Secure Execution Environment via Program Shepherding

by

Vladimir L. Kiriansky

B.S., Massachusetts Institute of Technology (2002)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 2003

Author .....
Department of Electrical Engineering and Computer Science
February 4, 2003

Certified by .....
Saman P. Amarasinghe
Associate Professor
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Secure Execution Environment

via Program Shepherding

by

Vladimir L. Kiriansky

Submitted to the Department of Electrical Engineering and Computer Science
on February 4, 2003, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

We present *program shepherding*, a method for monitoring control flow transfers during program execution in order to enforce a security policy. Program shepherding provides three basic techniques as building blocks for security policies. First, program 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 type, source, and target. Finally, shepherding guarantees that sandboxing checks around any program operation will never be bypassed.

Security attacks use inevitable bugs in trusted binaries to coerce a program into performing actions that it was never intended to perform. We use static and dynamic analyses to automatically build a custom security policy for a target program, which specifies the program's *execution model*. An accurate execution model restricts control flow transfers only to the intended ones and can thwart attacker attempts to alter program execution. For example, shepherding will allow execution of shared library code only through declared entry points. Finer specifications can be extracted from high-level information present in programs' source code — for example, which values a function pointer may take. Program shepherding will allow indirect calls only to their known targets, and function returns only to known callers. These analyses build a strict enough policy to prevent all deviations from the program's control flow graph and nearly all violations of the calling convention. This technique renders most security vulnerabilities unexploitable and thwarts current and future security attacks.

We present an efficient implementation of program shepherding's capabilities in the DynamoRIO [6, 7] runtime code modification system. The resulting system imposes minimal performance overhead, operates on unmodified binaries, and requires no special hardware or operating system support.

Thesis Supervisor: Saman P. Amarasinghe
Title: Associate Professor

Acknowledgments

The work described in this thesis would have been impossible without the help of many others. The DynamoRIO project is indebted to the Dynamo group at HP labs for their fundamental contribution. Thanks are due to Derek Bruening, who was the main maintainer of the base DynamoRIO system at MIT and helped implement many of its extensions for the system presented here. Derek also coined the term *program shepherding* for the techniques presented in this work. I am also grateful to my advisor Saman Amarasinghe for his undepletable enthusiasm and technical insights during the course of this and several other projects.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Remote exploitation of program vulnerabilities poses a very serious threat to our modern information infrastructure. It allows rapid automatic self-propagating programs (worms) to gain control over large number of hosts [57]. Grave secondary effects are possible due to the extent to which critical infrastructure relies on the Internet. Worms usually exploit well-known security vulnerabilites since flawed software is quite often left unpatched. For example, a recent worm worthy of a CERT Advisory [9] (CA-2003-04), propagates by exploiting a vulnerability in Microsoft's SQL Server announced six months earlier (CA-2002-22). The worm infected most vulnerable hosts within 10 minutes [42]. Although it has no malicious payload it wrought considerable damage by secondary effects of network outages, e.g. in ATM networks. Notably, even the vendor of the flawed software had unpatched systems, and this shows the inadequacy of the patch deployment process even for already known vulnerabilities. At the other end of the spectrum, vulnerabilities that are not publicly disclosed may be used in an attack specifically targeting a security sensitive entity. This thesis introduces techniques that can effectively render most security vulnerabilities unexploitable and thence mitigate disruptions due to distributed denial of service attacks. We have implemented them in a software system which provides a secure execution environment while imposing minimal performance overhead.

The goal of most security attacks is to gain unauthorized access to a computer system by taking control of a vulnerable program. This is generally done by exploiting bugs that allow overwriting stored program addresses with pointers to malicious code. An attacker

who gains control over a program can simply inject code to perform any operation that the overall application has permission to do. Hijacking trusted applications which are typically run with global permissions, such as login servers, mail transfer agents, and web servers, gives full access to machine resources. Vulnerabilities allowing remote execution of arbitrary code and their exploitation account for 34 of the 37 CERT Advisories [9] for 2002. Today's most prevalent attacks use buffer overflow and format string vulnerabilities to overwrite program addresses. Our threat model of security attacks assumes that an attacker is able to exploit inadvertent program vulnerabilities to gain random write access to arbitrary memory locations in the program address space.

Nearly all attacks have one thing in common: they coerce a target program into performing actions that it was never intended to perform. In short, they violate the *execution model* followed by legitimate program runs. The execution model encompasses the Application Binary Interface (ABI) and higher-level specifications from the program's source programming language. The model also incorporates components specific to the program, for example, which values a particular function pointer may take.

A program's execution model is invariably narrower than that imposed by the underlying hardware. As such, there is typically no efficient way to require that the rules of this execution model be adhered to. The result is that the execution model becomes, in practice, a convention rather than a strict set of rules. If this model were enforced, and only program actions that the programmer intended were allowed, a majority of current security holes would be closed. Whenever the execution model allows only a single choice, or choices equivalent in their system effects, an attacker cannot gain new abilities. For example, a common attack type overwrites a return address to point to a malicious destination. If this destination is not a valid return target in the program's execution model, it would be disallowed by an enforcement of the model.

1.1 Overview

In this thesis, we employ *program shepherding* [38] to enforce a security policy. Instead of attempting to protect data, program shepherding monitors control flow in order to enforce

a program's execution model. Program shepherding provides three basic techniques: restricted code origins to prevent execution of data or modified code, restricted control transfers to preclude deviations from the execution model, and un-circumventable sandboxing checks around any type of program operation.

The execution model implicitly provided by the programs according to the Application Binary Interface (ABI) can be extracted dynamically. For example, shepherding will allow execution of shared library code only through exported entry points, and can ensure that a return instruction only targets instructions after a call. Finer specifications of the execution model can be extracted statically from high-level information present in programs' source code — for example, which values a function pointer may take. Program shepherding will allow indirect calls only to their known targets, and function returns only to instructions after their known callers.

We use static and dynamic analyses to automatically build a custom security policy for a target program which specifies the program's execution model. This process requires no user interaction, but is able to build a strict enough policy to prevent all deviations from the program's control flow graph and nearly all violations of the calling convention. Therefore it greatly reduces the possibility of an unintended program action. Our static analyses require source code access but not recompilation.

We have efficiently implemented program shepherding capabilities in the DynamoRIO [6, 7] runtime code modification system. DynamoRIO executes only sanitized copies of the original program code and stores these in a trusted code cache. This code cache is the first key feature to providing efficient secure execution, because it allows many security checks to be performed only once. A second key feature of DynamoRIO is the creation of *traces*, hot streams of code that cross control flow transitions. Security checks on indirect control flow transitions can be elided when execution follows the trace. These features result in a secure system that imposes minimal performance overhead, operates on unmodified native binaries, and requires no special hardware or operating system support. Our program shepherding implementation on top of DynamoRIO supports both Windows and Linux on IA–32 processors. However, the detailed security discussion in this thesis is mainly focused on Linux applications.

1.2 Organization

In Chapter 2 we classify the types of security exploits that we are aiming to prevent. Program shepherding's three basic techniques are described in Chapter 3. The execution model and how to enforce it is discussed in Chapter 4. We analyze features of the execution model that can be enforced with reasonable cost using program shepherding in Chapter 5, where we also identify aspects of enforcement that can be performed dynamically when source code is unavailable. In Chapter 6 we discuss how we implement program shepherding efficiently, and how to prevent attacks directed at our system itself. We show how to incorporate static analyses to automatically extract features of the program's execution model such as its call graph in Chapter 7. We present experimental results and the performance of our system in Chapter 8. We demonstrate that enforcing a program's execution model via program shepherding can thwart many security attacks. We give experimental evidence that the execution model can be enforced efficiently and effectively using a runtime code modification system. We discuss related work in Chapter 9, propose future extensions in Chapter 10, and conclude this thesis in Chapter 11.

Chapter 2

Security Exploits

This section provides some background on the types of security exploits we are targeting to prevent. We dissect security exploits along three basic 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 C language memory model allows unsafe writes to objects which are often unintended by programmers. Most programmers code with the assumption that program objects are accessible only via valid references, which is the logical memory model [60]. However, numerous unsafe language features result in bugs that break this model, e.g. pointer arithmetic with no bounds checking, variadic arguments, dangling pointers, weak types, unions. All provide venues for overwriting other than the intended objects.

Currently, the most-exploited classes of program bugs involve buffer overflow [46, 13, 8], integer overflow [5], and format string [45, 12] vulnerabilities. These vulnerability classes are reported in respectively 22, 3, and 2 of the CERT advisories [9] for 2002 together with 4 advisories for other program bugs.

