Type-Assisted Dynamic Buffer Overflow Detection

Kyung-suk Lhee and Steve J. Chapin
Center for Systems Assurance
Syracuse University
{klhee, chapin}@ecs.syr.edu

Abstract

Programs written in C are inherently vulnerable to buffer overflow attacks. Functions are frequently passed pointers as parameters without any hint of their sizes. Since their sizes are unknown, most run time buffer overflow detection techniques instead rely on signatures of known attacks or loosely estimate the range of the referenced buffers. Although they are effective in detecting most attacks, they are not infallible. In this paper we present a buffer overflow detection technique that range checks the referenced buffers at run time. Our solution is a small extension to a generic C compiler that augments executable files with type information of automatic buffers (local variables and parameters of functions) and static buffers (global variables in data / bss section) in order to detect the actual occurrence of buffer overflow. It also maintains the sizes of allocated heap buffers. A simple implementation is described, with which we currently protect vulnerable copy functions in the C library.

1 Introduction

Programs written in C are inherently vulnerable to buffer overflow attacks. C allows primitive pointer manipulation, which is usually necessary for array operation because C has no first-class array type. For example, functions are passed the pointers as array parameters. To ensure that buffers are not overflowed, it is the programmers' responsibility to explicitly bounds check the buffers. In practice, bounds checking is often neglected or cannot be done since arrays are often passed without any hint of their sizes. Many copy functions in the C library such as **strcpy**(dest, src) are vulnerable this way, making them a popular point of attack.

Various types of buffer overflow attacks have been discovered. The simplest and the most popular among them is the stack smashing attack [1]. The stack smashing attack overflows a buffer to overwrite the return address of a function, so that the return address points to the attack code that is injected into the stack by the attacker, rather than the legitimate call point. The control flow is directed to the attack code when the function returns. The stack smashing attack exploits the stack configuration and the function call mechanism. There are other types of buffer overflow attacks that exploit data structures in the heap as well as in the stack. A survey on various types of attacks is found in [7].

There are several run time solutions that are highly effective without much run time overhead. However, most of them rely on the signatures of known attacks (or the loosely estimated range of the referenced buffers) rather than the detection of actual occurrence of buffer overflow, since sizes of buffers are unknown at run time. As a result, buffers can still be overflowed and they are vulnerable to attacks that do not show such signatures. Moreover, they are mostly built to defend against the stack smashing attack and focus only on its signatures. Buffer overflow techniques that can bypass those run time solutions are found in [4, 15, 5, 11, 21, 16, 18], and are discussed in Section 3.

Our goal is to increase the level of security in computing systems by devising a run time solution that is less dependent on attack signatures. We propose a solution that range checks the buffers at run time. Our solution is a small extension to the GNU C compiler that augments executable files with type information of automatic buffers (local variables and parameters of functions) and static buffers (global variables in data / bss section) in order to detect the actual occurrence of buffer overflow. It also maintains the sizes of allocated heap buffers. Currently we use it to perform range checking within the vulnerable copy functions in the C library.

2 Related work

2.1 StackGuard

The stack smashing attack overwrites the buffer, the return address and everything in between. StackGuard [6] is a GNU C compiler extension that inserts a *canary* word between the return address and the buffer so that an attempt to alter the return address is detected by inspecting the canary word before returning from a function[1]. Programs needs to be recompiled with StackGuard to be protected.

2.2 StackShield

StackShield [19] is also a GNU C compiler extension that protects the return address. When a function is called StackShield copies away the return address to a non-overflowable area, and restores the return address upon returning from a function. Even if the return address on the stack is altered, it has no effect since the original return address is remembered. As with StackGuard, programs needs to be recompiled.

2.3 Libsafe

Libsafe [3] is an implementation of vulnerable copy functions in C library such as **strcpy**(). In addition to the original functionality of those functions, it imposes a limit on the involved copy operations such that they do not overwrite the return address. The limit is determined based on the notion that the buffer cannot extend beyond its stack frame. Thus the maximum size of a buffer is the distance between the address of the buffer and the corresponding frame pointer. Libsafe is implemented as a shared library that is preloaded to intercept C library function calls. Programs are protected without recompilation unless they are statically linked with the C library. Libsafe protects only those C library functions whereas StackGuard and StackShield protect all functions.

