functions refer these local buffers, they cannot find the relevant address and size information. Fortunately, based on the fact that once overflow occurs local buffers would smash the frame pointer, we can hence use the saved frame pointer as Libsafe [2] did to act as the upper bounds when program writes to destination address in stack. Again, if the destination is not enough to hold the source data, then the buffer overflow attack is detected, prevented and reported to users.

#### IV. EVALUATION

We have conducted a series of experiments to evaluate the effectiveness and performance of LibsafeXP. All the experiments were carried out on two 2.4G Pentium processors with 1G RAM running Linux kernel 2.6.3. The tested programs were compiled by gcc 3.2.3 with default option, and dynamically linked with glibc 2.3.5.

##### A. Effectiveness

We extended the test-bed developed by Wilander et al. [16] to evaluate the buffer overflow prevention ability of LibsafeXP. In the original test-bed, they implemented 20 techniques to overflow a buffer located on `.stack`, `.data` or `.bss` sections. As LibsafeXP has guarded the heap variables, we added some special attacks (such as `double free`) which target heap data in the original test-bed, so as to test the buffer overflow prevention against heap buffers.

##### A.1 Protection against stack buffers

Our approach to protect against stack smash utilizes stack checking technique as Libsafe, and as expected, we achieve similar results to it. Specifically, the attacks to overflow stack pointer, frame pointer, function parameter pointer and `longjmp` buffer parameter were successfully detected, but to overflow local function pointer and `longjmp` local buffer were missed. This is because using frame pointer as a boundary is not so accurate. In other words, our approach does no worse than Libsafe on stack buffer overflow protections.

##### A.2 Protection against global and heap buffers

As our approach has enforced access range to program `.data` and `.bss` global variable, all the attack techniques developed in Wilander's test suite attempting to overwrite these areas were successfully detected and prevented.

For the `.heap` buffer overflows, our security test was to overflow the allocated heap buffers. Also as expected, all the out-of-bounds write were successfully caught and prevented.

##### B. Performance

To test the performance overhead incurred due to LibsafeXP, we first did the micro-benchmark test to measure the overhead

at the function call level, and then measured the overall performance by running 11 typical applications to get the macro-benchmark result. In order to compare our results with others, we chose Libsafe and LibsafePlus to do the evaluation in that our approach is the extension to them.

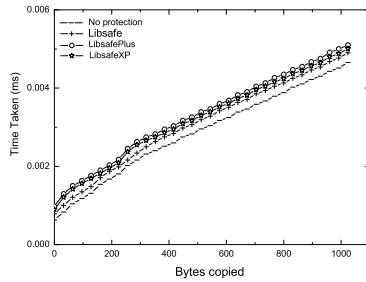
##### B.1 Micro benchmarks

In our approach, the performance overhead imposed on applications mainly arises from the cost of interception and runtime buffer overflow determination in the wrapper functions. Due to this similarity to LibsafePlus, we used LibsafePlus's micro benchmark test program, which is mainly designed to test the two most commonly used string handling functions (i.e., `memcpy` and `strcpy`) and `malloc` and `free` pair, to test our micro benchmarks.

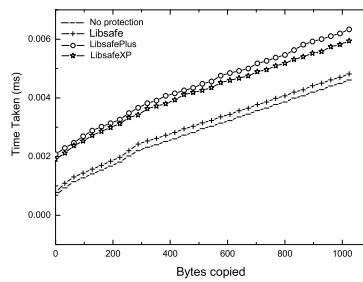
We carried out four tests for different situations: the program running without protection, the program running with Libsafe, the program running with LibsafePlus, and the program running with LibsafeXP. We measured the cost for single `memcpy` and `strcpy` to copy varied bytes to a global, local and heap buffer. The performance results are presented in Figure 7.