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 [13]. Integer overflow and integer signedness problems allow attackers to bypass imprecise bound checking code, which can then cause typical buffer overflow problems. Buffer overflows affecting a regular data pointer, for example an output argument of a function, 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 [36]. On a subsequent call to `free`, the list update operation

$$\texttt{this}\rightarrow\textit{prev}\rightarrow\texttt{next = this}\rightarrow\textit{next}\texttt{;}$$

will modify an arbitrary location (controlled by *prev*) with an arbitrary value (*next*).

Format string vulnerabilities also allow attackers to modify arbitrary memory locations with arbitrary values and often out-rank buffer overflows in recent security bulletins [12, 45]. 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)`. [1] As another example, a well-intended security logging facility `syslog(LOG_INFO, str)` ironically introduces a serious vulnerability. 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.

It is very difficult to prevent all exploits that allow address overwrites, as they are as varied as program bugs themselves. All of the above bugs have been found exploitable in circumstances reducing them to an arbitrary write problem. More than one write to disjoint memory ranges may also be performed, either in result of a single intrusion or of repeated

[1] Of course, the proper C idiom in this case is `fputs(str, stdout)`.

careful intrusions. We assume from now on a program vulnerability exists, which allows random write accesses with attacker chosen values. In most security attacks modifying data is simply the means to executing a sequence of instructions that will ultimately compromise the whole system. Attackers induce this by overwriting a stored program address that will be used in an indirect control transfer.

2.2 Stored Program Addresses

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) [20] of shared object entries. In the assumed arbitrary write threat model, any function pointer or other stored address that is later supplied to an indirect control transfer instruction (such as return, indirect call, or indirect jump) provides a vector to a potential attack.

Program addresses are credibly manipulated by a number of entities. For example, 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. The resulting protection is only as powerful as the weakest link.

2.3 Malicious Code

Using the privileges of the application, an attacker can cause damage by executing newly injected malicious code or by maliciously reusing already present code. Currently, the first approach is prevalently taken and attack code is implemented as new native code that is injected in the program address space as data [46]. Modifying any stored program address to point to the introduced code triggers intrusion when that address is used for control transfer. 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.

It is also possible to reuse existing code by changing a stored program address and constructing an activation record with suitable arguments. For example, a simple but powerful attack changes a function pointer to the C library function `system`, and arranges the first argument to be an arbitrary shell command to be run. Similar attacks may be launched to the arguments of any system call normally accessible to the application. Note that reuse of existing code can also include jumping into the middle of a sandboxed application 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, which will not be visible even in manual inspection of normal dissasembly; 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 [44]. 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.

Modifying the targets of a suitable sequence of indirect calls as well as their arguments also may allow an attacker to produce higher-level malicious code. Undetected sequential

intrusions may also allow orchestration of existing pieces of code to produce an unintended malicious outcome. While attacks of this kind are currently not widely publicized, a determined attacker can easily steer an application in this fashion.

Chapter 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 basic techniques: restricted code origins, restricted control transfers, and un-circumventable sandboxing. The following sections describe these techniques, while Chapter 5 discusses how to combine them to build efficient security policies.

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. This restriction can be used to ensure that malicious code masquerading as data is never executed, thwarting the largest class of current security attacks. 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 6.3 how to distinguish original code from modified and possibly malicious code.

Additional hardware flag for execute permissions on memory pages can provide similar features to our restricted code origins. However, it cannot by itself duplicate program

shepherding's capabilities because it cannot stop inadvertent or malicious changes to protection flags. Program shepherding uses its un-circumventable sandboxing to prevent this from happening by monitoring all such privileged operations, e.g. system calls that change page protection. Furthermore, program shepherding provides more than one bit of privilege information, hence distinguishable code origins allow different execute privileges to be specified by security policies for each type.

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.). Indirect calls, indirect jumps, and returns obtain their targets from data, which can be modified by an attacker. Enforcing the execution model involves allowing each branch to jump only to a specified set of targets. For example, the calling convention could be strengthened by requiring that a return instruction only target the instruction after a call. Stricter policies may allow indirect calls only to their apriori known targets, and function returns only to instructions after their known callers. These restrictions can prevent attackers from forming malicious code sequences from existing code. Different policies for determining the valid transition sets and the trade-offs between security, performance and applicability will be discussed in Section 5.2.

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 all other types of instructions of particular interest, for example privilege changing instructions. Customized policies can be added to validate system call arguments to limit data attacks. 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 6.6).

Chapter 4

Execution Model Enforcement

The execution model of a program includes several components. At the lowest level, the Application Binary Interface (ABI) specifies the register usage and calling conventions of the underlying architecture, along with the operating system interface mechanism. Higher-level conventions come from the source language of the program in the form of runtime data structure usage, expected interaction with the operating system, and usage of system libraries. Finally, the program itself is intended by the programmer to perform a limited set of actions.

Even the lowest level, the ABI, is usually not efficiently enforceable when there is no hardware support. There is no support for ensuring that calls and returns match, and it is prohibitively expensive to directly implement this in software. For this reason, the execution model is a convention rather than a strictly enforced set of rules. However, most security exploits come from violations of the execution model. Overwriting a stored program address lets an attacker control further program execution. However, transfer of control to that code should not allowed be under the program's execution model. Enforcing the model would thwart these security attacks.

Restrictions on data usage are very difficult to enforce efficiently because distinguishing memory references requires expensive runtime checks on every memory access. Most security attacks target not just any data, but data storing program addresses. Even limiting data protection to these locations, protecting the data incurs significant performance costs[13]. We restrict our enforcement of the execution model to the set of allowed control

transfers. Although caused by abnormal data operations, unintended program actions will surface as unintended control flow. We will show how to obtain accurate control transfer specifications and that an execution model can be efficiently enforced.

4.1 Degree of Freedom

The degree of freedom of an attacker is given by the size of the set of allowed values for an attacked stored program address. Ideally, these sets should be singletons, because in a real program execution at any point there is only one valid value (in the absence of race conditions). Therefore, we aim to minimize the size of the sets and convert them to singletons when possible.

4.2 Context-Insensitive Policies

Our first aim is to determine the points-to sets for function pointers by using an accurate static analysis. We use a *flow-insensitive* and *context-insensitive* analysis, discussed in detail in Chapter 7, to gather the sets of valid targets for indirect calls. Using that information we construct the complete call graph for the program. Targets of return instructions are then computed from the graph, since the instructions after caller sites of a function constitute the only valid targets for its exit point.

If we assume that the only security relevant events are due to system calls, we can perform reachability analysis to identify the system calls accessible from each of the functions in a set, up to a node dominated by all targets in the set. If different execution paths can reach different system calls, then an attacker has a choice of action for constructing a malicious sequence. (We may make a further unification by assuming the order of system calls is immaterial since attackers may be interested in any of them.) Whenever the system call sets are all equivalent (in the best case all being empty), we can accept any valid target in a set, because changing a stored pointer from one value to another provides no new abilities to an attacker.

In the following example with vulnerable code, the only accepted values for the function

pointer `f` are {`sin,cos`}, but not `system` for that matter.

```
double eval(double f(double), double x, char* msg) {
    printf(msg, x);  // format string vulnerability
    return f(x);
}
void sincos(char* usermsg) {
    eval(sin, 0, usermsg);
    eval(cos, 0, "cos(%f)");
}
```

If an attacker can never cause deviations from the trace of system calls, then the only vector of attack left is changing their arguments. Simple stateless system call sandboxing may be successfully employed to limit the arguments to those statically predicted, however this type of automatic policy generation is not covered in this thesis.

Context-insensitive policies make an attacker's life much more difficult, narrowing potential attack targets from any instruction in the program to a small handful. The program's control-flow graph and call graph can be enforced using only context-insensitive policies, as such graphs are themselves context-insensitive. However, the execution model is more than the control flow graph, which still allows unrealizable program paths. For one thing, the program model should incorporate the calling convention, which restricts each return to have exactly one target – the return site of the caller depending on the context of invocation. In our example, a return address overwrite may still lead to an (innocuous) omission of the second execution of `eval`. One approach is to enforce the calling convention by introducing sandboxing checks to match call and return pairs. There are also a number of schemes we can use to reduce the size of allowed targets further without incurring high overheads.

4.3 Selective Code Duplication

Even the most accurate static analysis will not produce only singleton sets for context-insensitive policies. However, dynamic program transformations may be applied to further

reduce the points-to sets. We can try to partition the set of targets by dynamically applying program transformations on the generated traces.

We can apply program specialization with respect to function pointers passed as arguments, which is a common use case. This way way the target set of a function pointer's later uses is a singleton set. In our example, this will result in two versions of `eval` specialized on the different function pointers passed, which would be useful if the sets reached valuable system calls.

