# e-NeXSh: Achieving an Effectively Non-Executable Stack and Heap via System-Call Policing

Gaurav S. Kc*
*Google, Inc.*
gskc@google.com

Angelos D. Keromytis
*Columbia University*
angelos@cs.columbia.edu

## Abstract

*We present e-NeXSh, a novel security approach that utilises kernel and LIBC support for efficiently defending systems against process-subversion attacks. Such attacks exploit vulnerabilities in software to override its program control-flow and consequently invoke system calls, causing out-of-process damage. Our technique defeats such attacks by monitoring all LIBC function and system-call invocations, and validating them against process-specific information that strictly prescribes the permissible behaviour for the program (unlike general sandboxing techniques that require manually maintained, explicit policies, we use the program code itself as a guideline for an implicit policy). Any deviation from this behaviour is considered malicious, and we halt the attack, limiting its damage to within the subverted process.*

*We implemented e-NeXSh as a set of modifications to the* `linux-2.4.18-3` *kernel and a new user-space shared library (*`e-NeXSh.so`*). The technique is transparent, requiring no modifications to existing libraries or applications. e-NeXSh was able to successfully defeat both code-injection and libc-based attacks in our effectiveness tests. The technique is simple and lightweight, demonstrating no measurable overhead for select UNIX utilities, and a negligible 1.55% performance impact on the Apache web server.*

## 1 Introduction

In recent years, the issue of process-subversion attacks has become very important, as is evidenced by the number of CERT advisories [2]. These are attacks that exploit programming errors in software to compromise running systems. Such errors allow the attacker to override the program's control logic, causing the program to execute code of their choosing. This code is either malicious executable code that has been injected into the process' memory, or existing functions in the Standard C library (LIBC) or elsewhere in the program code. In either case, the attacker is able to compromise the process, and generally can gain control of the whole system if the attacked process was running with root privileges (this is often the case with server daemons).

To cause any real damage outside of the compromised process, e.g., to spawn a shell or to open the `/etc/passwd` file for editing, the attacker needs to access kernel resources via system calls. There has been significant research in recent years resulting in a wide range of process-sandboxing techniques that monitor applications' system-call invocations [27, 31, 32, 33, 48, 49, 50, 58]. These systems generally require manual effort to specify explicit sandboxing policies, except when such policies can be automatically generated from application code or training runs. Then, the policies are often either imprecise, producing a large number of false positives, or involve significant (>2×) overheads — these are described in more detail in §5. Other techniques prevent the execution of code in data memory [3, 5, 10, 13, 38, 40, 55] to defeat code-injection attacks, randomise the addresses of functions in the Standard C library (LIBC) [3, 14, 20] to deter libc-based attacks, use static analyses to remove vulnerabilities from software [12, 19, 21, 35, 44, 46, 59], or instrument application code to detect run-time attacks [22, 47, 28, 57]. There exist problems and/or limitations of these systems too, including large overheads or breaking of applications, the possibility of mimicry or brute-force attacks that can bypass the defence mechanisms, and the inability to specify or protect all vulnerabilities, respectively. These limitations are also covered in §5.

In this paper we present e-NeXSh, a simple and lightweight technique that uses process-specific information — this information consists of disassembly data indicating the memory boundaries of all functions in the program, as well as the call sites for system-call invocations in the program code — to defeat both code-injection and libc-based attacks. The technique's novelty lies in how it

*This work was carried out while author was at Columbia University.

builds on the system-call interception model used by Intrusion Detection Systems (IDS), and extends the idea to user-space code to monitor invocations of LIBC functions. We utilise the program code and its disassembly information as guidelines for an implicit policy to prescribe normal program behaviour, rather than manually defining explicit policies. We show that e-NeXSh creates an "effectively" non-executable stack and heap that permit the execution of all code except system-call invocations (even via LIBC functions). This makes our technique practical even for applications that have a genuine need for an executable stack. We have evaluated a prototype implementation of our technique on x86/Linux, and have demonstrated efficacy at defeating both libc-based and code-injection attacks, including 100% effectiveness against Wilander's benchmark test-suite [64], in our evaluations. We provide further discussion in §5.

Our implementation consists of two parts: a user-space component to intercept and validate LIBC function invocations, and a kernel component to do the same for system-call invocations. The user-space component (implemented as a shared library: `e-NeXSh.so`) intercepts invocations of LIBC functions, and validates the call chain against the program's binary code and its disassembly information. This component can detect libc-based attacks that redirect the targeted program's control-flow into LIBC functions, to indirectly issue system calls. If an invocation has a legitimate call chain (i.e., one that matches the program code and disassembly data), we indicate this to the kernel component, and then forward the call on to the appropriate function in LIBC. Our kernel component extends the system-call handler in the Linux kernel to intercept and verify that all system-call invocations are made from legitimate locations in the program (or LIBC) code. These components collectively block all paths that an attacker may take to invoke system calls.

An important advantage of our technique is the low overhead added to system execution — we report negligible run-time overheads for both Apache benchmarks and common UNIX utilities. By implementing our technique within LIBC and kernel call handlers, we have managed to capitalise on the relatively large execution time required to process most system calls (including time spent in LIBC), versus normal procedure calls. Our design also permits transparent integration with applications, without needing to modify either source or binary code for both applications and the Standard C library (LIBC). This has many advantages — for instance, our technique can be applied to legacy and third-party applications for which there is no access to the source code. Furthermore, e-NeXSh can be used in conjunction with other defence mechanisms that instrument application executables or library binaries to perform run-time checking. Even though we modify the kernel in our implementation, we can selectively enforce the protection mechanism, allowing most programs [that do not need the security] to run unaffected.

The rest of this paper is structured as follows: we present an overview of our approach in §2. We describe our implementation of e-NeXSh in §3, and present an evaluation of our technique in §4. We discuss related work in §5, and some remaining issues in §6, and conclude in §7.

## 2 Approach

Published guidelines [4, 6, 9] and analyses [7, 15, 23, 24] of process-subversion attacks indicate that these attacks generally contain the following integral elements: the triggering of some security **vulnerability** in the target application, causing an overwrite of some **code pointer** memory location, making it point to **malicious code** of the attacker's choosing. These elements comprise the anatomy of attacks, and is illustrated in figure 1. Then, as the program continues execution, it eventually activates this overwritten code pointer, and begins executing the attack code in question.

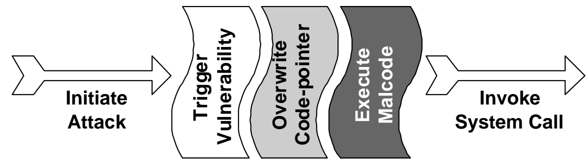


**Figure 1.** *Anatomy of an Attack.*

Research efforts have resulted in tools for identifying and/or eliminating vulnerabilities from application code, as well as at run-time techniques for detecting that vulnerabilities may have been exploited or that certain code pointer may have been overwritten. Rather than monitor all exploitable vulnerabilities and over-writable code pointers, we focus on the third element of attacks, i.e., the malicious code that attackers execute. Our technique is then able to defeat **both known and yet unknown attacks**, regardless of the specific vulnerabilities they exploit, or the specific code pointers they overwrite.

The executed malcode is invariably either executable code that the attacker has injected into the program memory, or existing code in the program or its referenced libraries, and in all cases, is executed to invoke some system call(s). Recognising this fact, we have based our technique on the monitoring of system-call invocations both within the kernel and at the user-space levels. We can trivially identify direct invocations of system calls by injected code, from within the kernel. Furthermore, we monitor calls to LIBC functions to detect if existing code in the program or its libraries is executed to indirectly invoke a system call (this generally is by way of the appropriate LIBC function). Specifically, we intercept calls to LIBC functions at the user-space level to inspect and validate the program

stack. By intercepting both paths to invoking system calls, we can detect when a compromised program attempts to invoke system calls. We also randomise the program memory layout to prevent mimicry attacks would otherwise replicate the program stack in order to present a seemingly valid call stack to our checking subsystem.

