Secure Execution Via Program Shepherding

Vladimir Kiriansky, Derek Bruening, Saman Amarasinghe
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
{vlk,iye,saman}@lcs.mit.edu

Abstract

We introduce *program shepherding*, a method for monitoring control flow transfers during program execution to enforce a security policy. Program shepherding provides three techniques as building blocks for security policies. First, shepherding can restrict execution privileges on the basis of code origins. This distinction can ensure that malicious code masquerading as data is never executed, thwarting a large class of security attacks. Second, shepherding can restrict control transfers based on instruction class, source, and target. For example, shepherding can forbid execution of shared library code except through declared entry points, and can ensure that a return instruction only targets the instruction after a call. Finally, shepherding guarantees that sandboxing checks placed around any type of program operation will never be bypassed. We have implemented these capabilities efficiently in a runtime system with minimal or no performance penalties. This system operates on unmodified native binaries, requires no special hardware or operating system support, and runs on existing IA-32 machines under both Linux and Windows.

1 Introduction

The goal of most security attacks is to gain unauthorized access to a computer system by taking control of a vulnerable privileged program. This is done by exploiting bugs that allow overwriting stored program addresses with pointers to malicious code. Today's most prevalent attacks target buffer overflow and format string vulnerabilities. However, it is very difficult to prevent all exploits that allow address overwrites, as they are as varied as program bugs themselves. It is also unreasonable to try to stop malevolent writes to memory containing program addresses, because addresses are stored in many different places and are legitimately manipulated by the application, compiler, linker, and loader.

Security attacks cannot be thwarted by simply inserting checks around application code that may cause system-wide changes. A malicious entity that gains control can simply inject its own code to perform any operation that the overall application has permission to do. Hijacking trusted applications such as web servers, mail transfer agents, and login servers, which are typically run with many global permissions, gives full access to machine resources.

Rather than attempt to stop a multitude of attack paths, where the protection is only as powerful as the weakest link, our approach is to prevent the execution of malicious code. We present *program shepherding* — monitoring control flow transfers to enforce a security policy. Program shepherding prevents execution of data or modified code and ensures that libraries are entered only through exported entry points. Instead of focusing on preventing memory corruption, we prevent the final step of an attack, the transfer of control to malevolent code. This allows thwarting a broad range of security exploits with a simple central system that can itself be easily made secure. Program shepherding also provides sandboxing that cannot be circumvented, allowing construction of customized security policies.

Program shepherding requires verifying every branch instruction, which is not easily done via static instrumentation due to the dynamism of shared libraries and indirect branches. Implementation in an interpreter is the most straightforward solution. We reduce the overhead of interpretation by performing security checks once and placing the resulting trusted code in a cache, where it can be executed overhead-free in the future. Our im-

This research was supported in part by the Defense Advanced Research Projects Agency under Grant F29601-01-2-0166. This paper was originally published in the Proceedings of the 11th USENIX Security Symposium (Security '02), San Francisco, California, August, 2002.

plementation naturally fits within the RIO infrastructure, a dynamic optimizer built on the IA-32 version [3] of Dynamo [2]. The resulting system imposes minimal or no performance overhead, operates on unmodified native binaries, and requires no special hardware or operating system support. Our shepherding implementation on top of RIO is implemented for both Windows and Linux; however, this paper mainly focuses on Linux.

In Section 2 we classify the types of security exploits that we are aiming to prevent. Program shepherding's three techniques are described in Section 3, and Section 4 shows how to combine them to produce potent security policies. Section 5 discusses how we implement program shepherding efficiently, and Section 6 describes how to prevent attacks directed at our system itself. We present experimental results and the performance of our system in Section 7.

2 Security Exploits

This section provides some background on the types of security exploits we are targeting. We classify security exploits based on three characteristics: the program vulnerability being exploited, the stored program address being overwritten, and the malicious code that is then executed.

2.1 Program Vulnerabilities

The two most-exploited classes of program bugs involve buffer overflows and format strings. Buffer overflow vulnerabilities are present when a buffer with weak or no bounds checking is populated with user supplied data. A trivial example is unsafe use of the C library functions `strcpy` or `gets`. This allows an attacker to corrupt adjacent structures containing program addresses, most often return addresses kept on the stack [7]. Buffer overflows affecting a regular data pointer can actually have a more disastrous effect by allowing a memory write to an arbitrary location on a subsequent use of that data pointer. One particular attack corrupts the fields of a double-linked free list kept in the headers of `malloc` allocation units [16]. On a subsequent call to `free`, the list update operation

```
this->prev->next = this->next;
```

will modify an arbitrary location with an arbitrary value.

Format string vulnerabilities also allow attackers to modify arbitrary memory locations with arbitrary values and often out-rank buffer overflows in recent security bulletins [6, 19]. A format string vulnerability occurs if the format string to a function from the `printf` family ({`,f,s,sn`}`printf, syslog`) is provided or constructed from data from an outside source. The most common case is when `printf(str)` is used instead of `printf("%s",str)`. The first problem is that attackers may introduce conversion specifications to enable them to read the memory contents of the process. The real danger, however, comes from the `%n` conversion specification which directs the number of characters printed so far to be written back. The location where the number is stored and its value can easily be controlled by an attacker with type and width specifications, and more than one write of an arbitrary value to an arbitrary address can be performed in a single attack.

In this paper we assume that attackers can exploit a vulnerability that gives them random write access to arbitrary addresses in the program address space. This ability can be used to overwrite any stored program address to transfer control of the process to the attacker.

2.2 Stored Program Addresses

Many entities participate in transferring control in a program execution. Compilers, linkers, loaders, runtime systems, and hand-crafted assembly code all have legitimate reasons to transfer control. Program addresses are credibly manipulated by most of these entities, e.g., dynamic loaders patch shared object functions; dynamic linkers update relocation tables; and language runtime systems modify dynamic dispatch tables. Generally, these program addresses are intermingled with and indistinguishable from data. In such an environment, preventing a control transfer to malicious code by stopping illegitimate memory writes is next to impossible. It requires the cooperation of numerous trusted and untrusted entities that need to check many different conditions and understand high-level semantics in a complex environment.

Security exploits can attack program addresses stored in many different places. Buffer overflow attacks target addresses adjacent to the vulnerable buffer. The classic return address attacks and local function pointer attacks exploit overflows of stack allocated buffers. Global data and heap buffer overflows also allow global function pointer attacks and `setjmp` structure attacks. Data pointer buffer overflows, `malloc` overflow attacks, and `%n` format string attacks are able to modify any stored program address in the vulnerable application — in addition to the aforementioned addresses, these attacks target entries in the `atexit` list, `.dtors` destructor routines, and in the Global Offset Table (GOT) [12] of shared object entries.

2.3 Malicious Code

An attacker can cause damage with injection of new malicious code or by malicious reuse of already present code. Usually the first approach is taken and the attack code is implemented as new native code that is injected in the program address space as data [20]. New code can be injected into various areas of the address space: in a stack buffer, static data segment, near or far heap buffer, or even the Global Offset Table. Since normally there is no distinction between read and execute privileges for memory pages (this is the case for IA-32), the only requirement is that the pages are writable during the injection phase. Modifying any stored program address to point to the beginning of the introduced code will trigger intrusion when that address is used for control transfer.