Furthermore, leaf functions can be partially inlined in traces from their callers, therefore they are also effectively reduced to singletons. A simple compare with the value of the singleton replaces the hash table lookup when detection of security violations is desired, otherwise it can be elided in a trace. In order to reduce the degree of freedom of return overwrite attacks, leaf functions with large fan-in can be selectively cloned and thus the return set of the original function is partitioned into smaller sets. In general, static analysis can determine each definition and use of a pointer used as an indirect branch target, and by selectively duplicating code from different definitions we can obtain selective flow- and context-sensitivity in the execution traces dynamically generated by the runtime system.

Chapter 5

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 by limiting attackers degree of freedom while preserving program correctness. We envision a system with customizable policy settings to allow the maximum protection level for each application. Our philosophy is to start with a completely safe subset and examine the causes of false alarms. The responsible operation is potentially dangerous, so its restrictions should be relaxed only so far as to reduce false alarms to a reasonable level while maintaining as much security as possible. Table 5.1 lists sample policy decisions that can be implemented with program shepherding. Our system currently implements a set of security policies for which allows normal execution for most application binaries, and a more restrictive set when static analysis of source code provides an accurate execution model. The applicability and attack resilience of these policies are described and analyzed later in this chapter.

Restricting	Least restrictive				Most restrictive
Code origins	Any		Dynamically written code, if self-contained and has no system calls	Only code from disk, can be dynamically loaded	***Only code from disk, originally loaded***
Function returns	Any	**Only to after-call targets**	*Only to known call sites*	Random `xor` as in StackGhost [26]	Return only from called function
Intra-segment call or jump	Any		**Only to function entry points**		*Only to bindings given in an interface list*
Inter-segment call or jump	Any		Only to export of target segment	**Only to import of source segment**	*Only to bindings given in an interface list*
Indirect calls	Any		Only to address stored in read-only memory	**Only within user segment or from library**	*Only to bindings given in an interface list*
`execve`	Any		**Static arguments**	Only if the operation can be validated	None
`open`	**Any**		Disallow writes to specific files, e.g. `/etc/passwd`	Only to a subregion of the file system	None

Table 5.1: 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 for binaries with no external information. Italicized entries show the more restrictive policies in place when an execution model is provided statically.

5.1 Code Origin Policies

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.

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 unless properly validated. 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* [2, 19] 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. Similarly, self-modifying code should usually be disallowed, but may be explicitly allowed for certain applications.

5.2 Control Transfer Policies

Direct control transfers that satisfy the code origin policies can always be allowed within a segment. Given that we limit execution models to those that disallow self-modifying code, direct control transfers will always perform as the program intends, as they are part of the code itself and cannot be modified by an attacker as guaranteed by the other techniques. 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 strengthened 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 source access, compiler support, profiling runs, or other external sources of information to determine all valid indirect call targets (Chapter 7). 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 information.

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 [47].

5.3 Sandboxing Policies

Sandboxing can provide detection of attacks that get past other barriers. For example, a data 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 thesis, but have been discussed elsewhere [30, 39].

A further on return target restriction can easily be provided to emulate a technique proposed in StackGhost [26]. 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. Instead we propose using a parallel stack in an inaccessible to the application area in memory and cached in hardware registers (XMM, or RSB – see Section 10.2), to match returns with their respective caller.

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, unless largely independent modules are to be isolated [65].

5.4 Attack Resilience

We now turn our attention to a specific security policy made up of the bold entries in Table 5.1. We implemented this policy in our prototype system when only a program

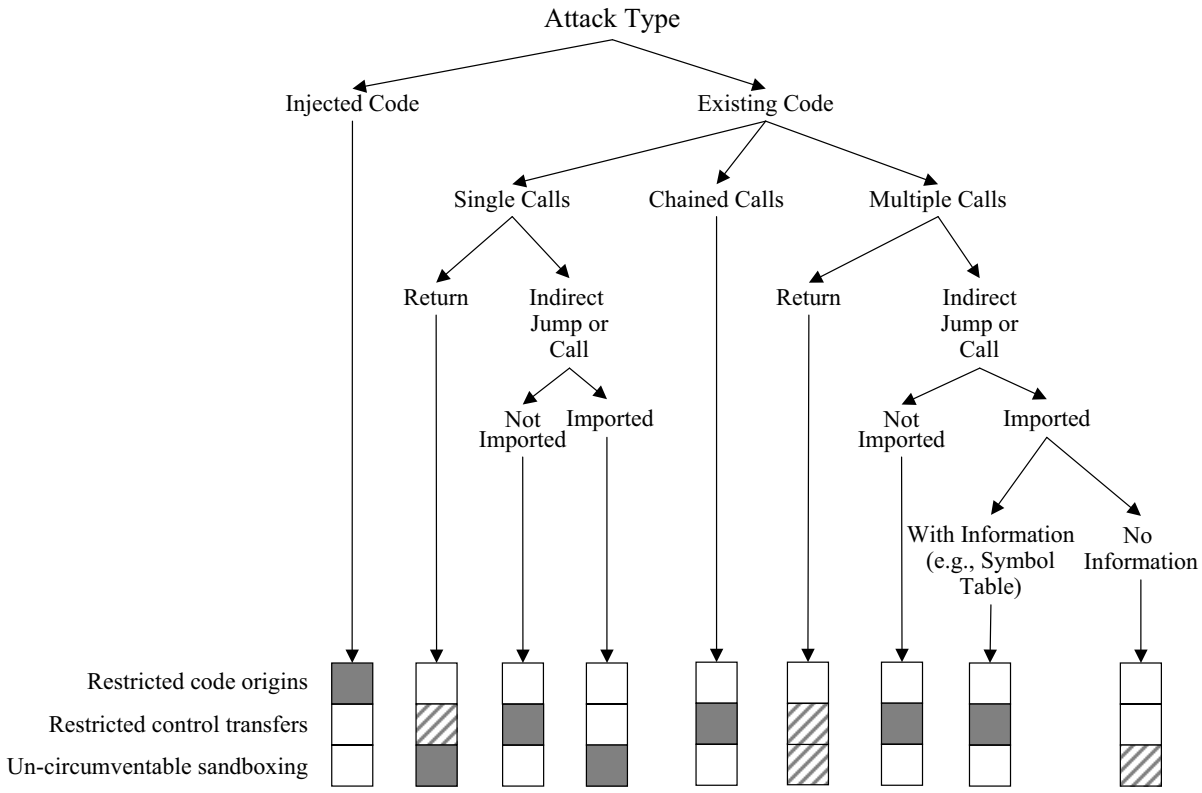

Figure 5-1: Capabilities of program shepherding's three techniques toward stopping different attack types, for the security policy indicated in bold in Table 5.1. The three boxes represent the three components. A filled-in box indicates that that component can completely stop the attack type above. Stripes 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 show techniques below it (e.g., sandboxing is capable of stopping some attacks of every type, but we show its use only when the other techniques do not provide full protection).

binary is available. For this security policy, Figure 5-1 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.

5.4.1 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 technique stops all exploits that introduce external code, and will eradicate 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.

5.4.2 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. With more accurate execution models they are even further limited only to their respective callers.

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 [39] 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. The *mimicry attacks* introduced [62] and further analyzed by Wagner [64] show how attackers can easily evade intrusion detection at the system call level. 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.

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 [44]. 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.

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.

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.

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 an execution model to provide the list of valid transfers. Whenever the degree of freedom of an attacker is limited to equivalent actions malicious sequences will never be constructed.

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 and will not be recognized as a threat by these techniques.

Chapter 6

Efficient Implementation of Program Shepherding

In order for a security system to be viable, it must be efficient. In order to be widely and easily adoptable, it must be maximally transparent to its users. 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 and privileged operation. Significant runtime components would also be needed to overcome the difficulties of statically handling indirect branches by merging information about dynamically loaded libraries and verifying that all modules are properly instrumented. In addition, the invariably introduced checks will impose significant performance penalties.

Another possibility is to use an implementation in an interpreter. Interpretetation is the most straightforward solution to provide complete mediation of control transfers on native binaries. It 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 evidenced in Table 6.1.