As shown in Figure 7(a), the average overhead imposed by `memcpy` on a global buffer is about  $0.7 \mu s$  per-call in LibsafeXP, which looks like better than LibsafePlus ( $0.8 \mu s$ ) because the red-black tree searching is quicker than the liner searching. Libsafe in this case imposes about  $0.4 \mu s$  per-call. When `memcpy` to heap is called, our cost is similar to LibsafePlus. The average overhead of them is about  $1.4 \mu s$  per-call as shown in Figure 7(b). This is expected because both LibsafeXP and LibsafePlus do similar red-black tree searching when determining the size for heap buffers, their results are almost equal. For the cost of `memcpy` to local buffers, the overhead is almost constant and (around  $0.9 \mu s$ ) per-call as shown in Figure 7(c), this is because we use the Libsafe's rough yet quicker approach to determine the local buffer overflows. For LibsafePlus in this case, it adds more performance overhead (about  $1.2 \mu s$ ), this can be explained when determining the local variable size, LibsafePlus will look up the extracted type information and this is sometimes slower than ours and Libsafe.

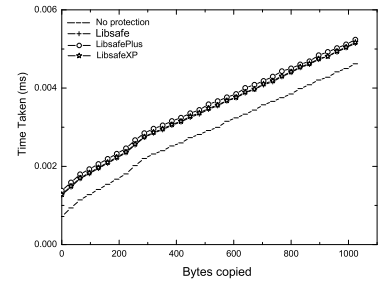
In the implementation of the intercepted `strcpy` function, we replace the original `strcpy` call with `memcpy` as LibsafePlus did. This is based on the fact that copying with `memcpy` is 6 to 8 times faster than `strcpy` for large buffers [14]. As shown in Figure 7(d), 7(e) and 7(f), when small buffer is copied by `strcpy`, the cost overhead is a little more than the original `strcpy` function. But for big buffers it imposes nearly no performance overhead and even improves the original `strcpy` function's performance. Note that the performance overhead of `strcpy` to heap buffer almost equals to LibsafePlus as shown in Figure 7(e). This is explained in the above test of `memcpy` to heap buffers. For the test result of `strcpy` to global buffers, LibsafeXP is a slightly slower than LibsafePlus. This is because LibsafePlus stores the global variable's size in a table whereas we use the *GT*. Just for this case in the test, it appears not as



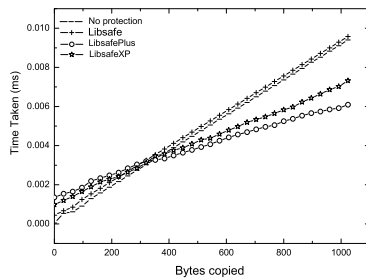
(a) mempcpy() to a global buffer



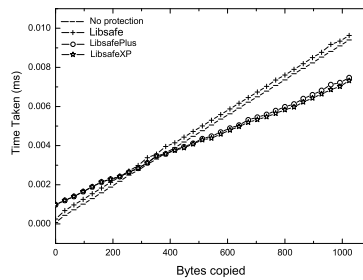
(b) mempcpy() to a heap buffer



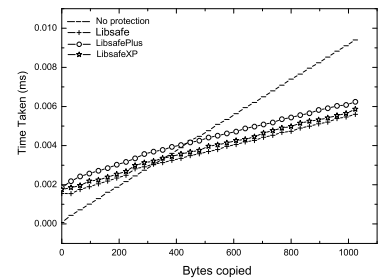
(c) mempcpy() to a local buffer



(d) strcpy() to a global buffer



(e) strcpy() to a heap buffer



(f) strcpy() to a local buffer

Fig. 7. Micro benchmark

good as LibsafePlus; but generally speaking, LibsafeXP runs quicker than LibsafePlus when `strcpy` to global buffer, because average tree searching (time  $O(\log(n))$ ) is quicker than average linear table searching (time  $O(n)$ ).

For the overhead of `malloc` and `free` pair, since the cost comes from the insertion and deletion for the newly allocated symbol in *HT*, we measured the time overhead of these two functions when pre-allocating the number of buffers (which are already in *HT*) from  $2^5$  to  $2^{21}$ . As depicted in Figure 8, the time cost grows almost logarithmically with the number of buffers pre-allocated in the red-black tree. This is because the insertion and deletion of nodes in a red-black tree is time  $O(\log(n))$ .

