Execution Model Enforcement Via Program Shepherding

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

Abstract

Nearly all security attacks have one thing in common: they coerce the target program into performing actions that it was never intended to perform. In short, they violate the program's *execution model*. The execution model encompasses the Application Binary Interface (ABI), higher-level specifications from the program's source programming language, and components specific to the program — for example, which values a particular function pointer may take. If this execution model were enforced, and only program actions that the programmer intended were allowed, a majority of current security holes would be closed.

In this paper, we employ *program shepherding* [26] to enforce a program's execution model. Program shepherding monitors control flow in order to enforce a security policy. We use static and dynamic analyses to automatically build a custom security policy for a target program which specifies the program's execution model. We have implemented our analyses in the DynamoRIO [4, 5] runtime code modification system. The resulting system imposes minimal or no performance overhead, operates on unmodified native binaries, and requires no special hardware or operating system support. Our static analyses require source code access but not recompilation. The analysis 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, greatly reducing the possibility of an unintended program action.

1. INTRODUCTION

The greatest threat to our modern information infrastructure is the remote exploitation of program vulnerabilities. The goal of most security attacks is to gain unauthorized access to a computer system by taking control of a vulnerable privileged program. This is done by exploiting bugs that allow overwriting stored program addresses with pointers to malicious code. Today's most prevalent attacks target buffer overflow and format string vulnerabilities.

Nearly all attacks have one thing in common: they coerce the target program into performing actions that it was never intended to perform. In short, they violate the *execution model* followed by legitimate programs. 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. For example, a common attack type overwrites a return address to point to a malicious destination. This destination is not a valid return target in the program's execution model, and would be disallowed by enforcement of the model.

Most security attacks target data storing program addresses. It is unreasonable to try to stop all malevolent writes to memory containing program addresses, because addresses are stored in many different places and are legitimately manipulated by the application, compiler, linker, and loader. Our model of security attacks assumes that an attacker is able to exploit program vulnerabilities to gain random write access to arbitrary memory locations in the program address space.

In this paper, we employ *program shepherding* [26] to enforce a program's execution model. Instead of attempting to protect data, program shepherding monitors control flow in order to enforce a security policy. 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, greatly reducing the possibility of an unintended program action.

We have implemented our analyses in the DynamoRIO [4, 5] runtime code modification system. DynamoRIO executes a program through copies of its code stored in a cache. This code cache is the key to efficient, secure execution, because it allows many security checks to 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 overhead. A second key feature of DynamoRIO is the creation of *traces*, hot streams of code that cross control flow transitions. Security checks on transitions can be elided when execution follows the trace. These features result in a secure system that imposes minimal or no performance overhead, operates on unmodified native binaries, and requires no special hardware or operating system support. Our static analyses require source code access but not recompilation.

The contributions of this paper are as follows:

- We demonstrate that enforcing a program's execution model can thwart many security attacks;
- We analyze features of the execution model that can be enforced with reasonable cost and those whose costs are unacceptable;

This research was supported in part by the Defense Advanced Research Projects Agency under Grant F29601-01-2-0166.

- We show how to incorporate static analyses to automatically extract features of the program's execution model such as its call graph;
- We identify aspects of enforcement that can be performed and further enhanced dynamically;
- We present two different schemes for verifying that the calling convention is followed;
- We give experimental evidence that the execution model can be enforced efficiently and effectively using a runtime code modification system.

In Section 2 we classify the types of security exploits that we are aiming to prevent. The execution model and how to enforce it is discussed in Section 3. Applying the enforcement policies using program shepherding is described in Section 4, and our implementation is discussed in Section 5. We present experimental results and the performance of our system in Section 6. We discuss related work in Section 7 and conclude the paper in Section 8.

2. SECURITY EXPLOITS

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

2.1 Program Vulnerabilities

The two most-exploited classes of program bugs involve buffer overflows and format strings. Buffer overflow vulnerabilities are present when a buffer with weak or no bounds checking is populated with user supplied data. A trivial example is unsafe use of the C library functions `strcpy` or `gets`. This allows an attacker to corrupt adjacent structures containing program addresses, most often return addresses kept on the stack [9]. Buffer overflows affecting a regular data pointer can actually have a more disastrous effect by allowing a memory write to an arbitrary location on a subsequent use of that data pointer. One particular attack corrupts the fields of a double-linked free list kept in the headers of `malloc` allocation units [25]. On a subsequent call to `free`, the list update operation `this`→*prev*→`next = this`→*next*`;` will modify an arbitrary location with an arbitrary value.