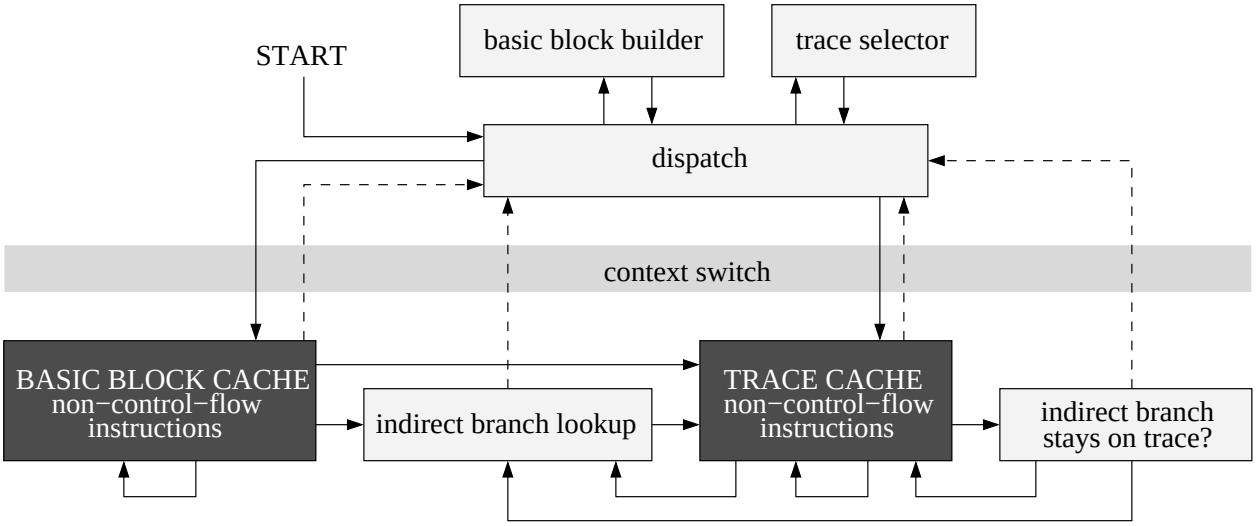

Figure 6-1: Flow chart of the DynamoRIO system infrastructure. Dark shading indicates application code. Note that the context switch is simply between the code cache and DynamoRIO; application code and DynamoRIO code all runs in the same process and address space. Dotted lines indicate the performance-critical cases where control must leave the code cache and return to DynamoRIO.

6.1 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 DynamoRIO [7]. DynamoRIO is based on top of an IA-32 port [6] of Dynamo [3]. DynamoRIO's optimizations are still under development. However, its baseline performance is already reasonable for many applications (see Section 8.3). DynamoRIO is implemented on IA-32 for both Windows and Linux, and is capable of running large

desktop applications.

A flow chart showing the operation of DynamoRIO is presented in Figure 6-1. 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 DynamoRIO's operation, focusing on the aspects that are relevant to our implementation of program shepherding. The techniques of program shepherding fit naturally within the DynamoRIO 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.

6.2 DynamoRIO: Runtime Introspection and Optimization

DynamoRIO 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 DynamoRIO (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, DynamoRIO *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 hash table lookup. Security policies that restrict indirect control transfers are put in place by varying this hash table lookup.

To improve the efficiency of indirect branches, and to achieve better code layout, basic

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 6.1: Performance achieved when various features are added to an interpreter, measured on two of the SPEC2000 benchmarks [56], crafty and vpr. Pure emulation results in an estimated slowdown factor of several hundred. Successively adding caching, linking, and traces brings the performance down dramatically.

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 hash table lookup, but if the check fails the full lookup must be performed. The superior code layout of traces usually amortizes the overhead of creating them and often speeds up the program [3, 51]. For context-insensitive security policies, no extra checks are required when execution continues across an indirect branch in a trace.

Table 6.1 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 DynamoRIO [6]. Signals on Linux are similarly intercepted. Security checks can be placed at the interception points, similarly to indirect branches. These abnormal control transfers are rare and so extra checks upon their interception do not affect overall performance.

6.3 Restricted Code Origins Implementation

Restricting execution to trusted code is accomplished by adding checks at the point where the system copies a basic block into the code cache. Checking code origins involves negligible overhead because code need only be checked once prior to insertion into the code cache. Once in the cache no checks need to be executed.

Code origin checking requires that DynamoRIO 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 [20] 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 DynamoRIO 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. For our prototype, we limit execution models to applications with no self-modifying or dynamically generated code, which is the case for the targeted security sensitive applications.

6.4 Restricted Control Transfers Implementation

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 hash table lookup routine translates the target program address into a basic block entry address.

6.4.1 Transfer Type Restrictions

Policies that only examine the target of a control flow transition are the cheapest to enforce. A separate hash table is used to look up the target for validation for different types of indirect control transfers (return instruction, indirect calls, and indirect branches). Security checks for indirect transfers that only examine their targets have little performance overhead, since we place in the hash table only targets that are allowed by the security policy. This enables type specific restrictions without sacrificing any performance when execution continues in the code cache. Targets of indirect branches are matched against entry points of PLT-defined [20] 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.

6.4.2 Transition Pair Restrictions

Our static analyses produce context-insensitive policies, which can be easily enforced with minimal overhead. This is because context-insensitive policies are always valid after initial verification, therefore they can be cached and cheaply evaluated with minimal execution overhead. Our execution model policies need to examine both the source and the target of a transition, which will have a slightly slower hash table lookup routine when shared tables are used. These checks can be optimized to be as efficient as only checking the target by using a separate hash table for each source location. The space drawback of this scheme is minor as equivalent target sets can be shared, and furthermore, the hash tables can be precomputed to be kept quite small without increase in access time.

6.5 Un-Circumventable Sandboxing Implementation

When required by the security policy, DynamoRIO 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. DynamoRIO 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 hash table lookup, go back to DynamoRIO, 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 DynamoRIO itself.

6.6 Protecting DynamoRIO

Program shepherding could be defeated by attacking DynamoRIO'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 DynamoRIO. Since the core of DynamoRIO is a relatively small piece of code, and DynamoRIO does not rely on any other component of the system, we believe extensive code review and analysis will leave no loopholes for exploitation.

6.6.1 Memory Protection

We divide execution into two modes: DynamoRIO mode and application mode. DynamoRIO mode corresponds to execution while in the top half of Figure 6-1. Application mode corresponds to the bottom half of Figure 6-1, including the code cache and the DynamoRIO routines that are executed without performing a context switch back to DynamoRIO. For the two modes, we give each type of memory page the privileges shown in Table 6.2. DynamoRIO data includes the indirect branch hash table and other data structures.

All application and DynamoRIO 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 from DynamoRIO, so it remains writable in DynamoRIO mode. DynamoRIO's data and the code cache can be written to by DynamoRIO itself, but they must be protected during application mode to prevent inadvertent or malicious modification by the application. We should also protect RIO's Global Offset Table (GOT) [20] by binding all imported symbols on program startup and then write-protecting the GOT.

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 6.2. Program shepherding's un-circumventable sandboxing guarantees that these system call checks are executed. Because the DynamoRIO 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 DynamoRIO's state cannot be corrupted within the process.

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

Table 6.2: Privileges of each type of memory page belonging to the application process executed under DynamoRIO. 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 DynamoRIO to execute.

6.6.2 Multiple Application Threads

DynamoRIO'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 DynamoRIO mode, only that thread's DynamoRIO data pages and code cache pages should be unprotected.

A potential attack could exploit this race condition: while one thread is in DynamoRIO mode another thread in application mode can modify the first thread's DynamoRIO data pages. We could solve this problem by forcing all threads to exit application mode when any one thread enters DynamoRIO 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 of extra synchronization would be unreasonable on a multiprocessor when threads are continuously context switching. We are investigating alternative solutions, including those in Section 10.1.

Chapter 7

Call Graph Construction

Constructing the call graph for a C program in the presence of indirect calls requires use of pointer analysis to disambiguate between the potential values of the used function pointers. Current research on pointer analyses [1, 58, 59, 15, 34] offers different tradeoffs between accuracy and scalability. Previous points-to analyses for C have also been specifically applied to call graph construction [33, 41].

7.1 Points-to Analysis

We have employed a context-insensitive, flow-insensitive Andersen's [1] style points-to analysis using *projection merging* [59] and *cycle elimination* [23]. It is implemented using the Banshee [4] analysis toolkit to build a customized constraint resolution engine. The C front-end is derived from David Gay's Region Compiler [28] and the GNU C Compiler. This type of points-to analysis scales very well to the size of our target applications, and on our targets it is practically dominated by source code preprocessing time. Flow-insensitive analysis ignores control flow and the order of assignments, and therefore the improved scalability comes at the cost of some losses in accuracy. However, the analysis results also hold even in concurrent programs.