Table 1 provides a high-level overview of the evolution of e-NeXSh in response to increasingly sophisticated and adaptive attack mechanisms:

**Attacks mechanisms and Defence measures**

| | |
|---|---|
| Code-Injection attacks | **Attack** executable attack code injected into data memory invokes system calls directly.<br>**Defence** the kernel component of e-NeXSh examines the "return address" for system calls to identify code-injection attacks. |
| (LIBC-based) Existing-Code attacks | **Attack** attacker invokes system calls indirectly by redirecting program flow-of-control to existing trap instruction in either application or library code, e.g., LIBC. This avoids the first-level defence against code-injection attacks.<br>**Defence** the LIBC component of e-NeXSh intercepts invocations of shared library calls, and verifies the program stack trace (`main` — to library function), and thus detects existing-code attacks. |
| (Stack-faking) Mimicry attacks | **Attack** attacker re-creates signature for a valid program stack-trace to give e-NeXSh the impression of a normal program run.<br>**Defence** randomise the program memory layout (activation record headers {return address, old frame pointer}, offset of program data stack, and offset of program code segment) to prevent attackers from re-creating a valid stack-trace signature. |

**Table 1.** *High-level overview of defence tactics in e-NeXSh and possible countermeasures. Each attack mechanism is a countermeasure to the immediately preceding defence technique.*

# 3 Implementation

In this section, we describe the two components of our e-NeXSh implementation for the Linux operating system running on x86 hardware. We then illustrate its operations, i.e., the sequence of events, for handling both normal program behaviour and attacks in progress. We end this section by enumerating some benefits of our technique.

## 3.1 Validating System-Call Invocations

We use the system calls emitted by a program as an indication of its behaviour, similar to traditional host-based intrusion detection systems (IDS) that infer program behaviour from audit patterns, i.e., the observed sequence of system calls emitted [34, 42, 43, 58, 65] by the running program. While these systems generally use source code-based static models to determine malicious activity, we use specific information associated with system-call and LIBC invocations. The novelty in our technique is that we extend the checking "deeper" into the application's call stacks, thus making it more difficult for an attacker to launch mimicry [60] (or libc-based) attacks.

We chose a kernel-based, system-call interposition technique to utilise the resource-manager role of the operating system kernel for monitoring the invocations of system calls. Similar to normal function-calling conventions in user code, a system-call invocation also stores information (the program counter value) about the caller code in the program stack. We extended the system-call handling utility in the kernel to verify that it was legitimate application (or library) code that made the system-call invocation. Specifically, we check to see that the virtual memory address of the `trap` machine instruction that issued the system call is located in one of the code segments for the process. The kernel maintains information on all the different code and data segments that comprise each process' run-time memory. Our kernel modifications simply checks the read-write flag for the given memory address. A "writable" flag denotes data memory, which we now assume contains injected code that's invoking a system call. On the other hand, legitimate invocations of system calls occur from non-writable code memory addresses that are associated with a "read-only" flag.

## 3.2 Validating LIBC Function Invocations

We extend this caller-validation idea into the user-space to validate invocations of LIBC functions. This is implemented in the form of a shared library (`e-NeXSh.so`) containing corresponding wrapper functions for each LIBC function that is intercepted — we currently provide wrappers only for functions that are useful to an attacker [65] (some of these functions are: `chmod`, `connect`, `execve`, `fork`, `open`, `mmap`, `mprotect`, `socket`). We set the `LD_PRELOAD` environment variable to ensure that LIBC function invocations made by programs are directed into our shared library. Each wrapper function consists of the following steps: authorise LIBC-based system calls, validate the current call stack, and resume execution by invoking the intended LIBC function, or kill the process if authorisation fails. We describe the most complex step (call-stack validation) first.

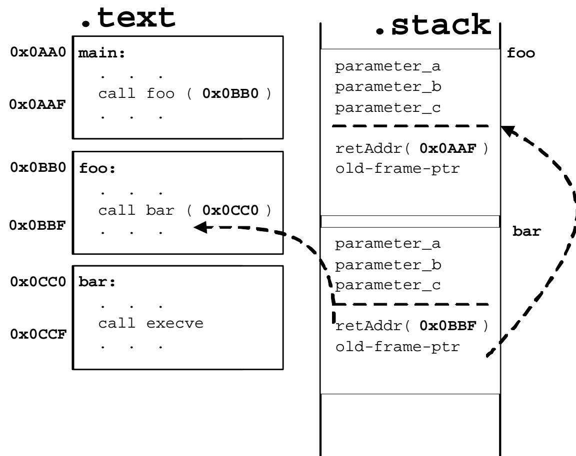


**Figure 2.** *The stack trace yields* `bar`*'s return address as* `0x0BBF`*. We de-reference this memory address (we can verify that it is contained within* `foo`*), and inspect the preceding* `call` *instruction to extract its operand. This operand is* `0x0CC0`*, which we then verify to match* `bar`*'s starting address. In this manner, given the return address for the stack frame for "*`foo` *calls* `bar`*", we can use the machine code (from the* `.text` *section) to verify that the caller-callee relationship is legitimate. We repeat these checks for each return address in the stack trace until we reach* `main()`.

### 3.2.1 Validating the Call Stack

Each wrapper function performs a stack walk along the dynamic chain of intermediate functions starting from `main()` and ending with the function that eventually invoked the LIBC function. This yields a list of return address values for each of the intermediate functions: we first verify that each of these return addresses exists in the write-protected memory region reserved for the application code, viz., the `.text` section (we assume that information about the range of the `.text` section for the program code is provided by the kernel). We then inspect the `.text` section of the process memory to extract and decode each `call` instruction preceding the instructions at these return addresses. We can now verify that the absolute target address of this direct `call` instruction exactly corresponds to the beginning address of the function that is one level deeper in the call chain. In other words, for each caller-callee pair in the dynamic chain, we can validate the call site and target site memory addresses against the start and end limits for both functions. Our technique for validating the call stack is similar to stack-tracing techniques in [26] and those in Java for checking access control [61], but without the benefit of Java's extensive run-time information. Figure 2 illustrates the verification of a single caller-callee function relationship, where `foo` calls `bar`.

The x86 architecture defines other (indirect) `call` instructions that are of variable length, and do not encode the true target of the `call` instructions. Instead, they utilise indirect addresses stored in registers or the memory at run-time. This prevents us from statically producing call graphs to define all possible caller-callee relationships, and forces us to accept any function as a potentially valid target for an indirect `call` instruction. An attacker could then potentially overwrite a function pointer variable or a C++ VPTR entry, and cause a redirection of control directly into a LIBC function.

There are a number of options available at this point: we can counter such attacks by requiring the innermost caller-callee pair in the dynamic chain (where the application program invokes the LIBC function) to be a direct `call` instruction — this has the potential to falsely label some applications as being compromised if they invoke the protected LIBC functions via function pointers (the applications we have evaluated do not use indirect call instructions to invoke LIBC functions). Another approach is to run the program in training mode and log the occurrences of code pointers in the call stack. Assuming that these invocations did not already lead to a process compromise during the training stage, the logged data can be integrated into the disassembly information for the program. This allows future occurrences of these code pointers to be accepted. A third option is to use a simple static-analysis [11] engine to pre-compute sets of acceptable values for the target address for indirect calls. Correctly using this new information would increase the memory footprint of `e-NeXSh.so`. However, the run-time overheads should be minimal since this data is referenced only for call-stack instances that include indirect calls. Even so, there may potentially be an efficient means to further reduce these overheads.