Format string vulnerabilities also allow attackers to modify arbitrary memory locations with arbitrary values and often outrank buffer overflows in recent security bulletins [8, 30]. A format string vulnerability occurs if the format string to a function from the `printf` family ({`,f,s,sn`}`printf, syslog`) is provided or constructed from data from an outside source. The most common case is when `printf(str)` is used instead of `printf("%s",str)`. The first problem is that attackers may introduce conversion specifications to enable them to read the memory contents of the process. The real danger, however, comes from the `%n` conversion specification which directs the number of characters printed so far to be written back. The location where the number is stored and its value can easily be controlled by an attacker with type and width specifications, and more than one write of an arbitrary value to an arbitrary address can be performed in a single attack.

It is very difficult to prevent all exploits that allow address overwrites, as they are as varied as program bugs themselves. In this paper we assume that attackers can exploit a vulnerability that gives them random write access to arbitrary addresses in the program address space. 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) [14] of shared object entries.

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 [31]. Modifying any stored program address to point to the introduced code will trigger intrusion when that address is used for control transfer. In our previous work [26] we have presented a system with minimal overhead that eradicates this class of attacks; they will not be discussed here.

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. 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.

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 [29]. 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.

3. 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 and expected interaction with the operating system and with 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 not efficiently enforceable. The underlying hardware has no support for ensuring that calls and returns match, and it is prohibitively expensive to implement this in software. For this reason, the execution model is a convention rather than a strict set of rules. However, most security exploits come from violations of the execution model. The most prevalent attacks today involve overwriting a stored program address with a pointer to injected malicious code. The transfer of control to that code is not allowed under the program's execution model. Enforcing the model would thwart many security attacks.

Much work has been done on enforcing the execution model's specifications on data usage, from sandboxing the address space [45] to enforcing non-executable privileges on data pages [32] and stack pages [13]. However, these schemes have significant performance costs, as restrictions on data usage are very difficult to enforce efficiently. This is because memory references all look very similar statically and must be disambiguated dynamically. 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 is extremely difficult. We restrict our enforcement of the execution model to the set of allowed control transfers. We focus on control transfers, rather than on data, for two reasons. First, control transfer specifications inhabit a much smaller (and therefore reasonably managed) space than arbitrary restrictions on data. And second, nearly all unintended program actions surface as unintended control flow transfers, although they may begin as abnormal data operations. Invariably, an attack that overwrites data has as its goal a malicious transfer of control.

3.1 Context-Insensitive Policies

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. Our first aim is to determine the points-to sets for function pointers by using an accurate static analysis. We use a flow-insensitive (to allow for concurrency) and context-insensitive analysis 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.

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. For one thing, the model includes the calling convention, which restricts each return to have only one target (the return site of the caller), depending on the context. There are a number of schemes we can use to reduce the size of allowed targets further.

If we assume that serious damage only occurs via system calls, we can perform reachability analysis to identify the system calls accessible from each of the functions. If different targets can reach different system calls, then an attacker has a choice of action for constructing a malicious sequence. However, if the system call sets are all equivalent (in the best case all being empty), we can accept any valid target, because changing a stored pointer from one value to another provides no new abilities to an attacker.

3.2 Context-Sensitive Policies

Even the most accurate flow-sensitive and context-sensitive static analysis will not produce singleton sets. 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.

3.2.1 Selective Code Duplication

We can apply program specialization with respect tofunction pointers passed as arguments, which is a common use case, that way the target set of its later uses is a singleton set. Furthermore, leaf functions can be partially inlined in traces from their callers, therefore they are effectively reduced to singletons. A simple compare with that singleton replaces the hashtable lookup if detection of security violations is desired, otherwise it can be elided in a trace. In order to reduce the degree of freedom of return overwrites, 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, by selectively duplicating code for each definition and use of an indirect branch target, we can obtain selective flow and context sensitivity in execution traces.

3.2.2 Full Context Sensitivity

Full context sensitivity with respect to return addresses may be worth the performance hit, given that return address overwrites are popular security exploit targets. There has been a lot of work on protecting return addresses and on detecting changes in return addresses [9, 18, 41], which are discussed further in Section 7. Here we present two techniques for detecting violations of the calling convention. Our techniques rely on using registers that are isolated from the program in order to provide secure storage without the prohibitive performance cost (and concurrency issues) of unprotecting memory, writing to it, and then re-protecting it. Program shepherding's complete control of all executed code allows us to ensure that our stolen registers are never accessed by application instructions.