Our current implementation is currently inaccurate in regards to treatment of assignments to `struct`/`union` fields. The field names are ignored and only the base object is looked at. Therefore, the analysis may produce larger than the actual points-to sets. Ac-

cording to the notion in [34] it is *field-independent* similar to the works in [59, 15, 58]. However, *field-based* analyses that ignore the base are suggested [33, 41] to be more accurate for call-graph construction. We further plan to use the intersection of result sets of both types of analysis to provide even more accurate points-to sets. In this thesis we will discuss only the performance of the field-independent instance of our analysis.

7.1.1 Dynamic Linking Support

The points-to information in our system is used at runtime and therefore needs to be efficiently propagated, and in case of position independent code relocated at run time. Use of dynamic libraries poses new problems in respect to combining the results of the independent local analyses on the shared objects and application executable. Previous modular combination techniques have been used for compile time analyses only by Das [15], and Heintze and Tardieu [34]. Our static analysis supports this model for shared objects by symbolically evaluating arguments that contain function pointers and thus allows binding and unification at runtime.

Currently our runtime component performs only a single level indirection of symbolic arguments to obtain the full points-to sets. While this scheme supports most use cases, it easily breaks when a structure containing function pointers is successively crossing module boundaries. Nevertheless, in practice this crude binding technique was sufficient to handle all SPEC2000 benchmarks, with the exception of one transition in `gcc`. In that case, an assignment to an `obstack` structure field was crossing two module boundaries, which resulted in an incomplete transition set in the forementioned settings. Therefore, it was flagged as alarming by the runtime system. This particular case is shown to be properly handled by a field-based analysis of structure field assignments [41] and will be valid even across module boundaries within our system.

The general problem of merging points-to sets, however, should be solved by computing the transitive closure for all points-to sets that cross module boundaries. The depth needed for this graph reachability analysis is determined by the maximum number of module boundary transitions between pointer definition and use. In order to reduce program

startup overhead, the fully expanded sets should be precomputed together with the executable. They will need to be recomputed only in the rare occasions that shared libraries are modified. The memory footprint of a straightforward representation of the final precomputed unique sets, as observed on our benchmarks, is in one or two 4KB pages. A self-contained security sensitive executable should be augmented with an ELF [20] section which holds the fully precomputed points-to sets for each indirect call and is memory mapped as read-only. Our prototype implementation currently refrains from binary modifications. Furthermore, statically available transition sets allow code auditors to easily analyze vulnerable program address targets.

7.1.2 Matching Analysis Results with Program Binaries

Since our analysis is not in the program build process, we have to match the call-site information we obtain from static source code analysis to the actual indirect call instructions in the executable. We have applied our post-process call-site matching on locally installed or previously built program executables and shared objects. We have experimented so far only with binaries produced by the `gcc` compiler, but any other compiler may be used to build the final binary, as far as it generates accurate enough debugging information at high optimization levels. Most calls can be unambiguously matched since they are usually sparsely located across function and line boundaries. However, debugging information is insufficient to disambiguate between indirect calls on the same line. Since evaluations between sequence points are compiler implementation dependent, we occasionally have to merge points-to sets for several indirect calls. If this analysis is used in an infrastructure with more precise code generation information, any artifactual inaccuracy of this external matching will not be present. On the other hand, although our current static analysis requires source code access, debug information is usually already present and recompilation is not necessary.

7.2 Program Profiling

Our static analysis provides a close upper bound on the points-to sets. It should be noted that for our purposes, a safe approximation on a points-to set can even be a lower bound on the accurate points-to set. In this respect it is opposite to the traditional notion of conservative estimation. An automatic points-to analysis that may miss some potential valid transitions may produce false alarms (false positives), but it will not introduce unintended transitions in the model. Therefore an omission in the deduced model may cause denial of service to unusual requests with legitimate intent, but it will never allow an malicious request.

An easy way to obtain the target sets for a flow-insensitive, context-insensitive validation in our program shepherding system is to run it in in “learning” mode to only flag invalid indirect transitions pairs. Successive executions use the results of previous runs as a model and allow only those transitions. This method has its own merit, especially in the absence of source code access. However, it is prone to a high number of false positives and for quick convergence requires profiling runs with high code coverage. Notwithstanding these adverse effects on program correctness, alarms on execution paths which have not been covered in test environment may be considered useful to establishments with high security requirements.

Chapter 8

Experimental Results

This section presents our test suite of vulnerable programs, shows the effectiveness of our program shepherding system on this test suite, and then evaluates the performance and memory requirements of our system on the SPEC2000 benchmarks [56].

8.1 Test Suite of Vulnerable Programs

We constructed several programs exhibiting a full spectrum of buffer overflow and format string vulnerabilities. Our experiments also included the SPEC2000 benchmark applications [56] and the following applications with recently reported security vulnerabilities:

stunnel-3.21 CVE-2002-0002 [14] A format string vulnerability in `stunnel` (SSL tunnel) allows remote malicious servers to execute arbitrary code because several calls to `fdprintf` (a custom file descriptor wrapper of `fprintf`) have no format argument.

groff-1.16 CVE-2002-0003 [14] 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 [48].

ssh-1.2.31 CVE-2001-0144 [14] An integer-overflow bug in the CRC32 compensation attack detection code causes the SSH daemon (typically run as root) to create a hash

table with size zero in response to long input. Later attempts to write values into the hash table provide attackers with random write access to memory.

sudo-1.6.1 CVE-2001-0279 [14] `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 [36].

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 [8, 66, 36, 48] to inject new code [46], reuse existing code in a single call, or reuse code in a chain of multiple calls [44]. 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 [20] entries. We investigated attacks against DynamoRIO itself, e.g., overwriting DynamoRIO's GOT entry to allow malicious code to run in DynamoRIO mode, but could not come up with an attack that could bypass the protection mechanisms presented in Section 6.6.

All vulnerable programs were successfully exploited when run on a standard RedHat 7.2 Linux installation. Execution of the vulnerable binaries under DynamoRIO with all security checks disabled also allowed successful intrusions. Although DynamoRIO 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 DynamoRIO enforcing the policies shown in bold on Table 5.1, 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.

8.2 Effectiveness of Static Analysis

We applied our static points-to analysis and runtime execution model enforcement of indirect branches on the SPEC2000 benchmarks [56] in C, two popular security sensitive applications that are usually run with high privileges, the GNU C library which is dynamically linked to all applications, as well as other supporting libraries.

We have not invested considerable time and effort to actually devise exploits for these target applications, without being detected by the policies shown in bold in Table 5.1. Nevertheless, specifically crafted vulnerable programs, which allow multiple intrusions to modify function pointers, were quite easily used as proxies to all already present system calls. It will be hard to argue that the difficulties in similar attacks to real applications are unsurmountable, therefore we dissect the worst case scenario of potential future attacks when the policies shown in italic in Table 5.1 are enforced.

We summarize the results for the benchmarks with nontrivial target sets in Table 8.1. The size of the maximum set of targets for an indirect call is given, as a measure of the largest degree of freedom for an execution deviation. (We consider average set sizes to be a misleading metric for security assessment and instead provide a worst case metric.) The indirect calls in the executable or the shared object are given for reference. The size of the union of all indirect call targets in our sets is provided for comparison with a much simpler analysis technique that allows indirect calls to any address taken function. An even less restrictive policy that can be applied on unstripped binaries may allow all function entry points in the executable and the shared libraries to be in the valid target sets.

The interpreters in the benchmarks — `gap` and `perl` have high maximum call set size due to dynamic method dispatch and that is not surprising. Inspection of the maximum size sets of the other benchmarks show that they contain functions with similar behavior and their size reflects intrinsically equivalent operations for the application, e.g. generic code generation in `gcc`, generic handling of multiple ciphers in `sshd`. However, inaccuracies due to field-independence result in larger sets than best obtainable. For example, the maximal size set of `sshd` is three times smaller when that points-to set is decomposed over the structure fields. In terms of freedom of choice for an attacker, most of the sets usually

Benchmark	Indirect calls	Functions	Union	Maximum
ammp	27	191	32	16
mesa	694	1073	440	440
gap	1275	865	614	268
gcc	137	2031	269	129
perlbmk	64	1042	448	433
vortex	18	935	41	37
glibc-2.2.4.so	687	2582	380	185
sendmail-8.12.6	100	685	116	84
openssh-3.5p1	133	738	100	41

Table 8.1: Static points-to analysis results. The total number of functions and indirect calls shown is as found in the executable or shared object. The size of the set of functions present in the union of all target sets, and the size of the maximum set of call targets are obtained by our analysis.

provide similar facilities, i.e. equivalent, if any, system calls. Therefore control over a function pointer constrained to each of these sets will have limited utility. We are currently automating this evaluation in order to fully quantify the effective degree of freedom of all target sets, and to identify potential applications of the techniques from Section 4.3.