### 3.2.2 Authorising System Calls

Before validating the call stack, the wrapper function invokes a new system call that we defined as `syscall_libc_auth`. This system call is used to indicate to the kernel that the checking mechanism in `e-NeXSh.so` has verified the user-space section of the call stack for upcoming system calls. We store a random[1] 32-bit nonce in the stack frame, and pass the address of this nonce in the system call. This ephemeral nonce value exists only for the duration of the execution of the wrapper functions in `e-NeXSh.so` (it is erased before the wrapper function returns; see below), meaning that it is never available in

[1]The 64-bit result from the `rdtsc` machine instruction indicates the value of the clock-cycle counter since the machine's last boot-up. The least-significant 32 bits of this result provide sufficient randomness for our purposes.

memory whenever any attacker code may be executing. We can thus prevent information leakage in a manner similar to Karger [36] for cross-domain calls, and hence prevent replay attacks that read and re-use nonces.

The kernel code for `syscall_libc_auth` verifies that this system-call invocation's call site is in the code section of `e-NeXSh.so`, and then stores the address and value (by dereferencing the location on the user-space stack) of the nonce in the process' PCB, and returns control to the user-space. Later on, when verifying system calls invoked via LIBC functions, the kernel checks to make sure that the nonce that was specified most recently still exists at the given location (the nonce location becomes invalid when we obliterate the nonce value from the stack, see next). Note that by setting the nonce before validating the call stack, we are able to prevent attacks that might otherwise jump into the middle of our wrapper functions to skip the initial checks.

After issuing the authorisation system call and validating the call stack, the wrapper function resumes the program execution by passing control to the original LIBC function (referenced via explicit `dlopen`/`dlsym` calls) which may, in turn, invoke system calls. These system calls are accepted by the kernel since not only are they invoked by a `trap` instruction located in the code section (of LIBC), but also because the nonce value that was passed to the kernel is still valid. When the LIBC function completes execution, it will return control to our wrapper function. We now zero out the nonce value on the stack (to eliminate the possibility of any mimicry attack re-using the nonce), and return to the application. We reduced the overhead from an earlier implementation that involved a separate system call to indicate to the kernel that the current `e-NeXSh.so` wrapper had completed its execution, and was returning control to the application.

Multi-threaded applications may enable attackers to use a parallel thread to invoke malicious system calls (this thread would still need to use a libc-based attack to avoid issuing a system call directly from data memory) after getting a legitimate program thread to carry out the system-call authorisation. We counter this threat by implementing semaphore-based synchronisation in the `e-NeXSh.so` wrapper functions to ensure that multiple threads are not concurrently allowed into the critical LIBC (and subsequent system-call) functions. This could potentially cause synchronisation problems with system calls that block on I/O, e.g., `read` on a socket interface. However, none of the LIBC functions we currently protect are blocking functions.

### 3.2.3 Attacks Against e-NeXSh

An attacker could attempt a direct target at the e-NeXSh protection mechanism by overwriting the program stack to make it appear as a valid run for the given program. This allows the attacker to issue any system call that the application could invoke during normal execution. The difference is, of course, in the system-call parameter being provided: an attacker could exploit this loophole to access critical files by compromising any program that does file I/O. This problem is similar to that of attackers overwriting critical program data that are used as parameters for system calls, and a general solution is to manually define policies [49] to dictate the set of file-system resources that each program can access, for instance.

This loophole in e-NeXSh allows an attacker to create a fake, but seemingly valid, stack, and consequently pass both the LIBC- and kernel-based checks. Such attacks have been demonstrated in [17] to **mimic a valid program stack**, allowing them to successfully bypass commercial-grade sandboxing products and consequently invoke system calls. Later, we describe how we can counter these attacks by using simple randomisation techniques to make it significantly harder for an attacker to re-create a valid stack.

Kruegel et al. [41] describe a binary code-analysis method involving symbolic execution [39] of victim programs, to construct attacks that can automatically regain program control even after issuing system calls (possibly by using faked stacks). These are mimicry attacks of a different kind [41, 60] in that they intend to invoke a number of system calls **matching a valid audit trail** for the given program, enabling them to evade detection by traditional host-based IDS systems [34, 42, 43, 58, 65] that may be monitoring such program audit trails. Kruegel's attacks use faked stack traces to thwart Feng's [27] and Sekar's [53] stack-verification techniques, and then repeatedly regain program control, allowing them to defeat the defence techniques' audit-trail monitoring mechanisms. In this manner, Kruegel has reduced the task of breaking techniques such as [27, 53] into a matter of invoking system calls a number of times.

A critical requirement for Kruegel's generated attacks to successfully regain program control after a system-call invocation is their need to maliciously modify code pointers, specifically, entries in the Procedure Linkage Table (PLT)[2]. In this regard, we can trivially render Kruegel's method ineffective by extending `e-NeXSh.so`'s initialisation (when the program is loaded) to carry out eager evaluation of the program's PLT, and subsequently write-protecting the PLT to prevent any updates. Preventing the attack from updating these code pointers has the effect that it eliminates the possibility of Kruegel's attacks regaining program control after attempting a system-call invocation. However, even this may be redundant given the increased difficulty in creating a fake stack that can evade `e-NeXSh.so`, as described next.

The core elements of creating a fake stack are: determine what memory locations in the stack should contain the

[2]Kruegel states that other code pointers, e.g., function pointers, do not reliably produce a successful exploit

return address and old frame-pointer values for activation records, and overwrite these locations with values that are valid for a normal program run. We employ randomisation on different portions of the process memory layout to make it hard to fake the stack. We first offset the starting location of the program stack by a random 16-bit value as in [14], increasing the difficulty for an attacker in figuring out **which** memory addresses on the stack to overwrite. We can also use either a suitably modified compiler [22, 37] or binary-editing tool [47] to obfuscate (using 32-bit XOR keys) the stored values in the stack-frame headers for the return address and old frame-pointer value. Note that the XOR keys need to be communicated to and stored in `e-NeXSh.so` to reconstruct the original return address and old frame-pointer values when traversing and validating the stack — the location where these values are stored can be randomised, and protected against scanning attacks by storing between unmapped memory pages (an attacker trying to read from these pages will cause a memory protection violation, and thus crash the process).

Obfuscating the stored return address and old-frame pointer values significantly increases the difficulty for an attacker in determining **what** values to write on the stack frames to simulate a valid stack. Even if he does figure out which locations on the stack to overwrite, he would need to know the XOR values for both the return address and old frame-pointer. Finally, we also randomise the starting offset (as in [14]) for the `.text` segment by a random 24-bit value, thus randomising function addresses in the program code. This further increases the difficulty for an attacker to figure out **what** values (for function return addresses) to write on the program stack to mimic a valid stack.

These are standard compile-time or program load-time techniques for obfuscating the program's stack segment and code segment layout, and can be implemented for no additional run-time cost while still increasing the work-load for a brute-force attack by a factor of up to $2^{104}$ (16 + 32 + 32 + 24). Combining e-NeXSh's system-call and LIBC function-call monitoring with these obfuscation techniques addresses their mutual shortcomings, and makes for a more powerful defence system than with either technique in isolation. For instance, a program protected solely by these obfuscation techniques, i.e., in the absence of our call monitoring, can trivially be defeated by an attacker simply injecting and executing code in data memory, or overwriting a program code pointer to invoke a LIBC function — this is demonstrated by brute-forcing attack techniques as reported in Shacham [54]. Conversely, e-NeXSh's call monitoring techniques are susceptible to stack-faking attacks in the absence of such obfuscations.