It is also possible to reuse existing code by changing a stored program address and constructing an activation record with suitable arguments. A simple but powerful attack reuses existing code by changing a function pointer to the C library function `system`, and arranges the first argument to be an arbitrary shell command to be run. Also note that reuse of existing code can include jumping into the middle of a sandboxed operation, bypassing the sandboxing checks and executing the operation that was intended to be protected. In addition, a jump into the middle of an instruction (on IA-32 instructions are variable-sized and unaligned) could cause execution of an unintended and possibly malicious instruction stream; however, such an attack is very unlikely.

An attacker may be able to form higher-level malicious code by introducing data carefully arranged as a chain of activation records, so that on return from each function execution continues in the next function of the chain [18]. The prepared activation record return address points to the code in a function epilogue that shifts the stack pointer to the following activation record and continues execution in the next function. Overwriting a suitable sequence of function pointers may also produce higher-level malicious code.

3 Program Shepherding

The program shepherding approach to preventing execution of malicious code is to monitor all control transfers to ensure that each satisfies a given security policy. This allows us to ignore the complexities of various vulnerabilities and the difficulties in preventing illegitimate writes to stored program addresses. Instead, we catch a large class of security attacks by preventing execution of malevolent code. We do this by employing three techniques: restricted code origins, restricted control transfers, and un-circumventable sandboxing. This section describes these techniques, while Section 4 discusses how to build security policies using these techniques.

3.1 Restricted Code Origins

In monitoring all code that is executed, each instruction's origins are checked against a security policy to see if it should be given execute privileges. Code origins are classified into these categories: from the original image on disk and unmodified, dynamically generated but unmodified since generation, and code that has been modified. Finer distinctions could also be made. We describe in Section 5.3 how to distinguish original code from modified and possibly malicious code.

A hardware execute flag for memory pages can provide similar features to our restricted code origins. However, it cannot by itself duplicate program shepherding's features because it cannot stop inadvertent or malicious changes to protection flags. Program shepherding uses un-circumventable sandboxing, described in Section 3.3, to prevent this from happening. Furthermore, program shepherding provides more than one bit of privilege information: it distinguishes different types of execute privileges for which different security policies may be specified.

3.2 Restricted Control Transfers

Program shepherding allows arbitrary restrictions to be placed on control transfers in an efficient manner. These restrictions can be based on both the source and destination of a transfer as well as the type of transfer (direct or indirect call, return, jump, etc.). For example, the calling convention could be enforced by requiring that a return instruction only target the instruction after a call. Another example is forbidding execution of shared library code except through declared entry points.

3.3 Un-Circumventable Sandboxing

Program shepherding provides direct support for restricting code origins and control transfers. Execution can be restricted in other ways by adding sandboxing checks on other types of operations. With the ability to monitor all transfers of control, program shepherding is able to guarantee that these sandboxing checks cannot be bypassed. Sandboxing without this guarantee can never provide true security — if an attack can gain control of the execution, it can jump straight to the sandboxed operation, bypassing the checks. In addition to allowing construction of arbitrary security policies, this guarantee is used to enforce the other two program shepherding techniques by protecting the shepherding system itself (see Section 6).

4 Security Policies

Program shepherding's three techniques can be used to provide powerful security guarantees. They allow us to strictly enforce a safe subset of the instruction set architecture and the operating system interface. There are tradeoffs between program freedom and security: if restrictions are too strict, many false alarms will result when there is no actual intrusion. This section discusses the potential design space of security policies that provide significant protection for reasonable restrictions of program freedom. We envision a system with customizable policy settings; however, our current system implements a single security policy, which is described later in this section.

Table 1 lists sample policy decisions that can be implemented with program shepherding. Consider the policy decision in the upper right of the table: allowing unrestricted execution of code only if it is from the original application or library image on disk and is unmodified. Such a policy will allow the vast majority of programs to execute normally. Yet the policy can stop all security exploits that inject code masquerading as data into a program. This covers a majority of currently deployed security attacks, including the classic stack buffer overflow attack.

A relaxation of this policy allows dynamically generated code, but requires that it contain no system calls. Legitimate dynamically-generated code is usually used for performance; for example, many high-level languages employ *just-in-time compilation* [1, 11] to generate optimized pieces of code that will be executed natively rather than interpreted. This code almost never contains system calls or other potentially dangerous operations. For this reason, imposing a strict security policy on dynamically-generated code is a reasonable approach. Shared libraries that are explicitly loaded (i.e., with `dlopen` or `LoadLibrary`) and dynamically selected based on user input should also be considered potentially unsafe. Similarly, self-modifying code should usually be disallowed, but may be explicitly allowed for certain applications.

Direct control transfers that satisfy the code origin policies can always be allowed within a segment. Calls and jumps that transition from one executable segment to another, e.g., from application code to a shared library, or from one shared library to another, can be restricted to enforce library interfaces. Targets of inter-segment calls and jumps can be verified against the export list of the target library and the import list of the source segment, in order to prevent malevolent jumps into the middle of library routines.

Indirect control transfers can be carefully limited. The calling convention can be enforced by preventing return instructions from targeting non-call sites, and limiting direct call sites to be the target of at most one return site. Controlling return targets severely restricts exploits that overwrite return addresses, as well as opportunities for stitching together fragments of existing code in an attack.

Indirect calls can be completely disallowed in many applications. Less restrictive general policies are needed, but they require higher-level information and/or compiler support. For C++ code it is possible to keep read-only virtual method tables and allow indirect calls using targets from these areas only. However, further relaxations are needed to allow callback routines in C programs. A policy that provides a general solution requires compiler support, profiling runs, or other external sources of information to determine all valid indirect call targets. A more relaxed policy restricts indirect calls from libraries no more than direct calls are restricted (if between segments they can only target import and export entries), while calls within the application text segment can target only intra-segment function entry points. The requirement of function entry points beyond a simple intra-segment requirement prevents indirect calls from targeting direct calls or indirect jumps that validly cross executable segment points and thus avoid the restriction. It is possible to extract the valid user program entry points from the symbol tables of unstripped binaries. Unfortunately, stripped binaries do not keep that infor-

<table>
<tr><td>Restricting</td><td colspan="4">Least restrictive</td><td>Most restrictive</td></tr>
<tr><td>Code origins</td><td colspan="2">Any</td><td>Dynamically written code, if self-contained and no system calls</td><td>Only code from disk, can be dynamically loaded</td><td>Only code from disk, originally loaded</td></tr>
<tr><td>Function returns</td><td>Any</td><td>Only to after calls</td><td>Direct call targeted by only one return</td><td>Random <code>xor</code> as in StackGhost [14]</td><td>Return only from called function</td></tr>
<tr><td>Intra-segment call or jump</td><td colspan="2">Any</td><td colspan="2">Only to function entry points (if have symbol table)</td><td>Only to bindings given in an interface list</td></tr>
<tr><td>Inter-segment call or jump</td><td colspan="2">Any</td><td>Only to export of target segment</td><td>Only to import of source segment</td><td>Only to bindings given in an interface list</td></tr>
<tr><td>Indirect calls</td><td colspan="2">Any</td><td>Only to address stored in read-only memory</td><td>Only within user segment or from library</td><td>None</td></tr>
<tr><td><code>execve</code></td><td colspan="2">Any</td><td>Static arguments</td><td>Only if the operation can be validated not to cause a problem</td><td>None</td></tr>
<tr><td><code>open</code></td><td colspan="2">Any</td><td>Disallow writes to specific files (e.g., <code>/etc/passwd</code>)</td><td>Only to a subregion of the file system</td><td>None</td></tr>
</table>

Table 1: Sample list of policies built using program shepherding. Each row shows a continuum of choices ranging from most restrictive on the right to least restrictive on the left for how to control the action in the left-hand column. Bold entries indicate the policy choices that we implemented for our experimental system.

mation.