8.3 Performance

Figure 8-1 shows the performance of our system enforcing execution model policies of applications on a Linux system with a Pentium 4 processor. The figure shows normalized execution time for the SPEC2000 benchmarks [56], compiled with full optimization and run with unlimited code cache space. The first bar gives the performance of DynamoRIO by itself. DynamoRIO 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 employing the context insensitive enforcement strategies on indirect control transfers using a shared hash table, as discussed in Section 3.2. The benchmarks marked with an asterisk were not in C and therefore were run with static analysis based on profiling information. The results show that the additional overhead is negligible on most benchmarks, even without optimizing our prototype to site-specific hash tables.

The final bar gives the overhead of protecting DynamoRIO itself. This overhead is

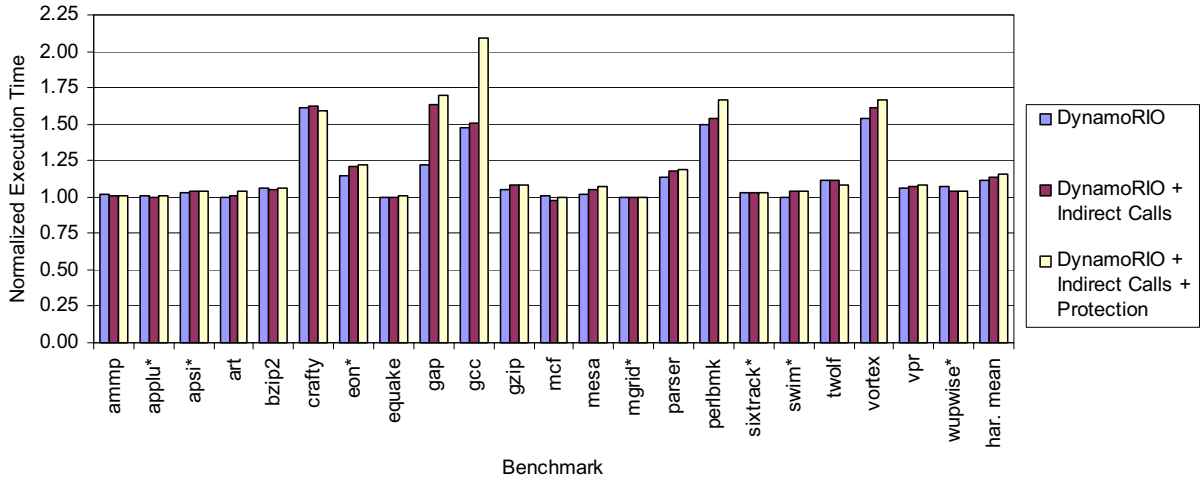

Figure 8-1: Normalized program execution time for our system (the ratio of our execution time to native execution time) on the SPEC2000 benchmarks (excluding FORTRAN 90 benchmarks) on Pentium 4 under Linux. They were compiled using `gcc -O3`. The final set of bars is the harmonic mean. The first bar is for DynamoRIO by itself; the middle bar shows the overhead of program shepherding (employing context insensitive restrictions on indirect control transfers); and the final bar shows the overhead of the page protection calls to prevent attacks against the system itself.

again minimal, within the noise in our measurements for most benchmarks. Only `gcc` has significant slowdown due to page protection, because it consists of several short runs with little code re-use. We are working on improving our page protection scheme by lazily unprotecting only those pages that are needed on each return to DynamoRIO mode.

Dynamically constructed policies based on the bold entries in Table 5.1 for applications with no source access provide another set of performance measurements. Figure 8-2 and Figure 8-3 show the performance of our system on a Pentium III processor under Linux and Windows, respectively. Each figure shows normalized execution time for the SPEC2000 benchmarks [56], 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 DynamoRIO by itself. The slightly lower performance of the base system on a Pentium 4 compared to a Pentium III likely results from aggressive microarchitectural optimizations that improve native execution but interact inefficiently with our runtime system (e.g. return stack buffer, trace cache for μops). The second bar shows the performance of program shepherding enforcing the poli-

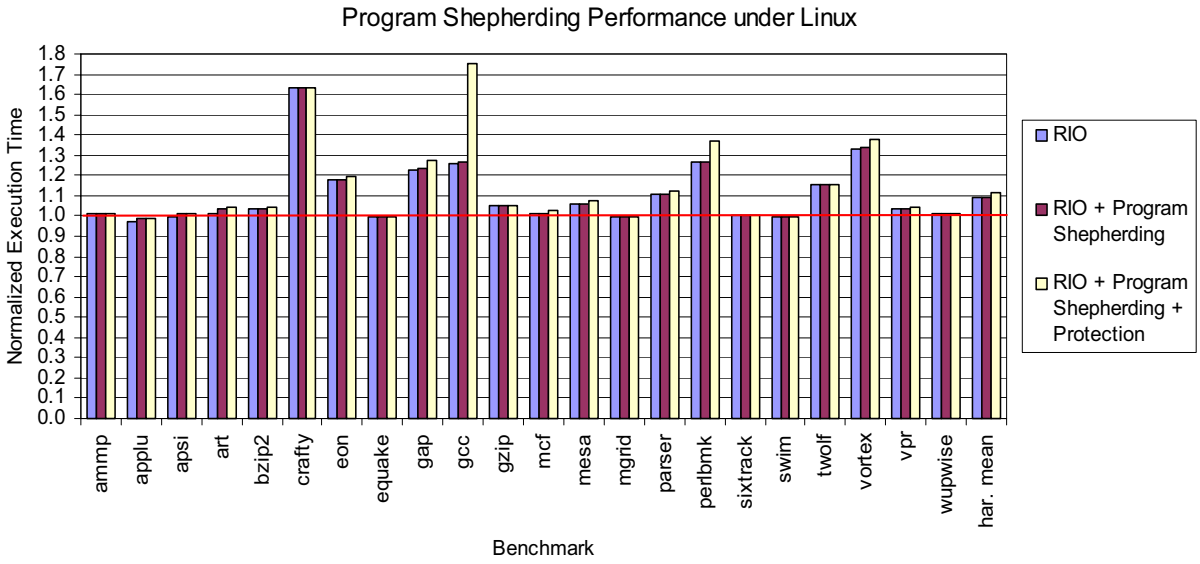

Figure 8-2: Normalized program execution time for our system (the ratio of our execution time to native execution time) on the SPEC2000 benchmarks [56] (excluding all FORTRAN 90 benchmarks) on Pentium III under Linux. They were compiled using `gcc -O3`. The first bar is for DynamoRIO by itself; the middle bar shows the overhead of program shepherding (with the security policy shown in bold in Table 5.1); and the final bar shows the overhead of the page protection calls to prevent attacks against the system itself.

cies shown in bold in Table 5.1. Resricted control transfers are enforced by hash table partitions on instruction type. The results show that the overhead of program shepherding enforcing these policies is indistinguishable from the overhead of the base system. This is also expected to be the case for transition pair enforcement using hash tables per-site.

The final bar gives the protection overhead on each operating system. The previously noted slowdowns on several benchmarks are seriously exarcebated on Windows, especially `gcc`. We conjecture that the significant difference between slowdowns on protection on Linux and Windows is because Windows is much less efficient at changing privileges on memory pages than Linux due to coarse grain invalidations and significantly many subsequent TLB misses.