Hash Value in Stolen Register. Violations of the calling convention can be detected by keeping a single hash value that is occasionally stored and later checked to ensure that the intervening calls and returns were properly paired up. The hash is kept in a register that is stolen from the application throughout its entire execution. On a call, the value in the register is hashed using an invertible hash function and the new value is written to the register. On a return, the register value is passed through the inverse function and the value prior to the call is restored. A different hash function per call target is used. The idea is that the value can be stored in memory and protected at some point (the "copy point"). Later in the same function, after any number of calls, the value in the register can be compared to the stored value (the "check point"). A discrepancy indicates return address tampering: a different series of calls was executed than should be allowed by the execution model.

If we assume that only system calls can cause serious damage to a machine, then the copy and check points only need to occur in each function that reaches a system call. The copy point is at the start of the body of the function and the check point is immediately prior to the call that reaches the system call. The final check point is prior to the system call itself. Given that return addresses can only be changed to target valid return sites (enforced by policies described earlier), an attacker must arrive at the system call via some call graph transition. By positioning check points prior to every transition that moves toward a system call, calling convention violations that might lead to malicious system call execution can be thwarted.

Entire Call Stack. Intel's processors have included a return stack buffer (RSB) since the Pentium Pro. 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.

As modern processors do not allow software control of the RSB, we have implemented a call stack using the SIMD registers of the Pentium 4. Most programs do not make use of these multimedia processor extensions. The Pentium 4 SSE2 extensions include eight 128-bit registers (the *XMM* registers) that can hold integral values. For a program that does not use these registers, they can be stolen and used as a call stack. Section 5.4 discusses details of our implementation of this stack.

4. PROGRAM SHEPHERDING

Our execution model enforcement employs security policies for program shepherding, which monitors all control transfers to ensure that each satisfies a given 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 the three program shepherding techniques: restricted code origins, restricted control transfers, and un-circumventable sandboxing. The following sections describe these techniques.

4.1 Restricted Code Origins

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

For our prototype, we limit execution model enforcement to those models that do not allow self-modifying code. This is enforced by keeping all original code write-protected and monitoring all system calls that change page protection. Program shepherding's un-circumventable sandboxing, described in Section 4.3, guarantees that these system call checks will never by bypassed. 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 are executed.

4.2 Restricted Control Transfers

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.

Indirect calls, indirect jumps, and returns obtain their targets from data, which can be modified by an attacker. Program shepherding allows arbitrary restrictions to be placed on control transfers in an efficient manner. Enforcing the execution model involves allowing each branch to jump only to a specified set of targets.

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, and thus can be cached and cheaply evaluated with minimal execution overhead. Policies that only examine the target of a control flow transition are the cheapest, as a shared hashtable per indirect transfer type can be used to look up the target for validation [26]. Our policies need to examine both the source and the target of a transition, which can be made as efficient as only checking the target by using a separate hashtable for each source location. The space drawback of this scheme is minor as equivalent target sets can be shared, and furthermore, the hashtables can be precomputed to be kept quite small without increase in access time.

4.3 Un-Circumventable Sandboxing

Program shepherding provides direct support for restricting code origins and control transfers. Execution can be restricted in other ways by adding sandboxing checks on other types of operations. With the ability to monitor all transfers of control, program shepherding is able to guarantee that these sandboxing checks cannot be bypassed. Sandboxing without this guarantee can never provide true security — if an attack can gain control of the execution, it can jump straight to the sandboxed operation, bypassing the checks. In addition to allowing construction of arbitrary security policies, this guarantee is used to enforce the other two program shepherding techniques by protecting the shepherding system itself (see Section 5.2).

5. IMPLEMENTATION

5.1 DynamoRIO

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. A performance-critical inner loop will execute without a single additional instruction beyond the original application code.