## 3.3 Operation

We now describe the normal invocation of LIBC functions by application code and a libc-based attack scenario. A code-injection attack that issues system calls directly from data memory will obviously be detected by our kernel code. By defining the `LD_PRELOAD` environment variable, the program's invocations of LIBC functions are directed into the appropriate wrapper function in `e-NeXSh.so` which issues the authorising system call, validates the program call stack, and invokes the intended LIBC function, in order. On the other hand, a standard libc-based attack will generally transfer control directly into the original LIBC function. Consequently, when this function issues system call(s), our kernel component will correctly reject them due to the lack of authorisation (i.e., the last specified nonce location has been invalid since the last invocation of any `e-NeXSh.so` wrapper function). A more sophisticated libc-based attack that inspects the entries in the Procedure Linkage Table (PLT) or Global Offset Table (GOT) can invoke the relevant wrapper function in our shared library. However, the call stack-validation code will detect the deviation from the program's normal behaviour, and can log the attack attempt and halt the process by issuing a `SIGKILL` signal. Directly issuing system calls for both logging and signaling and also not returning control to the application makes our technique invulnerable to extended attacks, e.g., [16, 41], that may have compromised the logging- or exit-handler code as a means of regaining control.

Figure 3 illustrates a successful, legitimate invocation of the `sys_execve` system call by function `foo` in the application code, as well as an unsuccessful attempt by the malicious code `shellcode` to invoke functions in LIBC. As a result, malicious code can invoke system calls neither directly nor indirectly via LIBC functions.

## 3.4 Transparency of Use

Our implementation imposes no interference on normal system operations. Firstly, we can enforce protection on only those programs that need security, e.g., server daemons like `httpd`, `sendmail`, `smbd`, `imapd`, and leave the rest of the system unaffected. The user-space component `e-NeXSh.so` can be disabled by simply not setting the `LD_PRELOAD` environment variable. We created a flag in the kernel's Process Control Block (PCB) data structure that has to be set in order for our kernel component to validate system-call invocations. This flag can be set by a variety of means in the kernel's process loading code: (a) the `LD_PRELOAD` environment variable is set (and contains `e-NeXSh.so`), (b) the program's name contains the prefix `enxProtect_`, or (c) the program image contains an ELF [56] section titled `enexsh`, consisting of pre-computed disassembly information for the program and the referenced libraries.

```
1 #include <string.h>
2
3 char *shellcode =
4   "\xeb\x1f\x5e\x89\x76\x08\x31\xc0"
5   "\x88\x46\x07\x89\x46\x0c\xb0\x0b"
6   "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c"
7   "\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
8   "\x80\xe8\xdc\xff\xff\xff/bin/sh";
9
10 int main(void) {
11   char buffer[96];
12   int i = 0, *p = (int *) buffer;
13   while (i++ < 32) *p++ = (int) buffer;
14   strncpy (buffer, shellcode,
15       strlen (shellcode));
16   return 0;
17 }
```

**Figure 4.** *Sample stacksmashing attack.*

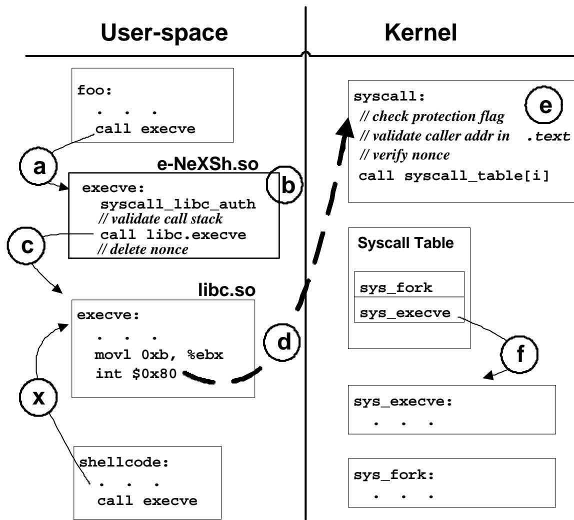


**Figure 3.** *The sequence (abcdef) indicates a successful system-call invocation by "valid" function* `foo`*, being routed through the new* `execve` *wrapper function in* `e-NeXSh.so` *and the original LIBC wrapper function* `execve`*, in order. Step (b) represents the operations of the new wrapper function, starting with it informing the kernel to authorise the invocation of* `sys_execve`*, and ending with the stack trace and caller-callee verifications, just before before calling the original LIBC function. Step (d) signifies the execution of the trap instruction, and the triggering of the system-call signal handler in the kernel, and step (e) represents the kernel performing its own checks, verifying that the calling code's virtual memory address is in read-only memory, and that the currently requested system call is authorised (by checking the nonce). The sequence (xde) indicates an unsuccessful attempt by attack code in* `shellcode` *to invoke the same system call via LIBC. The attack fails the nonce check at step (e), because the system-call invocation has not been authorised.*

Our technique allows for code execution on the stack and heap as long as it does not invoke a system call or LIBC function. A benefit of this feature is that e-NeXSh does not break applications with a genuine need for an executable stack, e.g., LISP interpreters. However, by preventing system-call or LIBC function invocations from the stack or heap, e-NeXSh creates an "effectively" non-executable stack and heap. This is an improvement over true non-executable stack and heap techniques that either break legitimate applications, or require significant effort to allow such code to execute on the stack or heap.

# 4 Evaluation

In this section, we report on our evaluations of the efficacy and usability of our technique. We conducted experiments to measure both its effectiveness in protecting the system from various attacks, and impact to the system performance.

## 4.1 Efficacy

The foremost objective of any protection mechanism is successful defence against attacks. We have used e-NeXSh to defeat stack- and heap-based code-injection attacks based on our modified version of Aleph One's stacksmashing attack (figure 4), and an example libc-based attack using the sample program in figure 5. To further test our technique's effectiveness against attacks, we ran a benchmark test-suite compiled by Wilander and Kamkar [64]. This code-injection attack test-suite contains 20 different buffer-overflow techniques to overwrite vulnerable code pointers, viz., function's return address, function pointer variables, old frame pointer, and `longjmp` buffers, in all of the stack, heap, `.bss` and data segments of process memory. e-NeXSh was able to achieve 100% effectiveness by defeating all 20 attacks, a significant improvement over the mere 50% achieved by ProPolice [25], the best buffer overflow detector tool evaluated by Wilander. A dynamic buffer-overflow detector [51] also managed 100% effectiveness using this test-suite, however with a considerably larger (upto 130%) overhead. We discuss performance issues due to e-NeXSh next.

## 4.2 Usability

The usability issues related to incorporating any security mechanism are manifold, including impact on system performance, increase in program disk or memory usage, any other interference with normal system operation (e.g., breaking certain applications), and ease of incorporating the

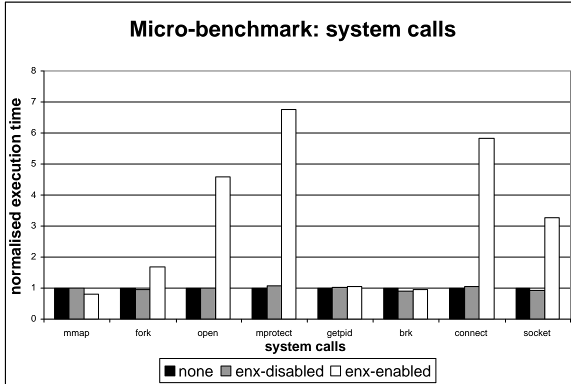


**Figure 6.** *e-NeXSh micro-benchmark: System calls. The three columns indicate mean execution time for each system call on: (a) a stock Linux system, (b) an e-NeXSh-enhanced Linux system, but without applying either kernel or LIBC protection, and (c) full e-NeXSh protection. The values are normalised to column (a), i.e., the mean execution time for stock Linux.*