## B.2 Macro benchmarks

Next, we tested the macro benchmarks of LibsafeXP by running 11 typical open source programs, which were selected by LibsafePlus. Table I describes the benchmark program and the test metric. The related result is described in Table II. We could find that the performance of LibsafeXP seems very similar to LibsafePlus, because they both adopt the similar buffer overflow prevention algorithm. And for some applications (e.g. for Bison), sometimes it runs better than LibsafePlus. This is also expected because when determining the size of global, heap and local variable, LibsafeXP costs a little less than LibsafePlus. So

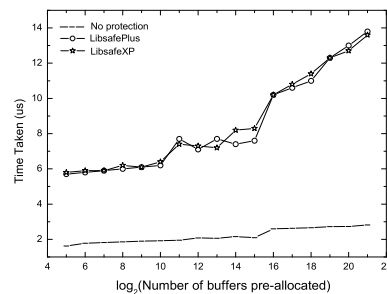


Fig. 8. Micro benchmark of malloc and free pair.

in general, LibsafeXP performs a slightly better than LibsafePlus. As for Libsafe, due to only guarding stack buffers and light cost of size determination, it imposes about less than 5% performance overhead on applications.

## V. ADVANTAGES AND LIMITATIONS

Applying LibsafeXP to protect software against buffer overflow attacks has several benefits. Firstly, LibsafeXP has guarded all three types of buffer overflows, i.e., global buffers, dynamic

TABLE I  
 DESCRIPTION OF APPLICATION BENCHMARKS

| Applications    | What was measured   |
|-----------------|---|
| Apache-2.0.48   | Connection rate, response time and error rate while requesting large files from the web server. |
| Bison-1.875     | Time to parse large grammar files 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 loop-back interface.                                |
| OpenSSL-0.9.7   | Time to sign and verify with 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.                 |
| Monkey-0.8.5-1  | Connection rate, response time and error rate while requesting large files from the web server. |
| Ccrypt-1.2-2    | Time to decrypt a large file encrypted using crypt.   |
| Tar-1.13.25-4   | Time to compress and bundle a large set of files.   |

TABLE II  
 PERFORMANCE OVERHEADS OF LIBSAFEPLUS AND LIBSAFEEXP

| Applications    | Libsafe | LibsafePlus | LibsafeXP |
|-----------------|---------|-------------|-----------|
| Apache-2.0.48   | 1.0X    | 1.0X        | 1.0X      |
| Bison-1.875     | 1.2X    | 2.4X        | 2.1X      |
| Enscript-1.6.1  | 1.1X    | 1.3X        | 1.3X      |
| Hypermail-2.1.8 | 1.0X    | 1.1X        | 1.1X      |
| OpenSSH-3.7.1   | 1.0X    | 1.0X        | 1.0X      |
| OpenSSL-0.9.7   | 1.0X    | 1.0X        | 1.0X      |
| Gnupg-1.2.3     | 1.0X    | 1.0X        | 1.0X      |
| Grep-2.5        | 1.1X    | 1.3X        | 1.3X      |
| Monkey-0.8.5-1  | 1.1X    | 1.3X        | 1.3X      |
| Ccrypt-1.2-2    | 1.0X    | 1.0X        | 1.0X      |
| Tar-1.13.25-4   | 1.0X    | 1.0X        | 1.0X      |

allocated heap buffers, and stack buffers, as a result it can detect most of the out-of-bounds buffer overflow attacks. Besides, as shown in our experiment, LibsafeXP incurs very low performance overhead (about 10 percent on average). Moreover, as every buffer's dereference is enforced to check against its maximum access range, there is almost no false positives. Further, LibsafeXP is a transparent and very practical solution in applications. There is no need of the program source code, which would be very useful to protect the software whose source code is unavailable, and no need of any other extra information except the symbol tables.