Indirect jumps are used in the implementation of `switch` statements and dynamically shared libraries. The first use can easily be allowed when targets are validated to be coming from read-only memory and are hence trusted. The second use, shared library calls, should be allowed, but such inter-segment indirect jumps can be restricted to library entry points. These restrictions will not allow an indirect jump instruction that is used as a function return in place of an actual return instruction. However, we have yet to see such code. It will certainly not be generated by compilers since it breaks important hardware optimizations in modern IA-32 processors [21].

Sandboxing can provide detection of attacks that get past other barriers. For example, an attack that overwrites the argument passed to the `system` routine may not be stopped by any aforementioned policy. Program shepherding's guaranteed sandboxing can be used for intrusion detection for this and other attacks. The security policy must decide what to check for (for example, suspicious calls to system calls like `execve`) and what to do when an intrusion is actually detected. These issues are beyond the scope of this paper, but have been discussed elsewhere [15, 17].

Sandboxing with checks around every load and store could be used to ensure that only certain memory regions are accessed during execution of untrusted code segments. This would provide significant security but at great expense in performance.

We now turn our attention to a specific security policy made up of the bold entries in Table 1. We implemented this policy in our prototype system. For this security policy, Figure 1 summarizes the contribution of each program shepherding technique toward stopping the types of attacks described in Section 2. The following sections describe in detail which policy components are sufficient to stop each attack type.

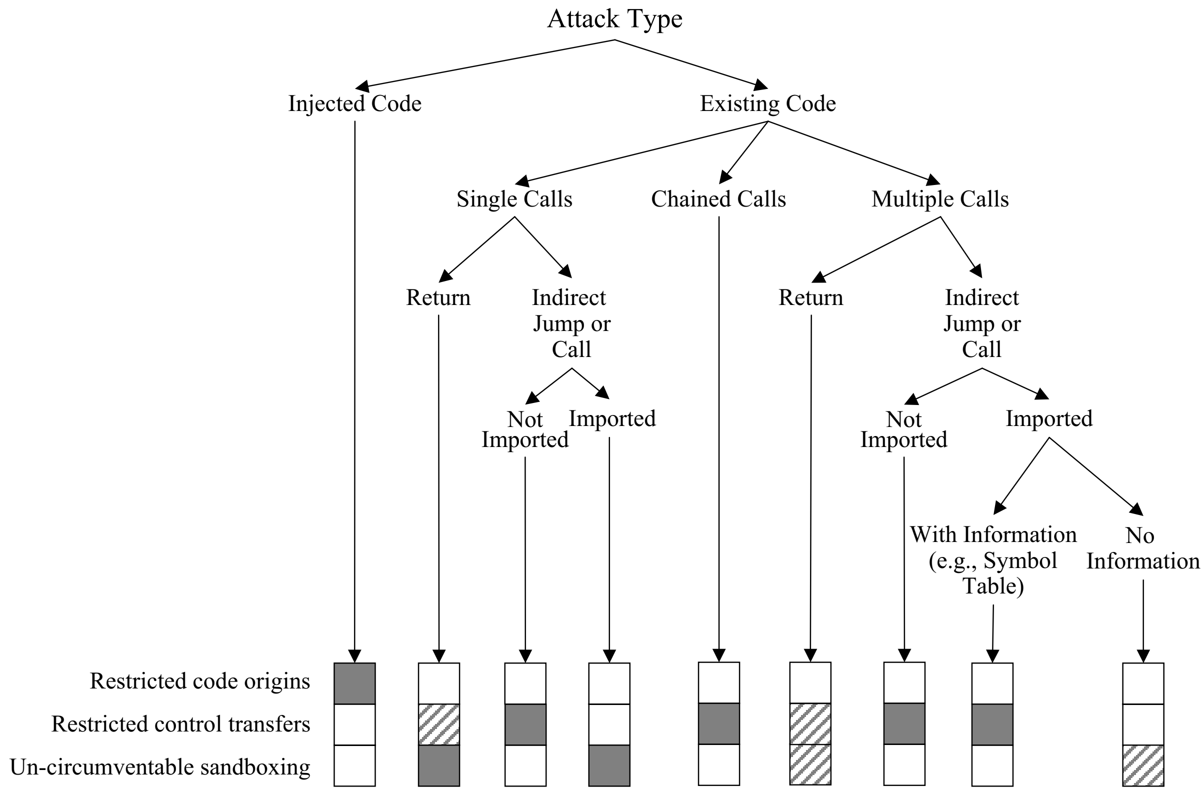

Figure 1: Capabilities of program shepherding's three components toward stopping different attack types, for the security policy indicated in bold in Table 1. The three boxes represent the three components. A filled-in box indicates that that component can completely stop the attack type above. Stripes indicate that the attack can be stopped only in some cases. The vertical order of the techniques indicates the preferred order for stopping attacks. If a higher box completely stops an attack, we do not invoke techniques below it (e.g., sandboxing is capable of stopping some attacks of every type, but we only use it when the other techniques do not provide full protection).

4.1 Injected Code Attacks

The code origin policy disallows execution from address ranges other than the text pages of the binary and mapped shared libraries. This stops all exploits that introduce external code, which covers a majority of currently deployed security attacks. However, code origin checks are insufficient to thwart attacks that change a target address pointer to point to existing code in the program address space.

4.2 Existing Code Attacks

Most vulnerable programs are unlikely to have code that could be maliciously used by an attacker. However, all of them have the standard C library mapped into their address space. The restrictions on inter-segment control transfers limit the available code that can be attacked to that explicitly declared for use by the application. Still, many of the large programs import the library routines a simple attack needs. For this reason, restricting inter-segment transitions to imported entry points would stop only a few attacks.

Return address attacks, however, are severely limited: they may only target code following previously executed call instructions. A further restriction can easily be provided by using restricted control transfers to emulate a technique proposed in StackGhost [14]. A random number can be `xor`-ed with the return address stored on the stack after a call and before a return. Any modification of the return address will result with very high probability in a request for an invalid target. In a threat model in which attackers can only write to memory, this technique renders execution of the attacker's intended code very unlikely. This protection comes at the low cost of two extra instructions per function call, but its additional value is hard to determine due to the already limited applicability of this kind of exploit. Furthermore, an attacker able to exploit a vulnerability that provides random read rights will not be stopped by this policy. Thus, we currently do not impose it.