8.4 Memory usage

The memory usage of our security system is shown in Table 8.2. All sizes shown are in KB. The left half of the table shows the total size of text sections of each benchmark and

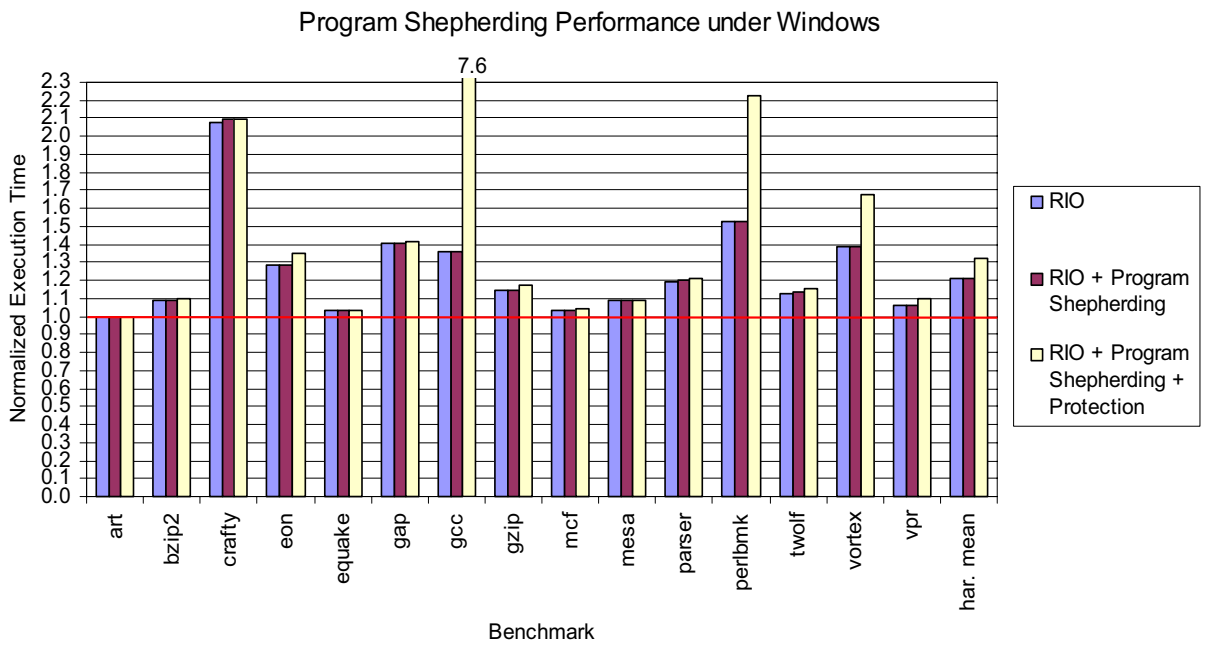

Figure 8-3: Normalized program execution time for our system (the ratio of our execution time to native execution time) on the SPEC2000 benchmarks [56] (excluding all FORTRAN benchmarks) on Pentium III under Windows 2000. They were compiled using `cl /Ox`. The first bar is for DynamoRIO by itself; the middle bar shows the overhead of program shepherding (with the security policy shown in bold in Table 5.1); and the final bar shows the overhead of the page protection calls to prevent attacks against the system itself.

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. The right half of Table 8.2 shows the memory overhead of DynamoRIO compared to the memory usage of each benchmark. For most benchmarks the memory used by DynamoRIO is a small fraction of the total memory used natively.

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. However, whenever multiple copies of a process are executed simultaneously these metrics may provide an incomplete picture. On modern operating systems static code is usually shared across processes and multiple instances do not require additional physical memory. When such programs are executed under DynamoRIO, the portion of their executed code will be duplicated in code cache in private pages per process. Therefore it may result in higher physical memory requirements. On the other hand, if multiple instances operate on

different data and hence exercise different code paths, each copy will be specialized for its particular instance. We have yet not quantified these effects in systems with a lot of sharing or with variations of application workload mixes. Still, we are considering schemes for efficient sharing between code caches of different processes in order to minimize memory overheads.

benchmark	static code	executed code	% executed	native	DynamoRIO	% 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	—	—	4.5%	—	—	1.8%

Table 8.2: Memory usage of the SPEC2000 benchmarks [56], 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 DynamoRIO when running the benchmark. Extra memory is the total memory used by DynamoRIO code, code caches and other DynamoRIO data. Native total is total memory used by the benchmark when run natively (outside of DynamoRIO).

Chapter 9

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 here the more successful ones.

StackGuard [13] 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. Attacks exploiting random access memory writes or targeting other program addresses are therefore not affected.

FormatGuard [12] is a library patch for dynamic checks of format specifiers to detect format string vulnerabilities. However, it is limited to programs which directly use the standard `printf` library functions without using custom wrappers around them.

Static analyses have also been applied for detection of very common classes of vulnerabilities as buffer overflow [63] and format string [54] vulnerabilities. When capturing most common cases, these tools report relatively low false positive rates.

StackGhost [26] 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 unless attackers are able to read the process addresses.

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 [26] and in the compiler patch StackShield [61]. 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 [13] have also shown significant performance penalties.

Enforcing non-executable permissions on IA-32 via OS kernel patches has been done for stack pages [18] and for data pages in PaX [49]. Our system provides execution protection from user mode on unmodified binaries and achieves better steady state performance. Protection against attacks using existing code was also proposed in PaX by randomizing placement of position independent code; however, it is open to attacks that are able to read process addresses and thus determine the program layout.

Type safety of C code has been proposed by the CCured system [43] which extends the C type system, infers statically verifiable type safe pointers, and adds run time checks only for unsafe pointers. Cyclone [35] provides a safe dialect of C in a similar fashion, but requires annotations in conversion of legacy code. The reported overhead of these systems is in the 30–300% range. We present a much easier to adopt system that requires no recompilation or code modification.

Other programming bugs stemming from violations of specific higher level semantic rules of safe programming have been targeted by static analyses like CQUAL [25], ESP [16], MC [31], and static model checkers SLAM [60], MOPS [10]. In an unsafe language like C, techniques that claim to be sound do not hold in the presence of violations of the memory and execution model assumed in the analyses [60]. Our system may be used

to complement these approaches and enforce the execution model of the application.

Most host-based intrusion detection systems focus on the sequences of system calls executed by an application [24, 30, 27, 50, 62, 40]. The *mimicry attacks* introduced [62] and further analyzed by Wagner [64] show how attackers can easily evade existing intrusion detection at the system call level by introducing undetected sequences of system calls. Our example from Section 5.4.2 is an information flow [32] mimicry attack. While we agree that system calls contain all externally visible security relevant program actions, we also argue that system call interposition provides an incomplete interface for containing attacks. Allowing malicious code to masquarade as the normal application, and divert its system call trace is already too empowering.

Our system is close in spirit to the hybrid approach of using static analysis and runtime model checking proposed by Wagner and Dean [62]. A static analysis is used to construct a finite state automaton recognizing the system calls possibly generated by a program, a runtime component to simulate the nondeterministic model space, and a system call interposition tool to verify the generated by the program sequence. The finite automata are generated based on assumed valid execution models — context-insensitive represented as a call graph, or context-sensitive as result of modeling a runtime stack. Our system is as at least as accurate in detection of malicious system call sequences, since it disallows any deviations from the execution model they assume. The non-determinism in the generated finite automata often has high runtime simulation overheads for the more accurate models. Our techniques subsume the need to further model and dynamically check system calls, and we present a practical system with minimal overhead. An extension to the runtime model checking applicable to the case of remote execution systems has been presented [29]. The addition of network based attacks drastically changes the threat model, and the large network latency dwarfs the overheads of NFA simulation. Hence, direct applicability to host based intrusion detection is likely to be inefficient.

The indirect call target sets in both [62, 29] are considered to contain all address taken functions. This is obtained either by source code [62] or binary code [29] analysis. The call graphs in our models are much more accurate as obtained by a points-to analysis. The possibility of adapting a points-to analysis to binary code is hypothesized [29] and that will

be an interesting area of research. Both [62, 29] address the issue of argument manipulation. However, the proposed techniques already allow arguments that can be changed dynamically — and in our system statically determined by the existing code arguments cannot be modified by attackers. Accurate containment of argument replacement attacks is not yet effectively addressed by most other IDS.

Software fault isolation techniques [65, 55] modify a program binary to restrict the address range of memory operations. Execution monitors [52] were applied in SASI [22] to enforce a memory model via static code instrumentation. These systems have much higher overheads due to heavy-weight sandboxing operations and inability to elide them from critical paths.

Our base system infrastructure itself, DynamoRIO [6, 7], is based on an IA-32 port of Dynamo [3]. Other software dynamic optimizers are Wiggins/Redstone [17], which employs program counter sampling to form traces that are specialized for the particular Alpha machine they are running on, and Mojo [11], which targets Windows NT running on IA-32. None of the above has been used for anything other than optimization. Strata [53] uses dynamic translation with lower performance to enforce a subset of the techniques we have presented earlier [38].

Chapter 10

Future Work

The performance results show that our prototype is an already practical system. We are continuing our optimization efforts and believe that good engineering will yield close to zero overheads in all aspects of the system. Many opportunities for optimization exist in the base system and it is gradually improved with respect to latest microarchitecture specifics. Better trace creation heuristics will result in easily reachable steady state which has no overheads. We have already delineated how the security policy enforcement can be implemented with minimal additional overhead. The costs of protection with the current system can readily go down with an on-demand protection changes. Nevertheless, we present here alternative solutions to these problems when supported by the operating system or underlying hardware.