Also, LibsafeXP has some limitations. It should be noted that one circumstance that LibsafeXP will be bypassed (false negative), though it is not included and tested in the Wilander's test suite, is when a function pointer is the member of certain struct which contains buffer variables above the function pointer as shown in Figure 9. Because we cannot accurately calculate the access range of the struct member buf (it is the remained size of the whole struct type), and when the buf

is overflowed, it would corrupt the pointer variable `foo_b` or even `foo_c`, but cannot corrupt the `foo_d`. We will address this false negative in future work.

Another limitation of our approach is that if the program has stripped the symbol tables for some reasons (e.g., to reduce the executable file size), LibsafeXP would not work for global buffer access range determination. But this circumstance seldom occurs, many programs when built normally have the symbol tables.

In addition, as LibsafeXP relies on the interception of buffer-related wrapper functions to enforce bounds checking, it is possible that certain buffer overflows simply do not use these functions. For this case, LibsafeXP also will not be able to prevent the buffer overflow (this limitation exists in almost all the library interposition based approach).

At last, if the program is statically linked, our approach will not work too. We are planning to look for some alternative techniques, e.g., binary rewriting, to remedy this. However, we should note the statically linked applications are not too much used if we consider Xiao's study that 99.78% applications in Unix platform are dynamically linked [6].

## VI. RELATED WORK

A large number defensive methods against buffer overflow attacks have been investigated, including static analysis (e.g., [11]), compiler extensions (e.g., [7], [10]), safe library functions (e.g., [2], [14]), execution monitoring (e.g., [12], [15]), intrusion detections (e.g., [17]), randomizing code/space transformations (e.g., [18], [19], [20]) and so forth. All of them were demonstrated useful. In this section, we do not intend to cover all these techniques, and instead we mainly discuss these safe library function approaches.

Since unsafe `glibc` functions is the major cause leading to buffer overflow attacks, rewriting these functions would be a promising approach. On the basis of this idea, Baratloo et al. presented Libsafe and Libverify [2] to defend against stack smashing attacks. Libsafe provides secure calls to the buffer re-

```

struct some_global_struct{
    ...
    int (*foo_a());
    char buf[N];
    ...
    int (*foo_b());
    ...
    int (*foo_c());
    ...
};
int (*foo_d());
    
```

Fig. 9. Undetectable case of LibsafeXP

lated `libc` functions, and Libverify uses a similar approach to StackGuard [8] by verifying the function return address before use. Both of the two approaches were powerful, but they only focused on the prevention of stack buffers.

To overcome the limitations of Libsafe and Libverify, Avijit et al. extended it and presented two tools named TIED and LibsafePlus [14] to cooperate for a more wider buffer overflow preventions. TIED extracts buffer's size from program debugging information which contains not only the global buffer's size but also the local variables, to help LibsafePlus determine buffer overflows. LibsafePlus is a very promising approach for buffer overflow preventions, but it heavily relies on the program debugging information.

Considering the fact that for LibsafePlus the debugging information is usually unavailable in the released software, and Libsafe only provides limited scope checking, we extend and integrate their approaches to implement our tool LibsafeXP. Although LibsafeXP looks like LibsafePlus, they are based on different knowledge. LibsafePlus tries to accurately determine all the buffer variable access range by retrieving relevant information from program debug section. However, LibsafeXP achieves this by retrieving the symbol information from binary image. Despite LibsafeXP cannot accurately determine the access range of every buffer, it can still prevent most of buffer overflow attacks.

## VII. CONCLUSION AND FUTURE WORK

In this paper, we have presented on the basis of different buffer types we can take different approaches to combat buffer-overflow attacks. For global variables, we rely on the symbol section of the protected ELF executable file, and extract those information for our bounds checking. For heap-based buffers, we intercept those related buffer operation APIs and track the allocated buffer's size. For stack-based local variables, as there is no such information available, we use the frame pointer as the upper bound. We have built a practical tool, LibsafeXP, to guard against almost all the three types of buffer overflows dynamically and transparently. Overall speaking, our approach is simple and efficient.