4.2.1 Single Calls

By *single call* attack we mean an attack that overwrites only a single program address (perhaps overwriting non-address data as well), thus resulting in a single malicious control transfer. We consider the readily available `execve` system call to be the most vulnerable point in a single-call attack. However, it is possible to construct an intrusion detection predicate [17] to distinguish attacks from valid `execve` calls, and either terminate the application or drop privileges to limit the exposure. Since only a single call can be executed, system calls that need to be used in combination for an intrusion do not need to be sandboxed. Sandboxing `execve` also prevents intrusion by an argument overwrite attack.

Nevertheless, sandboxing alone does not provide protection against sequences of operations that an application is allowed to do and can be controlled by an attacker. For example, an exploit that emulates the normal behavior of `sshd`, i.e., listens on a network socket, accepts a connection, reads the password file for authentication, but at the end writes the password file contents to the network, cannot be stopped by simple sandboxing. Therefore, restrictions on control transfers are crucial to prevent construction of such higher-level code from primitives, and hence to limiting possible attacks only to data attacks targeting unlikely sequences of existing code.

4.2.2 Chained Calls

An attacker may be able to execute a malicious code sequence by carefully constructing a chain of activation records, so that on return from each function execution continues in the next one [18]. Requiring that return instructions target only call sites is sufficient to thwart the chained call attack, even when the needed functions are explicitly imported and allowed by inter-segment restrictions. The chaining technique is countered because of its reliance on return instructions: once to gain control at the end of each existing function, and once in the code to shift to the activation record for the next function call.

4.2.3 Multiple Calls

We were able to construct applications that were open to an exploit that forms higher-level malicious code by changing the targets of a sequence of function calls as well as their arguments. Multiple sequential intrusions may also allow execution of higher-level malicious code. Higher-level semantic information is needed to thwart these attacks' intrusion method by limiting the valid indirect call targets. The policy that is able to stop such attacks in general, and without any false alarms, requires knowing in advance a list of bindings built on a previous run or otherwise generated.

It is also possible to extract the valid user program entry points from the symbol tables of unstripped binaries. Allowing indirect calls to target only valid entry points within the executable and within the shared libraries limits the targets for higher-level code construction. If there are no simple wrappers in the executable that allow arbitrary arguments to be passed to the lower level library functions, the possibility of successful attack of this type will be minimal.

Nevertheless, interpreters that are too permissive are still going to be vulnerable to data attacks that may be used to form higher-level malicious code that will not be recognized as a threat by these techniques.

5 Efficient Implementation of Program Shepherding

In order for a security system to be viable, it must be efficient. And to be widely and easily adoptable, it must be transparent. Transparency includes whether a target application must be recompiled or instrumented and whether the security system requires special hardware or operating system support. We examined possible implementations of program shepherding in terms of these two requirements of efficiency and transparency.

One possible method of monitoring control flow is instrumentation of application and library code prior to execution to add security checks around every branch instruction. Beyond the difficulties of statically handling indirect branches and dynamically loaded libraries, the introduced checks impose significant performance penalties. Furthermore, an attacker aware of the instrumentation could design an attack to overwrite or bypass the checks. Instrumentation is neither very viable nor

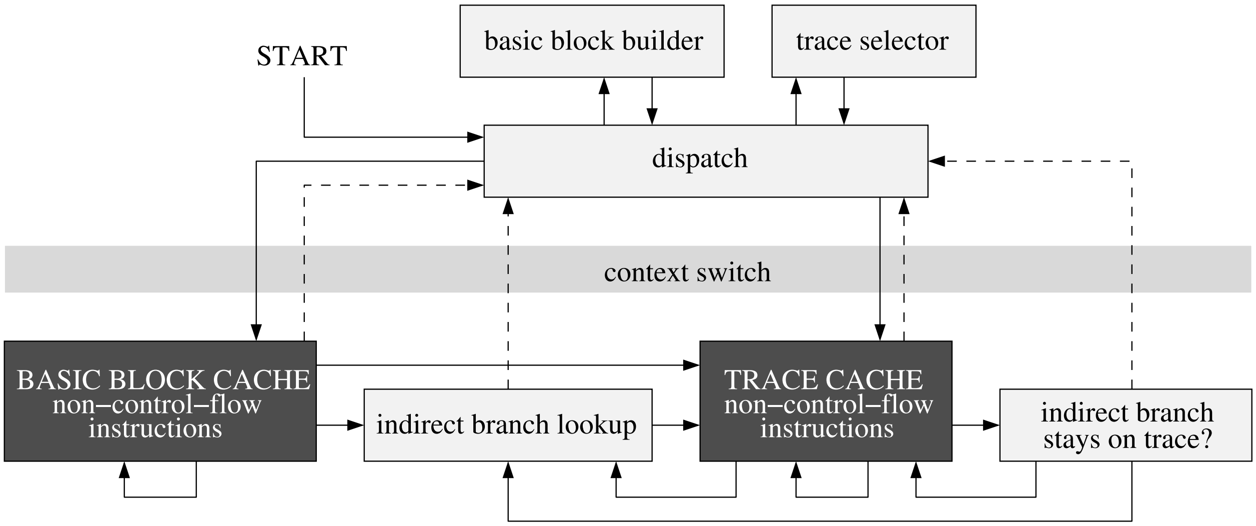

Figure 2: Flow chart of the RIO system infrastructure. Dark shading indicates application code. Note that the context switch is simply between the code cache and RIO; application code and RIO code all runs in the same process and address space. Dotted lines indicate the performance-critical cases where control must leave the code cache and return to RIO.

---

applicable.

Another possibility is to use an interpreter. Interpretation is a natural way to monitor program execution because every application operation is carried out by a central system in which security checks can be placed. However, interpretation via emulation is slow, especially on an architecture like IA-32 with a complex instruction set, as shown in Table 2.

5.1 Dynamic Optimization Framework

Recent advances in dynamic optimization have focused on low-overhead methods for examining execution traces for the purpose of optimization. This infrastructure provides the exact functionality needed for efficient program shepherding. Dynamic optimizers begin with an interpretation engine. To reduce the emulation overhead, native translations of frequently executed code are cached so they can be directly executed in the future. For a security system, caching means that many security checks need be performed only once, when the code is copied to the cache. If the code cache is protected from malicious modification, future executions of the trusted cached code proceed with no security or emulation overhead.

We decided to build our program shepherding system as an extension to a dynamic optimizer called RIO. RIO is built on top of the IA-32 version [3] of Dynamo [2]. RIO's optimizations are still under development. However, this is not a hindrance for our security purposes, as its performance is already reasonable (see Section 7.2). RIO is implemented for both IA-32 Windows and Linux, and is capable of running large desktop applications.

A flow chart showing the operation of RIO is presented in Figure 2. The figure concentrates on the flow of control in and out of the code cache, which is the bottom portion of the figure. The copied application code looks just like the original code with the exception of its control transfer instructions, which are shown with arrows in the figure.

Below we give an overview of RIO's operation, focusing on the aspects that are relevant to our implementation of program shepherding. The techniques of program shepherding fit naturally within the RIO infrastructure. Most monitoring operations only need to be performed once, allowing us to achieve good performance in the steady-state of the program. In our implementation, a performance-critical inner loop will execute without a single additional instruction beyond the original application code.

