# Prevention of Code-Injection Attacks by Encrypting System Call Arguments

Yoshihiro Oyama Akinori Yonezawa
{oyama,yonezawa}@yl.is.s.u-tokyo.ac.jp
Technical Report TR06-01
Department of Computer Science
Graduate School of Information Science and Technology
The University of Tokyo
March 15th, 2006

## Abstract

*Buffer overflow attacks are still a serious threat to the security of software systems. One of the most important classes of buffer overflow attacks is* code-injection attacks*, in which malicious code is injected into a memory area of vulnerable software and eventually executed. In this paper, we propose a simple and effective method for preventing code-injection attacks. The basic idea is to adopt a security-enhanced convention of system call invocations in which system call IDs and arguments are "encrypted" before being passed to the kernel and then "decrypted" at the beginning of in-kernel procedures. This is achieved with a modified standard library and a kernel module. In environments where the method is applied, injected code is likely to fail in executing system calls because their IDs and arguments are likely to be decrypted into meaningless values. We implemented the method on a Linux/IA-32 machine and measured the performance of real applications including GCC, LaTeX, wget, and Apache. Experimental results showed that the incurred performance overhead ranged from 0.1% to 15.0%.*

## 1 Introduction

Buffer overflow attacks [1, 10, 24] are the most common building block of computer viruses, worms, and exploit code. Many of the recent notorious computer viruses including Code Red, Mydoom, and Blaster take advantage of buffer overflow vulnerabilities. A great number of techniques for preventing buffer overflow attacks have been proposed in the last decade. However, none of them has become a complete solution to the buffer overflow problem. Buffer overflow attacks are still a serious threat for the security of software systems in the world.

One of the most important classes of buffer overflow attacks is *code-injection attacks*, which account for a considerable proportion of buffer overflow attacks. A code-injection attack injects a malicious code fragment into a memory area allocated by vulnerable software. It then overwrites a return address or a function pointer with the start address of the injected code. The vulnerable software eventually jumps to the injected code and consequently its control is hijacked by the attacker. If the attacker can inject an arbitrary byte sequence, then arbitrary instructions can be executed with the privileges of the software. A typical behavior of injected code is to attempt to call the interface of services provided by the operating system. On UNIX systems, for example, the primary aim for most injected code is to invoke a critical system call such as `execve`.

In this paper, we propose a simple but effective method for preventing injected code from executing system calls. The proposed method randomizes the relationship between (1) a system call ID and arguments passed from the user-level and (2) those passed to in-kernel system call handlers. On systems where the method is applied, an attacker has difficulty in getting injected code to execute a system call successfully; injected code (and hijacked software) is very likely to crash before causing serious damage to the targeted operating system. The method is achieved using a modified standard library and a kernel module. The library randomizes, or "encrypts," the system call ID and arguments just before the program enters the kernel mode. The kernel module de-randomizes, or "decrypts," them back into the original values. We implemented the modified standard library and the kernel module on a Linux/IA-32 machine, and measured the overhead imposed on the performance of real applications including GCC, LaTeX, wget, and Apache.

The advantages of the proposed method are summarized below:

**Compatibility.** All that users have to do to use our method is to install the kernel module, a modified standard library, and one supportive program. There is no need to recompile the operating system kernel. Moreover, there is no need to modify application programs, so legacy binary software can utilize our defense. In addition, applications running with our method's defense can coexist with those running without it on one operating system.

**Efficiency.** The runtime overhead incurred by our method is small. Inherently, it has almost no influence on the performance of program parts except system calls. The overhead imposed on a system call is also small according to our experimental results.

**Simplicity.** Our method is achieved with a tiny kernel module and a minimal modification to the standard library. Hence, it is not difficult to understand and maintain the implementation.

**Generality.** Our method does not assume specific operating systems or CPU architectures. The only assumptions are that (1) a user-level library can provide a choke point for intercepting all system calls just before control is passed to the kernel and (2) the kernel (or a kernel module) can provide a choke point for intercepting system calls before the control reaches system call handler functions.

The significance of our method lies in providing all the advantages and a moderate degree of security at the same time. As far as we know, there has been no single system that achieves all these advantages. Like other methods, our method is not a silver bullet for buffer overflow problems. However, it is useful because it can prevent many attacks with a minimal overhead and introduction cost.

This paper is organized as follows. We give an overview of code-injection attacks in Section 2. We explain the proposed method in Section 3. Implementation details are given in Section 4. We discuss the method in Section 5. Section 6 presents experimental results. After explaining related work in Section 7, we summarize this paper in Section 8.

For simplicity and readability, the remainder of this paper is described using the terms of the UNIX operating system. However, we expect this method to be applicable to other operating systems too as long as the assumptions on generality described above are satisfied.

## 2 Code-Injection Attacks

The behavior of code-injection attacks is illustrated in Figure 1. An attacker makes the application read a specially crafted byte sequence into a buffer from outside, for example, from a file, a network, or the standard input. The byte sequence contains a malicious code fragment and overruns the limit of the buffer to overwrite control information such as a return address and a function pointer. Thus the application jumps to and executes the injected code.