Future work of our approach is to extend it to other platforms, such as Windows for PE files. In fact, our approach does not rely on some special support. Whenever we can intercept those

buffer related functions dynamically and the program symbol table is available, it would work.

## VIII. ACKNOWLEDGEMENTS

We thank Kumar Avijit, John Wilander, Erwang Guo, Nai Xia, Yi Wang, and anonymous reviewers for their support during this work. This research was supported in part by Chinese National Science Foundation under grant 60373064 and Chinese National 863 High-Tech Program under grant 2003AA144010.

## REFERENCES

- [1] NIST, National Vulnerability Database. <http://nvd.nist.gov>, May 2006.
- [2] A. Baratloo, N. Singh, and T. Tsai. Transparent run-time defense against stack smashing attacks (Libsafe -Libverify). In *Proc. of 2000 USENIX Annual Technical Conference*, San Diego, California, USA, Jun. 2000.
- [3] TIS (Tool Interface Standards). Executable and Linkable Format Specification, UNIX System Laboratories.
- [4] T. Cormen, C. Stein, R. Rivest and C. Leiserson. *Introduction to Algorithms*. MIT Press, second edition. 2002.
- [5] P. Phantasmagoria. Glibc Malloc Exploitation Techniques. <http://packetstormsecurity.org/papers/attack/MallocMaleficarum.txt>, 2005.
- [6] Z. Xiao. An Automated Approach to Software Reliability and Security. Invited Talk, Department of Computer Science, University of California at Berkeley, 2003.
- [7] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proc. of the 7th USENIX Security Symposium*, Jan. 1998.
- [8] H. Ozdoganoglu, T. Vijaykumar, C. E. Brodley, A. Jalote. SmashGuard: A Hardware Solution to Prevent Security Attacks on the Function Return Address. Technical Report, Purdue University, Dec. 2002.
- [9] R. Lee, D. Karig, J. McGregor and Z. Shi. Enlisting Hardware Architecture to Thwart Malicious Code Injection. *Security in Pervasive Computing*, Boppard, Germany, Mar. 2003.
- [10] O. Ruwase and M. S. Lam. A Practical Dynamic Buffer Overflow Detector. In *Proc. of the 11th Annual Network and Distributed System Security Symposium*, Feb. 2004.
- [11] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proc. of the 2001 USENIX Security Symposium*, Washington DC, USA, Aug. 2001.
- [12] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proc. of the 12th Annual Network and Distributed System Security Symposium*. 2005.
- [13] R. Jones and P. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Proc. of the International Workshop on Automatic Debugging*, May 1997.
- [14] K. Avijit, P. Gupta, D. Gupta. TIED, LibsafePlus: Tools for Runtime Buffer Overflow Protection. In *Proc. of the 13th USENIX Security Symposium*, Aug. 2004.
- [15] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *Proc. of the 11th USENIX Security Symposium*, Aug. 2002.
- [16] J. Wilander and M. Kamkar. A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention. In *Proc. of the 10th Network and Distributed System Security Symposium*, Feb. 2003.
- [17] H. Feng, J. Giffin, Y. Huang, S. Jha, W. Lee, and B. Miller. Formalizing Sensitivity in Static Analysis for Intrusion Detection. In *Proc. of IEEE Symposium on Security and Privacy*, May 2004.
- [18] E.G. Barrantes, D.H. Ackley, S. Forrest, T. Palmer, D. Stefanovic and D. Zoviet. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM Conference on Computer and Communication Security*, Oct. 2003.
- [19] G.S. Kc, A.D. Keromytis, V. Prevelakis. Countering Code-Injection Attacks With Instruction-Set Randomization. In *Proceedings of the 10th ACM Conference on Computer and Communication Security*, Oct. 2003.
- [20] S. Bhatkar, R. Sekar and D. DuVarney. Efficient Techniques for Comprehensive Protection from Memory Error Exploits. In *Proc. of the 14th USENIX Security Symposium*, Aug. 2005.