[1] The name is derived from the coal mining practice of taking a canary down with the workers. The canary was more sensitive to poisonous gas than humans, so examining the state of the canary could reveal a dangerous buildup of poison gas.

2.4 Solar Designer's non-executable stack patch

The stack smashing attack injects an attack code in the stack, which is executed when the function returns. One of the core features of the Solar Designer's Linux kernel patch [17] is to make the stack segment non-executable. This patch does not impose any performance penalty nor does it require program recompilation (except for the operating system kernel).

2.5 PaX

PaX [14] is a page-based protection mechanism that marks data pages non-executable. Unlike Solar Designer's stack patch, PaX protects heap as well as stack. Since there is no execution permission bit on pages in x86 processor, PaX overloads the supervisor/user bit on pages and augments the page fault handler to distinguish the page faults due to the attempts to execute code in data pages. As a result, it imposes a run time overhead due to the extra page faults. PaX is also available as a Linux kernel patch.

2.6 Runtime array bounds checking

The pointer and array access checking technique by Austin et al. [2] is a source-to-source translator that transforms C pointers into the extended pointer representation called *safe pointer*, and inserts access checks before pointer or array dereferences. The safe pointer contains fields such as the base address, its size and the scope of the pointer. Those fields are used by the access check to determine whether the pointer is valid and is within the range. Since it changes the pointer representation, it is not compatible with existing programs.

The array bounds and pointer checking technique by Jones and Kelly [10] is an extension to the GNU C compiler that imposes the access check on C pointers and arrays. Instead of changing the pointer representation, it maintains a table of all the valid storage objects that holds such informations as the base address and size etc. The heap variables are entered into the table via a modified **malloc**() function and deleted from the table via a modified **free**() function. Stack variables are entered into / deleted from the table by the constructor / destructor function, which is inserted inside a function definition at the point a stack variable enters / goes out of the scope. The access check is done by substituting the pointer and array operations with the functions that perform bounds check using the table in addition to the original operation. Since native C pointers are used, this technique is compatible with existing programs.

The obvious advantage of array bounds checking approaches are that they completely eliminate buffer overflow vulnerabilities. However, these are also the most expensive solution, particularly for pointer- and array-intensive programs since every pointer and array operation must be checked. This may not be suitable for a production system.

2.7 Static analysis of array bounds checking

The integer range analysis by Wagner et al. [20] is a technique that detects possible buffer overflow in the vulnerable C library functions. A string buffer is modeled as a pair of integer ranges (lower bound, upper bound) for its allocated size and its current length. A set of integer constraints is predefined for a set of string operations (e.g. character array declaration, vulnerable C library functions and assignment statements involving them). Using those integer constraint, the technique analyzes the source code by checking each string buffer to see whether its inferred allocated size is at least as large as its inferred maximum length.

The annotation-assisted static analysis technique by Larochelle and Evans [12] based on LCLint [8] uses semantic comments, called annotations, provided by programmers to detect possible buffer overflow. For example, annotations for **strcpy**() contain an assertion that the destination buffer has been allocated to hold at least as many characters as are readable in the source buffer. This technique protects any annotated functions whereas the integer range analysis only protects C library functions.

Generally, a pure compile-time analysis like the above can produce many false alarms due to the lack of run time information. For example, **gets**() reads its input string from **stdin** so the size of the string is not known at compile time. For such a case a warning is issued as a possible buffer overflow. In fact, all the legitimate copy operations that accept their strings from unknown sources (such as a command line argument or an I/O channel) are flagged as possible buffer overflows (since they are indeed vulnerable). Without further action, those vulnerabilities are identified but still open to attack.

3 Exploitation techniques

The exploitation techniques presented in this section are exemplary and they can bypass some of the run-time defensive techniques. While the stack smashing attack can exploit just a single vulnerable **strcpy**(), these techniques usually require more vulnerabilities in the program that are less likely to be found in real world. Nonetheless, they identify different kinds of vulnerabilities that may not be protected by current defensive techniques.