Injected code damages the operating system through direct manipulation of critical resources or invocation of external programs. For example, an attacker against a vulnerable UNIX system can acquire a shell by injecting a byte sequence that

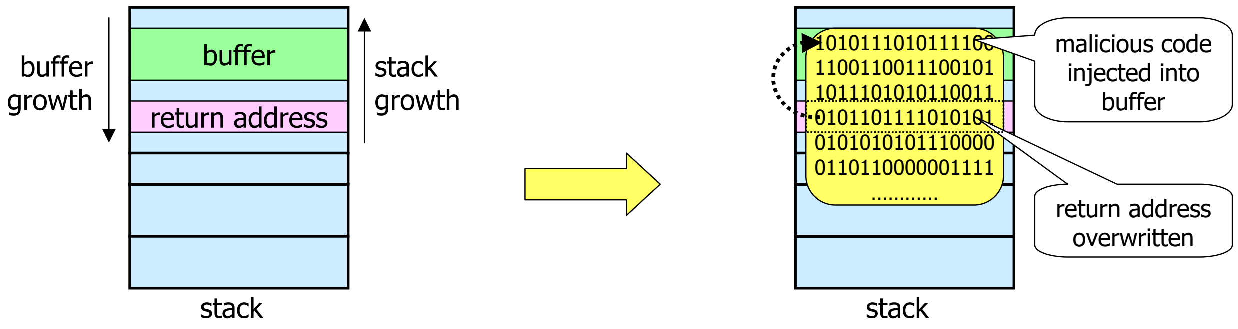

**Figure 1. Code-injection attacks.**

represents the invocation of `execve("/bin/sh", ...)`. An attacker may simply modify a password file instead of executing a shell.

Injected code must invoke services provided by the operating system (e.g., system calls) to perform effective attacks such as starting a shell or opening a file. There are several approaches to do that.

The most straightforward and commonly utilized approach for the code is to invoke system calls by itself. Below, we call it a *direct-invocation attack*. This approach has already appeared in Aleph One's historic document [1]. In direct-invocation attacks, injected code sets a system call ID and arguments in registers and then executes an instruction that raises a software interrupt for system calls. Figure 2 shows an extract of real exploit code based on the approach. The code was developed for exploiting sendmail installed on Linux/IA-32 systems. The last six instructions prepare for the invocation of the `execve` system call. First, `execve`'s ID 11 is set to register eax. Then, arguments are set to register ebx, ecx, and edx. Finally a software interrupt is raised.

Another approach is to jump into such code as is already available in the targeted software and can be utilized for the attack. This is often called a *return-into-libc attack*. If injected code can jump to an important function provided by a standard library (e.g., `system`), it obtains access to services of the operating system kernel.

Direct-invocation attacks require less development effort than return-into-libc attacks. Moreover, their behavior is less affected by the config-

```
unsigned char exploit[]=
...
"\xc7\x45\xf4\x2f\x62\x69\x6e" /*movl "/bin",-12(%ebp)*/
"\xc7\x45\xf8\x2f\x62\x61\x73" /*movl "/bas",-8(%ebp)*/
"\x83\xc2\x68" /* addl "h", %eax */
"\x89\x55\xfc" /* movl %eax, -4(%ebp) */
"\x8d\x45\xf0" /* leal -16(%ebp), %eax */
"\x50" /* push %eax */
"\x8d\x45\xec" /* leal -20(%ebp), %eax */
"\x50" /* push %eax */
"\x8d\x45\xf4" /* leal -12(%ebp), %eax */
"\x50" /* push %eax */
"\x68\x78\x56\x34\x12" /* push $0x12345678 */
"\x55" /* push %ebp */
"\x89\xe5" /* movl %esp, %ebp */
"\x53" /* push %ebx */
"\xb8\x1b\x11\x11\x11" /* movl $0x1111111b, %eax */
"\x2d\x10\x11\x11\x11" /* subl $0x11111110, %eax */
"\x8b\x5d\x08" /* movl 8(%ebp), %ebx */
"\x8b\x4d\x0c" /* movl 12(%ebp), %ecx */
"\x8b\x55\x10" /* movl 16(%ebp), %edx */
"\xcd\x80"; /* int $0x80 */
```

**Figure 2. Example of injected code that was developed to exploit sendmail installed on Linux/IA-32. It finally invokes the execve system call with an argument "/bin/bash." Redundant instruction sequences are chosen in some parts in order to eliminate NUL characters from the string.**

uration of the underlying platform. Hence many buffer-overflow exploits disclosed in web sites are direct-invocation ones. For example, in an exploit database web site [13], more than 20 exploits out of 85 are direct-invocation attacks.

## 3 Proposed Method

### 3.1 Motivation

This work is concerned with direct-invocation attacks, though preventing return-into-libc attacks and more sophisticated attacks [24] is also interesting. Here we clarify our attitude on return-into-libc attacks. We acknowledge that our defense can be bypassed by using return-into-libc attacks, and that medium-skill programmers can develop the attacks without great difficulty, even considering the case where the entry points into the library must be discovered dynamically. Nevertheless, we judge prevention of direct-invocation attacks as important for the following reasons.