5.2 RIO: Runtime Introspection and Optimization

RIO copies *basic blocks* (sequences of instructions ending with a single control transfer instruction) into a code cache and executes them natively. At the end of each block the application's machine state must be saved and control returned to RIO (a *context switch*) to copy the next basic block. If a target basic block is already present in the code cache, and is targeted via a direct branch, RIO *links* the two blocks together with a direct jump. This avoids the cost of a subsequent context switch.

Indirect branches cannot be linked in the same way because their targets may vary. To maintain transparency, original program addresses must be used wherever the application stores indirect branch targets (for example, return addresses for function calls). These addresses must be translated into their corresponding code cache addresses in order to jump to the target code. This translation is performed as a fast hashtable lookup.

To improve the efficiency of indirect branches, and to achieve better code layout, basic blocks that are frequently executed in sequence are stitched together into a unit called a *trace*. When connecting beyond a basic block that ends in an indirect branch, a check is inserted to ensure that the actual target of the branch will keep execution on the trace. This check is much faster than the hashtable lookup, but if the check fails the full lookup must be performed. The superior code layout of traces goes a long way toward amortizing the overhead of creating them and often speeds up the program [2, 24].

Table 2 shows the typical performance improvement of each enhancement to the basic interpreter design. Caching is a dramatic performance improvement, and adding direct links is nearly as dramatic. The final steps of adding a fast in-cache lookup for indirect branches and building traces improve the performance significantly as well.

The Windows operating system directly invokes application code or changes the program counter for callbacks, exceptions, asynchronous procedure calls, `setjmp`, and the `SetThreadContext` API routine. These types of control flow are intercepted in order to ensure that all application code is executed under RIO [3]. Signals on Linux must be similarly intercepted.

System Type	Normalized Execution Time	
	crafty	vpr
Emulation	~ 300.0	~ 300.0
+ Basic block cache	26.1	26.0
+ Link direct branches	5.1	3.0
+ Link indirect branches	2.0	1.2
+ Traces	1.7	1.1

Table 2: Performance achieved when various features are added to an interpreter, measured on two of the SPEC2000 benchmarks [25], crafty and vpr. Pure emulation results in a slowdown factor of several hundred. Successively adding caching, linking, and traces brings the performance down dramatically.

5.3 Restricted Code Origins

Restricting execution to trusted code is accomplished by adding checks at the point where the system copies a basic block into the code cache. These checks need be executed only once for each basic block.

Code origin checking requires that RIO know whether code has been modified from its original image on disk, or whether it is dynamically generated. This is done by write-protecting all pages that are declared as containing code on program start-up. In normal ELF [12] binaries, code pages are separate from data pages and are write-protected by default. Dynamically generated code is easily detected when the application tries to execute code from a writable page, while self-modifying code is detected by monitoring calls that unprotect code pages.

If code and data are allowed to share a page, we make a copy of the page, which we write-protect, and then unprotect the original page. The copy is then used as the source for basic blocks, while the original page's data can be freely modified. A more complex scheme must be used if self-modifying code is allowed. Here RIO must keep track of the origins of every block in the code cache, invalidating a block when its source page is modified. The original page must be kept write-protected to detect every modification to it. The performance overhead of this depends on how often writes are made to code pages, but we expect self-modifying code to be rare. Extensive evaluation of applications under both Linux and Windows has yet to reveal a use of self-modifying code.

5.4 Restricted Control Transfers

The dynamic optimization infrastructure makes monitoring control flow transfers very simple. For direct branches, the desired security checks are performed at the point of basic block linking. If a transition between two blocks is disallowed by the security policy, they are not linked together. Instead, the direct branch is linked to a routine that announces or handles the security violation. These checks need only be performed once for each potential link. A link that is allowed becomes a direct jump with no overhead.

Indirect control transfer policies add no performance overhead in the steady state, since no checks are required when execution continues on the same trace. Otherwise, the hashtable lookup routine translates the target program address into a basic block entry address. A separate hashtable is used for different types of indirect branch (return instruction, indirect calls, and indirect branches) to enable type specific restrictions without sacrificing any performance. Security checks for indirect transfers that only examine their targets have little performance overhead, since we place in the hashtable only targets that are allowed by the security policy. Targets of indirect branches are matched against entry points of PLT-defined [12] and dynamically resolved symbols to enforce restrictions on inter-segment transitions, and targets of returns are checked to ensure they target only instructions after call sites. Security checks on both the source and the target of a transfer will have a slightly slower hashtable lookup routine. We have not yet implemented any policies that examine the source and the target, or apply transformations to the target, and so we do not have experimental results to show the actual performance impact of such schemes.

Finally, we must handle non-explicit control flow such as signals and Windows-specific events such as callbacks and exceptions [3]. We place security checks at our interception points, similarly to indirect branches. These abnormal control transfers are rare and so extra checks upon their interception do not affect overall performance.

5.5 Un-Circumventable Sandboxing

When required by the security policy, RIO inserts sandboxing into a basic block when it is copied to the code cache. In normal sandboxing, an attacker can jump to the middle of a block and bypass the inserted checks. RIO only allows control flow transfers to the top of basic blocks or traces in the code cache, preventing this.

An indirect branch that targets the middle of an existing block will miss in the indirect branch hashtable lookup, go back to RIO, and end up copying a new basic block into the code cache that will duplicate the bottom half of the existing block. The necessary checks will be added to the new block, and the block will only be entered from the top, ensuring that we follow the security policy.

When sandboxing system calls, if the system call number is determined statically, we avoid the sandboxing checks for system calls we are not interested in. This is important for providing performance on applications that perform many system calls.

Restricted code cache entry points are crucial not just for building custom security policies with un-circumventable sandboxing, but also for enforcing the other shepherding features by protecting RIO itself. This is discussed in the next section.

6 Protecting RIO

Program shepherding could be defeated by attacking RIO's own data structures, including the code cache, which are in the same address space as the application. This section discusses how to prevent attacks on RIO. Since the core of RIO is a relatively small piece of code, and RIO does not rely on any other component of the system, we believe we can secure it and leave no loopholes for exploitation.

6.1 Memory Protection

We divide execution into two modes: RIO mode and application mode. RIO mode corresponds to the top half of Figure 2. Application mode corresponds to the bottom half of Figure 2, including the code cache and the RIO routines that are executed without performing a context switch back to RIO. For the two modes, we give each type of memory page the privileges shown in Table 3. RIO data includes the indirect branch hashtable and other data structures.

All application and RIO code pages are write-protected in both modes. Application data is of course writable in application mode, and there is no reason to protect it

Page Type	RIO mode	Application mode
Application code	R	R
Application data	RW	RW
RIO code cache	RW	R (E)
RIO code	R (E)	R
RIO data	RW	R

Table 3: Privileges of each type of memory page belonging to the application process. R stands for Read, W for Write, and E for Execute. We separate execute privileges here to make it clear what code is allowed by RIO to execute.

from RIO, so it remains writable in RIO mode. RIO's data and the code cache can be written to by RIO itself, but they must be protected during application mode to prevent inadvertent or malicious modification by the application.

If a basic block copied to the code cache contains a system call that may change page privileges, the call is sandboxed to prevent changes that violate Table 3. Program shepherding's un-circumventable sandboxing guarantees that these system call checks are executed. Because the RIO data pages and the code cache pages are write-protected when in application mode, and we do not allow application code to change these protections, we guarantee that RIO's state cannot be corrupted.