Although we can apply multiple defensive techniques for added protection, these exploitation techniques can also be used in tandem to produce more sophisticated attacks that are more difficult to detect. However, none of these exploits are possible if buffer overflow is prevented. If programmers rely on C library functions to overflow buffers, then our current implementation can detect and prevent such attacks.

3.1 Return-into-libc

The return-into-libc exploit [18, 13] overflows a buffer to overwrite the return address as the stack smashing attack does. However it overwrites the return address with the address of C library function such as **system**(). Since it uses an existing code rather than a shellcode, Solar Designer's non-executable stack patch or PaX cannot detect this [2].

3.2 Other code pointers

Code pointers other than the return address can also be overwritten, such as a function pointer variable [5], a pointer to a shared library function in the global offset table [21], the table of pointers to destructor functions [15], or a C++ virtual function pointer [16]. Exploits that alter those code pointers and not the return address can bypass StackGuard, StackShield and Libsafe.

[2] They both provide guards against return-into-libc attacks, but they can still be exploited. For example, we can use the procedure linkage table entry of system() instead of the address of system() to bypass the stack patch (where the address of system() can contain zero bytes) or PaX (where the address of system() are unknown in advance due to the random mapping of shared libraries).

3.3 Malloc() overflow

The **malloc**() overflow [11] exploits the heap memory objects allocated via the memory allocator in the GNU C library. The memory allocated by **malloc**() not only includes the user requested block but also the data used to manage the heap (size of the block, pointer to other blocks and the like). The vulnerability is that a heap variable can be overflowed to overwrite those management data. Exploits based on this technique can bypass stack-based defensive techniques such as StackGuard, StackShield, Libsafe and Solar Designer's stack patch.

3.4 Indirect overflow via pointer

The indirect overflow via pointers [4] overflows a buffer to overwrite a pointer, which is used subsequently to overwrite a code pointer. With this technique it is possible to overwrite the return address without altering the StackGuard canary word. It is also possible to overwrite a memory area that is far from the overflowed buffer. Bulba and Kil3r [4] gives examples that bypass StackGuard, StackShield and Solar Designer's stack patch.

4 Overview of Our Approach

Array bounds checking is a direct way to detect buffer overflows, but it is difficult to do because the type information (hence the size) of buffers are not available in binary files except as optional debugging information in the symbol table. To enable range checking on buffers at run time, introduce an intermediary step in the compilation that emits an additional data structure into the binary file. This data structure describes the types of automatic buffers and static buffers. These types are known at compile time, so our data structure is complete for describing automatic and static buffers (there are two exceptions in which size of an automatic buffer cannot be determined at compile time, which are discussed in Section 6.). For example, buffers in a **struct** variable are safe from each other as depicted in Figure 1.

For dynamically allocated (heap) objects, we maintain a table that tracks those objects and their sizes. Range checking is then done by looking up those data structures at run time. We use those data structures to perform range checking of arguments to the vulnerable string functions in the C library.

```
struct mybuf {
    char buf1[32];
    void (*fptr)();
    char buf2[32];
};
```

Figure 1: A struct containing two string buffers and a function pointer.

Regardless of which of these types of attack is attempted, buffers have to be overflowed in some way for the attacks to succeed. Since our approach prevents buffers from being overflowed it is insensitive to which attack was chosen. To truly protect from all the possible buffer overflow attacks in the most efficient way, we need to identify all and only those vulnerable points in the program. However it cannot be done without extensive source code analysis. For the current implementation we protect only C library functions. We believe that it is useful as a stand-alone protection system and can be easily extended with compile time analysis to remove bounds checking on "known-safe" function calls.

Our data structure for describing buffers is similar to the type table in the Process Introspection Library [9], which describes data types of savable memory blocks in order to checkpoint and restart processes in a distributed, heterogeneous environment. The Process Introspection Library also deduces the type of a heap allocated memory block, a capability that we currently lack, but which can be similarly added.