10.1 Operating System Extensions

The described so far technique allows a program shepherding implementation solely in one protection domain — it may be an application or an OS kernel. A program shepherding system protecting user mode applications, as our current prototype, can benefit from further facilities already provided by the hardware memory management unit. Operating support will be needed in order to take advantage of these privileged facilities. On most computer architectures including IA32, a supervisor privilege bit determines whether a particular page is writable. All pages that need to be read-only in application mode, but writable in

shepherding mode, can be marked as supervisor pages. The protection bits of the pages will thus be static and will not need to be modified between the two contexts. Switching contexts will then have the cost of a kernel trap and will be free of race conditions. Additional techniques for improving context switch performance, including use of segmentation hardware are discussed in CoVirt [37].

10.2 Hardware Support

The currently implemented system does not require any hardware support and yet achieves minimal overheads. In order to fully remove the checks from critical execution paths for maximum performance some components of the system can be implemented in hardware. Our requirements often overlap substantially with already existing hardware facilities. Therefore they only require minimal modifications for an interface with a program shepherding system, whenever a fast hardware path fails. Modern processors cache control data that needs to be validated only when it is put in the cache. Examples of these are the instruction translation look-aside buffers (TLB); Branch Target Address Caches (BTAC) used for indirect branch prediction; and return stack buffers (RSB) used for return target prediction. Traps on misprediction or cache misses will often allow software to handle the slow paths and perform the security checks needed.

Intel's processors have included a return stack buffer (RSB) since the Pentium Pro [47]. The RSB is of limited size and is used as a branch predictor for return instructions. On a call the return address is pushed onto the RSB, and on a return the top RSB value is popped and used as the predicted target of the return. Since the hardware is storing each return address, it is only natural to propose using the RSB to enforce the calling convention.

Exposing the RSB to software might be done by allowing read and write access. Then a program shepherding system could monitor every call and return and insert code to handle underflow and overflow and code to compare the RSB prediction to the real return address. On overflow, the RSB is copied to memory which is then protected. On underflow, the most recent saved RSB copy is written in to the RSB. For better performance only half of the RSB is stored and swapped in, with the upper half being shifted down on overflow, to

prevent thrashing due to frequent minor call depth changes.

A further level of hardware support would be to add traps for underflow, overflow, and RSB misprediction. Then the software need not impose instrumentation on every call and return; it would simply need to handle the traps.

Chapter 11

Conclusions

This thesis introduced 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 DynamoRIO 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. We have shown that by enforcing the program's execution model by restricting control transfers, we are able to thwart current and future security attacks. We incorporate static program analysis with dynamic analysis and program transformations to provide an efficient enforcement of the execution model.

We have discussed the potential design space of security policies that can be built using program shepherding. Our system currently implements a set of policy settings for trusted binaries with no source access, and a stricter set when source is available. We are expanding the set of security policies that our system can provide without significant loss of performance. Future expansions include using more 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 [20] and allowing updates only by the dynamic resolver can easily be implemented in a secure and efficient fashion.

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. Furthermore, data modifications that lead to a conditional control flow shift may still be approachable by an extension of this technique. Infeasible execution paths as a result of data changes in between guarding conditional branches may be detected to point out memory model violations due to security attacks or unspecified race conditions.

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 [21] 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 to provide an easy adoption path. 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.

Bibliography

[1] L.O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. DIKU report 94/19.

[2] 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.

[3] 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.

[4] Banshee. `http://bane.cs.berkeley.edu/banshee`.

[5] blexim. Basic integer overflows. *Phrack*, 10(60), December 2002.

[6] 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 2001.

[7] Derek Bruening, Timothy Garnett, and Saman Amarasinghe. An infrastructure for adaptive dynamic optimization. In *1st International Symposium on Code Generation and Optimization (CGO-03)*, March 2003.

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

[9] CERT/CC Advisories. Computer Emergency Response Team (CERT). `http://www.cert.org/advisories/#2002`.

[10] Hao Chen and David Wagner. MOPS: An infrastructure for examining security properties of software. In *ACM Conference on Computer And Communications Security (CCS 2002)*, 2002.

[11] 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.

[12] 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.

[13] 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.

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

[15] Manuvir Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2000)*, Vancouver, BC, Canada, June 2000.

[16] Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: Path-sensitive program verification in polynomial time. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, June 2002.

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

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

[19] 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.

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

[21] 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.

[22] Ulfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings of the New Security Paradigms Workshop*, September 1999.

[23] Manuel Fahndrich, Jeffrey S. Foster, Zhendong Su, and Alexander Aiken. Partial online cycle elimination in inclusion constraint graphs. *ACM SIGPLAN Notices*, 33(5):85–96, 1998.

[24] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A sense of self for Unix processes. In *Proceedinges of the 1996 IEEE Symposium on Research in Security and Privacy*, pages 120–128. IEEE Computer Society Press, 1996.

[25] J. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation PLDI.*, 2002.

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

[27] Tal Garfinkel. Traps and pitfalls: Practical problems in in system call interposition based security tools. In *Proc. Network and Distributed Systems Security Symposium*, February 2003.

[28] David Gay and Alexander Aiken. Language support for regions. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 70–80, Snowbird, Utah, June 2001.

[29] Jonathan T. Giffin, Somesh Jha, and Barton P. Miller. Detecting manipulated remote call streams. In *11th USENIX Security Symposium*, August 2002.

[30] 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.

[31] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A system and language for building system-specific, static analyses. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation PLDI.*, 2002.

[32] Nevin Heintze and Jon G. Riecke. The SLam calculus: programming with secrecy and integrity. In ACM, editor, *POPL 1998 the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 365–377, 1998.

[33] Nevin Heintze and Olivier Tardieu. Demand-driven pointer analysis. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 24–34, Snowbird, Utah, June 2001.

[34] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using CLA: A million lines of code in a second. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 254–263, Snowbird, Utah, June 2001.

[35] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *USENIX Annual Technical Conference*, June 2002.

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

[37] Samuel T. King and Peter M. Chen. Operating system extensions to support host based virtual machines. Technical Report CSE-TR-465-02, University of Michigan, 2002.

[38] Vladimir Kiriansky, Derek Bruening, and Saman Amarasinghe. Secure execution via program shepherding. In *11th USENIX Security Symposium*, August 2002.

[39] 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.

[40] Yihua Liao and V. Rao Vemuri. Using text categorization techniques for intrusion detection. In *11th USENIX Security Symposium*, August 2002.

[41] A. Milanova, A. Rountev, and B. G. Ryder. Precise call graph construction in the presence of function pointers. In *Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation*, October 2002. (SCAM'02).

[42] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. The spread of the sapphire/slammer worm, January 2003. `http://www.cs.berkeley.edu/~nweaver/sapphire/`

[43] George C. Necula, Scott McPeak, and Westley Weimer. CCured: type-safe retrofitting of legacy code. In *Symposium on Principles of Programming Languages POPL*, pages 128–139, 2002.

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

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

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

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

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

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

[50] Niels Provos. Improving host security with system call policies. Technical Report CITI-TR-02-03, CITI, University of Michigan, November 2002.

[51] 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.

[52] Fred B. Schneider. Enforceable security policies. *Information and System Security*, 3(1):30–50, 2000.

[53] Kevin Scott and Jack Davidson. Safe Virtual Execution using software dynamic translation. In *Proceedings of the 2002 Annual Computer Security Application Conference*, December 2002.

[54] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *In Proceedings of the 10th USENIX Security Symposium, 2001*, pages 201–220, 2001.

[55] C. Small. MiSFIT: A tool for constructing safe extensible c++ systems. In *Proceedings of the Third USENIX Conference on Object-Oriented Technologies*, Portland, Oregon, June 1997. USENIX.

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

[57] Stuart Staniford, Vern Paxson, and Nicholas Weaver. How to 0wn the internet in your spare time. In *11th USENIX Security Symposium*, August 2002.

[58] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Programming Languages*, pages 32–41, 1996.

[59] Zhendong Su, Manuel Fähndrich, and Alexander Aiken. Projection merging: Reducing redundancies in inclusion constraint graphs. In *Proceedings of the 27th Annual ACM Symposium on Principles of Programming Languages*, pages 81–95, 2000.

[60] Sriram K. Rajamani Thomas Ball. The SLAM project: Debugging system software via static analysis. In *POPL 2002*, pages 1–3, January 2002.

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

[62] David Wagner and Drew Dean. Intrusion detection via static analysis. In *IEEE Symposium on Security and Privacy*, 2001.

[63] David Wagner, Jeffrey S. Foster, Eric A. Brewer, and Alexander Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, pages 3–17, San Diego, CA, February 2000.

[64] David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *ACM Conference on Computer And Communications Security (CCS 2002)*, 2002.

[65] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. *ACM SIGOPS Operating Systems Review*, 27(5):203–216, December 1993.

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