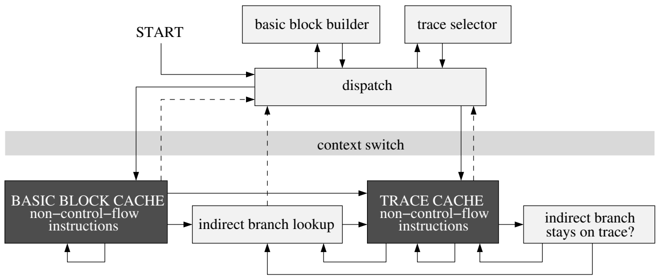

Figure 1: Flow chart of the DynamoRIO system infrastructure. Dark shading indicates application code. Note that the context switch is simply between the code cache and DynamoRIO; application code and DynamoRIO code all runs in the same process and address space. Dotted lines indicate the performance-critical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

We decided to build our program shepherding system as an extension to a dynamic optimizer called DynamoRIO [4, 5]. DynamoRIO is based on an IA-32 port of Dynamo [2]. DynamoRIO's optimizations are still under development. However, this is not a hindrance for our security purposes, as its performance is already reasonable for many applications (see Section 6.2). DynamoRIO is implemented for both IA-32 Windows and Linux, and is capable of running large desktop applications.

A flow chart showing the operation of DynamoRIO is presented in Figure 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.

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 hashtable lookup. Security policies that restrict indirect branch targets are put in place by varying this hashtable lookup.

To improve the efficiency of indirect branches, and to achieve better code layout, basic blocks that are frequently executed in sequence are stitched together into a unit called a *trace*. When connecting beyond a basic block that ends in an indirect branch, a check is inserted to ensure that the actual target of the branch will keep execution on the trace. This check is much faster than the hashtable lookup, but if the check fails the full lookup must be performed. The superior code layout of traces usually amortizes the overhead of creating them and often speeds up the program [2, 33]. For context-insensitive security policies, no extra checks are required when execution continues across an indirect branch in a trace.

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

5.2 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. To secure DynamoRIO, we protect its memory pages. We divide execution into two modes: DynamoRIO mode and application mode. DynamoRIO mode corresponds to the top half of Figure 1. Application mode corresponds to the bottom half of Figure 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 1. DynamoRIO data includes the indirect branch hashtable 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.

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 1. 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.

We should also protect DynamoRIO's Global Offset Table (GOT) [14] by binding all symbols on program startup and then write-protecting the GOT, although our prototype implementation does not yet do this. Also, there are some pathological attacks that could be performed with multiple application threads that we have not yet solved efficiently [26].

5.3 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. Research on pointer analyses [39, 10, 23, 22, 27] offer different tradeoffs between accuracy and scalability. We have employed a context-insensitive, flow-insensitive Andersen-style points-to analysis [39, 16] while scaling very well to the size of our applications. Our current implementation is currently inaccurate only in regards to treatment of `struct`/`union` fields which are ignored and only the base object is looked at and may produce larger sets. According to the notion in [23] it is *field-independent* similar to the works [39, 10, 38]. However, for call-graph construction *field-based* analyses that ignore the base [27] are suggested in [22, 27] to be more accurate. Here we will discuss the performance of the field-independent instance of our analysis.

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 interesting problems in respect to combining the results of the independent local analyses on the shared objects and application executable. Similar modular combination techniques for compile time analyses have been used by Das [10] and [23]. 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 only performs a single level indirection of symbolic arguments to obtain the full 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 a transition in `gcc` which had an incomplete set in the forementioned settings and was flagged as alarming. This particular case would actually be handled by a field-based analysis of structure assignments [27] even across module boundaries.

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. In order to reduce program startup overhead, the fully expanded sets should be precomputed together with the executable, and will need to be recomputed only in the rare case shared libraries are modified. The memory footprint of 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 [14] section memory mapped read-only which holds the fully precomputed points-to sets for each indirect call. Our prototype implementation, however, refrains from binary modifications.

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 experimented so far only with binaries produced by the `gcc` compiler, but any other compiler may be used to build the actual binary, as far as it generates accurate enough debugging information at higher optimization levels. We have applied our postprocess on program executables and shared libraries which were previously built or locally installed. Most calls are sparsely located across function and line boundaries and can therefore be unambiguously matched. 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 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.