5 Implementation

We implemented a prototype by extending the GNU C compiler on Linux. We augment each object file with type information of automatic and static buffers, leaving the source code intact. Specifically, we intercept the output of the **gcc** preprocessor and append to it a data structure describing the type information. The augmented file is then piped into the next stage to complete the compilation.

The type information of buffers are read by precompiling the (preprocessed) source file with debugging option turned on, and parsing the resulting stabs debugging statements. From the stabs debugging statements we generate a type table, a data structure that associates the address of each function with the information of the function's automatic buffers (their sizes and offsets to the stack frame). The type table also contains the addresses of static buffers declared in the source file and their sizes. This way, each object file carries information of its automatic / static buffers independently. The type table is kept under a static variable so objects can be linked without any conflict. To make those type tables visible at run time, each object file is also given a constructor function[3]. The constructor function associates its type table with a global symbol. This process is illustrated in Figure 2.

Our implementation is transparent in the sense that source files are unmodified, and programs are compiled normally using the supplied makefile in the source distribution. It is also highly portable because the augmentation is done in the source level. Because type tables in the object files are assembled at run time, objects can be linked both statically and dynamically.

The range checking is done by a function in a shared library. The range checking function accepts a pointer to the buffer as the parameter, and finds the size of the buffer according to the following algorithm (for an automatic buffer; locating a static buffer is straightforward). Figure 3 illustrates this.

1. Locate the stack frame of the buffer by chasing down the saved frame pointer,
2. Retrieve the return address of the next stack frame to find out who allocated the stack frame,
3. Locate the function who allocated the stack frame by comparing the return address with function addresses in the type table,
4. Locate the buffer of the function by comparing the buffer address with offsets in the table + frame pointer value,
5. The size of the buffer (or the size of a field if it is a **struct** variable) is returned

The shared library also maintains a table of currently allocated heap buffers by intercepting **malloc**(), **realloc**() and **free**() functions (a feature of the dynamic memory allocator in GNU C library). For the heap buffers, the size of the referenced buffer is determined as the size of the allocated memory block. Without type information it is currently unable to determine the exact size, which may be significant as evident in Figure 1. We implemented a shared library that is preloaded to intercept vulnerable copy functions in C library to perform range checking.

[3]This is a **gcc** feature; constructor functions run before **main**() does.

6 Limitations

There are two cases in which we cannot determine the size of automatic buffers; stack buffers dynamically allocated with **alloca**(), and variable-length automatic arrays (a GNU C compiler extension). They are limitations inherent in our solution.

The current implementation is also unable to determine the type of function scope static variables since they are not visible outside the declared function. For the same reason, we cannot protect buffers declared in a function scope functions (nested functions, another GNU C compiler extension). Although those symbols are not visible in the source file, they are visible in the compiled file. Thus, this problem is not inherent in our solution. In order to fix the problem, we need to express the type table in assembly language and append it to the compiled file. The current prototype is done at the source level, augmenting the type table written in C at the (preprocessed) source file.

7 Experiments

To estimate the run time overhead incurred by the range checking for each C library function, we ran a small program that calls each C library function in a tight loop (loop count is 100,000,000).

The range checking (done in C library wrapper functions) involves the following steps.

1. Intercept a C library function
2. Retrieve the buffer size by type table lookup
3. Compare the buffer size with the source string length
4. Call the C library function

The overhead is thus mostly attributed to 1) the time taken for type table lookup (in order to find the size of the buffer), and 2) the time taken for calling **strlen**()