First, it complements a method for preventing return-into-libc attacks. Vulnerability to direct-invocation attacks is the weakest link on systems that have already been prepared for return-into-libc attacks. One widely-known approach to countermeasuring return-into-libc attacks is memory layout randomization, which makes it difficult for attackers to find the entry point of a critical function. Memory layout randomization, however, cannot prevent injected code from invoking system calls directly by itself.

Second, many exploits published on web sites are direct-invocation attacks, as described in Section 2. Low-skill attackers are likely to execute the exploits with little modification.

Finally, attackers are obliged to spend more time and effort to write a return-into-libc attack when a prevention mechanism against direct-invocation attacks is incorporated in a targeted system. Injected code must contain an instruction sequence for searching for or guessing the entry address of a code fragment that the attacker would like to jump to.

### 3.2 Method Description

Our method is achieved with a kernel module and a modified version of a dynamically linked standard library (i.e., modified libc). Figure 3 illustrates the basic behavior of the modified library and the kernel module. The modified library intercepts a system call just before the execution mode changes to the kernel mode (i.e., just before the library raises a software interrupt for system calls). Then it randomizes (or encrypts) the ID and arguments of the system call. In the kernel mode, the kernel module intercepts the system call and de-randomizes (or decrypts) the ID and arguments. The decrypted values are passed to the corresponding system call handler.

System call IDs and arguments are encrypted using a sequence of random numbers. The modified library and kernel module share a seed for generating a sequence of random numbers. Each time a system call is invoked, they generate a new random number and XOR it with the ID and arguments of the system call. A seed is generated by the kernel module when an application invokes `execve` or `fork`. The kernel module manages associations between process IDs and random seeds. Our method encrypts system call arguments to extend the space of random numbers. As described in Section 4.1, the space becomes much smaller when encrypting system call IDs only.

Our method makes it difficult for attackers to inject exploit code that successfully damages a targeted system. If the system call is directly invoked by injected code, the decrypted values are likely to be meaningless ones, so the corresponding system call handler is likely to return an error with respect to the attacker's intended action. Attackers who know that our method is working still have difficulty in executing a system call because they must guess a random number. Note that attackers must make the right guess within a small number of attempts because a wrong guess is likely to result in a crash of the targeted software.

Encryption is applied to only applications that are started from the launcher program described in Section 4.2. Applications whose system call information is encrypted can coexist with other ap-

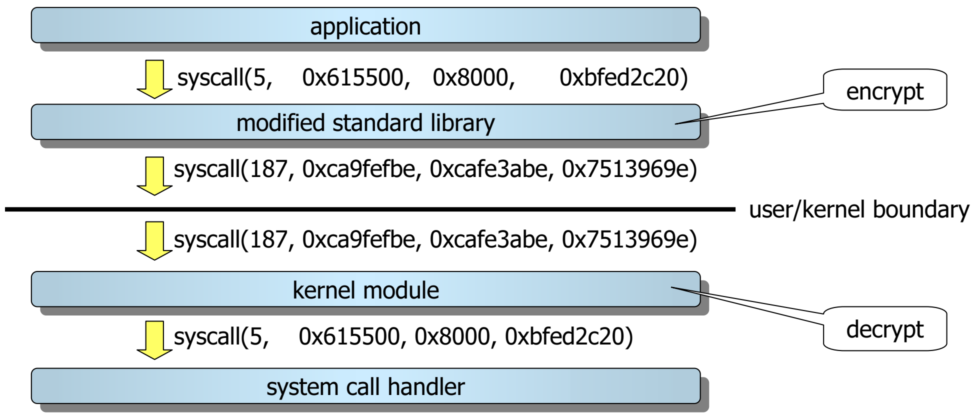
application
syscall(5, 0x615500, 0x8000, 0xbfed2c20)
encrypt
modified standard library
syscall(187, 0xca9fefbe, 0xcafe3abe, 0x7513969e)
user/kernel boundary
syscall(187, 0xca9fefbe, 0xcafe3abe, 0x7513969e)
kernel module
decrypt
syscall(5, 0x615500, 0x8000, 0xbfed2c20)
system call handler

**Figure 3. Handling of system calls in the proposed method. The ID and arguments of a system call are encrypted in the period between the modified library and the kernel module. The first argument 5 is the ID of the open system call.**

plications on the same operating system.

Our method does not prevent either buffer overflow or code injection events themselves, but hinders malicious operations that are performed after their successful execution. It provides a safety net at the final stage of defense and can complement other methods that mitigate attacks at an earlier stage.

Our method can be better understood through the analogy of client-server systems: it can be regarded as a variant of a general framework for security that encrypts communication links between a client and a server. An application in the proposed method corresponds to a client, and the kernel (including kernel modules) corresponds to a server. System call invocations are regarded as communication between a client and a server. As encryption of communication links makes it harder to send a malicious request to a server, the proposed method makes it harder to send a malicious system call request to the kernel.

## 4 Implementation

We implemented our method on a Linux environment (Fedora Core 1, kernel 2.4.22, glibc 2.3.2) running on an IA-32 machine (Pentium4 2.6GHz, 2GB main memory). Below we explain the implementation details.