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 is opposite to the traditional notion of conservative estimation. A points-to analysis that may miss some potential valid transitions may produce false alarms (false positive) but will not miss a violation of this model. Therefore an omission in the deduced model may cause denial of service to unusual requests with legitimate intent, but it will never miss on a 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, and then use the results of previous runs 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.

benchmark	total	shared	percent shared
ammp	321	4	1.2%
applu	372	2	0.5%
apsi	1153	14	1.2%
art	160	0	0.0%
equake	270	2	0.7%
mesa	400	0	0.0%
mgrid	389	0	0.0%
sixtrack	2707	86	3.2%
swim	393	2	0.5%
wupwise	548	4	0.7%
bzip2	197	0	0.0%
crafty	895	16	1.8%
eon	1866	63	3.4%
gap	1779	39	2.2%
gcc	7723	843	10.9%
gzip	199	0	0.0%
mcf	214	0	0.0%
parser	1061	10	0.9%
perlbmk	2696	106	3.9%
twolf	1012	12	1.2%
vortex	3371	135	4.0%
vpr	1014	12	1.2%
average	1880	116	2.6%

Table 2: Return address sharing for each reference data set run of the SPEC2000 benchmarks, compiled with `gcc -O3`. The first column gives the total number of unique return addresses dynamically encountered. The second column lists the number of addresses that share their least significant 16 bits, while the final column shows the percentage of total addresses that share their bottom bits. For benchmarks with multiple runs, the highest percentage run is shown.

5.4 Calling Convention Enforcement

We have implemented a call stack using the SIMD registers of the Pentium 4. The Pentium 4 SSE and SSE2 extensions add eight 128-bit registers (the *XMM* registers) that can hold single-precision, double-precision, or integral values. For a program that does not make use of these registers, they can be stolen and used as a call stack.

The SSE2 instruction set includes instructions for transferring a 16-bit value into or out of one of the eight 16-bit slots in each XMM register. Unfortunately, storing a 32-bit value is much less efficient. However, just the lower 16 bits of return addresses are sufficient to distinguish over 97% of valid addresses, as shown in Table 2. For a number of applications there are no return addresses that share their least significant 16 bits. Using just the lower 16 bits, then, does not sacrifice much security. It also allows twice as many return address to be stored in our register stack.

We implemented a scheme where the XMM registers form a rotating stack. The final 16-bit slot is used to store the call depth, leaving room for 63 return address entries. On a call, the return address is stored in the first slot and the rest of the slots are shifted over. When the call depth exceeds 63, the oldest 32 values are copied to memory which is then protected. On a return, the first slot's value is compared with the actual return address. Then the

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 3: Static points-to analysis. 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.

slots are all shifted down. When the call depth reaches 0, the most recent stored values are swapped back in to the first 32 register slots. Only copying half of the stack avoids thrashing due to a frequent series of small call depth changes. Expensive memory protetion is only required on every call depth change of 32.

To handle `setjmp()` and `longjmp()`, the `jmp_buf` should be write-protected between the `setjmp()` and the corresponding `longjmp()`, and the return address stack must be unwound to the proper location. We have not implemented this yet, and our system reports calling convention violations when it encounters `longjmp()` (for example, in `perlbmk` in SPEC2000).

6. EXPERIMENTAL RESULTS

This section shows the effectiveness of our system and evaluates its performance on the SPEC2000 benchmarks [37].

6.1 Effectiveness of static analysis

We applied our static points-to analysis and runtime implementation enforcement on indirect branches in the C SPEC2000 benchmarks [37], two 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 summarize the results for the benchmarks with nonempty sets in Table 3. 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. The indirect calls in the executable or the shared object are given for reference. The size of the union of all indirect call targets is provided for comparison with a much simpler technique that allows indirect calls to any address taken function, which is possible with source code access. An even less restrictive analysis that can be applied on unstripped binaries may allow all function entry points in the executable and the shared libraries.

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, for example the maximal set of `sshd` would be three times smaller. In terms of freedom of choice for an attacker, most of the sets usually 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 a target set.

6.2 Performance

Figure 2 shows the performance of our system on a Linux system with a Pentium 4 processor. The figure shows normalized execution time for the SPEC2000 benchmarks [37], compiled with full optimization and run with unlimited code cache space. (Note that we do not have a FORTRAN 90 compiler on Linux.) 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 4.2. The benchmarks marked with an asterisk were not in C and therefore were run with profiling information. The results show that the additional overhead is negligible on most benchmarks.

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

Figure 3 shows the performance of calling convention enforcement using XMM registers, as discussed in Section 3.2.2. Its layout is similar to Figure 2.

The memory usage of our program shepherding system has been presented elsewhere [26].

7. RELATED WORK

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

StackGuard [9] is a compiler patch that modifies function prologues to place "canaries" adjacent to the return address pointer. A stack buffer overflow will modify the "canary" while overwriting the return pointer, and a check in the function epilogue can detect that condition. This technique is successful only against sequential overwrites and protects only the return address.