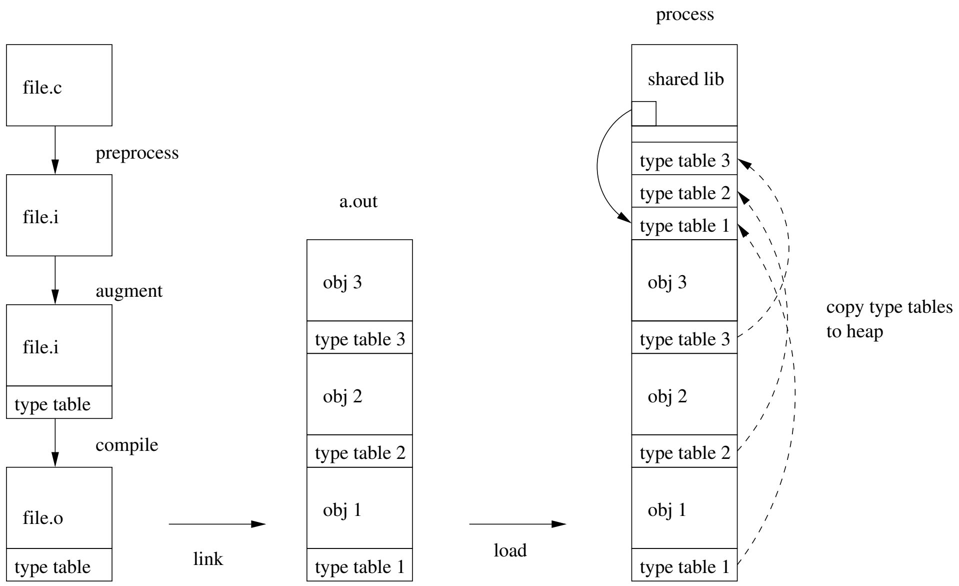

Figure 2: Compilation process, the executable file and the process

```
main(int argc, char **argv)
{
  char buf[32];
  func(buf, argv[1]);
}

func(char *dst, char *src)
{
  strcpy(dst, src);
}
```

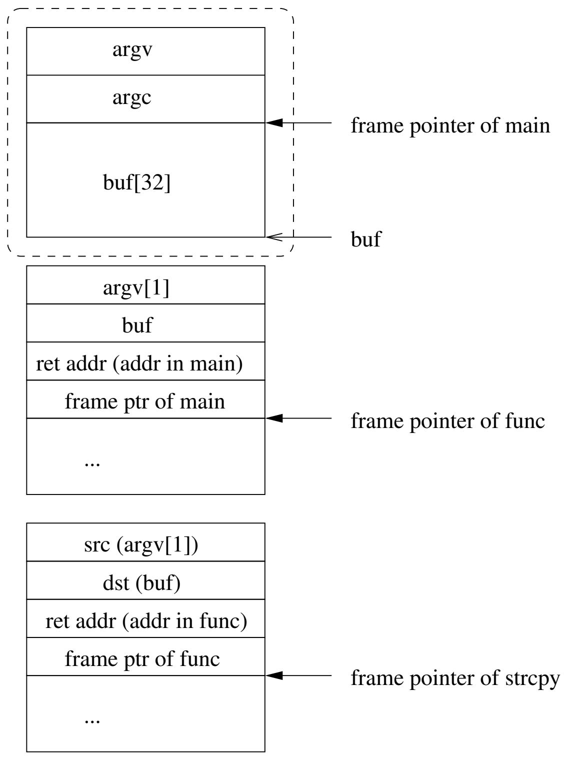

Figure 3: The stack frame of the buffer is found by comparing the address of the referenced buffer and saved frame pointers in the stack (address of the buf should be less than its frame pointer since it is a local variable). The first frame (in dashed box) is the frame for the buffer. The return address of the next frame is used to locate the entry in the type table (address of main), which is used subsequently to find the size of the buffer. It is assumed that the stack grows down, and the address of the buffer is that of its least significant byte (little endian architecture).

(in order to check whether the buffer size is enough) if needed. According to these two criteria, the C library wrapper functions are roughly partitioned into three classes; 1) functions such as **strcpy**() require the call to **strlen**() in addition to type table lookup, 2) functions such as **memcpy**() needs only type table lookup, and 3) functions such as **strncpy**() may or may not require **strlen**() depending on whether the buffer size is greater or equal to the size parameter or not.

Each function was tested 8 times with varying string length (8, 16, 32, 64, 128, 256, 512 and 1024). Our test were performed on a pc with AMD Duron 700MHz running Redhat Linux 6.2. Figure 4 shows the result.