### 4.1 Modified Standard Library

We modified a glibc source package bundled with Fedora Core 1 distribution (glibc-2.3.2-10271512) to create a standard library for our method. The program parts that need modifying are instruction sequences for invoking system calls. They are found in the following files under the directory `sysdeps/sysv/unix/linux/i386/` in the package.

```
sysdeps.h, clone.S,
getcontext.S, mmap.S, mmap64.S,
posix_fadvise64.S, semtimedop.S,
setcontext.S, socket.S,
swapcontext.S, syscall.S,
vfork.S, _exit.S
```

As far as the cases we examined, modifying them is sufficient to intercept all system calls on the platform above.

Below, we describe a modification to `sysdeps.h`. We omit the explanations of modifications to other program parts because they are almost all the same. `sysdeps.h` contains the following code fragment.

```
#define DO_CALL(syscall_name, args)       \
    PUSHARGS_##args                        \
    DOARGS_##args                          \
    movl $SYS_ify (syscall_name), %eax;   \
    ENTER_KERNEL                           \
    POPARGS_##args
```

ENTER_KERNEL is a macro expanded to `int $0x80`, an instruction that raises a software interrupt for system calls. DOARGS_##args is a macro expanded to instructions that place system call arguments in registers. The next `movl` instruction sets a system call ID to register eax.

We insert the following lines between the DOARGS_##args macro and the `movl` instruction. The inserted lines encrypt system call arguments.

```
    call __get_random_number;              \
    xorl %eax, %ebx;                       \
    xorl %eax, %ecx;                       \
    xorl %eax, %edx;                       \
    xorl %eax, %esi;                       \
    xorl %eax, %edi;                       \
```

The first line calls a function `__get_random_number`, which generates a new random number from the seed and returns the number. It is a hidden function we added to the standard library. The definition of the function is written in a C source file in the directory `sysdeps/sysv/unix/linux/i386/`. In the following `xorl` instructions, the random number is XOR-ed with values of all registers that keep system call information. Below the `xorl` instructions, we further insert instructions for encrypting system call IDs. We XOR a different random number with a system call ID so that the encrypted value may not exceed the maximum system call ID. If the encrypted value is larger than the maximum, the kernel immediately returns an error before the kernel module decrypts the encrypted ID. Hence, the lowest eight bits of the original random number are XOR-ed with a system call ID, only when the original ID is less than 256.

The amount of modification to the glibc source tree was 195 lines (145 lines of assembly and 50 lines of C source). Though the program parts that must be modified are platform-dependent, we expect that the amount of modification needed on other platforms will be small.

## 4.2 Application Launcher

Encryption of system call information is applied to applications started via our application launcher. The launcher receives the path and arguments of a program to be executed. The task of the launcher is to tell the kernel module to apply the proposed method to the launched program. The launcher communicates with the kernel module via a newly introduced character device `/dev/scencrypt`. The launcher invokes `ioctl` for the device, giving a flag that represents our method. Then the kernel module generates a random seed and creates an association between the process ID of the launcher and the seed. After that, the launcher invokes the `execve` system call to start the program. The kernel module begins decryption of system call information in the first call of `execve` after `ioctl` to `/dev/scencrypt`.

## 4.3 Kernel Module

The kernel module intercepts system calls by modifying the system call table. When the kernel module is inserted into the kernel, it sets all table entries to a newly introduced function `redirect_syscall`. The original system call table is saved in a memory space that the kernel module allocates statically. `redirect_syscall` first checks whether the calling process is registered in the table that manages associations between process IDs and random seeds (described below). If registered, the control moves to another newly introduced function `decrypt_syscall`. Otherwise, a system call handler is looked up from the original system call table and called. The function `decrypt_syscall` decrypts the ID and arguments of the invoked system call by XOR-ing them with a random number generated from the seed associated with the application's process ID. It then calls the original system call handler.

The kernel module maintains a table that manages associations between process IDs and random seeds. When the launcher invokes `ioctl` to

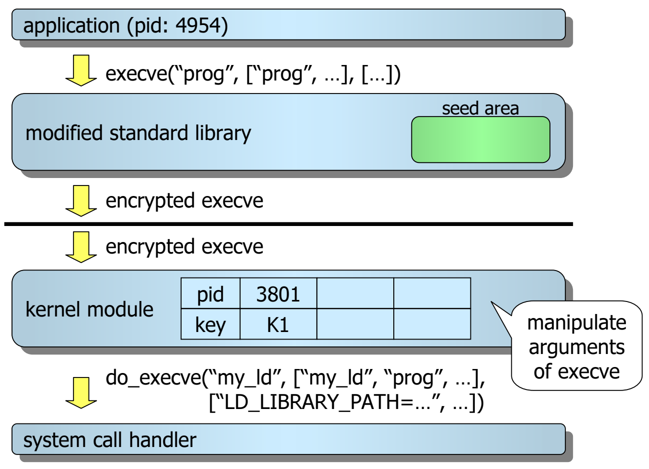
application (pid: 4954)
execve("prog", ["prog", ...], [...])
modified standard library
seed area
encrypted execve
encrypted execve
kernel module
pid
3801
key
K1
manipulate arguments of execve
do_execve("my_ld", ["my_ld", "prog", ...], ["LD_LIBRARY_PATH=...", ...])
system call handler