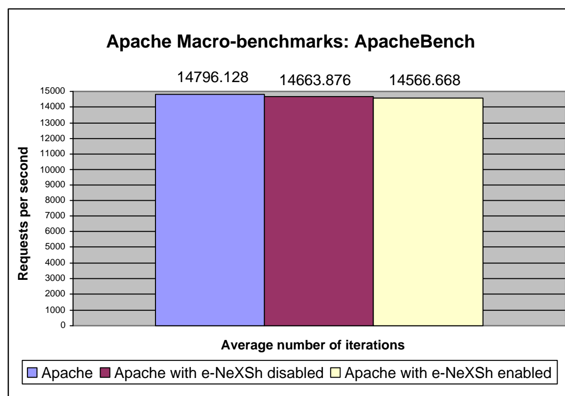


**Figure 7.** *e-NeXSh macro-benchmark: ApacheBench. The height of each column indicates the number of requests that Apache could process each second.*

```
 1 #include <unistd.h>
 2
 3 char *argv[] = { "/bin/sh" };
 4 int main(void) {
 5   void **fp = (void **) &fp;
 6   fp[2] = execve;
 7   fp[4] = argv[0];
 8   fp[5] = argv;
 9   fp[6] = 0;
10   return 0;
11 }
```

**Figure 5.** *Sample libc-based attack.*

technique (e.g., any manual effort required). We focus primarily on the impact to system performance due to our interception of LIBC function and system-call invocations.

### 4.2.1 Execution Overheads

We ran a set of micro-benchmarks to determine the worst-case performance hit on system-call invocations (via LIBC), and macro-benchmarks to determine the performance impact on real-world software. We used the ApacheBench benchmarking tool from the Apache HTTP Server project [1] and several common UNIX tools for the macro-benchmarks. Our results show that e-NeXSh imposes a small performance overhead on average for Apache-1.3.23-11 (∼1.5% reduction in request-processing capability), and tools like `tar` and `gzip`. These benchmarks were compiled with the `gcc-3.2` compiler with no debugging, and no optimisation. The benchmarks were run on a 1.2GHz Pentium III machine with 256MB of RAM. Timing measurements were made using the `gettimeofday` function for the micro-benchmark tests, and the UNIX `time` tool for the macro-benchmark tests. ApacheBench provided its own timing results.

**Micro-benchmarks** We wrote a C program that uses the `gettimeofday` function to measure the system time to invoke each LIBC function (corresponding to the system call, see next) `LIMIT` times in a loop (`LIMIT` == 10,000). We then divided the total time by `LIMIT` to get the mean execution time per invocation. We collected such values from 10 runs, and averaged the median 8 values for each system call. Figure 6 shows the results for individual micro-benchmarks to assess upper-bound performance overhead for several potentially dangerous system calls (that would be used by an attacker), a system call (`brk`) frequently invoked for dynamic memory allocations, and a lightweight system call (`getpid`).

Note the spikes for `open`, `mprotect`, `connect` and `socket` system calls, indicating overheads of almost $\times 7$. We ran additional tests with varying `LIMIT` (1, 10, 100, 1000, and 10000) for each of these system calls, and discovered that the e-NeXSh overhead per system-call invocation (the overhead settles in the range 3–8$\mu s$ per 2–4$\mu s$ system-call execution time) became more prominent at higher iterations — these additional results are tabulated in appendix A.

**Macro-benchmarks** We chose the ApacheBench benchmarking suite for the Apache [1] HTTP server project as a realistic benchmark for evaluating the performance impact of e-NeXSh. Apache is ideal for this purpose owing to its wide-spread adoption, and to the fact that it exercises many of the wrapper functions in `e-NeXSh.so` such as `connect`, `execve`, `fork`, `open`, and `socket`. We ran the `ab` tool from ApacheBench with the parameters: `-n 10000`

| Benchmark | Normal in seconds | e-NeXSh in seconds | Overhead in percent |
|---|---|---|---|
| ctags | 9.98±0.14 | 9.91±0.10 | -0.60±1.93 |
| gzip | 10.98±0.62 | 11.19±0.44 | 2.09±6.45 |
| scp | 6.30±0.04 | 6.29±0.04 | -0.15±0.96 |
| tar | 12.89±0.28 | 13.12±0.46 | 1.84±3.91 |

**Table 3.** *e-NeXSh macro-benchmark: UNIX utilities,*

`-C 1000 http://localhost/50KB.bin` to simulate 1000 concurrent clients making a total of 10,000 requests for a 50KB file through the `loopback` network device (i.e., on the same host as the web server, to avoid network latency-induced perturbations to our results). We collected and averaged the results for 5 runs of `ab` for each server configuration (see next). The machine was otherwise unloaded.

Table 2 shows the results of our macro-benchmark tests (also plotted in figure 7). The Apache server suffers a small ∼1.55% decrease (though the standard deviation indicates that this decrease is insignificant) in request-handling capacity while running under full e-NeXSh protection (column 3, Apache +ENX), as compared to running on a stock Linux system (column 1, Apache). Column 2 (Apache -ENX) signifies the Apache server running unprotected on top of an e-NeXSh-enhanced kernel, however with e-NeXSh disabled.

| Apache | Apache (-ENX) | Apache (+ENX) |
|---|---|---|
| 14796.13 ± 244.96 | 14663.88 ± 142.90 | 14566.67 ± 197.86 |
| (decrease in throughput) | -0.87% ± 1.98 | -1.55% ± 2.14 |

**Table 2.** *e-NeXSh macro-benchmark: ApacheBench. The values in the first data row indicate the number of HTTP requests handled per second (averaged over 5 runs, and corresponding standard deviation) by the server in each configuration. The second row shows the average decrease in throughput in comparison to Apache running on stock Linux.*

We also ran some tests to determine e-NeXSh's impact on the performance of common, non-server UNIX programs. Table 3 shows the execution time (averaged over 5 runs, and corresponding standard deviation) for the `ctags`, `gzip`, `scp` and `tar` utilities, measured using the UNIX `time` command. Three of these tests involved a local `glibc-2.2.5` code repository: we ran `ctags` to generate indexes for the GLIBC source code, `tar` to create an archive of the source code repository, and `scp` to upload the archive file to a remote server (using public keys for automatic authentication). We also created a 50MB file by reading bytes from `/dev/random`, and we compressed this file using `gzip` (it compressed the GLIBC archive too fast for us to get a meaningful time measurement). These results show that overhead added by e-NeXSh to these common programs is insignificant: in the final column, the reported overhead is consistently smaller in magnitude than the standard deviation.

#### 4.2.2 Other Usability Issues

Recall that e-NeXSh operates completely transparently to applications and existing libraries, meaning that their disk usage remains constant. The run-time memory usage of protected programs increases by the size of the data structure representing function boundaries (at 16 bytes per function, Apache-1.3.23-11 has 790 functions, i.e., increased memory footprint of ∼3KB) in `e-NeXSh.so`. If the `e-NeXSh.so` shared library is customised for each application, this will increase the disk usage marginally (currently 34KB).

Once the kernel has been patched, and a suitable `e-NeXSh.so` shared library made available, applying e-NeXSh protection to an application is a simple matter of by setting the `LD_PRELOAD` environment variable. Unlike the non-executable stack and heap techniques, e-NeXSh does not break applications that require execution of code on the stack or heap, or require complex workarounds to "unbreak" them. Also, extending `e-NeXSh.so` with a new wrapper is very easy, requiring only a one-line C macro to define the wrapped LIBC function's signature.

## 5 Related Work

Our technique is similar to existing work in three general areas of security research: (a) system-call interposition techniques for process sandboxing or intrusion detection, (b) techniques that prevent the execution of injected code, and (c) address obfuscation techniques to combat libc-based attacks. Though there exist certain overlaps in these areas, we discuss each area separately.