The table lookup is done by binary search, so the overhead incurred by the table lookup will increase logarithmically as the number of functions and variables in the executable file increases. In sum, the micro test shows the worst case scenario and we expect better performance in real programs (which will, after all, do some useful work besides just calling C-library string functions). Figure 5 is the result of testing three programs (enscript 1.6.1, tar 1.13 and java 1.3.0), and shows the increase in size of executable files due to the augmented type table, the number of calls to C library functions that those program made during the test run, and the run time. Overhead in the macro test is in the range of 4-5% for substantial runtimes, with the short java test showing a 20% overhead (note that the absolute runtime overhead is minimal).

8 Conclusions and future work

Although many solutions have been proposed, buffer overflow vulnerabilities remain a serious security threat. Pure static analysis techniques can identify the vulnerable points in a program before the program is deployed, but cannot eliminate all vulnerabilities. We proposed a run-time buffer overflow detection mechanism that is efficient, portable, and compatible enough with existing programs to be practical. The value of our work is that it can catch some of the attacks that other run-time solutions cannot. We believe that our work is not only useful as a stand-alone protection system but also can be complementary to other solutions. We plan to extend our work to include static analysis technique in order to be able to selectively perform the range checking.

References

[1] AlephOne. Smashing the stack for fun and profit. *Phrack*, 7(49), Nov. 1996.

[2] T. M. Austin, S. E. Breach, and G. S. Sohi. Efficient detection of all pointer and array access errors. In *ACM SIGPLAN 94 Conference on Programming Language Design and Implementation*, June 1994.

[3] A. Baratloo, N. Singh, and T. Tsai. Transparent run-time defense against stack smashing attacks. In *Proceedings of the 2000 USENIX Annual Technical Conference*, pages 251–262, San Jose, CA, June 2000. USENIX.

[4] Bulba and Kil3r. Bypassing stackguard and stackshield. *Phrack*, 10(56), May 2000.

[5] M. Conover and w00w00 Security Team. w00w00 on heap overflows. *http://www.w00w00.org/files/articles/heaptut.txt*, Jan. 1999.

[6] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and QianZhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, pages 63–77, San Antonio, TX, Jan. 1998. USENIX.

[7] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *Proceedings DARPA Information Survivability Conference and Exposition*, pages 119–129, Hilton Head, SC, Jan. 2000.

[8] D. Evans, J. Guttag, J. Horning, and Y. M. Tan. Lclint: A tool for using specifications to check code. In *SIGSOFT Symposium on the Foundations of Software Engineering*, pages 87–96. ACM, Dec. 1994.

[9] A. J. Ferrari, S. J. Chapin, and A. S. Grimshaw. Heterogeneous process state capture and recovery through process introspection. *Cluster Computing*, 3(2):63–73, 2000.

[10] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in c programs. In *Proceedings of the third International Workshop on Automatic Debugging*, pages 13–26, Sweden, May 1997.

[11] M. Kaempf. Vudo - an object superstitiously believed to embody magical powers. *http://www.synnergy.net/downloads/papers/vudo-howto.txt*.

[12] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, Washington D.C, Aug. 2001. USENIX.

[13] Nergal. The advanced return-into-lib(c) exploits: Pax case study. *Phrack*, 10(58), Dec. 2001.

[14] PaX. https://pageexec.virtualave.net.

[15] J. M. B. Rivas. Overwriting the .dtors section. *http://www.synnergy.net/downloads/papers/dtors.txt*.

[16] Rix. Smashing c++ vptrs. *Phrack*, 10(56), May 2000.

[17] SolarDesigner. Non-executable stack patch. *http://www.openwall.com/linux*.