We should also protect RIO's Global Offset Table (GOT) [12] by binding all symbols on program startup and then write-protecting the GOT, although our prototype implementation does not yet do this.

6.2 Multiple Application Threads

RIO's data structures and code cache are thread-private. Each thread has its own unique code cache and data structures. System calls that modify page privileges are checked against the data pages of all threads. When a thread enters RIO mode, only that thread's RIO data pages and code cache pages are unprotected.

A potential attack could occur while one thread is in RIO mode and another thread in application mode modifies the first thread's RIO data pages. We could solve this problem by forcing all threads to exit application mode when any one thread enters RIO mode. We have not yet implemented this solution, but its performance cost would be minimal on a single processor or on a multiprocessor when every thread is spending most of its time executing in the code cache. However, the performance cost would be unreasonable on a multiprocessor when threads are continuously context switching. We are investigating alternative solutions.

On Windows, we also need to prevent the API routine `SetThreadContext` from setting register values in other threads. RIO's hashtable lookup routine uses a register as temporary storage for the indirect branch target. If that register were overwritten, RIO could lose control of the application. Our interception of this API routine has not interfered with the execution of any of the large applications we have been running [3]. In fact, we have yet to observe any calls to it.

7 Experimental Results

Our program shepherding implementation is able to detect and prevent a wide range of known security attacks. This section presents our test suite of vulnerable programs, shows the effectiveness of our system on this test suite, and then evaluates the performance of our system on the SPEC2000 benchmarks [25].

7.1 Effectiveness

We constructed several programs exhibiting a full spectrum of buffer overflow and format string vulnerabilities. Our experiments also included the SPEC2000 benchmark applications [25] and the following applications with recently reported security vulnerabilities:

stunnel-3.21 CAN-2002-0002 [8] A format string vulnerability in `stunnel` (SSL tunnel) allows remote malicious servers to execute arbitrary code because several `fdprintf` (a custom file descriptor wrapper of `fprintf`) calls have no format argument.

groff-1.16 CAN-2002-0003 [8] The preprocessor of the `groff` formatting system has an exploitable buffer overflow which allows remote attackers to gain privileges via `lpd` in the `LPRng` printing system. The `pic` picture compiler from the `groff` package also has a format string vulnerability [22].

ssh-1.2.31 CVE-2001-0144 [8] An integer-overflow bug in the CRC32 compensation attack detection code causes the SSH daemon (typically run as root) to create a hashtable with size zero in response to long input. Later attempts to write values into the hashtable provide attackers with random write access to memory.

sudo-1.6.1 CVE-2001-0279 [8] `sudo` (superuser do) allows local users to gain root privileges. A vulnerability caused by an out-of-bound access due to incomplete end of loop condition is triggered by long command line arguments. An exploit based on `malloc` corruption has been published [16].

Attack code is usually used to immediately give the attacker a root shell or to prepare the system for easy takeover by modifying system files. Hence, the exploits in our tests tried to either start a shell with the privilege of the running process, typically root, or to add a root entry into the `/etc/passwd` file. We based our exploits on several "cookbook" and proof-of-concept works [4, 27, 16, 22] to inject new code [20], reuse existing code in a single call, or reuse code in a chain of multiple calls [18]. Existing code attacks used only standard C library functions.

When run natively, our test suite exploits were able to get control by modifying a wide variety of code pointers including return addresses; local and global function pointers; `setjmp` structures; and `atexit`, `.dtors`, and GOT [12] entries. We investigated attacks against RIO itself, e.g., overwriting RIO's GOT entry to allow malicious code to run in RIO mode, but could not come up with an attack that could bypass the protection mechanisms presented in Section 6.

All vulnerable programs were successfully exploited when run on a standard RedHat 7.2 Linux installation. Execution of the vulnerable binaries under RIO with all security checks disabled also allowed successful intrusions. Although RIO interfered with a few of the exploits due to changed addresses in the targets, it was trivial to modify the exploits to work under our system. Execution of the vulnerable binaries under RIO enforcing the policies shown in bold on Table 1, effectively blocked all attack types. All intrusion attempts that would have led to successfully exploitable conditions were detected. Nevertheless, the vulnerable applications were able to execute normally when presented with benign input. The SPEC2000 benchmarks also gave no false alarms on the reference data set.

7.2 Performance

Figure 3 and Figure 4 show the performance of our system on Linux and Windows, respectively. Each figure shows normalized execution time for the SPEC2000 benchmarks [25], compiled with full optimization and run with unlimited code cache space. (Note that we do not have a FORTRAN 90 compiler on Linux or any FORTRAN compiler on Windows.) The first bar gives the performance of RIO by itself. RIO breaks even on many benchmarks, even though it is not performing any optimizations beyond code layout in creating traces. The second bar shows the performance of program shepherding enforcing the policies shown in bold in Table 1. The results show that the overhead of program shepherding is negligible.

The final bar gives the overhead of protecting RIO itself. This overhead is again minimal, within the noise in our measurements for most benchmarks. On Linux, only `gcc` has significant slowdown due to page protection, because it consists of several short runs with little code re-use. On Windows, however, several benchmarks have serious slowdowns, especially `gcc`. Our only explanation at this point for the difference between the Linux and Windows protection slowdowns is that Windows is much less efficient at changing privileges on memory pages than Linux is. We are working on improving our page protection scheme by lazily unprotecting only those pages that are needed on each return to RIO mode.

The memory usage of our security system is shown in Table 4. All sizes shown are in KB. The left half of the table shows the total size of text sections of each benchmark and all shared libraries it uses compared to the amount of code actually executed. The third column gives the percentage of the total static code that is executed. By operating dynamically our system is able to focus on the small portion of code that is run, whereas a static approach would have to examine the text sections in their entirety.

The right half of Table 4 shows the memory overhead of RIO compared to the memory usage of each benchmark. For most benchmarks the memory used by RIO is a small fraction of the total memory used natively.

8 Related Work

Reflecting the significance and popularity of buffer overflow and format string attacks, there have been several other efforts to provide automatic protection and detection of these vulnerabilities. We summarize the more successful ones.