**Figure 4. Operations performed before the execution of the execve handler.**

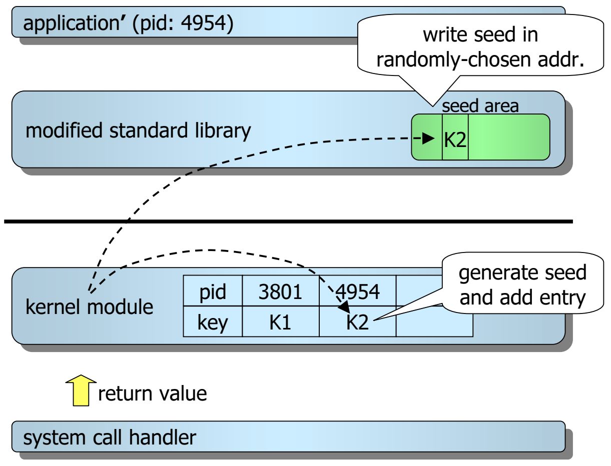

**Figure 5. Operations performed after the execution of the execve handler.**

`/dev/scencrypt`, the kernel module adds an entry to the table. Each entry contains a flag indicating whether the kernel module has started the decryption of system call information invoked by the process. The flag is turned off at first and turned on when the launcher invokes `execve`.

When the launcher or an application process executes `execve`, the kernel module performs the following operations:

- Manipulates arguments of `execve` to make the application link the modified library.
- Calls the system call handler of `execve`.
- Generates a random seed.
- Registers in the table the association between the application's process ID and the seed.
- Writes the seed in the address space of the application.

Operations performed before and after the `execve` system call handler are illustrated in Figures 4 and 5, respectively. The address where an application keeps a seed is randomly chosen by the kernel module from free memory areas in the application's address space. The chosen address is reported to the application through a fixed address between the application stack and the kernel space. The function `__get_random_number` eventually reads the fixed area, obtains the chosen address, and obtains the seed. Then, it fills the fixed area with zero for security. Consequently, the address of a seed differs in different processes and different runs, and attackers cannot obtain the address of a seed via the fixed area because it is zero-filled just after the communication of the address of the seed.

The purpose of the randomized placement of seeds is to increase the effort and code required to find a seed. The randomized placement does not completely conceal a seed from an attacker. Sophisticated exploits will still find a seed even when the address of a seed is randomized.

When an application invokes `fork`, the kernel module generates a random seed, adds an entry to the table, and gives different seeds to the parent and child processes. When an application process terminates, the corresponding entry is removed from the table.

The kernel module is made up of 749 lines of C source and headers, and 11 lines of assembly code. The object file of the module is about 11 kilobytes.

## 5 Discussion

**Limitations** Our method has the following limitations. First, it cannot work with systems that use system call interceptions (e.g., debuggers, tracers, and sandboxes). Second, the current implementation does not allow our method to be applied to programs to which an original standard library is statically linked. If these programs are started from our application launcher, they are likely to crash. At present, static linking of our modified library is the only way to apply our method to applications that require static links. Finally, sophisticated attackers can devise exploit code that searches for a random seed in the address space of the application and encrypts IDs and arguments of system calls with the discovered random seed. However, developing this kind of exploit takes a considerable amount of effort and hacks.

**Introduction of stream cipher** The prediction of XOR-ed random numbers becomes difficult if the numbers are generated with a stream cipher such as RC4. However, introduction of a stream cipher contributes little to the enhancement of the security level because the weakest link of our method is not the strength of an adopted encryption algorithm. We expect that attackers will not attempt to obtain a correct guess using a brute-force attack, but will attempt to break a weaker link. For example, they may resort to return-into-libc attacks or attempt to find a memory area storing a random seed or a cipher key.

**Probability issues** There is a chance, albeit small, that a failed exploit attempt can nevertheless result in the continued execution of the vulnerable process in a corrupted state. It remains possible that, for example, a malicious system call would be transformed into a `read` system call that reads private data or a `write` system call that corrupts important data structures. The same situation is seen in the work of Software Fault Isolation [34] and Failure-Oblivious Computing [29]. We think that the continued execution is not a large problem in most cases.[1] Even if our defense is absent, injected code can execute arbitrary system calls including the `read` and `write` system calls described above. The introduction of our defense does not affect the range of operations that can be performed by injected code. Hence it does not raise the maximum level of potential damage.

**Multithread support** The current implementation does not support multithreaded applications. Below we briefly describe the extensions needed to support them. First, a distinct area for storing a random seed must be allocated for each thread. Second, the modified standard library and the kernel module must manage associations between thread IDs and seeds. Finally, when a system call is invoked, the kernel module must examine which thread invoked it.

## 6 Experiments

We conducted several experiments on the platform described in Section 4. Performance figures shown in this section are the average of figures measured in five executions.

### 6.1 Microbenchmark

We measured the overhead incurred when our method is applied. We executed the following microbenchmarks:

**fib** Calculates the value of fib(40). No system call is invoked during the time of measurement.

**getpid-loop** Repeats the invocation of the `getpid` system call ten million times. It does not invoke any other system call during the time of measurement.