### 5.1 System-Call Interposition

System-call interception-based intrusion-detection systems [18, 32, 34, 58, 62, 63] are similar to our technique in that they passively observe process behaviour. The observed behaviour of the running program, as signified by its audit trails, i.e., the sequence of system calls issued by the running program, is validated against an abstract Finite State Machine (FSM) model that represents normal execution of the monitored programs. These model can be constructed either during a training phase, or it can be generated from other compile-time information about the program. Running these systems within acceptable overheads has

generally resulted in loss of precision, yielding a large number of false positives, and sometimes even false negatives. Giffin [33], another system call-driven, intrusion-detection system, is an improvement over Wagner and Dean's technique [58] that uses static disassembly techniques on Solaris executables to create a precise, yet efficient model (based on the Dyck language) to represent the beginning and end of function invocations. Feng [27] and Rabek [50] take the concept of system-call interception a step further by inspecting return addresses from the call stack to verify them against the set of valid addresses for the program. This is similar to our (kernel-level) concept of validating the virtual memory address of code that issues the `trap` instruction, and the validation of the call stack return addresses in `e-NeXSh.so`. However, these systems incur larger overheads as they get the kernel to extract and verify individual return address values from the program call stack, whereas we only have to validate a single address in the kernel. Instead, we verify the user-space call stack completely within our user-space component. Another important advantage of our system is the simplicity of our technique — instead of an FSM-based model, we simply use the program code (as a call-graph model) to validate program call stacks.

Gao's evaluation [29] concludes that mimicry attacks can break anomaly-detection techniques that intercept system calls and analyse audit trails. These mimicry attacks exploit the fact that such anomaly-detection techniques define normal program behaviour in terms of audit trails. Our technique is not similarly vulnerable to these attacks since we monitor not audit trails over a period of time, but rather the entire call stack to validate against the static program code. In §3.2.3, we discussed the ineffectiveness against e-NeXSh of Kruegel [41], a similar method for automating mimicry attacks against certain classes of intrusion-detection systems. A Phrack article [17] presents a mimicry attack [29, 41] (using faked stack frames) to defeat two commercial sandboxing products for the Windows operating system, viz., *NAI Entercept* and *Cisco Security Agent*, that perform kernel-based verification of the return addresses on the user-space stack and the return address of the `trap` call. These defence techniques are tricked into accepting the faked stack frames since they only check that the return addresses from the stack-trace exist in a `.text` section. Our full caller-callee validation in `e-NeXSh.so` combined with the stack- and code-segment obfuscations (§3.2.3) would thwart a Linux version of this attack, given that we make it much harder to fake the stack.

Linn et al. [45] present a defence technique that is very similar to e-NeXSh is terms of its objectives and methods. They also use the locations of `trap` instructions in code memory to identify illegal invocations of system calls by code-injection attacks — whereas our kernel module simply inspects the "return address" of system-call invocations and checks the memory page's read/write flag, Linn's technique uses the PLTO [52] binary rewriting tool to pre-process executable files to construct an Interrupt Address Table (IAT) of valid sites for system calls. The IAT is loaded by the kernel, and referenced for a matching "return address" entry when validating each system-call invocation during the program run. Linn's technique inherits the PLTO tool's inability to handle dynamically linked executables, and hence has to include all referenced library code, e.g., LIBC, in a single static executable to deal with the `trap` instructions in the LIBC code. e-NeXSh, on the other hand, still only needs to verify that the "return address" of the `trap` instruction exists in a write-protected memory area — for LIBC, this would be the code-segment of `libc.so` in the program memory.

Besides their method of monitoring system-call instructions to identify code-injection attacks, Linn also includes a mechanism parallel to our `e-NeXSh.so`, i.e., for identifying attacks that use existing `trap` instructions in the program (or library) code to invoke system calls. Linn classifies these attacks into "known address" and "scanning" categories, and focuses on using obfuscation techniques to defeat such "scanning" attacks, including (a) using the PLTO tool to replace the `trap` instructions with other machine instructions that are guaranteed to also cause a kernel trap, (b) removing from executables any symbolic information that might aid an attacker in figuring out where the `trap` instructions were replaced, (c) interspersing `nop`-equivalent instructions in the program code, and (d) interspersing the address space of the executable with `munmap`'d memory pages. This collection of obfuscation techniques serves to prevent a "scanning" attacker from using an existing `trap` **location** in the program code to invoke system calls. However, when compared to `e-NeXSh.so` that accomplishes the same purpose, we see that Linn's performance overhead (15%) is approximately $\times 10$ greater than that for e-NeXSh. Linn attributes their large overhead primarily to a degraded instruction-cache performance, and points out that their layout randomisation easily leads to a high rate of TLB misses. Another disadvantage of their technique is the need to modify the executable files in a highly intrusive fashion, which is likely to complicate matters for both debugging purposes and interoperability with other, independent defence techniques.

Some intrusion-detection systems require manual effort to define and update explicit policies [18, 49] to restrict programs' run-time operations. Our technique obviates the need for such explicit policies: instead, we use the program code in the `.text` segment and its disassembly information as guidelines for an implicit policy.

### 5.2 Defence Against Code-Injection Attacks

Process-specific randomised instruction sets [13, 38] and process shepherding [40] have demonstrated resilience against code-injection attacks by only permitting trusted code to execute, where the trust is dictated by the origins of the code. These systems rely heavily on the use of machine emulators or binary translators incurring large overheads, and hence are unsuitable for real-world use.

Techniques like [3, 5, 10, 55] protect against code-injection attacks by making the program stack, heap and static data areas non-executable. By default, these data areas are mapped to memory pages marked writable in the Linux operating system. Since the 32-bit x86 architecture only provides support to specify whether individual memory pages are writable and/or readable, there is no efficient means of specifying whether a given page is executable. This has resulted in operating systems like Linux considering readable pages as also being executable. These non-executable stack and heap techniques [3, 55] have developed a software solution for distinguishing the readable and executable capabilities for individual pages, and have been successful in preventing the execution of code in these areas, although in a mostly non-portable manner. A critical drawback of these approaches is that they break code with legitimate need for an executable stack, prompting the development of complex workarounds to facilitate such code, e.g., trampolines for nested functions (a GCC extension to C) and signal-handler return code for Linux.

Recent processors [30] provide native hardware support for non-executable pages via a NoExecute (NX) flag. This, however, will serve only to make redundant the code used to emulate the per-page execute bit — the complex workarounds and associated overheads to allow executable stacks and heaps for applications that require them still remain. Furthermore, these techniques cover only a subset of exploitation methods (e.g., existing-code or libc-based attacks are still possible).

Our approach can also be thought of as making data memory non-executable for the purposes of injected code invoking system calls or LIBC functions. However, our technique does not prohibit the execution of most code that has been deposited into data memory (the exception is the `trap` instruction to make system calls), making it possible to run applications that require an executable stack.

### 5.3 Address Obfuscation

Address-obfuscation techniques [3, 14, 20] can disrupt libc-based attacks by randomising the locations of key system library code and the absolute locations of all application code and data, as well as the distances between different data objects. Several transformations are used, such as randomising the base addresses of memory regions (stack, heap, dynamically linked libraries, routines, and static data), permuting the order of variables and routines, and introducing random gaps between objects (e.g., by randomly padding stack frames or `malloc()`'d regions). However, Shacham et al. [54] recently demonstrated the futility of such address-obfuscation techniques for 32-bit systems (they can only utilise $\leq$16 bits of randomness) by creating an attack to defeat PaX's address space layout randomisation in 216 seconds. e-NeXSh is not vulnerable to this attack since we do not obfuscate the memory addresses of LIBC functions. The secret component in our technique, i.e., the nonce, is reliably secure against re-use by attackers since we create and destroy the nonce values entirely within `e-NeXSh.so`. Furthermore, we employ up to 104 bits (compare to 16bits for the tests in [54]) of randomness, which greatly increases the difficulty for an attacker.