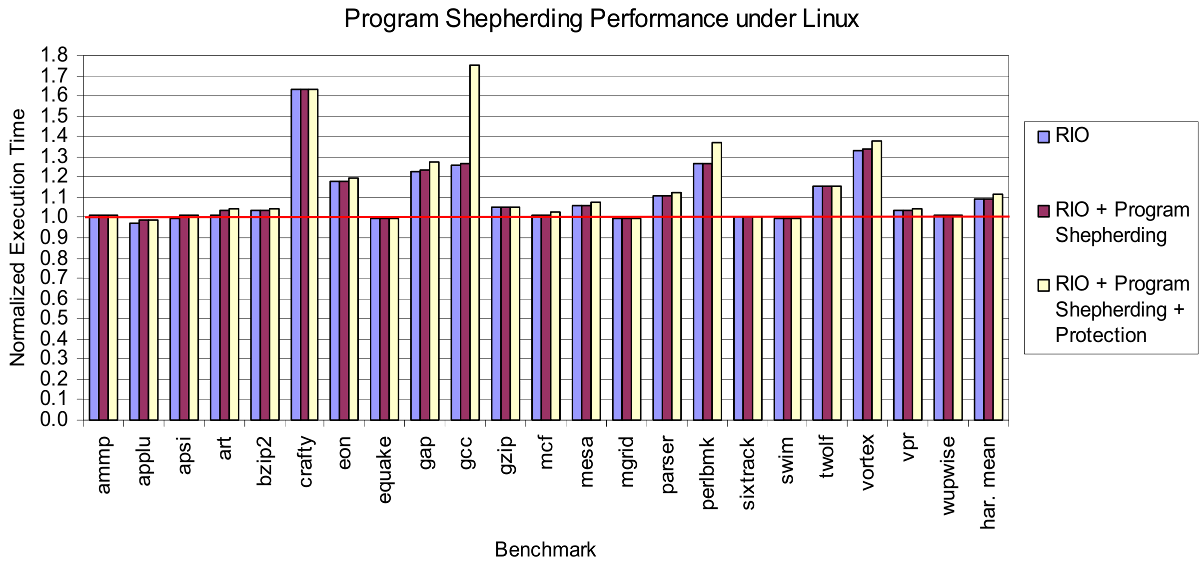

Figure 3: Normalized program execution time for our system (the ratio of our execution time to native execution time) on the SPEC2000 benchmarks [25] (excluding all FORTRAN 90 benchmarks) on Linux. They were compiled using `gcc -O3`. The final set of bars is the harmonic mean. The first bar is for RIO by itself; the middle bar shows the overhead of program shepherding (with the security policy of Table 1); and the final bar shows the overhead of the page protection calls to prevent attacks against the system itself.

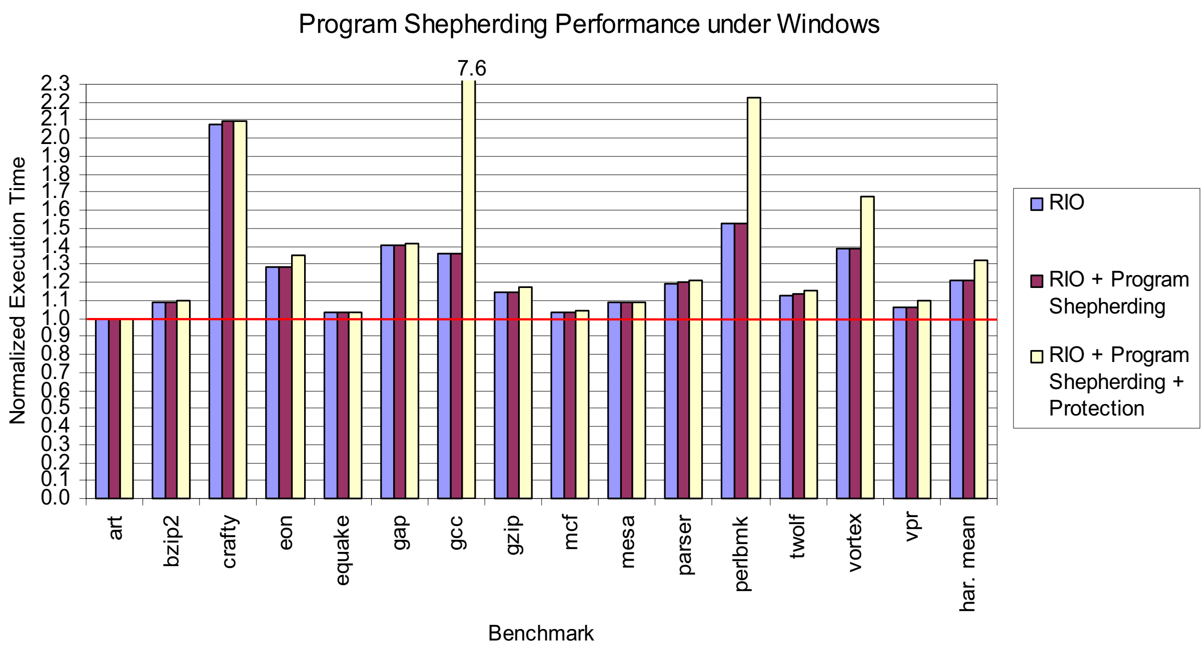

Figure 4: Normalized program execution time for our system (the ratio of our execution time to native execution time) on the SPEC2000 benchmarks [25] (excluding all FORTRAN benchmarks) on Windows 2000. They were compiled using `cl /Ox`. The final set of bars is the harmonic mean. The first bar is for RIO by itself; the middle bar shows the overhead of program shepherding (with the security policy of Table 1); and the final bar shows the overhead of the page protection calls to prevent attacks against the system itself.

benchmark	static code	executed code	% executed	native total	RIO extra	% RIO extra
ammp	1515	52	3.4%	14893	1696	11.4%
applu	1597	181	11.3%	195715	2720	1.4%
apsi	1639	179	10.9%	197016	2208	1.1%
art	1424	22	1.5%	4612	928	20.1%
bzip2	1317	30	2.3%	190767	928	0.5%
crafty	1467	169	11.5%	3418	3232	94.6%
eon	2114	269	12.7%	2721	2208	81.1%
equake	1428	39	2.7%	34255	928	2.7%
gap	1713	167	9.7%	198916	4256	2.1%
gcc	2518	729	29.0%	145547	14496	10.0%
gzip	1323	27	2.0%	186374	928	0.5%
mcf	1289	24	1.9%	98516	928	0.9%
mesa	1885	63	3.3%	22812	1696	7.4%
mgrid	1475	63	4.3%	58233	1184	2.0%
parser	1390	114	8.2%	32407	3232	10.0%
perlbmk	1878	286	15.2%	76272	6304	8.3%
sixtrack	2812	347	12.3%	60786	4256	7.0%
swim	1452	44	3.0%	196433	928	0.5%
twolf	1591	124	7.8%	4256	3232	75.9%
vortex	1890	395	20.9%	50390	6304	12.5%
vpr	1540	114	7.4%	40425	2208	5.5%
wupwise	1477	67	4.5%	181527	1696	0.9%
arithmetic mean	1670	159	8.5%	90741	3023	16.2%
harmonic mean	1604	66	4.5%	15410	1747	1.8%

Table 4: Memory usage of the SPEC2000 benchmarks [25], in KB, on Linux. For benchmarks with multiple data sets, the run with the maximum memory usage is shown. Static code is the total size of the text sections of the benchmark and all shared libraries it uses. Executed code is the total size of all instructions processed by RIO when running the benchmark. RIO total is the total memory used by RIO itself when running the benchmark. Native total is total memory used by the benchmark when run natively (outside of RIO).

StackGuard [7] is a compiler patch that modifies function prologues to place "canaries" adjacent to the return address pointer. A stack buffer overflow will modify the "canary" while overwriting the return pointer, and a check in the function epilogue can detect that condition. This technique is successful only against sequential overwrites and protects only the return address.