**open-loop** Repeats the operation in which one existing file is opened and then closed immediately. It invokes the `open` system call (and the `close` system call) ten million times.

**exec-loop** Repeats the operation in which the program itself is invoked recursively with

[1]An exception is the case in which fail-fastness is required.

| | fib | getpid-loop | open-loop | exec-loop |
|---|---|---|---|---|
| original (ms) | 3388 | 4273 | 16004 | 2562 |
| our method (ms) | 3401 | 7722 | 23258 | 8031 |

**Table 1. Microbenchmark results.**

| | tar | gzip | cp | grep | gcc | latex | wget | Apache |
|---|---|---|---|---|---|---|---|---|
| original (ms) | 758 | 4075 | 822 | 532 | 3135 | 992 | 82937 | 82937 |
| our method (ms) | 849 | 4389 | 886 | 611 | 3329 | 993 | 85997 | 84537 |
| overhead | 12.0% | 7.7% | 7.8% | 15.0% | 6.2% | 0.1% | 0.7% | 1.9% |

**Table 2. Application benchmark results.**

the `execve` system call. It terminates when `execve` has been invoked ten thousand times.

Table 1 shows the results. "Original" indicates execution times taken when our method was not applied. "Our method" indicates execution times taken when our method was applied. Naturally, the performance of fib was little affected by our method. The overheads imposed on getpid-loop and open-loop were 80.7% and 45.3%, respectively. Getpid-loop is considered to show an upper bound of overheads, since it repeats only the execution of a lightweight system call. Based on the performance of getpid-loop, we can estimate the overhead incurred on each system call. In the experiment of getpid-loop, the increase in execution time was 3449 ms and `getpid` was executed ten million times. Therefore, we can estimate that an overhead of approximately 0.34 $\mu$s was imposed on each system call. That estimation agrees well with the open-loop results (note that open-loop invokes two system calls in a loop). The performance of exec-loop is significantly degraded when our method is applied. This is because our kernel module performs a considerable amount of operations during the interception of `execve`. The operations include generation of a random seed and modification of `execve`'s arguments. The implementation of the operations has not yet been fully optimized. We plan to reduce the overhead by optimizing them.

## 6.2 Application Benchmark

We also measured the performance of the following application benchmarks:

**tar** Executes the `tar` command to create a tar file from the source tree of the Linux 2.4.22 kernel.

**gzip** Compresses ten `vmlinux` files with the `gzip` command.

**cp** Copies the source tree of glibc 2.3.2 recursively with the `cp -a` command.

**grep** Executes the command `grep -r` to search the source tree of the Linux 2.4.22 kernel and glibc 2.3.2 for the word "vulnerable."

**gcc** Executes the `gcc` command to compile one hundred C source files into object files. Each source file contains about one hundred lines of C program.

**latex** Compiles a LaTeX source file into a DVI file with the `platex` command. The LaTeX file is composed of 11100 lines and the DVI file is a 114-page paper.

**wget** Executes the `wget -q -r` command to download files from a web server running on the local machine. By using the recursive downloading feature of `wget`, it downloads ten thousand files each ten kilobytes in size. In this benchmark, our method is applied to `wget` and not to the web server.

| stage of attack | | preventing method |
|---|---|---|
| code pointer overwrite | | StackGuard [8, 32], PointGuard [9], safe C compilers [16, 20, 21, 30, 36], Libsafe [3], LibsafePlus [2], memory layout randomization [5, 7, 26, 35], static vulnerability detection [6, 11, 19, 31, 33] |
| execution of injected instructions | | randomized instruction sets [4, 18, 22], non-executable pages [12, 25, 26] |
| execution of operating system services | direct invocation of system call | randomized system call table [7], our method |
| | return-into-libc | memory layout randomization [5, 7, 26, 35] |

**Table 3. Summary of related work.**

**Apache** Starts an Apache web server and then executes the `wget -q -r` command on the local machine to download files from the server. In this benchmark, our method is applied to Apache and not to `wget`. A set of files downloaded by `wget` is the same as in the wget benchmark.

Table 2 shows the results. The overheads ranged from 0.1% to 15.0%. Though a considerable amount of overhead was incurred in microbenchmarks, the overheads on real applications were not significant.

## 7 Related Work

Table 3 shows a summary of related work. The work closest to ours is that of Chew et al. [7], which proposes several methods of mitigating buffer overflows by introducing randomness into the implementation of system software. One of their methods changes the mapping between system call IDs and system call handlers by mixing up the system call table using random numbers. This is achieved by recompilation of the kernel and binary rewriting of applications to fit them to the new kernel. In their method, one mapping is shared by all processes and does not change except when the kernel is recompiled. In our method, each process has a different mapping and the mapping changes every time a system call is invoked. Hence, attackers against our method have more difficulty in exploiting the targeted system. In addition, when using our method, there is no need to recompile the kernel. Applications running with our method's defense can coexist with those running without it on one operating system.

StackGuard [8, 32] encrypts control information in a stack by XOR-ing it with a random number. StackGuard causes malicious control information to be decrypted into meaningless values, resulting in the failure of the attackers' attempt to acquire control of vulnerable software. Our method encrypts system call arguments, not control information in a stack. Unlike StackGuard, our method can mitigate attacks that are not limited to stack-smashing attacks and can work without recompilation of an application.