## 6 Open Issues and Future Work

An underlying assumption in our work is that an attack needs to interact with the system outside its compromised process, and that this interaction can be tightly monitored and controlled by the OS kernel. Linux allows an application to carry out memory-mapped I/O without having to issue system-calls except for one initial call to the `mmap` system call. The techniques presented in this paper cannot detect when a compromised process is performing memory-mapped I/O. However, such an attack is effective only against a program that has already set up memory-mapped access to critical files.

e-NeXSh is incompatible with systems that involve copying executable code to data sections for the purposes of execution — this will immediately be flagged as execution of injected code, and the process will be halted. For instance, techniques like LibVerify [12] and Program Shepherding [40] that require execution of managed or shepherded code stored in data pages cannot be used in conjunction with e-NeXSh.

One deficiency of our system is that it does not protect against attacks that exploit vulnerabilities to overwrite crucial (non-code pointer) data. This could enable the attacker to bypass application-specific access-control checks, or, in extreme cases, even be able to specify the parameter for the program's own invocation of the `system` call. However, few techniques [18, 49] monitor system-call parameters to protect against such attacks, and only with manually edited, explicit policies.

### 6.1 Future Work

Our implementation relies on program and library disassembly for validating stack traces, and is currently un-

able to carry out proper user-space call-stack validation either if optimising compilers have been used to produce code without the `old-frame-pointer` entry in stack frames (i.e., cannot do stack traces), or if the program executable has been `strip`'d of symbols (i.e., cannot disassemble the code). The obvious solution to this problem involves imposing certain build-time constraints — application code will need to be compiled with the `old-frame-pointer` enabled, and the executables cannot be run through `strip`. Another possibility is to use more robust disassemblers like IDA-Pro [8].

A possible future direction for our work is to relocate the user-space stack-verification code into the kernel. Having a self-contained e-NeXSh mechanism in the kernel will allow for a simpler design, avoiding the need for an extra system call or storage space for a nonce in the PCB. Furthermore, the call-stack verification can be extended to monitor library code in statically linked executables. However, this decision could also lead to larger performance overheads as the kernel has to validate the user-space stack.

We are working to improve the handling of code pointers in the call stack. In addition to collecting information about the set of acceptable use for function pointers in the call stack during training stages, we are considering the use of static-analysis techniques combined with some run-time program data [11] to compute full call graphs for programs.

## 7 Conclusions

We have presented a technique that makes use of information about a process' run-time memory, creating an implicit policy to efficiently monitor all system call and LIBC function invocations made by the process. This helps in defeating process-subversion attacks from causing any damage outside of the compromised process. This technique has demonstrated successful protection of software against both code-injection and libc-based attacks, using Wilander's test-suite [64] in addition to our own synthetic effectiveness benchmarks. We have established that our approach is both feasible and economical, imposing negligible overheads on Apache and common UNIX utilities, and is applicable for both legacy and closed-source applications since we do not require any changes to application source code.

## 8 Acknowledgements

We would like to thank Alfred Aho for his invaluable comments and insights during the many discussions on the techniques presented in this paper. This work was supported in part by the National Science Foundation under grant ITR CNS-0426623.

## References

[1] Apache HTTP Server Project. `http://httpd.apache.org`.

[2] CERT/CC Advisories. `http://www.cert.org/advisories`.

[3] Pax: Non-executable data pages. `https://pageexec.virtualave.net`.

[4] Phrack: . . . a Hacker community by the community, for the community . . . . `http://www.phrack.org/`.

[5] RedHat Linux: Exec Shield. `http://people.redhat.com/mingo/exec-shield`.

[6] SecuriTeam Exploits. `http://www.securiteam.com/exploits/`.

[7] SecurityFocus BugTraq Mailing List. `http://www.securityfocus.com/archive/1`.

[8] The IDA Pro Disassembler and Debugger. `http://www.datarescue.com/idabase/`.

[9] The Metasploit Project: payloads for Linux. `http://www.metasploit.org/shellcode_linux.html`.

[10] The OpenBSD Project 3.3 Release: Write XOR Execute. `http://openbsd.org/33.html`.

[11] D. C. Atkinson. Call Graph Extraction in the Presence of Function Pointers. In *Proceedings of the 2002 International Conference on Software Engineering Research and Practice*, June 2002.

[12] A. Baratloo, N. Singh, and T. Tsai. Transparent run-time defense against stack smashing attacks. In *Proceedings of the 2000 USENIX Annual Technical Conference*, June 2000.

[13] G. Barrantes, D. H. Ackley, T. S. Palmer, D. D. Zovi, S. Forrest, and D. Stefanovic. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the ACM Computer and Communications Security (CCS) Conference*, October 2003.

[14] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address Obfuscation: an Effi cient Approach to Combat a Broad Range of Memory Error Exploits. In *Proceedings of the 12th USENIX Security Symposium*, pages 105–120, August 2003.

[15] Brett Hutley. SANS Malware FAQ: The BH01 worm. `http://www.sans.org/resources/malwarefaq/bh01.php`.

[16] Bulba and Kil3r. Bypassing StackGuard and StackShield. *Phrack*, 5(56), May 2000.

[17] J. Butler. Bypassing 3rd party windows buffer overfbw protection. *Phrack*, 11(62), July 2004.

[18] S. Chari and P. Cheng. BlueBox : A Policy-Driven, Host-Based Intrusion Detection System. In *Proceedings of the 9th Network and Distributed System Security Symposium (NDSS)*, February 2002.

[19] H. Chen and D. Wagner. MOPS: an Infrastructure for Examining Security Properties of Software. In *Proceedings of the ACM Computer and Communications Security (CCS) Conference*, pages 235–244, November 2002.

[20] M. Chew and D. Song. Mitigating Buffer Overfbws by Operating System Randomization. Technical Report Computer Science Technical Report 65, Carnegie Mellon University, December 2002.

[21] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: Protecting Pointers From Buffer Overflow Vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium*, pages 91–104, August 2003.

[22] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, Jan. 1998.

[23] Edward Ray. SANS Malware FAQ: MS-SQL Slammer. `http://www.sans.org/resources/malwarefaq/ms-sql-exploit.php`.

[24] eEye Digital Security. ANALYSIS: Sasser Worm. `http://www.eeye.com/html/Research/Advisories/AD20040501.html`.

[25] J. Etoh. GCC extension for protecting applications from stack-smashing attacks. `http://www.trl.ibm.com/projects/security/ssp`, June 2000.

[26] H. H. Feng, J. T. Giffin, Y. Huang, S. Jha, W. Lee, and B. P. Miller. Formalizing Sensitivity in Static Analysis for Intrusion Detection. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2004.

[27] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2003.

[28] M. Frantzen and M. Shuey. StackGhost: Hardware facilitated stack protection. In *Proceedings of the USENIX Security Symposium*, pages 55–66, August 2001.

[29] D. Gao, M. K. Reiter, and D. Song. On gray-box program tracking for anomaly detection. In *Proceedings of the 13th USENIX Security Symposium*, pages 103–118, August 2004.

[30] L. Garber. New Chips Stop Buffer Overflow Attacks. *IEEE Computer*, 37(10):28, October 2004.

[31] T. Garfinkel, B. Pfaff, and M. Rosenblum. Ostia: A Delegating Architecture for Secure System Call Interposition. In *Proceedings of the 11th Network and Distributed System Security Symposium (NDSS)*, pages 187–201, February 2004.