StackGhost [18] is an example of hardware-facilitated return address pointer protection. It is a kernel modification of OpenBSD that uses a Sparc architecture trap when a register window has to be written to or read from the stack, so it performs transparent `xor` operations on the return address before it is written to the stack on function entry and before it is used for control transfer on function exit. Return address corruption results in a transfer unintended by the attacker, and thus attacks can be foiled.

Techniques for stack smashing protection by keeping copies of the actual return addresses in an area inaccessible to the application are also proposed in StackGhost [18] and in the compiler patch StackShield [41]. 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 [9] have also shown significant performance penalties.

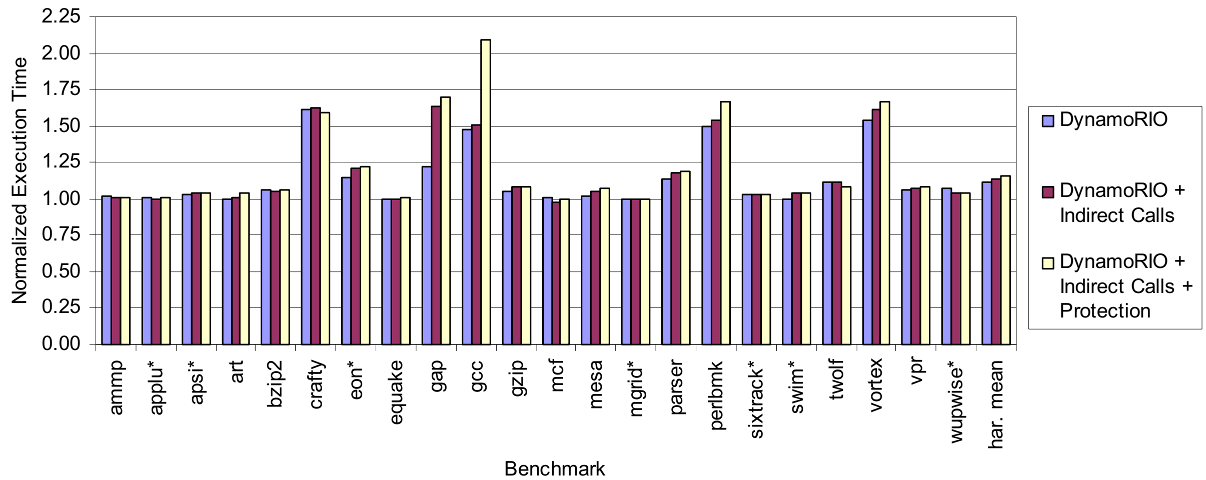

Figure 2: Normalized program execution time for our system (the ratio of our execution time to native execution time) on the SPEC2000 benchmarks (excluding all FORTRAN 90 benchmarks) on Pentium 4 under Linux. They were compiled using `gcc -O3`. The final set of bars is the harmonic mean. The first bar 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.

FormatGuard [8] is a library patch for dynamic checks of format specifiers to detect format string vulnerabilities in programs which directly use the standard `printf` functions.

Static analyses have also been applied for detection with relatively low false positive rate of common buffer overflow [43] and format string [36] vulnerabilities.

Enforcing non-executable permissions on IA-32 via OS kernel patches has been done for stack pages [13] and for data pages in PaX [32]. Our previous work [26] 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 [28] which extends the C type system, infers statically verifiable type safe pointers, and adds run time checks only for unsafe pointers. Cyclone [24] 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.

Other programming bugs stemming from violations of specific higher level semantic rules of safe programming have been targeted by static analyses like CQUAL [17], ESP [11], MC [20], and static model checkers SLAM [40], MOPS [6]. 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 [40]. Our system may be used to complement these and enforce the execution model of the application.

Our system is close in spirit to the hybrid approach of using static analysis and runtime model checking in [42]. A static analysis to construct a model of the system calls possibly generated by a program and a runtime component to verify that. The system call model is generated from an assumed valid execution model — context-insensitive represented as call graph, or context-sensitive with stack enforcement. Our system is as at least as accurate in detection of disallowed system call sequences, since it disallows deviations from the chosen execution model. Therefore our techniques subsume the need to further model and dynamically check system calls, and we present a practical system with minimal overhead. The *mimicry attacks* introduced [42] and further analyzed by Wagner [44] show how attackers can easily evade intrusion detection at the system call level. We have also outlined [26] a simple mimicry attack violating information flow [21].