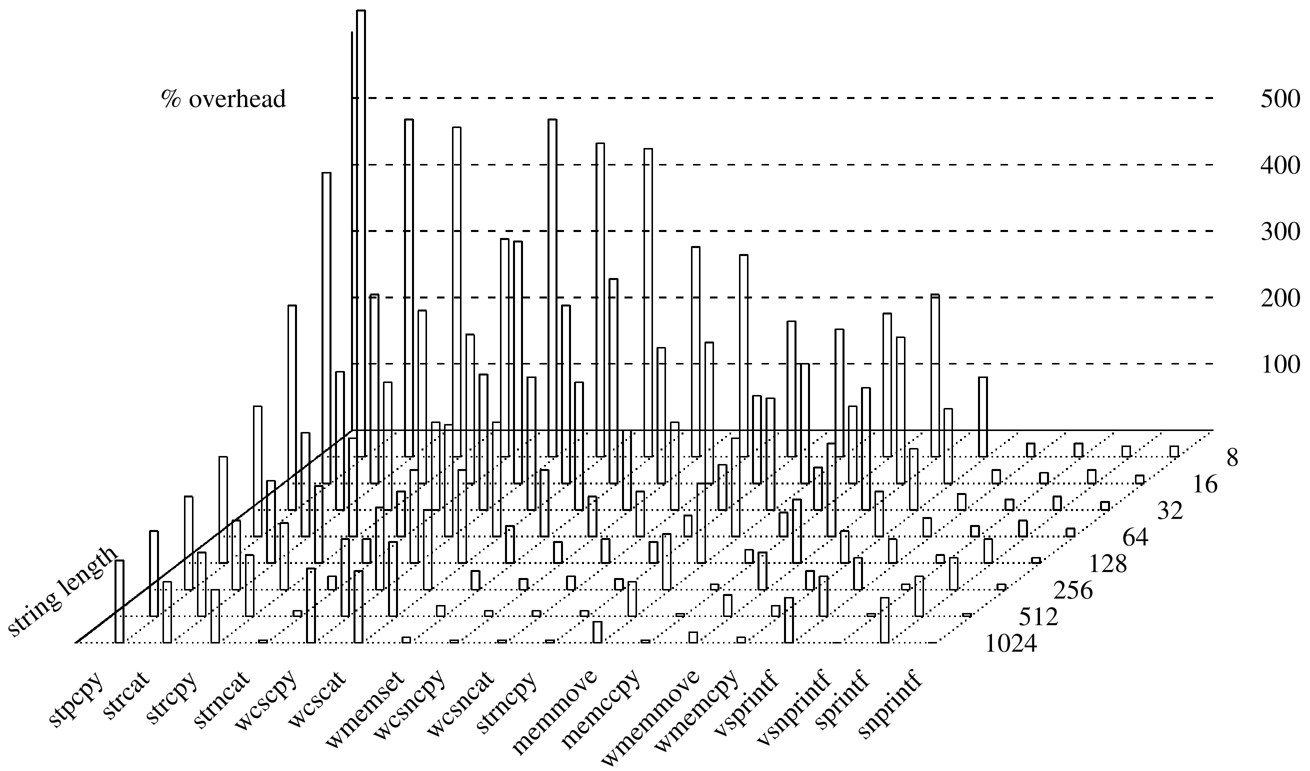

Figure 4: Micro test that shows function overhead by range checking. Each overhead was measured as follows; If an intercepted function is 2.5 times slower then the overhead is 150 percent ($(2.5 - 1) \times 100$).

program	file size (original)	file size (with type table)	libc function count	run time (original)	run time (type table)
enscript	348,503 bytes	368,665 bytes	6,345,760 calls	3 min. 01 sec	3 min. 10 sec
tar	425,958	463,140	23,883	1 min. 12 sec	1 min. 15 sec
java	26,016	28,698	20,552	5 sec	6 sec

Figure 5: Macro test with enscript, tar and java. Enscript printed a text file of size 100Mbytes (to /dev/null). Tar zipped the linux kernel source directory twice. Java ran antlr to parse the GNU C grammar. The run time is the average of ten runs.

[18] SolarDesigner. Getting around non-executable stack (and fix). *Bugtraq mailing list, http://www.securityfocus.com/archive/1/7480*, Aug. 1997.

[19] StackShield. http://www.angelfire.com/sk/stackshield.

[20] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, pages 3–17, San Diego, CA, Feb. 2000.

[21] R. Wojtczuk. Defeating solar designer non-executable stack patch. *Bugtraq mailing list, http://www.securityfocus.com/archive/1/8470*.