PointGuard [9] stores encrypted values of critical pointers in memory and decrypts them before they are loaded on registers. A maliciously crafted pointer is likely to be decrypted into a meaningless value, causing the program to crash. PointGuard incurs an overhead in every load and store of all kinds of pointers. Since PointGuard is implemented as a C compiler enhancement, it requires recompilation of an application.

Randomization of instruction sets [4, 18, 22] for preventing code-injection attacks has been proposed by several research groups. Though instruction randomization is a powerful technique, it incurs more performance penalty than ours, because it requires the introduction of an emulator and the binary transformation of applications. Our method focuses on providing a moderate level of security with simpler and more lightweight implementation.

There are several techniques that introduce randomness or diversity into operating systems to mitigate attacks [5, 7, 14, 26, 28, 35]. For example, address obfuscation [5] randomizes the memory placement of a wide range of memory objects including stacks, heaps, variables, and dynamically linked libraries. Our work introduces randomness into the convention of system call invocations. Memory-placement randomization and our method can complement each other.

Baratloo et al. proposed a library-replacement method of preventing buffer overflow attacks [3]. Their library Libsafe replaces a standard library. Libsafe intercepts a library call and checks whether the library call could overflow a buffer allocated in a stack frame. LibsafePlus [2] is an extension to Libsafe. It estimates buffer sizes more accurately than Libsafe. Libsafe and LibsafePlus are different from our method in that they prevent buffer overflow or code injection themselves. Our method hinders malicious operations performed after the success of a buffer overflow or code injection. Their methods and ours are complementary to each other.

Safe C compilers or translators [16, 20, 21, 30, 36] generate safe (or safer) code that checks memory access errors at runtime. Unfortunately, they have not been widely adopted due to their limitations. For example, some systems require applications to be recompiled, some incur a large overhead, and some do not support the full specification of the C language.

Some systems statically analyze the source code of C programs to detect potential vulnerabilities [6, 11, 19, 31, 33]. However, they have not succeeded in exterminating all vulnerabilities. As a result, buffer overflow vulnerabilities are still reported even nowadays.

A great proportion of code-injection attacks can be prevented by introducing *non-executable stacks* using the hardware's protection capability. Some researchers proposed hardware-assisted protections against buffer overflows [12, 25, 26]. In the industrial world, protection based on the Execute-Disable Bit capability of modern processors has been adopted in recent operating systems including Windows XP SP2. However, non-executable stacks can cause the anomalous execution of normal programs, because some programs depend on executable stacks for run-time code generation or signal handling. Moreover, non-executable stacks cannot block the execution of malicious code injected into a heap. Our method does not impede programs that assume executable stacks and it can prevent attacks by heap-injected code.

## 8 Summary and Future Work

We described a method for preventing code-injection attacks by randomizing the convention of system call invocations. By utilizing a sequence of random numbers shared between a modified standard library and a kernel module, our method can change the convention each time a system call is invoked. Consequently, attackers have difficulty in guessing the values corresponding to the ID and arguments of the system call they would like to invoke. We conducted experiments and measured the overhead incurred by our method. The results showed that our method imposed overheads that ranged from 0.1% to 15.0%.

There are several items for future work. First, we would like to confirm that our method works well on a wider range of operating systems and CPU architectures. Second, we would like to develop a framework that enables the proposed method to coexist with supportive systems that use system call interceptions (e.g., debuggers, tracers, and middleware that creates virtual or sandboxed execution environments [15, 17, 23, 27]). One solution is to provide the systems with a plug-in or a patch that enables them to cooperate with our method. Finally, we would like to devise a framework that allows our method to be applied to statically linked programs. We are now examining binary patching techniques to achieve this.

## References

[1] Aleph One. Smashing The Stack For Fun And Profit. http://www.phrack.org/show.php?p=49&a=14, 1996.

[2] Kumar Avijit, Prateek Gupta, and Deepak Gupta. TIED, LibsafePlus: Tools for Runtime Buffer Overflow Protection. In *Proceedings of the 13th USENIX Security Symposium*, pages 45–56, San Diego, August 2004.

[3] Arash Baratloo, Navjot Singh, and Timothy Tsai. Transparent Run-Time Defense Against Stack-

Smashing Attacks. In *Proceedings of the USENIX Annual 2000 Technical Conference*, pages 251–262, San Diego, June 2000.

[4] Elena Gabriela Barrantes, David H. Ackley, Stephanie Forrest, Trek S. Palmer, Darko Stefanović, and Dino Dai Zovi. Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS 2003)*, pages 281–289, Washington DC, October 2003.

[5] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar. Address Obfuscation: an Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *Proceedings of the 12th USENIX Security Symposium*, pages 105–120, Washington DC, August 2003.

[6] Hao Chen and David Wagner. MOPS: an Infrastructure for Examining Security Properties of Software. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS)*, pages 235–244, Washington DC, November 2002.

[7] Monica Chew and Dawn Song. Mitigating Buffer Overflows by Operating System Randomization. Technical Report CMU-CS-02-197, Carnegie Mellon University, December 2002.