Software fault isolation [45] modifies a program binary to restrict the address range of memory operations. Execution monitors [34] were applied in SASI [15] to enforce a memory model via static code instrumentation.

The Andersen's [1] style points-to analysis that we're employing in this work uses *projection merging* [39] and cycle elimination [16] implemented using the Banshee [3] analysis toolkit to build a customized constraint resolution engine. The C front end is derived from David Gay's Region Compiler [19] and the GNU C Compiler. Other points-to analyses for C have been presented in [10, 23], and specifically applied to callgraph construction [22, 27].

Our system infrastructure itself, DynamoRIO [4, 5], is based on an IA-32 port of Dynamo [2]. Other software dynamic optimizers are Wiggins/Redstone [12], which employs program counter sampling to form traces that are specialized for the particular Alpha machine they are running on, and Mojo [7], which targets Windows NT running on IA-32. None of the above has been used for anything other than optimization. Strata [35] uses dynamic translation with lower performance to enforce a subset of the techniques we have presented earlier [26].

8. CONCLUSIONS

In this paper, 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 with an efficient enforcement of the execution model.

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.

9. REFERENCES

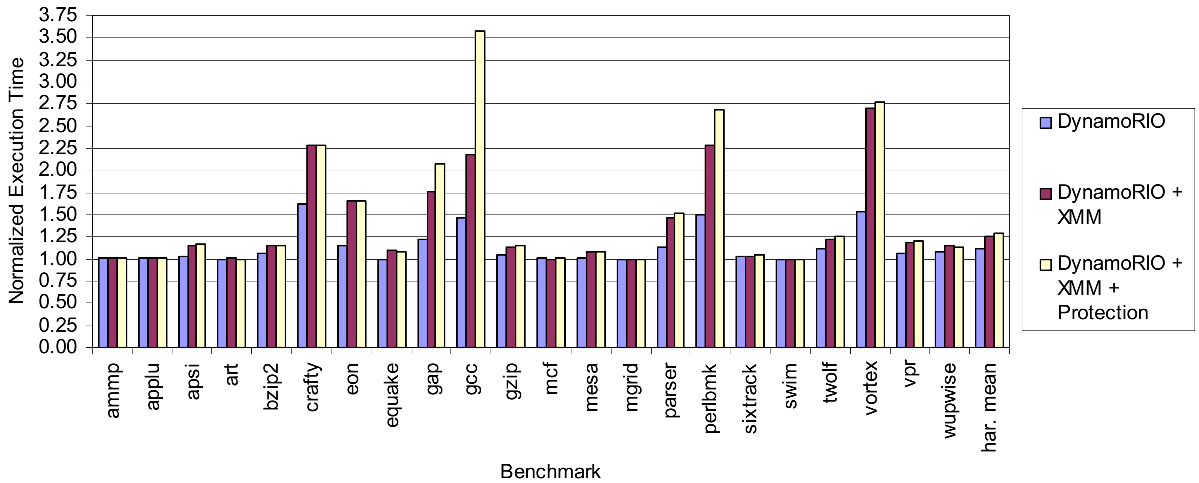

Figure 3: Normalized program execution time for our system (the ratio of our execution time to native execution time) on the SPEC2000 benchmarks (excluding all FORTRAN 90 benchmarks) on on Pentium 4 under Linux. They were compiled using `gcc -O3`. The final set of bars is the harmonic mean. The first bar is for DynamoRIO by itself. The middle bar shows the overhead of our calling convention enforcement using the XMM registers as a return address stack. The final bar shows the same performance with full memory protection for DynamoRIO and the return address stack overflow/underflow memory areas.

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

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

[4] 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.

[5] 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.

[6] Hao Chen and David Wagner. MOPS: An infrastructure for examining security properties of software. In *To appear at ACM CCS*, 2002.

[7] 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.

[8] 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.

[9] 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.

[10] Manuvir Das. Unification-based pointer analysis with directional assignments. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation PLDI00*, Vancouver, BC, Canada, June 2000.

[11] 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.

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

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

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

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

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

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

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

[27] 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).

[28] 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.

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

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

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

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

[33] 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.

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

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

[36] 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.

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

[38] 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.

[39] 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.

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

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

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

[43] 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.

[44] David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *To appear at ACM CCS*, 2002.

[45] 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.