[32] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of the 10th Network and Distributed System Security Symposium (NDSS)*, pages 191–206, February 2003.

[33] J. Giffin, S. Jha, and B. P. Miller. Efficient context-sensitive intrusion detection. In *Proceedings of the 11th Network and Distributed System Security Symposium (NDSS)*, 2004.

[34] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, 1998.

[35] R. W. M. Jones and P. H. J. Kelly. Backwards-compatible bounds checking for arrays and pointers in C programs. In *Third International Workshop on Automated Debugging*, 1997.

[36] P. A. Karger. Using registers to optimize cross-domain call performance. In *Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 194–204, 1989.

[37] G. S. Kc, S. A. Edwards, G. E. Kaiser, and A. Keromytis. CASPER: Compiler-Assisted Securing of Programs at Runtime. Technical Report TR CUCS-025-02, Columbia University, New York, NY, November 2002.

[38] G. S. Kc, A. D. Keromytis, and V. Prevelakis. Countering Code-Injection Attacks With Instruction-Set Randomization. In *Proceedings of the ACM Computer and Communications Security (CCS) Conference*, pages 272–280, October 2003.

[39] J. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7), 1976.

[40] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *Proceedings of the 11th USENIX Security Symposium*, pages 191–205, August 2002.

[41] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Automating mimicry attacks using static binary analysis. In *Proceedings of the 14th USENIX Security Symposium*, July 2005.

[42] L. C. Lam and T. Chiueh. Automatic Extraction of Accurate Application-Specific Sandboxing Policy. In *Proceedings of the $7^{th}$ International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 1–20, September 2004.

[43] W. Lee, S. Stolfo, and P. Chan. Learning patterns from unix process execution traces for intrusion detection. In *Proceedings of the AAAI97 workshop on AI methods in Fraud and risk management*, 1997.

[44] K. Lhee and S. J. Chapin. Type-assisted dynamic buffer overflow detection. In *Proceedings of the 11th USENIX Security Symposium*, pages 81–90, August 2002.

[45] C. M. Linn, M. Rajagopalan, S. Baker, C. Collberg, S. K. Debray, and J. H. Hartman. Protecting against unexpected system calls. In *Proceedings of the 14th USENIX Security Symposium*, July 2005.

[46] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-Safe Retrofitting of Legacy Code. In *Proceedings of the Principles of Programming Languages (PoPL)*, January 2002.

[47] M. Prasad and T. Chiueh. A Binary Rewriting Defense Against Stack-based Buffer Overflow Attacks. In *Proceedings of the USENIX Annual Technical Conference*, pages 211–224, June 2003.

[48] V. Prevelakis and D. Spinellis. Sandboxing Applications. In *Proceedings of the USENIX Technical Annual Conference, Freenix Track*, pages 119–126, June 2001.

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

[50] J. Rabek, R. Khazan, S. Lewandowski, and R. Cunningham. Detection of injected, dynamically generated and obfuscated malicious code. In *Proceedings of the Workshop on Rapid Malcode (WORM)*, 2003.

[51] O. Ruwase and M. S. Lam. A Practical Dynamic Buffer Overflow Detector. In *Proceedings of the 11th Network and Distributed System Security Symposium (NDSS)*, pages 159–169, February 2004.

[52] B. Schwarz, S. K. Debray, and G. R. Andrews. PLTO: a link-time optimizer for the intel ia-32 architecture. In *Proceedings of the 2001 Workshop on Binary Translation (WBT-2001)*, 2001.

[53] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, 2001.

[54] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh. On the Effectiveness of Address-Space Randomization. In *Proceedings of the* $11^{th}$ *ACM Conference on Computer and Communications Security (CCS)*, pages 298–307, October 2004.

[55] Solar Designer. Openwall: Non-executable stack patch. `http://www.openwall.com/linux`.

[56] Tool Interface Standards Committee. Executable and Linking Format (ELF) specifi cation, May 1995.

[57] Vendicator. Stack shield. `http://www.angelfire.com/sk/stackshield`.

[58] D. Wagner and D. Dean. Intrusion detection via static analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 156–169, Oakland, CA, 2001.

[59] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A First Step towards Automated Detection of Buffer Overrun Vulnerabilities. In *Proceedings of the 7th Network and Distributed System Security Symposium (NDSS)*, pages 3–17, February 2000.

[60] D. Wagner and P. Soto. Mimicry attacks on host based intrusion detection systems. In *Proceedings of the Ninth ACM Conference on Computer and Communications Security*, 2002.

[61] D. S. Wallach and E. W. Felten. Understanding Java stack inspection. In *Proceedinges of the 1996 IEEE Symposium on Research in Security and Privacy*, pages 52–63, May 1998.

[62] C. Warrender, S. Forrest, and B. A. Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *IEEE Symposium on Security and Privacy*, pages 133–145, 1999.

[63] A. Wespi, M. Dacier, and H. Debar. Intrusion detection using variable-length audit trail patterns. In *Proceedings of the 3rd International Workshop on the Recent Advances in Intrusion Detection (RAID)*, 2000.

[64] J. Wilander and M. Kamkar. A Comparison of Publicly Available Tools for Dynamic Intrusion Prevention. In *Proceedings of the 10th Network and Distributed System Security Symposium (NDSS)*, pages 123–130, February 2003.

[65] H. Xu, W. Du, and S. J. Chapin. Context Sensitive Anomaly Monitoring of Process Control Flow to Detect Mimicry Attacks and Impossible Paths. In *Proceedings of the* $7^{th}$ *International Symposium on Recent Advances in Intrusion Detection (RAID)*, pages 21–38, September 2004.

# A Appendix

Figures 8, 9, 10, and 11 demonstrate declining average execution time for system calls in our extended micro-benchmark tests, and indicate that the overhead due to e-NeXSh is in the range 3–8$\mu s$ per invocation.

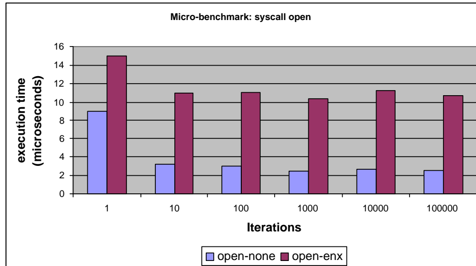


**Figure 8.** *Micro-benchmark results: open.*

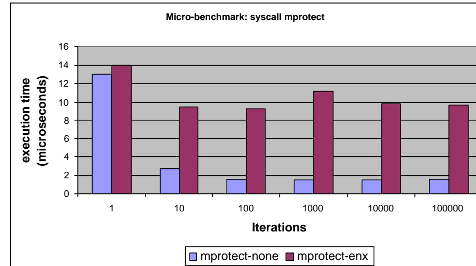


**Figure 9.** *Micro-benchmark results: mprotect.*

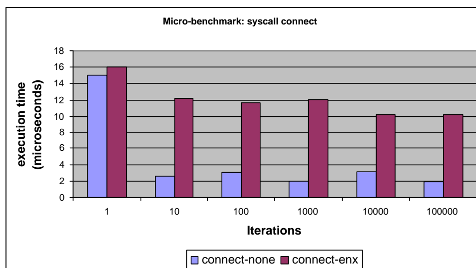


**Figure 10.** *Micro-benchmark results: connect.*

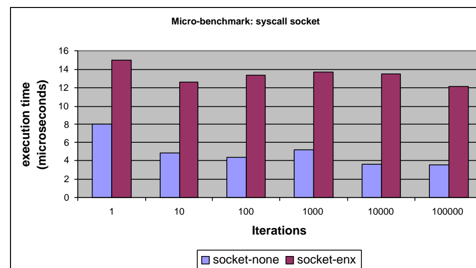


**Figure 11.** *Micro-benchmark results: socket.*