[8] Crispan Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Symposium*, pages 63–78, San Antonio, January 1998.

[9] Crispin Cowan, Steve Beattie, John Johansen, and Perry Wagle. PointGuard$^{\mathrm{TM}}$: Protecting Pointers from Buffer Overflow Vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*, pages 91–104, Washington, D.C., August 2003.

[10] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade. In *Proceedings of the DARPA Information Survivability Conference and Exposition (DISCEX)*, pages 1119–1129, Hilton Head Island, January 2000.

[11] Nurit Dor, Michael Rodeh, and Mooly Sagiv. CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI '03)*, pages 155–167, San Diego, USA, June 2003.

[12] ExecShield. http://people.redhat.com/mingo/exec-shield/.

[13] Exploit Database. http://www.exploitdatabase.com/.

[14] Stephanie Forrest, Anil Somayaji, and David H. Ackley. Building Diverse Computer Systems. In *Proceedings of the Sixth Workshop on Hot Topics in Operating Systems (HotOS-VI)*, pages 67–72, Cape Cod, USA, May 1997.

[15] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A Secure Environment for Untrusted Helper Applications: Confining the Wily Hacker. In *Proceedings of the 6th USENIX Security Symposium*, pages 1–13, San Jose, July 1996.

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

[17] Kazuhiko Kato and Yoshihiro Oyama. SoftwarePot: An Encapsulated Transferable File System for Secure Software Circulation. In *Software Security – Theories and Systems*, volume 2609 of *Lecture Notes in Computer Science*, pages 112–132, February 2003.

[18] Gaurav S. Kc, Angelos D. Keromytis, and Vassilis Prevelakis. Countering Code-Injection Attacks With Instruction-Set Randomization. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS 2003)*, pages 272–280, Washington DC, October 2003.

[19] David Larochelle and David Evans. Statically Detecting Likely Buffer Overflow Vulnerabilities. In *Proceedings of the 10th USENIX Security Symposium*, Washington DC, August 2001.

[20] George C. Necula, Scott McPeak, and Westley Weimer. CCured: Type-Safe Retrofitting of Legacy Code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2002)*, pages 128–139, Portland, January 2002.

[21] Yutaka Oiwa, Tatsurou Sekiguchi, Eijiro Sumii, and Akinori Yonezawa. Fail-Safe ANSI-C Compiler: An Approach to Making C Programs Secure

(Progress Report). In *Proceedings of the International Symposium on Software Security*, volume 2609 of *Lecture Notes in Computer Science*, pages 133–153, Tokyo, November 2002.

[22] Noritaka Osawa. A Smart Virtual Machine for Heterogeneous Distributed Environments: PivotVM. *Transactions on Information Processing Society of Japan*, 40(6):2543–2552, June 1999.

[23] David S. Peterson, Matt Bishop, and Raju Pandey. A Flexible Containment Mechanism for Executing Untrusted Code. In *Proceedings of the 11th USENIX Security Symposium*, pages 207–225, San Francisco, August 2002.

[24] Jonathan Pincus and Brandon Baker. Beyond Stack Smashing: Recent Advances in Exploiting Buffer Overruns. *IEEE Security and Privacy*, 2(4):20–27, July-August 2004.

[25] Openwall Project. http://www.openwall.com/.

[26] PaX Project. http://pax.grsecurity.net/.

[27] Niels Provos. Improving Host Security with System Call Policies. In *Proceedings of the 12th USENIX Security Symposium*, pages 257–272, Washington DC, August 2003.

[28] Calton Pu, Andrew Black, Crispin Cowan, and Jonathan Walpole. A Specialization Toolkit to Increase the Diversity of Operating Systems. In *Proceedings of the 1996 ICMAS Workshop on Immunity-Based Systems*, Nara, Japan, December 1996.

[29] Martin Rinard, Cristian Cadar, Daniel Dumitran, Daniel M. Roy, Tudor Leu, and Jr. William S. Beebee. Enhancing Server Availability and Security Through Failure-Oblivious Computing. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation*, pages 303–316, San Francisco, December 2004.

[30] Olatunji Ruwase and Monica S. Lam. A Practical Dynamic Buffer Overflow Detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS '04)*, pages 159–169, San Diego, February 2004.

[31] John Viega, J. T. Bloch, Yoshi Kohno, and Gary McGraw. ITS4: A Static Vulnerability Scanner for C and C++ Code. In *Proceedings of the 16th Annual Computer Security Applications Conference (ACSAC 2000)*, New Orleans, USA, December 2000.

[32] Perry Wagle and Crispin Cowan. StackGuard: Simple Stack Smash Protection for GCC. In *Proceedings of the GCC Developers Summit*, pages 243–255, Ottawa, Canada, May 2003.

[33] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. In *Proceedings of the Network and Distributed System Security Symposium Conference (NDSS 2000)*, pages 3–17, San Diego, February 2000.

[34] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating System Principles (SOSP '93)*, pages 203–216, Asheville, December 1993.

[35] Jun Xu, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Transparent Runtime Randomization for Security. In *Proceedings of the 22nd Symposium on Reliable Distributed Systems (SRDS 2003)*, pages 260–269, Florence, Italy, October 2003.

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