StackGhost [14] is an example of hardware-facilitated return address pointer protection. It is a kernel modification of OpenBSD that uses a Sparc architecture trap when a register window has to be written to or read from the stack, so it performs transparent `xor` operations on the return address before it is written to the stack on function entry and before it is used for control transfer on function exit. Return address corruption results in a transfer unintended by the attacker, and thus attacks can be foiled.

Techniques for stack smashing protection by keeping copies of the actual return addresses in an area inaccessible to the application are also proposed in StackGhost [14] and in the compiler patch StackShield [26]. Both proposals suffer from various complications in the presence of multi-threading or deviations from a strict calling convention by `setjmp()` or exceptions. Unless the memory areas are unreadable by the application, there is no hard guarantee that an attack targeted against a given protection scheme can be foiled. On the other hand, if the return stack copy is protected for the duration of a function execution, it has to be unprotected on each call, and that can be prohibitively expensive (`mprotect` on Linux on IA-32 is 60–70 times more expensive than an empty function call). Techniques for write-protection of stack pages [7] have also shown significant performance penalties.

FormatGuard [6] is a library patch for eliminating format string vulnerabilities. It provides wrappers for the `printf` functions that count the number of arguments and match them to the specifiers. It is applicable only to functions that use the standard library functions directly, and it requires recompilation.

Enforcing non-executable permissions on IA-32 via kernel patches has been done for stack pages [10] and for data pages in PaX [23]. Our system provides execution protection from user mode and achieves better steady state performance. Randomized placement of position independent code was also proposed in PaX as a technique for protection against attacks using existing code; however, it is open to attacks that are able to read process addresses and thus determine the program layout.

Our system infrastructure itself is a dynamic optimization system based on the IA-32 version [3] of Dynamo [2]. Other software dynamic optimizers are Wiggins/Redstone [9], which employs program counter sampling to form traces that are specialized for the particular Alpha machine they are running on, and Mojo [5], which targets Windows NT running on IA-32. None of these has been used for anything other than optimization.

9 Conclusions

This paper introduces program shepherding, which employs the techniques of restricted code origins, restricted control transfers, and un-circumventable sandboxing to provide strong security guarantees. We have implemented program shepherding in the RIO runtime system, which does not rely on hardware, operating system, or compiler support, and operates on unmodified binaries on both generic Linux and Windows IA-32 platforms. We have shown that our implementation successfully prevents a wide range of security attacks efficiently.

Program shepherding does *not* prevent exploits that overwrite sensitive data. However, if assertions about such data are verified in all functions that use it, these verifications cannot be bypassed if they are the only declared entry points.

We have discussed the potential design space of security policies that can be built using program shepherding. Our system currently implements one set of policy settings, but we are expanding the set of security policies that our system can provide without loss of performance. Future expansions include using semantic information provided by compilers to specify permissible operations on a fine-grained level, and performing explicit protection and monitoring of known program addresses to prevent corruption. For example, protecting the application's GOT [12] and allowing updates only by the dynamic resolver can easily be implemented in a secure and efficient fashion.

A potential application of program shepherding is to allow operating system services to be moved to more efficient user-level libraries. For example, in the exokernel [13] operating system, the usual operating system abstractions are provided by unprivileged libraries, giving efficient control of system resources to user code. Program shepherding can enforce unique entry points in these libraries, enabling the exokernel to provide better performance without sacrificing security.

We believe that program shepherding will be an integral part of future security systems. It is relatively simple to implement, has little or no performance penalty, and can coexist with existing operating systems, applications, and hardware. Many other security components can be built on top of the un-circumventable sandboxing provided by program shepherding. Program shepherding provides useful security guarantees that drastically reduce the potential damage from attacks.

References

[1] Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. Adaptive optimization in the Jalapeño JVM. In *2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'00)*, October 2000.

[2] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent runtime optimization system. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '00)*, June 2000.

[3] Derek Bruening, Evelyn Duesterwald, and Saman Amarasinghe. Design and implementation of a dynamic optimization framework for Windows. In *4th ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-4)*, December 2000.

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

[5] Wen-Ke Chen, Sorin Lerner, Ronnie Chaiken, and David M. Gillies. Mojo: A dynamic optimization system. In *3rd ACM Workshop on Feedback-Directed and Dynamic Optimization (FDDO-3)*, December 2000.

[6] Crispin Cowan, Matt Barringer, Steve Beattie, and Greg Kroah-Hartman. FormatGuard: Automatic protection from printf format string vulnerabilities, 2001. In 10th USENIX Security Symposium, Washington, D.C., August 2001.

[7] Crispin Cowan, Calton Pu, Dave Maier, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, Qian Zhang, and Heather Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Symposium*, pages 63–78, San Antonio, Texas, January 1998.

[8] Common vulnerabilities and exposures. MITRE Corporation.
`http://cve.mitre.org/`.

[9] D. Deaver, R. Gorton, and N. Rubin. Wiggins/Restone: An on-line program specializer. In *Proceedings of Hot Chips 11*, August 1999.

[10] Solar Designer. Non-executable user stack.
`http://www.openwall.com/linux/`.

[11] L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk-80 system. In *ACM Symposium on Principles of Programming Languages (POPL '84)*, January 1984.

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

[13] Dawson R. Engler, M. Frans Kaashoek, and James O'Toole. Exokernel: An operating system architecture for application-level resource management. In *Symposium on Operating Systems Principles*, pages 251–266, 1995.

[14] M. Frantzen and M. Shuey. Stackghost: Hardware facilitated stack protection. In *Proc. 10th USENIX Security Symposium*, Washington, D.C., August 2001.

[15] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the 6th Usenix Security Symposium*, San Jose, Ca., 1996.

[16] Michel Kaempf. Vudo - an object superstitiously believed to embody magical powers. *Phrack*, 8(57), August 2001.

[17] Calvin Ko, Timothy Fraser, Lee Badger, and Douglas Kilpatrick. Detecting and countering system intrusions using software wrappers. In *Proc. 9th USENIX Security Symposium*, Denver, Colorado, August 2000.

[18] Nergal. The advanced return-into-lib(c) exploits. *Phrack*, 4(58), December 2001.

[19] Tim Newsham. Format string attacks. Guardent, Inc., September 2000.
`http://www.guardent.com/docs/FormatString.PDF`.

[20] Aleph One. Smashing the stack for fun and profit. *Phrack*, 7(49), November 1996.

[21] Intel Pentium 4 and Intel Xeon processor optimization reference manual. Intel Corporation, 2001.

[22] Zenith Parsec. Remote linux groff exploitation via lpd vulnerability.
`http://www.securityfocus.com/bid/3103`.

[23] PaX Team. Non executable data pages.
`http://pageexec.virtualave.net/pageexec.txt`.

[24] Eric Rotenberg, Steve Bennett, and J. E. Smith. Trace cache: A low latency approach to high bandwidth instruction fetching. In *29th Annual International Symposium on Microarchitecture (MICRO '96)*, December 1996.

[25] SPEC CPU2000 benchmark suite. Standard Performance Evaluation Corporation.
`http://www.spec.org/osg/cpu2000/`.

[26] Vendicator. Stackshield: A "stack smashing" technique protection tool for linux.
`http://www.angelfire.com/sk/stackshield/`.

[27] Rafal Wojtczuk. Defeating solar designer non-executable stack patch.
`http://www.securityfocus.com/archive/1/8470`.